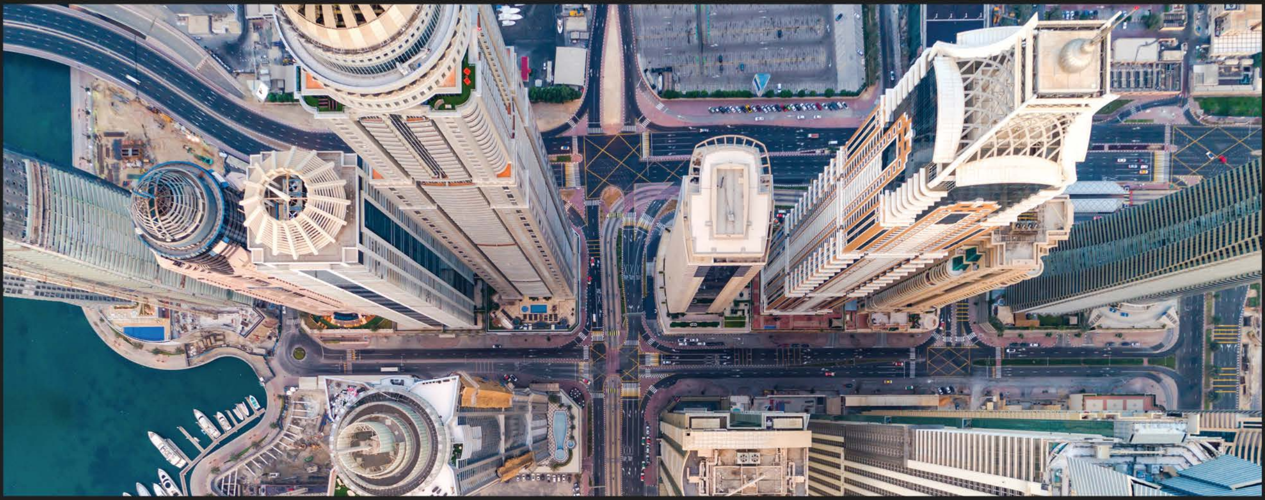


1ST EDITION

Implementing GitOps with Kubernetes

Automate, manage, scale, and secure infrastructure and cloud-native applications on AWS and Azure

PIETRO LIBRO | ARTEM LAJKO

Implementing GitOps with Kubernetes

Automate, manage, scale, and secure infrastructure and cloud-native applications on AWS and Azure

Pietro Libro

Artem Lajko

packt

Implementing GitOps with Kubernetes

The authors acknowledge the use of cutting-edge AI, such as ChatGPT, with the sole aim of enhancing the language and clarity within the book, thereby ensuring a smooth reading experience for readers. It's important to note that the content itself has been crafted by the author and edited by a professional publishing team.

Group Product Manager: Preet Ahuja
Publishing Product Manager: Surbhi Suman
Book Project Manager: Srinidhi Ram
Senior Editor: Adrija Mitra
Technical Editor: Yash Bhanushali
Copy Editor: Safis Editing
Proofreader: Adrija Mitra
Indexer: Hemangini Bari
Production Designer: Prafulla Nikalje
DevRel Marketing Coordinator: Rohan Dobhal

First published: August 2024
Production reference: 1260724

Published by Packt Publishing Ltd.

Grosvenor House
11 St Paul's Square
Birmingham
B3 1RB, UK

ISBN 978-1-83588-422-5

`www.packtpub.com`

To my wife and life partner, Eleonora, who supports my follies, encourages me, and allows me to move forward.

To my daughter, Giulia, who sacrificed spending part of her time with me to allow me to write this book.

To my parents, Filomena and Antonino, who enabled me to study.

To everyone who has believed in me.

– Pietro Libro

To my girlfriend, my family, and my friends, who put up with me through countless nights and weekends as I crafted this book!

To my sparring partner, Steve Golling, who co-created the craziest solutions with me and inspired me over the years!

And to everyone I've had the pleasure of working with over the past years, whose impact through our exchanges and collaborations has been invaluableyou know who you are (and if you don't, just assume I mean you!). Thank you!

– Artem Lajko

Contributors

About the authors

Pietro Libro is a tech enthusiast with over two decades of experience in software development and software architecture. His pragmatic problem-solving skills have been honed through work in the public administration, finance, and automation industries. He holds a master's degree in computer science from the University of Rome, *La Sapienza*. Over the years, Pietro has transitioned from software development to a solution and cloud architect role. He is currently awaiting the defense of his PhD in bioinformatics at the University of Tuscia. Pietro's dedication to learning is evident through his numerous certifications and his role as a technical speaker. Specializing in various technologies, especially software and cloud architecture, he relocated from Italy to Switzerland. Currently serving as a cloud solution architect in Zürich, Pietro lives with his wife, Eleonora, his daughter, Giulia, and their cat, "Miau". In his free time, Pietro enjoys biking, practicing taekwondo, watching science fiction movies and series, and spending time with his family.

Artem Lajko is a passionate and driven platform engineer and Kubestronaut, boasting over eight years of IT experience, backed by a master's degree in computer science. His track record showcases expertise in designing, developing, and deploying efficient and scalable cloud infrastructures. As a curious and continuous learner, Artem holds certifications in Azure, AWS, Kubernetes, and GitOps. Currently, he's playing a pivotal role in enhancing innovation and application management at the Port of Hamburg. His technical acumen spans cloud infrastructures, cross-cloud solutions, and DevOps practices. He is also passionate about blogging and networking with manufacturers to craft top-notch solutions using market-available tools.

About the reviewers

Chen Fliesher has worked in the IT industry for over 20 years, with more than 7 years of experience as a DevOps engineer specializing in CI/CD and the cloud. For the last five years, he has been dedicated to implementing the GitOps methodology at AT&T. He serves as a developer, DevOps engineer, and architect on IoT, AI, and other projects.

Chen received his BSc in Mathematics and Computer Science (cum laude) from Bar-Ilan University. He is currently employed by AT&T as a DevOps Tech Lead in the software delivery service. In this role, he assists the development team in accelerating the migration of both legacy and new services from on-premises to the cloud. Additionally, he has given several talks at events such as Kubernetes Community Days and Jenkins User Conference, and he mentors and teaches at AT&T.

I'd like to thank my family and friends, who understand the time and commitment it takes me to become a subject matter expert in the field of software delivery and the cloud.

I thank my wife, Nili, and my three daughters, who allowed me the time to learn, research, and practice in the field of progressive delivery.

Thanks to all my peers who assist me in delivering talks on KCD, JUC, and others.

Never stop learning.

Dhirendra Kumar is a cloud (certified) Kubernetes and platform architect. He has a master's degree in computer science and has worked with many global tech companies (IBM, HPE, Medidata, and so on) for the past 20+ years. Dhirendra has helped design, develop, and deploy many solutions and advise companies as they navigate the challenges of moving cloud projects from concept to production. This gives him a rare mix of knowledge in both the development and cloud computing spaces.

Prasanna Jatla is currently working as a senior software engineer at American Express, leveraging over 11 years of extensive experience in the financial industry. His expertise lies in the development of scalable, distributed software applications. He has spearheaded numerous modernization projects and automation initiatives. His skill set extends to the development of event-driven applications and batch-processing systems, where he has demonstrated proficiency in driving efficiency and performance.

Table of Contents

Part 1: Understanding GitOps via Uncomplicated Orchestrations/ Kubernetes

1

An Introduction to GitOps

2

Navigating Cloud-native Operations with GitOps

3

Version Control and Integration with Git and GitHub 51

4

Kubernetes with GitOps Tools 73

Part 2: Harnessing Advanced Orchestrations, Culture, and Control in GitOps Practices

5

6

7

Part 3: Hands-on Automating Infrastructure and CI/CD with GitOps

8

9

10

11

Deploying Real-World Projects with GitOps on Kubernetes 295

Part 4: Operational Excellence Through GitOps Best Practices

12

13

14

Preface

Hello there!

In the rapidly evolving world of software development, maintaining consistency, scalability, and reliability in deployments is a significant challenge. GitOps has emerged as a revolutionary approach to bridge the gap between development and operations, especially in Kubernetes environments. By using Git as the single source of truth for system and application configurations, GitOps automates and standardizes deployments, ensuring they are consistent, auditable, and efficient.

GitOps leverages the principles of version control, infrastructure as code, and continuous deployment to create a seamless, transparent workflow. This methodology not only simplifies operations but also enhances collaboration and accelerates delivery times, allowing teams to focus more on innovation rather than firefighting.

This book, *Implementing GitOps with Kubernetes: Automate, manage, scale, and secure infrastructure and cloud-native applications on AWS and Azure*, aims to provide a comprehensive guide to mastering GitOps. Through practical examples, step-by-step tutorials, and insights from industry experts, we will explore how to effectively implement GitOps practices in your Kubernetes deployments.

Our journey begins with an introduction to the core concepts and principles of GitOps. We will delve into the technical intricacies of tools such as Argo CD, Flux CD, Helm, and Kustomize, and how they integrate with Kubernetes. From there, we will tackle advanced topics such as multi-cluster management, security, and scalability, ensuring you have a holistic understanding of GitOps.

Drawing from our extensive experience in cloud architecture and DevOps practices, we will share real-world scenarios and best practices that have been tested and validated in various industries. By the end of this book, you will be equipped with the knowledge and skills to implement GitOps strategies in your current or future Kubernetes deployments, ensuring reduced complexity and increased scalability.

Whether you are a DevOps engineer, site reliability engineer, platform engineer, or cloud architect, this book will provide you with the tools and insights needed to succeed in today's cloud-native landscape. Let's embark on this journey together and unlock the full potential of GitOps for your Kubernetes deployments.

Who this book is for

Implementing GitOps with Kubernetes: Automate, manage, scale, and secure infrastructure and cloud-native applications on AWS and Azure is designed for professionals looking to enhance their skills in deploying and managing Kubernetes environments using GitOps principles.

The primary audience of this book includes the following:

- **DevOps engineers**: Professionals responsible for managing and automating the deployment of applications and infrastructure. This book will provide them with advanced GitOps techniques to streamline their workflows, reduce deployment errors, and ensure consistency across environments.
- **Site reliability engineers (SREs)**: Engineers focused on maintaining the reliability and performance of applications. This book will offer insights into integrating GitOps for better observability, automated recovery, and efficient scaling of Kubernetes clusters.
- **Platform engineers**: Individuals who build and maintain the underlying platforms that support application development and deployment. They will learn how to implement GitOps to manage infrastructure as code, ensuring their platforms are robust, scalable, and secure.
- **Cloud engineers**: Engineers working with cloud platforms who need to manage complex Kubernetes environments. This book will teach them how to leverage GitOps to automate deployments, manage multi-cloud setups, and optimize cloud resource utilization.
- **Software engineers**: Developers who are looking to understand the deployment process better and contribute to infrastructure management. This book will provide them with a comprehensive understanding of GitOps practices, enabling them to collaborate more effectively with operations teams.
- **Solution architects**: Professionals responsible for designing and implementing technical solutions. They will gain a deeper understanding of how to incorporate GitOps into their architecture designs, ensuring scalable and maintainable solutions.
- **IT leaders and managers**: Leaders who oversee the implementation of DevOps and cloud-native strategies within their organizations. This book will help them understand the benefits of GitOps, guiding them in making informed decisions about adopting and scaling GitOps practices in their teams.

The target audience of this book should ideally have the following background:

- **Knowledge of cloud computing**: Readers should have a foundational understanding of cloud computing concepts and environments, as GitOps practices often involve deploying applications on cloud platforms.
- **Familiarity with continuous integration and continuous deployment (CI/CD) principles**: A basic grasp of CI/CD principles is essential, as GitOps builds upon these methodologies to automate and streamline deployments.

- **Basic Kubernetes understanding**: Prior experience with Kubernetes is highly beneficial, as this book delves deep into integrating GitOps with Kubernetes environments. Readers should be familiar with Kubernetes fundamentals such as pods, services, and deployments.
- **Experience with version control systems**: Since GitOps relies heavily on Git for version control, readers should have experience with Git or similar version control systems. This includes understanding branching, merging, and managing repositories.
- **DevOps tools and practices**: Familiarity with DevOps tools such as Docker, Helm, and other orchestration utilities will help readers grasp the advanced topics discussed in the book more efficiently.

By the end of this book, readers will be equipped with the knowledge and practical skills needed to implement GitOps strategies effectively in their Kubernetes deployments, ensuring reduced complexity, increased scalability, and improved operational efficiency.

What this book covers

Chapter 1, *An Introduction to GitOps*, provides a foundational understanding of GitOps, exploring its principles and how it transforms culture, workflows, and mindsets in modern software development.

Chapter 2, *Navigating Cloud-Native Operations with GitOps*, delves into building and managing containerized applications using GitOps practices, covering topics such as Kubernetes fundamentals, container image optimization, and cloud-native pipelines.

Chapter 3, *Version Control and Integration with Git and GitHub*, explains the pivotal role of Git and GitHub in GitOps, offering insights into effective version control and collaborative development practices.

Chapter 4, *Kubernetes with GitOps Tools*, explores various GitOps tools such as Helm, Kustomize, Argo CD, and Flux CD, detailing their integration with Kubernetes and providing a comparative analysis to help choose the right tool for specific needs.

Chapter 5, *GitOps at Scale and Multitenancy*, discusses advanced GitOps practices for scaling deployments and managing multi-cluster environments, including strategies for effective Git repository management and building a service catalog for Kubernetes.

Chapter 6, *GitOps Architectural Designs and Operational Control*, focuses on architectural frameworks and operational methodologies for GitOps, emphasizing scalability, resilience, and efficiency in cloud-native deployments.

Chapter 7, *Cultural Transformation in IT for Embracing GitOps*, highlights the cultural shift required to adopt GitOps, discussing principles of infrastructure as code, immutable infrastructure, DORA metrics, and overcoming organizational resistance.

Chapter 8, *GitOps with OpenShift*, provides an in-depth exploration of applying GitOps principles within the Red Hat OpenShift environment, including setting up GitOps workflows, leveraging OpenShift's CI/CD tools, and securing GitOps pipelines.

Chapter 9, *GitOps for Azure and AWS Deployments*, covers the implementation of GitOps practices within Azure and AWS ecosystems, detailing the integration of cloud-native tools and services to streamline application and infrastructure management.

Chapter 10, *GitOps for Infrastructure Automation – Terraform and Flux CD*, delves into the integration of Terraform and Flux CD for automating infrastructure management, covering version control, multi-environment management, and advanced automation techniques.

Chapter 11, *Deploying Real-World Projects with GitOps on Kubernetes*, provides a hands-on guide to executing real-world projects using GitOps and Kubernetes, from setting up development environments to designing, developing, and deploying scalable applications.

Chapter 12, *Observability with GitOps*, explores the integration of observability practices into GitOps workflows, covering SRE principles, internal versus external observability, SLO-driven performance, and advanced monitoring techniques.

Chapter 13, *Security with GitOps*, discusses the security aspects of GitOps, including hardening declarative CD, implementing policy-as-code, managing secrets, maintaining platform catalogs, and automated security scanning.

Chapter 14, *FinOps, Sustainability, AI, and Future Trends for GitOps*, highlights the fusion of FinOps with GitOps for sustainable and cost-effective operations, covering cost forecasting, optimization, carbon footprint assessment, AI-driven automation, and future trends in GitOps.

To get the most out of this book

To get the most out of this book, readers should have a foundational understanding of cloud computing and DevOps principles. Familiarity with Kubernetes and containerization technologies, such as Docker, is essential. Experience with version control systems, particularly Git, will be beneficial as GitOps heavily relies on these tools. Basic knowledge of CI/CD pipelines and infrastructure as code concepts will also help readers grasp the advanced topics covered in the book.

Software/hardware mentioned in this book	**OS requirements**
Kubernetes	Windows, macOS, or Linux
Git	Windows, macOS, or Linux
Docker	Windows, macOS, or Linux
Argo CD	Windows, macOS, or Linux
Flux CD	Windows, macOS, or Linux
Helm	Windows, macOS, or Linux
Kustomize	Windows, macOS, or Linux

Software/hardware mentioned in this book	OS requirements
Terraform	Windows, macOS, or Linux
Azure Kubernetes Service (AKS)	Windows, macOS, or Linux
AWS Elastic Kubernetes Service (EKS)	Windows, macOS, or Linux
OpenShift	Windows, macOS, or Linux

This book includes all the necessary instructions to work through each chapter. Step-by-step guides are provided to ensure a smooth setup and implementation process. The source code for the examples and projects discussed in the book is available in a public repository. Please refer to the next page for the repository link and further details.

If you are using the digital version of this book, we advise you to type the code yourself or access the code from the book's GitHub repository (a link is available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`. If there's an update to the code, it will be updated in the GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "There are globally defined values under `optimization/opencost`, and then there's a custom pricing model for each specific country."

A block of code is set as follows:

```
helm repo add sealed-secrets https://bitnami-      labs.github.io/
sealed-secrets
helm install sealed-secrets sealed-secrets/sealed-secrets
#Install e.g. CLI on MacOS
brew install kubeseal
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
clusterName: "aks-excelsior-development-2"
globalConfig:
  chat_gpt_token: sk-dw******
  signing_key: ea657a******
  account_id: 7935371f******
sinksConfig:
- slack_sink:
    name: main_slack_sink
    slack_channel: pocs
    api_key: xoxb******
- robusta_sink:
    name: robusta_ui_sink
    token: eyJhY2NvdW******
enablePrometheusStack: true
# This part is added to the default generated_values.yaml
enablePlatformPlaybooks: true
runner:
  sendAdditionalTelemetry: true
rsa:
  private: ******
  public: ******

# This part is added to the default generated_values.yaml
playbookRepos:
  chatgpt_robusta_actions:
    url: "https://github.com/robusta-dev/kubernetes-chatgpt-bot.git"

# This part is added to the default generated_values.yaml
customPlaybooks:
# Add the 'Ask ChatGPT' button to all Prometheus alerts
- triggers:
  - on_prometheus_alert: {}
  actions:
  - chat_gpt_enricher: {}
```

Any command-line input or output is written as follows:

```
$ aws eks --region eu-central-1 update-kubeconfig --name
eksgitopscluster
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For instance, words in menus or dialog boxes appear in **bold**. Here is an example: "When a **Common Vulnerabilities and Exposures** (**CVE**) is revealed, and you opt for the cockpit and fleet approach, adopting GitOps at scale also facilitates a vulnerability management at scale strategy."

Tips or important notes

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, email us at `customercare@packtpub.com` and mention the book title in the subject of your message.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/support/errata` and fill in the form.

Piracy: If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Share Your Thoughts

Once you've read *Implementing GitOps with Kubernetes*, we'd love to hear your thoughts! Please click here to go straight to the Amazon review page for this book and share your feedback.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781835884225`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly

Part 1: Understanding GitOps via Uncomplicated Orchestrations/ Kubernetes

In this part, you will explore the foundational aspects of GitOps. Starting with a broad overview, you will understand how GitOps has emerged as a key practice in modern software development and platform engineering. We will delve into navigating cloud-native operations, emphasizing the integration of Git and GitHub for effective version control. Additionally, you will utilize Kubernetes with various GitOps tools to streamline deployment processes, offering a comprehensive understanding of how these technologies interconnect to simplify and enhance software deployment and management.

This part includes the following chapters:

- *Chapter 1*, An Introduction to GitOps
- *Chapter 2*, Navigating Cloud-Native Operations with GitOps
- *Chapter 2*, Version Control and Integration with Git and GitHub
- *Chapter 4*, Kubernetes with GitOps Tools

1

An Introduction to GitOps

In this opening chapter, we'll delve into GitOps, an innovative approach that's revolutionizing software development and operations. This chapter begins with the tale of GitOps, focusing on its transformative effect on organizational culture and workflows. The *Fundamentals of GitOps* section outlines its core principles, such as the centralization of Git, its declarative approach, and automated synchronization. *Traditional CI/CD with DevOps against GitOps* traces the evolution of software deployment, contextualizing GitOps in the broader history of the field. This chapter will also contrast traditional CI/CD and DevOps methodologies with GitOps, highlighting GitOps as an evolution in managing complex systems. We'll conclude by discussing the advantages of adopting GitOps for platform engineering while illustrating its efficiency, consistency, and adaptability.

In this chapter, we'll cover the following topics:

- GitOps unveiled – reshaping development culture and practices
- The fundamentals of GitOps and the advantages of adopting it for platform engineering
- Why GitOps?
- The integration between GitOps, IaC, and Kubernetes

Technical requirements

To follow along with this chapter, you need to have basic knowledge of **version control**, **Infrastructure as Code** (**IaC**), **continuous integration** (**CI**), **continuous delivery** (**CD**), and Kubernetes. Basic knowledge of major cloud providers, such as **Google Cloud Platform** (**GCP**), **AWS**, and Azure, will be beneficial.

GitOps unveiled – reshaping development culture and practices

In the realm of modern software development and operations, GitOps emerges as a confluence of ideas and practices, redefining the approach to managing complex systems.

The term GitOps was first introduced by *Alexis Richardson*, the co-founder of **Weaveworks**, in August 2017 through a series of blog posts. Richardson described GitOps as a methodology where developer tools are pivotal in driving operational procedures. This approach emphasized the significance of using declarative tools and advocated for the best practices wherein configurations are treated as code. Consequently, these configurations should be subject to version control, aligning with the core principles of GitOps.

This innovative concept marked a shift in how software development and IT operations teams approach how infrastructure and applications are managed and deployed, promoting a more efficient, transparent, and reliable process.

GitOps is a paradigm that represents a breakthrough in how software development and operations teams manage and deploy applications and infrastructure. It revolves around the idea of using **Git**, a widely used version control system, as the single source of truth for declarative infrastructure and applications.

Version control, also known as source control, is a system that records changes to a file or set of files over time so that you can recall specific versions later. It allows multiple people to work on a document simultaneously, and version control systems provide a way to manage these changes among multiple people.

Traditional CI/CD with DevOps against GitOps

In traditional IT operations, as shown in *Figure 1.1*, the development team periodically hands over new software versions to a **quality assurance** (**QA**) team. This team tests the software and passes it to the operations team for deployment.

As software release cycles become shorter, traditional operations models struggle to keep up. The operations team oversees configuring the infrastructure for new software versions while focusing on system reliability, resilience, and security:

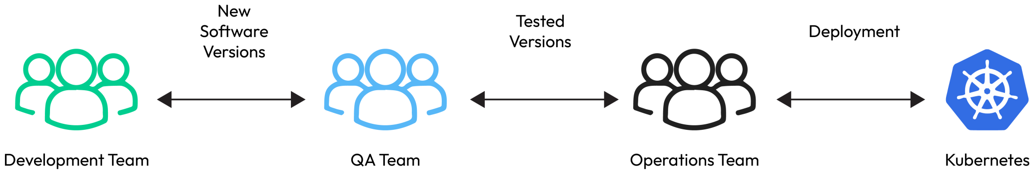


Figure 1.1 – Example of a traditional IT operations workflow

Managing infrastructure **without** advanced frameworks can be complex and requires specialized knowledge. IT operations encompass all processes and services provided by IT staff to meet a business's technological requirements. This includes handling maintenance work and responding to customer issues. In traditional models, development, QA, and operations teams often work under different management structures, so detailed handoff processes and thorough documentation are crucial:

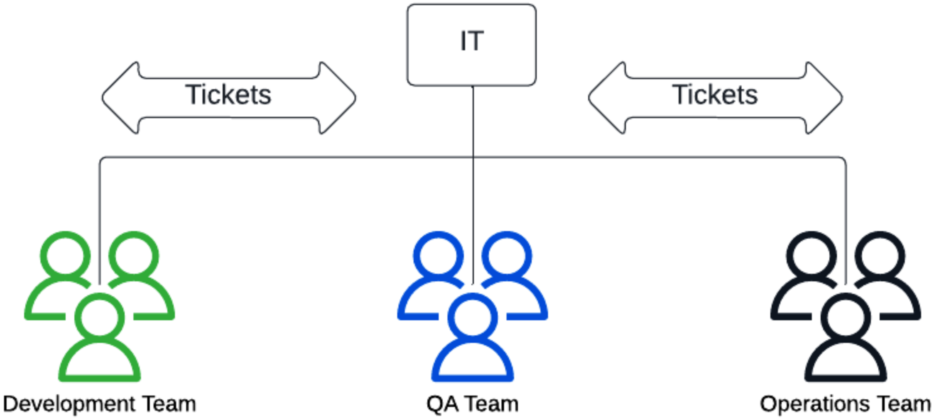


Figure 1.2 – Traditional organization model with separated teams

The traditional organizational model, which is characterized by distinct, separated teams, tends to **complicate** the delivery process. This separation often leads to slower deployments and reduced deployment frequencies. Additionally, the need to transition tasks between teams heightens the risk of miscommunication, which can result in testing oversights and deployment errors.

Most development teams use automated build systems and CI to compile, test, and produce deployable artifacts.

CI is a foundational process in modern software development that automates building, testing, and packaging applications, as shown in *Figure 1.3*:

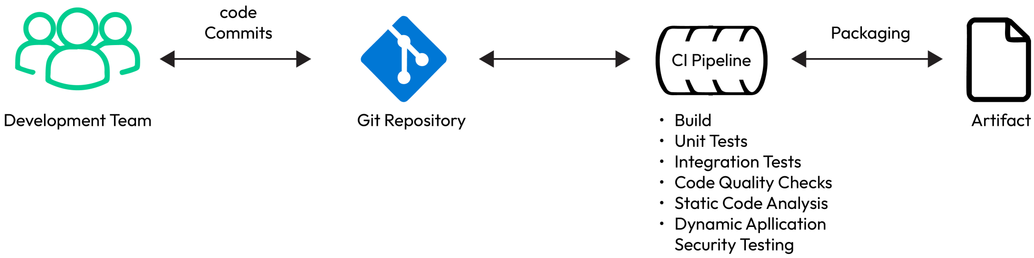


Figure 1.3 – Example of a CI workflow

In this system, software engineers frequently commit code changes to a central repository (for example, a **Git repository**), where these changes are automatically built, tested, and prepared for integration with the main code branch. This approach ensures that the software is always in a state ready for production deployment, highlighting the importance of CI in maintaining the continuous flow of software development.

CI's automated processes are crucial for ensuring the quality and integrity of code before it merges with the main branch. By automating these stages, CI minimizes the risks associated with human error and speeds up the development cycle. However, the process often halts at the deployment stage, which is still predominantly manual in many organizations. This manual deployment can involve complex procedures or semi-automated scripts and often requires operations engineers to physically transfer and restart applications on servers. Such manual interventions are prone to errors and lack robust control mechanisms, making them inefficient and risky.

Building on the foundations of CI, **CD** takes automation a step further:

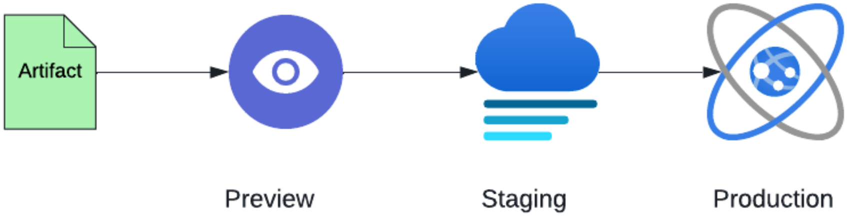


Figure 1.4 – Example of a CD workflow

CD is a practice where code changes, once they pass through CI's build and testing phases, are automatically prepared and ready for release to a production environment. It always keeps the software in a **release-ready** state, dramatically simplifying the process of releasing updates and new features.

CD versus continuous deployment

CD and continuous deployment are closely related practices in software development that streamline the release process. CD ensures every code change is automatically built, tested, and prepared for release, allowing for manual deployment to production. Continuous deployment goes a step further by automatically deploying every change that passes testing directly into production, eliminating manual steps. This accelerates the release process but demands high confidence in automation and testing protocols.

CD not only reduces the risks associated with deploying releases by enabling smaller and more frequent updates that are easier to manage than larger, infrequent ones, but it also significantly shortens prolonged testing cycles typically necessary for extensive code changes. This method enhances collaboration among development, operations, and other teams, aligning all parties toward the shared goal of rapidly and efficiently delivering high-quality software. CD efficiently manages and mitigates potential risks associated with large-scale code modifications, fostering a more agile and responsive software development life cycle.

CD empowers businesses to quickly adapt to market changes and customer feedback, maintaining a competitive edge in innovation and customer satisfaction.

DevOps marks a significant shift in both organizational framework and mindset, placing a strong emphasis on automation. DevOps, in essence, acts as the glue that binds the technical efficiencies of CI/CD with the strategic goals of the organization. It emphasizes continuous feedback, monitoring, and optimization throughout the development process, ensuring continuous improvement post-deployment. This approach ensures that software development is not just a linear process ending with deployment but a cyclical one, where feedback and performance data feed back into the development process.

GitOps is an influential operational model that builds upon the core practices of DevOps, which are widely adopted in application development, and applies them to infrastructure automation. It integrates essential principles such as version control, collaborative work, adherence to compliance standards, and the use of CI/CD tools. This strategy facilitates effective and dependable management of IaC processes.

By integrating DevOps into the CI/CD pipeline, organizations can create a more holistic, efficient, and collaborative environment. This not only accelerates the pace of innovation and delivery but also significantly enhances product quality and customer satisfaction, positioning the organization competitively in an ever-evolving digital landscape.

In this model, the traditional roles of deployment and operations are transferred from the operations team to the development team of the application. DevOps integrates software **development** (**Dev**) with IT **operations** (**Ops**), aiming to reduce the overall system development life cycle. This integration facilitates frequent delivery of features, fixes, and updates, ensuring they align closely with business goals.

The following are some key advantages of adopting DevOps:

- Enhanced collaboration between the development and operations teams
- Improvement in the quality of the products developed
- An increase in the frequency of releases
- A reduction in the time needed to bring new features to market
- Lower costs in design, development, and operational processes
- Reduced **mean time to recovery** (**MTTR**), enhancing system resilience and minimizing the impact of disruptions

The evolution from DevOps to GitOps represents a natural progression in the quest for more efficient and reliable software development and operations practices. While DevOps focuses on breaking down the barriers between software development and operations while fostering a culture of collaboration and continuous improvement, GitOps takes this a step further by centering the entire operational workflow around the Git version control system. GitOps leverages Git as the single source of truth for declarative infrastructure and application.

Reduced MTTR

Reducing the MTTR refers to improving one of the key **DevOps Research and Assessment** (**DORA**) metrics, which is the average time it takes to recover from a failure or outage in a system or service. This metric is crucial in understanding and enhancing the resilience and reliability of IT operations. By reducing the MTTR, organizations can ensure that their systems are restored more quickly after a disruption, minimizing the impact on users and business operations. This improvement is often achieved through more effective incident management processes, automation, and better preparedness for handling failures.

This approach not only streamlines deployment and management processes but also enhances transparency, accountability, and reproducibility.

The transition from DevOps to GitOps is driven by the need for greater automation, consistency, and security in managing complex and dynamic cloud-native environments.

GitOps automates the deployment process using Git's powerful version control capabilities, which means every change in the system can be tracked, reversed, and subject to peer review processes, just like code changes in software development. This shift brings the rigor and precision of software development to the world of operations, significantly reducing the potential for human error and improving the stability and security of production environments.

Moreover, GitOps aligns perfectly with the containerized and microservices-oriented architectures that are prevalent in modern cloud environments, enabling faster and more efficient deployment pipelines that are crucial for businesses to remain agile and responsive in a rapidly changing digital landscape.

In the following sections of this chapter, we will delve deeper into the benefits of adopting GitOps. We will explore why it should be used, its integration with Kubernetes, and other tools such as Argo CD and Flux CD.

The fundamentals of GitOps and the advantages of adopting it for platform engineering

DevOps emerged as a response to these silos, blending the roles of development and operations. It extended the principles of CI/CD to include a cultural shift toward collaboration, aiming to break down the barriers between developers and IT operations. The emphasis was on a holistic approach to the software life cycle, ensuring that both development and operational considerations were integrated from the start. DevOps embraced practices such as IaC and monitoring, fostering a more responsive and agile environment.

GitOps, a more recent innovation, builds upon these foundations but introduces a significant shift: it takes the principles of DevOps and CI/CD and centralizes them around the Git repository.

In a GitOps story, as illustrated in *Figure 1.5*, everything begins with a Git repository, where the entire state of a system – code, configuration, environment settings, and even documentation – is stored and versioned. This approach allows for a high level of transparency, traceability, and audibility as every change is recorded and can be traced back to a commit:

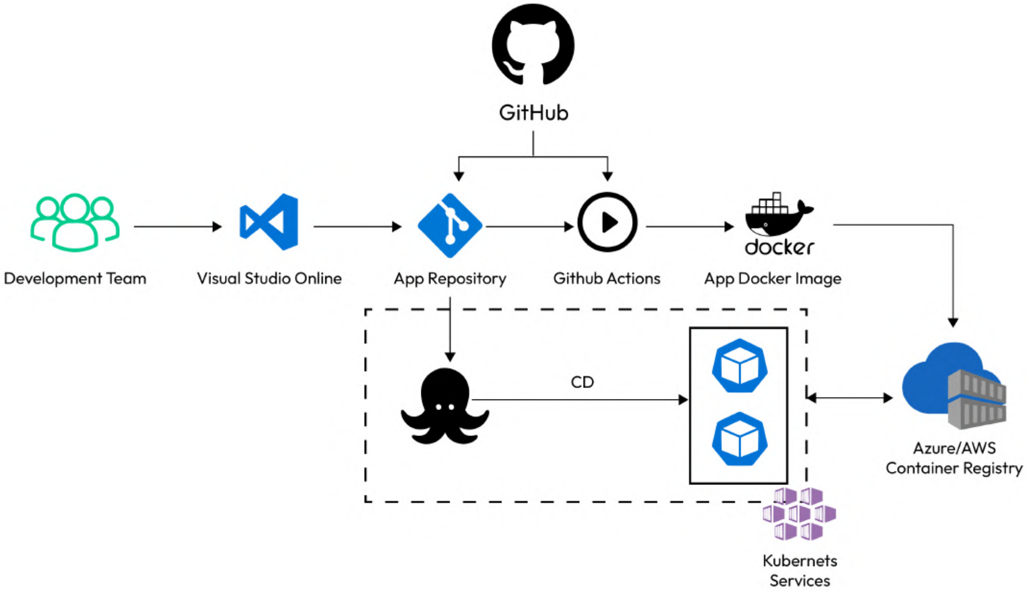


Figure 1.5 – Example of a GitOps pipeline embracing DevOps

In GitOps, Git is not just a version control system; it's the single source of truth for both application code and infrastructure configuration. This approach leverages the strengths of Git – such as version control, collaboration features, and the pull request workflow – to manage the entire deployment and operational process. GitOps emphasizes a **declarative approach**, where the desired state of the system is defined in Git and automated processes ensure that the actual state aligns with this desired state.

Everything from application code to infrastructure configuration is stored in Git repositories. This centralization not only simplifies the process of managing and tracking changes but also ensures consistency and transparency across all stages of development and deployment.

The workflow in a GitOps scenario is elegantly automated and highly efficient. Developers submit changes via pull requests, which then undergo automated tests and reviews. Once approved, these changes are automatically deployed to production environments, aligning with the principles of CI/CD seen previously.

Pull request

A **pull request** is a feature in version control systems, particularly in Git, that facilitates collaboration in software development. Acting as a checkpoint, a pull request allows developers to inform others about changes they've pushed to a repository on a hosting service such as GitHub or GitLab. After pushing a branch with updates, a developer can initiate a pull request to merge these changes into the main branch. This process invites review and discussion on the proposed changes, ensuring that code is vetted and refined collaboratively before being integrated, thus enhancing code quality and shared understanding among team members.

This automation not only streamlines the deployment process but also significantly reduces the chances of human error. Moreover, because the entire system state is version-controlled, rolling back to a previous state in case of an issue is straightforward.

This process, often termed **Git-centric**, empowers teams to manage infrastructure and applications with the same tools and processes used in code development, bringing a unified approach to software delivery.

GitOps represents a significant evolution from traditional practices that leverages the strengths of Git, automation, and declarative models. This methodology provides a framework for realizing the results of DevOps culture, offering an agnostic approach that can be implemented with various tools, such as Kubernetes and CI/CD solutions.

The GitOps Working Group, with its defined set of GitOps principles, underpins this approach, emphasizing the importance of declarative configurations, versioned and immutable states, automatic pulling, and continuous reconciliation.

As we explore this further, these fundamentals will be illustrated with practical examples, showcasing the real-world impact and advantages of adopting a GitOps mindset in platform engineering.

Why GitOps?

Adopting GitOps for platform engineering brings a multitude of advantages, reshaping how teams interact with infrastructure and applications. This methodology leverages the strengths of Git, automation, and declarative configurations, providing a robust framework for managing complex systems. Here are some of the key benefits:

- **Declarative and immutable nature**: GitOps focuses on declaring the desired state of systems rather than detailing imperative scripts. It adopts an immutable infrastructure approach where changes are made by replacing resources rather than modifying existing ones, enhancing consistency and stability.
- **Automated synchronization**: Central to GitOps is automatically synchronizing the system state with the configurations described in Git to cover the full life cycle of infrastructure and applications, including deployment, updates, scaling, and recovery.

- **Pull request workflow integration**: Operational processes in GitOps are deeply integrated with a pull request workflow, embodying the philosophy of treating everything as code. This integration facilitates code reviews, collaborative discussions, and shared knowledge and responsibility.
- **Continuous feedback and monitoring**: GitOps emphasizes continuous monitoring to detect and remedy any divergence from the desired state, promoting ongoing system maintenance and improvement.
- **Enhanced efficiency and productivity**: By automating platform engineering aspects, GitOps reduces manual efforts and human errors, allowing teams to focus more on innovation.
- **Improved consistency and reliability**: Using Git as the single source of truth with declarative configurations ensures consistent and reliable infrastructure across different environments.
- **Faster recovery and rollback**: GitOps enables quick rollbacks to previous states for enhanced system resilience and faster recovery from failures.
- **Collaboration and transparency**: The methodology encourages collaborative platform engineering, with peer reviews and discussions improving knowledge sharing and transparency.
- **Streamlined audits and compliance**: All changes are tracked in Git, simplifying auditing and facilitating compliance with regulatory requirements and internal policies.
- **Scalability and flexibility**: GitOps scales efficiently with organizational needs and adapts to various types of infrastructure, including cloud, on-premises, and hybrid environments.
- **Enhanced security**: The use of pull requests and code reviews adds an extra layer of security, ensuring thorough scrutiny of changes to infrastructure and applications.
- **Simplified onboarding and knowledge sharing**: The Git repository provides a clear overview of the system's current state, aiding in onboarding new team members and fostering an environment of accessible knowledge sharing.
- **Continuous improvement and adaptation**: The inherent feedback loops and monitoring in GitOps enable teams to iteratively improve their systems and adapt to changing requirements.

Adopting GitOps presents various challenges for a company that can impact its transition to this methodology. Firstly, there's the learning curve and the need for training in specific tools such as Kubernetes, Helm, or Argo CD, which can temporarily reduce productivity. Implementing GitOps also requires a cultural shift toward rigorous code management and CI/CD practices. The initial setup and integration process can be complex and time-consuming, necessitating significant investments in time and resources.

Furthermore, organizing the Git repository structure efficiently is crucial as it directly affects the manageability and scalability of deployments. Mismanagement here can lead to operational inefficiencies and increased errors. Another critical aspect is secrets management; GitOps workflows require sensitive data to be handled carefully to prevent security breaches. Proper strategies must be implemented to securely store, access, and manage secrets within the GitOps processes to ensure that automated deployments remain secure and efficient.

In this section, we highlighted the significant advantages and challenges of adopting GitOps for platform engineering. This methodology capitalizes on the strengths of Git, automation, and declarative configurations to offer a robust framework for managing complex systems. The next section will introduce the integration between GitOps, IaC, and Kubernetes.

The integration between GitOps, IaC, and Kubernetes

In this section, we'll delve into the intricate integration of GitOps, IaC, and Kubernetes, along with key tools such as Argo CD and Flux CD. We'll explore how this cohesive blend enhances cloud-native operations, streamlining deployment processes and bolstering security, thereby revolutionizing the way infrastructure is managed and applications are deployed in modern software development environments.

GitOps and IaC

IaC involves managing and provisioning infrastructure via code rather than manual procedures. This approach involves creating configuration files that detail your infrastructure requirements, thus simplifying the process of modifying and disseminating these configurations. IaC guarantees consistent provisioning of the same environment each time, and at the same time, IaC transforms configuration specifications into coded documentation, improving configuration management.

This approach helps avoid undocumented and spontaneous configuration alterations.

GitOps extends the familiar Git version control system to infrastructure and deployment management and aligns closely with the principles of IaC. This integration allows developers to manage infrastructure configuration and code deployment as they would their software development process, bringing numerous advantages:

- **Accelerated production time**: Quicker infrastructure provisioning with reduced human error leads to more predictable outcomes
- **Template-based reusability**: Use an IaC configuration file as a template, providing a baseline for further related configurations
- **Uniform deployment**: Achieve consistent infrastructure deployment, avoiding minor discrepancies that could lead to significant future issues
- **Traceability and auditability**: Integrating IaC with a code repository for source tracking offers a detailed history of changes, including versioning and audit trails that identify who made changes and when
- **Enhanced organizational knowledge**: Versioning documents the rationale behind changes, creating a valuable resource for both new and existing employees
- **Improved disaster recovery**: IaC ensures the ability to rapidly reconstruct environments in the event of a failure, minimizing downtime and business disruption

GitOps and Kubernetes

GitOps and Kubernetes (`https://kubernetes.io/docs/concepts/overview/`) represent a powerful combination in the realm of modern software development and infrastructure management. Kubernetes, an open source platform for automating the processes of deployment, scaling, and operations for application containers across clusters of hosts, provides a robust foundation for managing containerized applications at scale. GitOps, on the other hand, is an operational framework that applies the principles of Git – version control, collaboration, and compliance – to infrastructure automation. When combined, GitOps and Kubernetes offer a highly efficient, scalable, and reliable system for managing complex applications and infrastructure.

Kubernetes has become essential for orchestrating containerized applications, offering capabilities such as automated rollouts and rollbacks, service discovery and load balancing (with its ability to assign IP addresses to Pods and services for load balancing), storage orchestration, secret and configuration management without exposing secrets or requiring Docker image rebuilds, and self-healing mechanisms such as restarting failed containers and rescheduling Pods when node issues occur.

Additionally, Kubernetes facilitates horizontal scaling based on CPU usage and operates declaratively. These features continue to evolve with each Kubernetes release.

The fusion of GitOps with Kubernetes significantly quickens and improves the efficiency of deployment processes. This combination promotes more frequent deployments by enabling CD automation and integrating essential feedback and control systems. By using Git's declarative definitions, developers can employ well-known workflows, which simplifies the ability to create new development or test environments and deploy new features to a cluster. Consequently, this leads to an increased daily rate of changes, accelerating the delivery of new features and functionalities to end users.

GitOps also empowers teams to become more self-sufficient. Traditionally, development teams relied on operations teams for deploying changes in production. GitOps shifts this dynamic, allowing development teams to be less dependent on platform and operations teams for deploying and managing code and configuration changes in production. Simply committing changes to the Git repository and merging pull requests can trigger the rest of the process through GitOps tools. This self-service nature enhances efficiency and strategic capability.

GitOps enhances the management of deployments and infrastructure by utilizing Git as the source of truth for **declarative configurations**. In a declarative system, the focus is on defining the desired outcome rather than the steps to achieve it. This approach contrasts with the imperative approach, which involves specifying a sequence of steps and often adds extra workload for operators. Within GitOps, every component of the environment can be coded and declared in the Git repository. Once changes are made and committed to Git, the system automatically processes these declarations to create or update the necessary objects.

This **seamless integration** with the declarative nature of Kubernetes is where the power of GitOps truly shines. Kubernetes operates on a **reconciliation loop** that continuously adjusts the actual state of the cluster to match the desired state defined by its configurations. When discrepancies between the current and desired states are detected, Kubernetes takes automatic action to reconcile these differences. This could involve creating, updating, or deleting resources to ensure the cluster's configuration matches what has been defined in the Git repository.

Together, the combination of GitOps and the Kubernetes reconciliation loop provides a robust framework for automated system management. This ensures that all system components are deployed consistently according to the precise configurations defined in Git, adhering to a fully declarative approach that streamlines operations, enhances reliability, and maintains consistency across development, staging, and production environments.

Observability is another crucial aspect of GitOps in a Kubernetes environment. It involves continuously monitoring the actual state present in the cluster and comparing it to the desired state. GitOps aids in measuring and monitoring what's running on the cluster, alerting teams to any discrepancies. Modern GitOps tools come equipped with monitoring and notification capabilities to support this function.

From a **compliance and auditing** perspective, GitOps offers significant advantages. With Git as the sole source of truth, auditing becomes straightforward: auditors can assess the desired state by examining the source code repository and verify the current state by reviewing the underlying infrastructure provider and the Kubernetes cluster state. This simplifies compliance and auditing processes across various domains.

For organizations that use *multiple Kubernetes clusters* for different business and team needs, managing these clusters and maintaining consistent configuration and security policies is a significant challenge. GitOps addresses this by allowing GitOps agents running across these clusters to manage everything on behalf of the Ops team, making it an ideal solution for multi-cluster configuration management.

Disaster recovery (**DR**) is essential for organizations to recover quickly from incidents that disrupt business operations, with the primary aim being to rapidly restore critical systems and IT infrastructure. From the GitOps perspective, DR primarily handles configuration management, not data recovery. This distinction is crucial because, while configurations are saved in Git, actual data is not. Tools such as Git are useful for version control of configurations but do not aid in data recovery. For example, if a database is deleted, Git cannot assist in its recovery; therefore, separate data recovery strategies are necessary and specifically designed for restoring lost data.

In the Kubernetes ecosystem, the adoption of GitOps for automating infrastructure setup is on the rise. This includes the creation of deployments, services, and various Kubernetes entities. DevOps and **site reliability engineering** (**SRE**) teams are increasingly turning to GitOps for managing all infrastructure configuration files as code. This approach leverages the inherent IaC characteristic of GitOps.

Furthermore, GitOps enhances *security* protocols. It uses sophisticated cryptographic methods in Git for monitoring and managing changes, and it can also authenticate changes to confirm authorship and source. This practice ensures a securely defined intended state of the cluster. In the event of a security breach, the immutable and auditable nature of Git's source of truth is instrumental in reconstructing a system separate from the compromised one, thereby minimizing downtime and boosting the efficiency of incident responses.

The practice of separating responsibilities in software packaging and release to production further adheres to the principle of least privilege, which diminishes the likelihood and severity of security breaches and offers a smaller attack surface.

GitOps facilitates this by ensuring that the entire infrastructure environment and components are defined declaratively in the Git repository. In the event of a disaster, recovery becomes a matter of reapplying the configuration files from Git to quickly restore the ecosystem.

Kubernetes and Argo CD

Argo CD, a GitOps CD for Kubernetes (`argo-cd.readthedocs.io`), is a declarative tool for continuous delivery specifically designed for Kubernetes. It functions as a standalone solution or can be integrated into your existing CI/CD pipeline to efficiently deploy necessary resources across your Kubernetes clusters.

Integrating GitOps and Kubernetes using Argo CD provides a streamlined approach for managing cloud-native applications and jobs on Kubernetes. Argo CD, part of the Argo project family, offers a suite of tools, including Workflows, Events, and Rollouts, that enhance Kubernetes-native application delivery. This becomes particularly relevant for organizations transitioning to containerized environments and Kubernetes deployments, where Argo CD can act as a comprehensive solution.

In large-scale deployments typical of big enterprises with numerous developers and a multitude of microservices, the GitOps methodology shines. However, it often necessitates supplementary tools and services for effective implementation. It's not practical for each team in such an organization to operate independently and manage its Kubernetes cluster. A more efficient strategy is a centralized platform for operating and maintaining multi-tenant clusters. This setup gives teams the autonomy to manage their workloads within these clusters, while still providing centralized oversight and maintenance.

Argo CD is instrumental in facilitating this centralized approach to cluster management, promoting the adoption of GitOps. The following are some key factors to consider if you want to create an effective GitOps strategy in Kubernetes with Argo CD:

- **User onboarding**: Simplifying the onboarding process is crucial. Centralizing the installation, setup, and maintenance of deployment operators can ease the process of transitioning multiple microservices to Kubernetes, rather than placing this burden on individual teams. Incorporating **single sign-on** (**SSO**) is particularly beneficial in systems with a large user base, streamlining the introduction of new users.

- **Multi-tenancy and management**: In environments with multiple tenants, users need secure and flexible access to resources. Kubernetes' **role-based access control** (**RBAC**) system is a powerful feature but might not suffice for managing numerous clusters. Argo CD enhances this with additional access control features, integrating seamlessly with SSO providers and facilitating access to multiple clusters.
- **Observability**: A key feature of an effective CD tool is enabling developers to monitor and track the state of their applications. Argo CD's user-friendly interface provides detailed insights into an application's configuration, such as sync status with Git, specific out-of-sync elements, and any operational issues.

In the following section, we will explore Flux CD, a tool that, when integrated with Kubernetes, enables scalability and ensures consistent environments across various types of deployments.

Kubernetes and Flux CD

Flux (`fluxcd.io`) is an open and extensible CD solution for Kubernetes and represents a critical component in the modern DevOps ecosystem.

At the time of writing, Alexis Richardson, CEO of Weaveworks, has announced on LinkedIn that the company is facing economic challenges and will be closing its doors and shutting down commercial operations. Weaveworks will be collaborating with a financial trustee, to be announced soon, to manage the closure process. This decision comes despite the company generating significant revenue and expanding its customer base, highlighting the financial volatility and strategic challenges faced in maintaining its operations. The original announcement can be found here: `https://www.linkedin.com/posts/richardsonalexis_hi-everyone-i-am-very-sad-to-announce-activity-7160295096825860096-ZS67/`.

As a GitOps tool, Flux ensures that the state of manifests in a Git repository is consistently synchronized with what is running in a Kubernetes cluster. GitOps, far from being just another tool, offers a method for developers to manage operational workflows in Kubernetes using Git. This approach emphasizes using a version-controlled system such as Git to deploy applications in Kubernetes, allowing developers to directly push code into production. Changes can easily be tracked and reverted if necessary, bolstering the reliability of deployments.

The following are the main features of Flux CD:

- Automated synchronization between a version control repository and a cluster
- Instant reflection of repository changes in the cluster
- Direct code deployment into production from repositories
- Version-controlled configuration, ensuring all configurations are up to date
- Disaster recovery capabilities, enabling new clusters to be set up with identical configurations

In the context of Kubernetes, a robust container orchestration platform, Flux CD plays an instrumental role. Kubernetes excels in automating and simplifying application deployment and management, particularly in scaling and maintaining consistent environments across diverse deployments. Flux CD, adhering to GitOps principles, introduces a layer of automation and security by continuously synchronizing application and infrastructure states with configurations stored in a Git repository. This ensures that the actual state aligns with the desired state in a secure and controlled manner. The synergy between Kubernetes and Flux CD offers several key benefits:

- **Automated deployments**: Flux CD's automated updates and configurations are committed to the Git repository
- **Scalability and reliability**: Kubernetes' prowess in handling containerized applications is complemented by Flux CD's consistency in configurations
- **Enhanced security**: Flux CD's use of Git as the single source of truth adds an extra security layer
- **Operational efficiency**: The integration streamlines operations, reducing manual intervention and increasing efficiency

In conclusion, the combination of Kubernetes and Flux CD provides a comprehensive solution for modern software deployment and management. Their integration enhances automation, scalability, security, and efficiency, marking them as indispensable tools in cloud-native technologies and DevOps practices.

Summary

This chapter provided an in-depth introduction to GitOps, exploring its transformative impact on software development and operations. We began by contrasting GitOps with traditional CI/CD and DevOps methodologies, highlighting its unique approach and advantages.

Then, we covered the history and evolution of deployment strategies that led to GitOps. Key principles such as Git centralization, the declarative approach, automated synchronization, and continuous feedback were discussed, emphasizing GitOps' role in enhancing efficiency, consistency, and collaboration in platform engineering.

Additionally, we delved into GitOps' integration with tools such as Kubernetes, showcasing its scalability and security advantages in modern cloud environments. We concluded by underscoring the significance of adopting GitOps for its robust framework and adaptability to evolving technological landscapes.

In *Chapter 2*, we'll dive into the practicalities of GitOps in cloud-native environments, examining its seamless integration with Kubernetes, the architecture of lightweight Kubernetes distributions, and the design of cloud-native CI/CD pipelines.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- *Argo CD – Declarative GitOps CD for Kubernetes*: `https://argo-cd.readthedocs.io/en/stable/`
- *Flux CD*: `https://fluxcd.io/`

2

Navigating Cloud-native Operations with GitOps

In *Chapter 1*, we delved into the foundational concepts of GitOps, contrasting its approach with traditional CI/CD and DevOps methodologies. We explored its historical evolution, key principles such as Git centralization and automated synchronization, and its integration with Kubernetes. This chapter will emphasize GitOps' role in enhancing scalability and security in modern cloud environments. By the end of this chapter, you will have a comprehensive understanding of GitOps' transformative impact on software deployment and operations, setting the stage for its application in cloud-native operations.

In this chapter, we'll focus on the following key areas:

- GitOps and cloud-native tech
- An introduction to Kubernetes
- Exploring K3s as a lightweight Kubernetes distribution
- Containers
- Sample workflow – effortless CD with Docker and K3s

Technical requirements

To engage with the examples in this chapter, you'll need a Kubernetes cluster. While we'll guide you through how to install K3s in a way that's suitable for these examples, any Kubernetes setup will suffice.

K3s is optimized for Linux systems, so ensure you have access to a Linux environment. If you're using a non-Linux system, consider alternatives such as **Windows Subsystem for Linux** (**WSL**) or **Virtual Box** (see [*1*] and [*2*] in the *Further reading* section at the end of this chapter).

The code for this chapter is available in the `Chapter02` folder in this book's GitHub repository: `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`.

An overview of the integration of GitOps and cloud-native technology

In *Chapter 1*, we explored GitOps, a fusion of DevOps and Git, emphasizing its ability to enhance operational efficiency and system stability by applying software development techniques to infrastructure management. Moving forward, *Chapter 2* expands on this foundation, examining how GitOps integrates with cloud-native technology. This technology signifies a significant shift in application development, characterized by containerization, microservices, and dynamic orchestration, enhancing scalability and resilience.

GitOps complements this by enabling systematic, version-controlled management of complex systems. The synergy between GitOps and cloud-native technologies, particularly Kubernetes, leads to a more dynamic, agile, and reliable approach to system management. This chapter aims to show how GitOps simplifies and elevates the capabilities of cloud-native environments.

Additionally, GitOps, emerging from the confluence of DevOps and version control, leverages Git's power for managing and automating software system deployments and operations. By treating infrastructure as code, GitOps facilitates reviewing, versioning, and deploying changes using Git's familiar pull requests and merges. This approach ensures consistency, traceability, and ease of rolling back, proving especially potent in cloud-native settings.

Cloud-native technology, in contrast, represents a paradigm shift in how applications are constructed and deployed. It involves using containers, microservices, and dynamic orchestration to create robust, scalable, and independently deployable applications. This technology maximizes cloud flexibility, enabling swift scaling and resilience. When integrated with GitOps, cloud-native technology becomes more robust, allowing teams to manage complex systems more effectively with increased confidence.

Before delving into the practical applications of GitOps, it is essential to introduce Kubernetes, the orchestration platform that's central to cloud-native technology. Additionally, we will discuss **K3s**, a lightweight variant of Kubernetes. K3s is particularly suited for personal development environments as it allows Kubernetes clusters to be deployed on individual laptops. This setup allows for hands-on experimentation and learning, providing a practical foundation for understanding and applying GitOps techniques in a Kubernetes context. This knowledge will be crucial as we progress to more advanced topics and practical demonstrations of GitOps in action.

An introduction to Kubernetes

In the upcoming sections, we will introduce Kubernetes, including a brief historical overview of the original project and the core concepts of Kubernetes architecture. In the second part, we will delve deeper into K3s and explore how you can use it to run a local Kubernetes cluster on your laptop.

What is Kubernetes?

Kubernetes is a robust and open source platform that was crafted to streamline the automation of deploying, scaling, and managing application containers. It plays a central role in the kingdom of container orchestration, offering a solid framework for the effective management of containerized applications across multiple settings, including physical data centers and both public and private cloud environments.

Originally, the Kubernetes project at Google, codenamed *Project 7* as a nod to *Star Trek's* Seven of Nine, symbolized a more approachable version of Google's Borg system. Owing to licensing constraints, the term Kubernetes, Greek for helmsman, was adopted and reflected in its seven-spoked wheel logo, subtly honoring its *Star Trek*-inspired origins. Following its 2014 announcement, Joe Beda, Brendan Burns, and Craig McLuckie, among other Google engineers, spearheaded its development. Distinct from Borg's C++ coding, Kubernetes utilized **Go**. Its first version, Kubernetes 1.0, was released in 2015. Through collaboration with the Linux Foundation, Kubernetes became a cornerstone of the **Cloud Native Computing Foundation** (**CNCF**), rapidly garnering integration into services offered by major tech entities such as Red Hat, VMware, Mesosphere, Docker, Microsoft Azure, and AWS.

Kubernetes plays a critical role in cloud computing, facilitating both declarative configuration and automation. It supports a range of container tools, including Docker, and its ability to manage complex container architectures across multiple hosts makes it highly valuable.

Kubernetes simplifies the deployment and scaling of applications, and its automated rollouts and rollbacks for containerized applications enhance reliability and efficiency. It allows containers to be orchestrated across multiple hosts, handles how applications are deployed and scaled, and covers their networking and storage needs.

The platform's self-healing feature automatically restarts, replaces, and reschedules containers if they fail. It also scales containers in response to varying loads and updates them without downtime using a variety of deployment patterns.

Kubernetes supports a range of workloads, including **stateless**, **stateful**, and **data-processing** workloads. It's flexible enough to deliver complex applications, offering scalability and reliability while managing workloads effectively.

Overall, Kubernetes has revolutionized the way containerized applications are deployed and managed, making it a key tool in the world of modern software development and operations.

Kubernetes architecture

Kubernetes architecture is built to manage and orchestrate containerized applications. It consists of several components that work together.

In Kubernetes architecture, the cluster is divided into two primary components: the **control plane** and the **worker nodes** (or **data plane**). The control plane is responsible for global decision-making and managing the cluster's state. It includes essential elements such as the **API server**, **etcd**, **scheduler**, **controller manager**, and **cloud controller manager**.

Conversely, node components are responsible for running the actual workloads. Each node contains vital services such as **Kubelet**, a **container runtime**, and **kube-proxy**, which ensure that containers run as expected and handle network communication within and outside the cluster.

This architecture allows for a robust and scalable system where the control plane maintains control and nodes efficiently manage the workload.

The control plane includes the following components:

- **API server (kube-apiserver)**: This central management entity processes REST requests, validates them, and updates the corresponding objects in etcd. It's the main interface of the Kubernetes control plane.
- **etcd**: This is a consistent and highly available **key-value store** that acts as the primary storage for all cluster data. It's crucial for the cluster's state management.
- **Scheduler (kube-scheduler)**: The scheduler is responsible for assigning Pods to nodes based on resource availability, user-defined constraints, taints, and selectors. This ensures each Pod is placed on the optimal node that satisfies not only resource needs but also respects scheduling policies such as taints and affinity/anti-affinity selectors.
- **Controller manager (kube-controller-manager)**: This component runs various controller processes in the background. It observes the state of the cluster, manages the life cycle of workloads, and handles operations on nodes to ensure the desired state of the Kubernetes cluster is maintained.
- **Cloud controller manager**: An architectural component that embeds cloud-specific control logic, allowing cloud vendors to link their platforms with Kubernetes. It abstracts away the cloud-specific code from core Kubernetes logic, enabling each cloud service to develop its plugins independently. Each node component, which hosts the pods, consists of essential components for maintaining and managing the containers and network communication:
    - **Kubelet**: This agent ensures that containers are running in a **Pod**, as per the specifications defined in the Pod's configuration. It manages the state of each Pod on the node, communicating with the control plane of the master node (or master nodes in the case of highly available Kubernetes clusters).

- **Container runtime**: This is the underlying software that is responsible for running containers. Kubernetes supports several container runtimes, such as Docker, containerd, and CRI-O, enabling it to run containerized applications.
- **kube-proxy**: This component oversees network interactions to and from the Pods. It routes TCP and UDP packets and facilitates connection forwarding, adding a Kubernetes service abstraction that acts as a proxy.

Figure 2.1 illustrates the Kubernetes architecture described here, with components for the control plane and each component node:

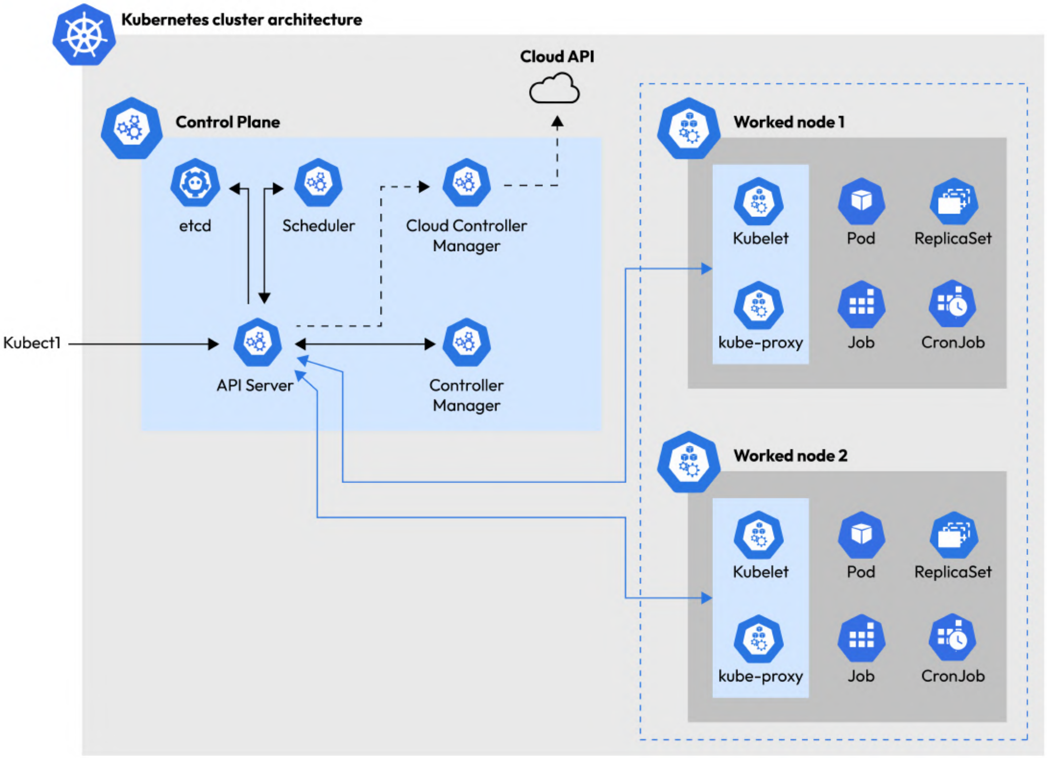


Figure 2.1 – The Kubernetes cluster architecture

For a more in-depth understanding of each component in the Kubernetes architecture, please refer to the official Kubernetes documentation (`https://kubernetes.io/`). This resource provides comprehensive information and detailed explanations of various aspects of the Kubernetes system, including its master and node components.

Now that we have a basic understanding of what Kubernetes is and the main components that run in a Kubernetes cluster, it's time to learn how to set up a local cluster on your laptop using K3s, a lightweight Kubernetes distribution.

Exploring K3s as a lightweight Kubernetes distribution

As mentioned previously, throughout this book, and specifically in this chapter, we will utilize K3s, a **lightweight** Kubernetes distribution (`https://k3s.io/`), to run our examples.

K3s is particularly well-suited for scenarios where the **full-scale** implementation of Kubernetes may be too resource-intensive or complex.

Its lightweight nature makes it ideal for edge computing and IoT scenarios, where resources are often limited, and efficiency is paramount. In these environments, K3s provides the necessary Kubernetes features without the overhead. Additionally, solutions such as vCluster from Loft have leveraged K3s to run Kubernetes within Kubernetes, facilitating multi-tenancy on a host cluster. This approach allows for isolated Kubernetes environments within a single cluster, optimizing resource usage and offering scalability in multi-tenant setups. These use cases highlight K3s's versatility and efficiency in diverse computing environments. More information about K3s can be found in the official documentation: `https://docs.k3s.io/`.

Origin of the K3s name

The name K3s, as explained in the official documentation (`https://docs.k3s.io/`), is derived from the intent to create a Kubernetes installation that's significantly smaller in memory size. The naming convention follows that of Kubernetes, often abbreviated as K8s, which consists of 10 letters. Halving this led to K3s, which was stylized to represent a more compact version of Kubernetes. Unlike Kubernetes, K3s does not have an expanded form, and its pronunciation is not officially defined. This naming reflects the goal of a lighter, more efficient version of Kubernetes.

K3s simplifies the process of deploying a Kubernetes cluster, making it accessible even for small-scale operations or development purposes. By removing non-essential components and using lighter-weight alternatives, K3s significantly reduces the size and complexity of Kubernetes while maintaining its core functionalities.

K3s maintain compatibility with the larger Kubernetes ecosystem, ensuring that tools and applications designed for Kubernetes can generally be used with K3s as well.

One of the key features of K3s is its single binary installation, which includes both the Kubernetes server and agent, simplifying the setup process. This makes it an ideal choice for developers who want to quickly set up a Kubernetes environment for testing or development without the overhead of a full Kubernetes installation.

K3s also offers flexible networking and storage options, catering to a wide range of use cases – from small local clusters to larger, more complex environments. Its versatility and ease of use make it a popular choice for those looking to explore Kubernetes without the need for extensive infrastructure.

Lastly, K3s's lightweight nature and efficiency make it a suitable choice for **continuous integration/continuous deployment** (**CI/CD**) pipelines, allowing for faster build and test cycles in environments where resources are a consideration. In *Chapter 5*, we'll learn how to use K3s to run Kubernetes on Kubernetes.

Local cluster setup

Before diving into our first deployment example, it's essential to set up the environment and understand how Kubernetes, particularly K3s, facilitates our deployments. K3s is primarily designed for Linux environments, so make sure you have a modern Linux system such as Red Hat Enterprise Linux, CentOS, Fedora, Ubuntu/Debian, or even Raspberry Pi. If you're a Windows user, you can still engage with K3s by setting up **WSL** or running a Linux **virtual machine** (**VM**) through **VirtualBox**. These setups will prepare you to harness the power of Kubernetes for your deployments.

Choosing your local Kubernetes environment – K3s, Minikube, and alternatives

In this chapter, we have chosen to use K3s due to its lightweight nature and ease of setup, which makes it particularly suitable for developing and testing Kubernetes environments. However, there are several other alternatives for setting up local Kubernetes clusters that cater to different needs and platforms. For instance, Colima (`https://github.com/abiosoft/colima`) is an excellent choice for macOS users, offering a Docker and Kubernetes environment directly on macOS with minimal configuration. **Minikube** (`https://minikube.sigs.k8s.io`) is another popular option that runs on Windows, macOS, and Linux and is ideal for those looking to simulate a Kubernetes cluster in a single node where they can experiment and test Kubernetes applications.

While K3s is our choice for this chapter, you are encouraged to use the local cluster setup that best fits your platform or preferences. In subsequent chapters, we will primarily focus on using K3s or Minikube. These platforms provide a convenient and consistent environment for learning and deploying applications using Kubernetes, ensuring that the concepts and procedures we'll explore are accessible regardless of the specific local cluster technology used.

Setting up WSL

All details regarding the nature of WSL and the procedures for installing it on Windows are beyond the scope of this book. However, comprehensive guidance on setup steps and in-depth information about WSL can be accessed through the official Microsoft documentation (see [*1*] in the *Further reading* section at the end of this chapter):

Figure 2.2 – A conceptual illustration representing WSL on a Windows operating system

Remember, staying updated with the latest WSL versions and features through the official site will enhance your experience and ensure compatibility with the most recent Windows updates.

Setting up VirtualBox

VirtualBox is an open source **virtualization software** developed by Oracle. It allows users to run multiple operating systems on a single physical computer, creating VMs that can operate independently. This makes it an invaluable tool for software testing, development, and educational purposes as it provides a flexible and isolated environment for running and experimenting with different operating systems without risk to the host system:

Welcome to VirtualBox.org!

VirtualBox is a powerful x86 and AMD64/Intel64 virtualization product for enterprise as well as home use. Not only is VirtualBox an extremely feature rich, high performance product for enterprise customers, it is also the only professional solution that is freely available as Open Source Software under the terms of the GNU General Public License (GPL) version 3. See "About VirtualBox" for an introduction.

Presently, VirtualBox runs on Windows, Linux, macOS, and Solaris hosts and supports a large number of guest operating systems including but not limited to Windows (NT 4.0, 2000, XP, Server 2003, Vista, 7, 8, Windows 10 and Windows 11), DOS/Windows 3.x, Linux (2.4, 2.6, 3.x, 4.x, 5.x and 6.x), Solaris and OpenSolaris, OS/2, OpenBSD, NetBSD and FreeBSD.

VirtualBox is being actively developed with frequent releases and has an ever growing list of features, supported guest operating systems and platforms it runs on. VirtualBox is a community effort backed by a dedicated company: everyone is encouraged to contribute while Oracle ensures the product always meets professional quality criteria.

News Flash

- New **October 17th, 2023 VirtualBox 7.0.12 released!** Oracle today released a 7.0 maintenance release which improves stability and fixes regressions. See the Changelog for details.
- New **October 17th, 2023 VirtualBox 6.1.48 released!** Oracle today released a 6.1 maintenance release which improves stability and fixes regressions. See the Changelog for details.
- New **July 18th, 2023**

Figure 2.3 – The VirtualBox home page at https://www.virtualbox.org/.

The detailed steps for installing VirtualBox are beyond the scope of this book. However, comprehensive installation instructions and additional information can be found in the official documentation [*2*].

For the most current information and tips, visiting the official VirtualBox documentation is highly recommended.

Unless otherwise specified, for this chapter and the subsequent ones, we will assume the use of an **Ubuntu-22.04 LTS** installation within WSL. This setup provides a consistent and controlled environment for our examples and demonstrations.

By focusing on a specific version of Ubuntu, we ensure that the instructions and scenarios presented are as relevant and applicable as possible, aligning closely with the most common and stable Linux distribution used in WSL.

K3s setup and installation verification

In this section, we'll cover the basic steps that are necessary to establish a Kubernetes cluster using K3s in its default configuration, assuming that WSL is already installed and functioning correctly.

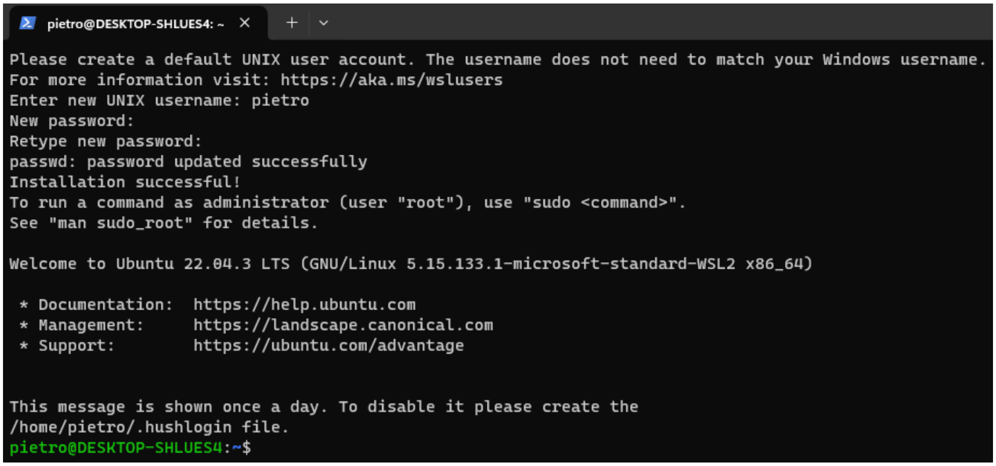
Downloading and installing K3s

Follow these steps to download and install K3s:

1. Let's start by opening a new Terminal window and typing the following command:

   ```
   $ wsl --install -d Ubuntu-22.04
   ```

   At a certain stage, the setup will require you to specify a UNIX username (for example, `pietro`), which does not need to match your Windows username. The next step involves setting the password that will be used to run a command as an administrator (`sudo`). If the operations are completed correctly, the Terminal window should look like this:



```
Please create a default UNIX user account. The username does not need to match your Windows username.
For more information visit: https://aka.ms/wslusers
Enter new UNIX username: pietro
New password:
Retype new password:
passwd: password updated successfully
Installation successful!
To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

Welcome to Ubuntu 22.04.3 LTS (GNU/Linux 5.15.133.1-microsoft-standard-WSL2 x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage


This message is shown once a day. To disable it please create the
/home/pietro/.hushlogin file.
pietro@DESKTOP-SHLUES4:~$
```

Figure 2.4 – Successfully installing an instance of Ubuntu 22.04.3 LTS on WSL

2. Before proceeding with the K3s setup, it is always better to execute commands to update the operating system with the latest patches:

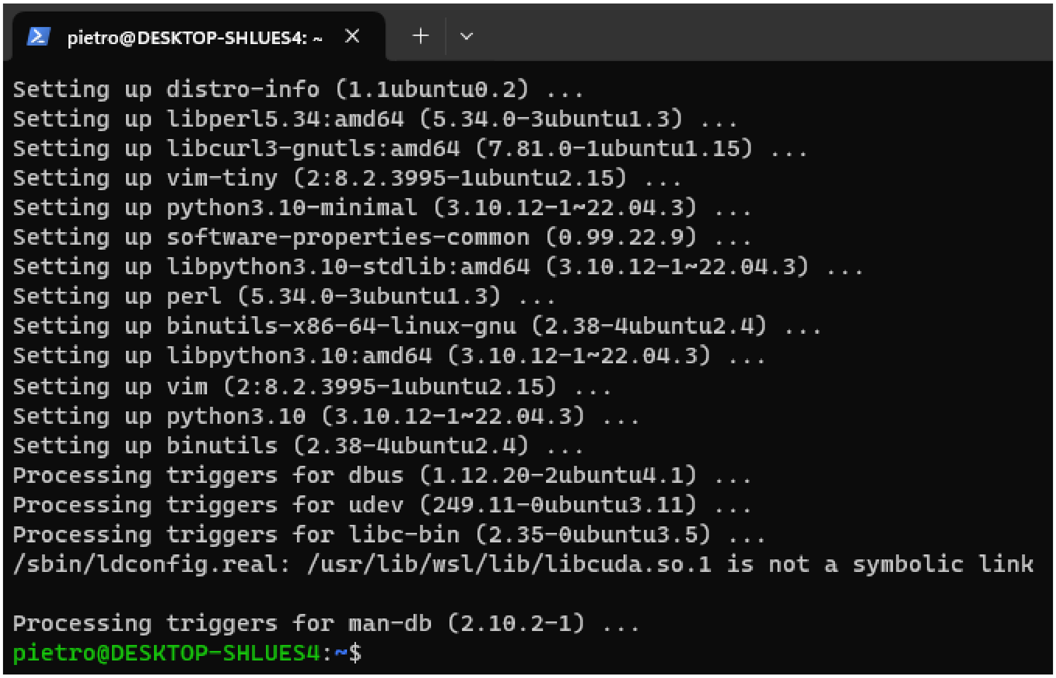
   ```
   $ sudo apt update
   $ sudo apt upgrade
   ```

   This ensures that you are working with the most recent and secure versions of the software.

The apt update and apt upgrade commands

The `apt update` and `apt upgrade` commands are fundamental in maintaining the software on systems using the APT package manager, commonly found in Debian-based Linux distributions such as Ubuntu. The `apt update` command refreshes the local package index by retrieving the latest information about available packages and their versions from configured sources. This doesn't install or upgrade any packages and instead updates the package lists to inform the system of new, removed, or updated software. Once the package index has been updated, the `apt upgrade` command is used to upgrade installed packages to their latest versions. It downloads and installs the updates for any packages where newer versions are available, ensuring the system is up-to-date and potentially more secure.

If required, enter the password you set up while installing Ubuntu. After executing these commands, the Terminal should look as follows:



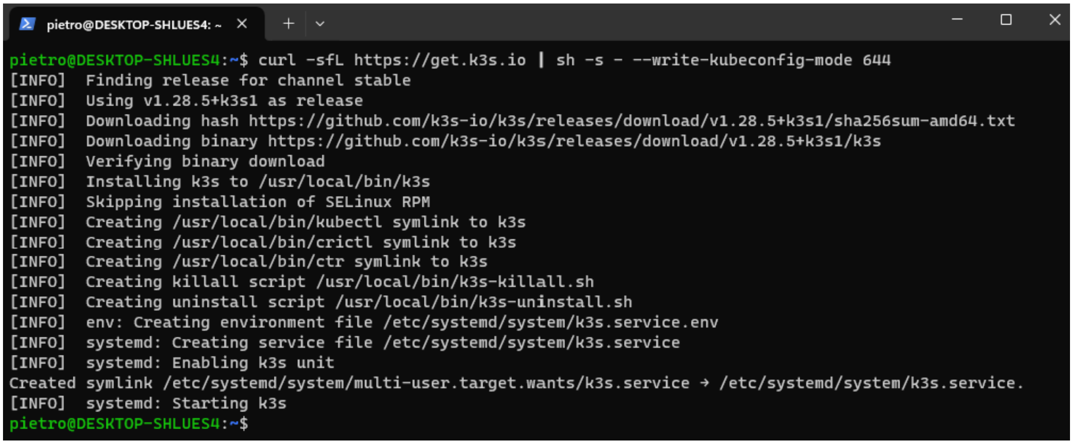
```
Setting up distro-info (1.1ubuntu0.2) ...
Setting up libperl5.34:amd64 (5.34.0-3ubuntu1.3) ...
Setting up libcurl3-gnutls:amd64 (7.81.0-1ubuntu1.15) ...
Setting up vim-tiny (2:8.2.3995-1ubuntu2.15) ...
Setting up python3.10-minimal (3.10.12-1~22.04.3) ...
Setting up software-properties-common (0.99.22.9) ...
Setting up libpython3.10-stdlib:amd64 (3.10.12-1~22.04.3) ...
Setting up perl (5.34.0-3ubuntu1.3) ...
Setting up binutils-x86-64-linux-gnu (2.38-4ubuntu2.4) ...
Setting up libpython3.10:amd64 (3.10.12-1~22.04.3) ...
Setting up vim (2:8.2.3995-1ubuntu2.15) ...
Setting up python3.10 (3.10.12-1~22.04.3) ...
Setting up binutils (2.38-4ubuntu2.4) ...
Processing triggers for dbus (1.12.20-2ubuntu4.1) ...
Processing triggers for udev (249.11-0ubuntu3.11) ...
Processing triggers for libc-bin (2.35-0ubuntu3.5) ...
/sbin/ldconfig.real: /usr/lib/wsl/lib/libcuda.so.1 is not a symbolic link

Processing triggers for man-db (2.10.2-1) ...
pietro@DESKTOP-SHLUES4:~$
```

Figure 2.5 – Terminal window after executing the apt update and apt upgrade commands

3. The next step is to install K3s using the following command:

```
$ curl -sfL https://get.k3s.io | sh -s - --write-kubeconfig-mode
644
```

The preceding command will download and set up the necessary tools, followed by launching the K3s server. The successful setup of a K3s instance is depicted in *Figure 2.6*:



```
pietro@DESKTOP-SHLUES4:~$ curl -sfL https://get.k3s.io | sh -s - --write-kubeconfig-mode 644
[INFO]  Finding release for channel stable
[INFO]  Using v1.28.5+k3s1 as release
[INFO]  Downloading hash https://github.com/k3s-io/k3s/releases/download/v1.28.5+k3s1/sha256sum-amd64.txt
[INFO]  Downloading binary https://github.com/k3s-io/k3s/releases/download/v1.28.5+k3s1/k3s
[INFO]  Verifying binary download
[INFO]  Installing k3s to /usr/local/bin/k3s
[INFO]  Skipping installation of SELinux RPM
[INFO]  Creating /usr/local/bin/kubectl symlink to k3s
[INFO]  Creating /usr/local/bin/crictl symlink to k3s
[INFO]  Creating /usr/local/bin/ctr symlink to k3s
[INFO]  Creating killall script /usr/local/bin/k3s-killall.sh
[INFO]  Creating uninstall script /usr/local/bin/k3s-uninstall.sh
[INFO]  env: Creating environment file /etc/systemd/system/k3s.service.env
[INFO]  systemd: Creating service file /etc/systemd/system/k3s.service
[INFO]  systemd: Enabling k3s unit
Created symlink /etc/systemd/system/multi-user.target.wants/k3s.service → /etc/systemd/system/k3s.service.
[INFO]  systemd: Starting k3s
pietro@DESKTOP-SHLUES4:~$
```

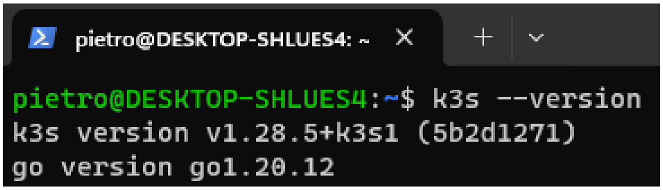
Figure 2.6 – Successfully setting up K3s

Verifying the K3s installation

It is necessary to use two commands to check the correctness of the K3s setup and configuration. The first one is as follows:

```
$ k3s --version
```

The preceding command is used to check which version of K3s we are running. If the K3s server is running correctly, we should be able to see a message similar to the following:



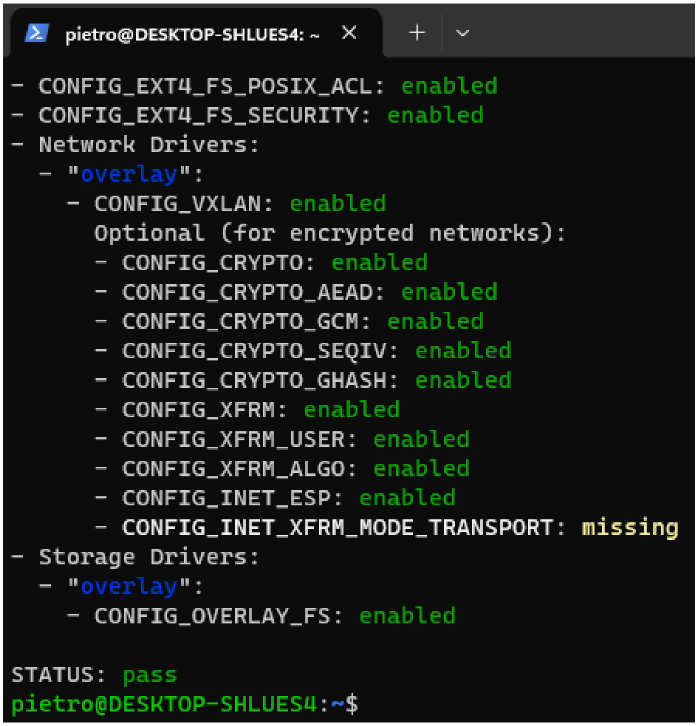
```
pietro@DESKTOP-SHLUES4:~$ k3s --version
k3s version v1.28.5+k3s1 (5b2d1271)
go version go1.20.12
```

Figure 2.7 – The result of executing the k3s –version command

The second command that checks the correctness of the K3s setup is as follows:

```
$ k3s check-config
```

The `k3s check-config` command performs a diagnostic check on the system's configuration to ensure it is suitable for running a K3s cluster. It verifies critical aspects such as kernel compatibility, required system dependencies, and the presence of necessary features and modules. This command helps in identifying potential issues or missing configurations before proceeding with the K3s installation, ensuring a smoother setup process:



```
- CONFIG_EXT4_FS_POSIX_ACL: enabled
- CONFIG_EXT4_FS_SECURITY: enabled
- Network Drivers:
  - "overlay":
    - CONFIG_VXLAN: enabled
      Optional (for encrypted networks):
      - CONFIG_CRYPTO: enabled
      - CONFIG_CRYPTO_AEAD: enabled
      - CONFIG_CRYPTO_GCM: enabled
      - CONFIG_CRYPTO_SEQIV: enabled
      - CONFIG_CRYPTO_GHASH: enabled
      - CONFIG_XFRM: enabled
      - CONFIG_XFRM_USER: enabled
      - CONFIG_XFRM_ALGO: enabled
      - CONFIG_INET_ESP: enabled
      - CONFIG_INET_XFRM_MODE_TRANSPORT: missing
- Storage Drivers:
  - "overlay":
    - CONFIG_OVERLAY_FS: enabled

STATUS: pass
pietro@DESKTOP-SHLUES4:~$
```

Figure 2.8 – Successfully configuring the k3s check-config command

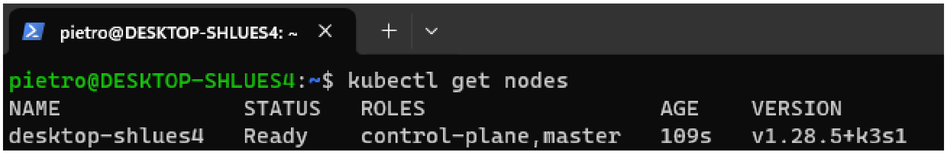
Congratulations! You have confirmed that the K3s server has been installed in your local development environment. Now, it's time to verify the Kubernetes cluster and deploy a test application.

Checking the Kubernetes cluster

To confirm that our K3s node is up and running, let's type the following command:

```
$ kubectl get nodes
```

If the Kubernetes cluster is working correctly, the preceding command will produce the following output:



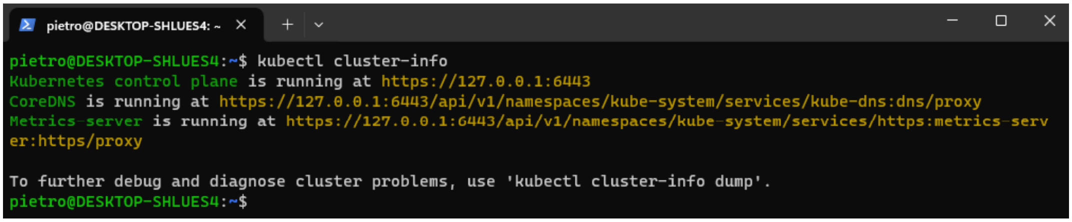
```
pietro@DESKTOP-SHLUES4:~$ kubectl get nodes
NAME              STATUS   ROLES                  AGE    VERSION
desktop-shlues4   Ready    control-plane,master   109s   v1.28.5+k3s1
```

Figure 2.9 – Example output after running the kubectl get nodes command

After confirming that the node is up and running correctly, we can run the following command to obtain more information about the running cluster:

```
$ kubectl cluster-info
```

The `kubectl cluster-info` command is a useful tool in Kubernetes for obtaining essential information about a cluster. When executed, it displays key details such as the Kubernetes master and services endpoint addresses. This command helps users quickly understand the state and connectivity of their cluster's control plane and core services such as KubeDNS and, when applicable, the dashboard. It is particularly valuable for troubleshooting and ensuring that the Kubernetes cluster is configured correctly and operational. Easy to use, `kubectl cluster-info` is often one of the first commands you should run to verify the health and status of a Kubernetes environment, as shown here:



```
pietro@DESKTOP-SHLUES4:~$ kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:6443
CoreDNS is running at https://127.0.0.1:6443/api/v1/namespaces/kube-system/services/kube-dns:dns/proxy
Metrics-server is running at https://127.0.0.1:6443/api/v1/namespaces/kube-system/services/https:metrics-serv
er:https/proxy

To further debug and diagnose cluster problems, use 'kubectl cluster-info dump'.
pietro@DESKTOP-SHLUES4:~$
```

Figure 2.10 – Information provided after executing the kubectl cluster-info command

kubectl

kubectl is a command-line tool that serves as the primary interface for interacting with Kubernetes. It allows users to deploy applications, inspect and manage cluster resources, and view logs. Essentially, kubectl provides the necessary commands to control Kubernetes clusters effectively. Users can create, delete, and update parts of their Kubernetes applications and infrastructure using this versatile tool. It is designed to be user-friendly, offering comprehensive help commands and output formatting options, making it easier to understand and manage complex Kubernetes environments. kubectl is an indispensable tool for developers and system administrators working with Kubernetes, offering a robust and flexible way to handle containerized applications and services in various environments.

Kubernetes manifest

A Kubernetes manifest is a configuration file, typically written in YAML or JSON, that defines resources that should be deployed to a Kubernetes cluster. It specifies the desired state of objects, such as Pods, Services, or Deployments, that Kubernetes needs to create and manage. This manifest enables users to declare their applications' requirements, networking, and storage configurations, among other settings, in a structured and versionable format.

As an example, a basic Kubernetes manifest for deploying a simple application might look like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: hw-gitops-folks
spec:
  containers:
  - name: hw-gitops-folks-container
    image: k8s.gcr.io/echoserver:1.4
    ports:
    - containerPort: 8080
```

In this manifest, a Pod named `hw-gitops-folks` is defined. It contains one container named `hw-gitops-container`, which uses the `echoserver:1.4` image from Kubernetes' container registry. The container exposes port `8080`. This manifest, when applied to a Kubernetes cluster, will create a Pod running a simple echo server that can be used for basic testing.

Our first deployment with K3s

Now that we have successfully set up, configured, and verified our K3s cluster, we are poised to embark on an exciting phase: preparing for our first deployment. This step marks a significant milestone in our journey as we transition from the foundational aspects of K3s to actively utilizing the cluster for practical applications. The upcoming deployment process will not only reinforce our understanding of Kubernetes concepts but also demonstrate the real-world utility of our K3s environment. It's a moment where theory meets practice, allowing us to see firsthand how our configured cluster can host and manage applications. Let's proceed with an eagerness to explore the capabilities of our Kubernetes setup while keeping the practices we've learned and the robust infrastructure we've established in mind:

1. Let's begin by typing the following command, which should list all the running Pods:

   ```
   $ kubectl get pods
   ```

   The result of its execution should look something like this:

   ```
   No resources found in default namespace
   ```

2. The preceding output is normal since no deployments have been performed so far. Let's try another command:

   ```
   $ kubectl get pods --all-namespaces
   ```

   This time, the result should be different as we are requesting to include Pods running in all namespaces, both user-defined and system-defined, such as those within the predefined `kube-system` namespace. These Pods are essential for the operation of the Kubernetes system. The specific Pods and their statuses are detailed in *Figure 2.11*, offering a comprehensive view of the active system components within this crucial namespace:

```
pietro@DESKTOP-SHLUES4:~$ kubectl get pods --all-namespaces
NAMESPACE     NAME                                     READY   STATUS      RESTARTS   AGE
kube-system   local-path-provisioner-84db5d44d9-7zkvm   1/1     Running     0          19h
kube-system   coredns-6799fbcd5-gth8p                   1/1     Running     0          19h
kube-system   helm-install-traefik-crd-s5swf            0/1     Completed   0          19h
kube-system   helm-install-traefik-6rj7p                0/1     Completed   1          19h
kube-system   svclb-traefik-8f8b8625-zczsm              2/2     Running     0          19h
kube-system   traefik-f4564c4f4-4jfjf                   1/1     Running     0          19h
kube-system   metrics-server-67c658944b-gdxkm           1/1     Running     0          19h
pietro@DESKTOP-SHLUES4:~$
```

Figure 2.11 – Example of running Pods in the kube-system namespace

What is a namespace in Kubernetes?

In Kubernetes, a namespace is a fundamental concept that's used to organize clusters into logically isolated sub-groups. It provides a way to divide cluster resources between multiple users and applications. Essentially, namespaces are like virtual clusters within a physical Kubernetes cluster. They allow for resource management, access control, and quota management, enabling efficient and secure multi-tenancy environments. For instance, different development teams or projects can operate in separate namespaces, without interference. Namespaces also facilitate resource naming, ensuring that resources with the same name can coexist in different namespaces. They play a crucial role in Kubernetes for scalability and maintaining order, especially in larger systems with numerous applications and teams.

Creating different namespaces in Kubernetes is widely regarded as a best practice for several compelling reasons. Namespaces provide a logical partitioning of the cluster, allowing for more organized and efficient resource management. This separation is particularly beneficial in environments with multiple teams or projects as it ensures a clear distinction between resources, reduces naming conflicts, and enhances security by isolating workloads. Additionally, namespaces facilitate fine-grained access control as administrators can assign specific permissions and resource limits to different namespaces, preventing accidental or unauthorized interactions between distinct parts of the cluster. By using namespaces, teams can also streamline deployment processes and monitor resource usage more effectively, leading to a more robust and scalable Kubernetes environment. In essence, namespaces are crucial in maintaining order, security, and efficiency in complex Kubernetes clusters. So, let's get started by creating one:

1. Let's continue by creating a new namespace before continuing with our first deployment:

   ```
   $ kubectl create namespace gitops-kubernetes
   ```

   The response to this command should look something like this:

   ```
   namespace/gitops-kubernetes created
   ```

2. The command to delete a namespace is as follows:

   ```
   $ kubectl delete namespace gitops-kubernetes
   ```

3. For the first deployment, we will create a Kubernetes manifest file that defines a deployment for a simple "hello-world" web page, along with a corresponding service to expose it. This manifest file will create a deployment that runs a container based on a generic `hello-world` image and a service to make the deployment accessible (the complete version of the manifest mentioned here can be found in this book's GitHub repository):

   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: hello-world-deployment
     namespace: gitops-kubernetes
   ...
   spec:
   ...
       spec:
         containers:
         - name: hello-world
           image: nginxdemos/hello
           ports:
           - containerPort: 80
   ---
   apiVersion: v1
   kind: Service
   metadata:
     name: hello-world-service
     namespace: gitops-kubernetes
   spec:
     type: NodePort
      ...
     ports:
       - protocol: TCP
         port: 80
         nodePort: 30007
   ```

To apply the manifest, we need to save it in a `.yaml` (or .`yml`) file, such as `hello-world-deployment.yaml` (its name isn't important).

4. To edit the file, we can use an editor such as `nano` by running the following command:

```
$ nano hello-world-deployment.yaml
```

This manifest file has two parts:

- **Deployment**: It creates a deployment named `hello-world-deployment` that runs a container using the `nginxdemos/hello` image, which serves a simple HTML page. The container is configured to expose port `80`. In the metadata section, we have specified to run the Pod in the namespace we created previously – that is, `namespace: gitops-kubernetes`.
- **Service**: It creates a service named `hello-world-service` of the `NodePort` type to expose the deployment. This service makes the hello-world application accessible on a port on the nodes in the cluster (in this example, `port 30007`). In the metadata section, we have specified to run the service in the namespace we created previously – that is, `namespace`: `gitops-kubernetes`.

NodePort

In this hello-world service example, the `NodePort` service type was chosen to demonstrate a simple way of exposing a service to external traffic in Kubernetes. `NodePort` opens a specific port on all the nodes; any traffic sent to this port is forwarded to the service. While this is useful for development and testing, it may not be ideal in a real-world cloud scenario, especially when running on a VM in the cloud. This is because `NodePort` exposes a port on the host VM/node, potentially posing a security risk by making the service accessible externally. In production environments, more secure and controlled methods of exposing services are typically preferred.

5. To apply this manifest, use the `kubectl apply -f <filename>.yaml` command:

```
$ kubectl apply -f hello-world-deployment.yaml
```

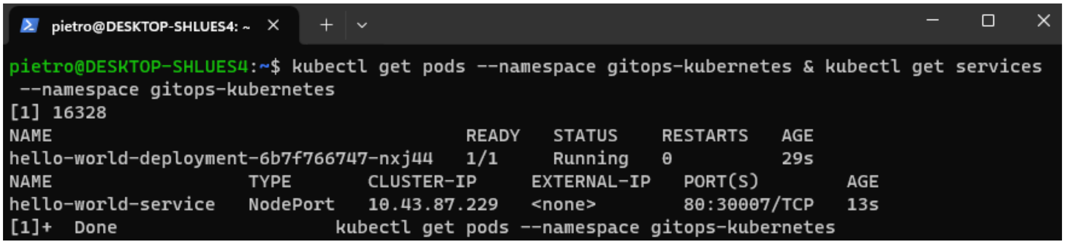
The response to this command should look something like this:

```
deployment.apps/hello-world-deployment created
service/hello-world-service unchanged
```

6. Now, we can list the Pods and services that are running in the `gitpos-kubernetes` namespace using the following command:

```
$ kubectl get pods --namespace gitops-kubernetes & kubectl get
services --namespace gitops-kubernetes
```

The result of this command is shown in *Figure 2.12*:



```
pietro@DESKTOP-SHLUES4:~$ kubectl get pods --namespace gitops-kubernetes & kubectl get services
 --namespace gitops-kubernetes
[1] 16328
NAME                                      READY   STATUS    RESTARTS   AGE
hello-world-deployment-6b7f766747-nxj44   1/1     Running   0          29s
NAME                  TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)        AGE
hello-world-service   NodePort   10.43.87.229   <none>        80:30007/TCP   13s
[1]+  Done                    kubectl get pods --namespace gitops-kubernetes
```

Figure 2.12 – Results of applying the deployment file, where we can see useful information such as the Cluster-IP and the assigned ports

Now that we have deployed our application in the Kubernetes cluster, the next crucial step is to test its functionality. This is where **port forwarding** plays a key role.

Port forwarding

Port forwarding with kubectl allows us to temporarily route traffic from our local machine to a pod in the Kubernetes cluster. This method is especially useful for testing purposes as it enables us to interact with the application as if it were running locally, without the need to expose it publicly. By forwarding a local port to a port on the pod, we can verify the deployment's operational aspects, ensuring that our application behaves as expected in a controlled environment before making it accessible to external traffic. The following steps outline the process for executing port forwarding on the running pod and testing its functionality using **curl**:

1. **Start port forwarding**: Use the following `kubectl` command to start port forwarding from a local port to a port on the Pod:

   ```
   $ kubectl port-forward pod/[POD_NAME] [LOCAL_PORT]:[REMOTE_PORT]
   ```

   Replace `[POD_NAME]` with the name of your Pod. For instance, in *Figure 2.12*, the name of the pod is `hello-world-deployment-6b7f766747-nxj44`. Here, `[LOCAL_PORT]` should be replaced with the local port on your machine – for example, `9000` (ensure that the local port is not already used by another running service!) – while `[REMOTE_PORT]` should be replaced with the port on the Pod that you want to forward traffic to. In our case, as reported in *Figure 2.10*, the Pod port is `80`.

2. At this point, we are using the Pod's name, `hello-world-deployment-6b7f766747-nxj44`. So, if we want to forward traffic from local port `9000` to the Pod's port, `80`, the command would be as follows:

   ```
   $ kubectl port-forward hello-world-deployment-6b7f766747-nxj44
   --namespace gitops-kubernetes 9000:80
   ```

This will produce the following output:

```
Forwarding from 127.0.0.1:9000 -> 80
Forwarding from [::1]:9000 -> 80
```

The preceding output indicates that port forwarding is set up on your machine to redirect traffic from a local port to a port on a Kubernetes Pod or another network service. Keep this command running as it maintains the port forwarding session.

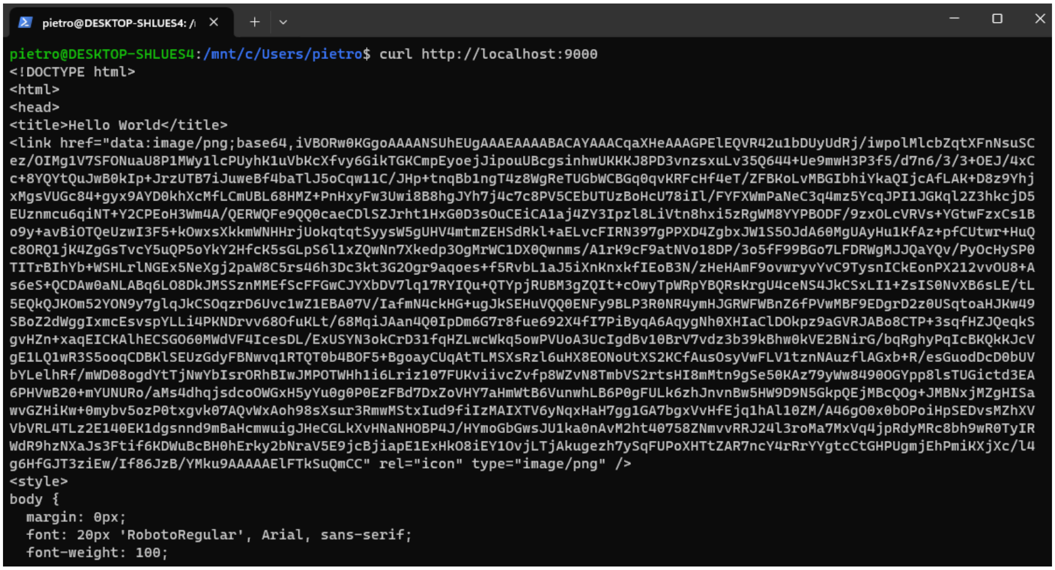
3. Open a new Terminal or Command Prompt and type the following command to open a new WSL shell:

```
$ wsl -d Ubuntu-22.04
```

4. Use `curl` to send a request to the local port that is being forwarded:

```
$ curl http://localhost:9000
```

This command sends a request to your local machine on port `9000`, which `kubectl` then forwards to the Pod's port (`80`). You should see the output of the request in your Terminal. Typically, this is the content that's served by your application running in the Kubernetes Pod, as shown in *Figure 2.13*:



```
pietro@DESKTOP-SHLUES4:/mnt/c/Users/pietro$ curl http://localhost:9000
<!DOCTYPE html>
<html>
<head>
<title>Hello World</title>
<link href="data:image/png;base64,iVBORw0KGgoAAAANSUhEUgAAAEAAAABACAYAAACqaXHeAAAGPElEQVR42u1bDUyUdRj/iwpolMlcbZqtXFnNsuSC
ez/OIMg1V7SFONuaU8P1MWy1lcPUyhK1uVbKcXfvy6GikTGKCmpEyoejJipouUBcgsinhwUKKKJ8PD3vnzsxuLv35Q644+Ue9mwH3P3f5/d7n6/3/3+OEJ/4xC
c+8YQYtQuJwB0kIp+JrzUTB7iJuweBf4baTlJ5oCqw11C/JHp+tnqBb1ngT4z8WgReTUGbWCBGq0qvKRFcHf4eT/ZFBKoLvMBGIbhiYkaQIjcAfLAK+D8z9Yhj
xMgsVUGc84+gyx9AYD0khXcMfLCmUBL68HMZ+PnHxyFw3Uwi8B8hgJYh7j4c7c8PV5CEbUTUzBoHcU78iIl/FYFXWmPaNeC3q4mz5YcqJPI1JGKql2Z3hkcjD5
EUznmcu6qiNT+Y2CPEoH3Wm4A/QERWQFe9QQ0caeCDlSZJrht1HxG0D3sOuCEiCA1aj4ZY3Ipzl8LiVtn8hxi5zRgWM8YYPBODF/9zxOLcVRVs+YGtwFzxCs1B
o9y+avBiOTQeUzwI3F5+kOwxsXkkmWNHHrjUokqtqtSyysW5gUHV4mtmZEHSdRkl+aELvcFIRN397gPPXD4ZgbxJW1S5OJdA60MgUAyHu1KfAz+pfCUtwr+HuQ
c8ORQ1jK4ZgGsTvcY5uQP5oYkY2HfcK5sGLpS6l1xZQwNn7Xkedp3OgMrWC1DX0Qwnms/A1rK9cF9atNVo18DP/3o5fF99BGo7LFDRWgMJJQaYQv/PyOcHySP0
TITrBIhYb+WSHLrlNGEx5NeXgj2paW8C5rs46h3Dc3kt3G2Ogr9aqoes+f5RvbL1aJ5iXnKnxkfIEoB3N/zHeHAmF9ovwryvYvC9TysnICkEonPX212vvOU8+A
s6eS+QCDAw0aNLABq6LO8DkJMSSznMMEfScFFGwCJYXbDV7lq17RYIQu+QTYpjRUBM3gZQIt+cOwyTpWRpYBQRsKrgU4ceNS4JkCSxLI1+ZsIS0NvXB6sLE/tL
5EQkQJKOm52YON9y7glqJkCSOqzrD6Uvc1wZ1EBA07V/IafmN4ckHG+ugJkSEHuVQQ0ENFy9BLP3R0NR4ymHJGRWFWBnZ6fPVwMBF9EDgrD2z0USqtoaHJKw49
SBoZ2dWggIxmcEsvspYLLi4PKNDrvv68OfuKLt/68MqiJAan4Q0IpDm6G7r8fue692X4fI7PiByqA6AqygNh0XHIaClDOkpz9aGVRJABo8CTP+3sqfHZJQeqkS
gvHZn+xaqEICKAlhECSGO60MWdVF4IcesDL/ExUSYN3okCrD31fqHZLwcWkq5owPVUoA3UcIgdBv10BrV7vdz3b39kBhw0kVE2BNirG/bqRghyPqIcBKQkKJcV
gE1LQ1wR3S5ooqCDBKlSEUzGdyFBNwvq1RTQT0b4BOF5+BgoayCUqAtTLMSXsRzl6uHX8EONoUtXS2KCfAusOsyVwFLV1tznNAuzflAGxb+R/esGuodDcD0bUV
bYLelhRf/mWD08ogdYtTjNwYbIsrORhBIwJMPOTWHh1i6Lriz107FUKviivcZvfp8WZvN8TmbVS2rtsHI8mMtn9gSe50KAz79yWw8490OGYpp8lsTUGictd3EA
6PHVwB20+mYUNURo/aMs4dhqjsdcoOWGxH5yYu0g0P0EzFBd7DxZoVHY7aHmWtB6VunwhLB6P0gFULk6zhJnvnBw5HW9D9N5GkpQEjMBcQOg+JMBNxjMZgHISa
wvGZHiKw+0mybv5ozP0txgvk07AQvWxAoh98sXsur3RmwMStxIud9fiIzMAIXTV6yNqxHaH7gg1GA7bgxVvHfEjq1hAl10ZM/A46gO0x0bOPoiHpSEDvsMZhXV
VbVRL4TLz2E140EK1dgsnnd9mBaHcmwuigJHeCGLkXvHNaNHOBP4J/HYmoGbGwsJU1ka0nAvM2ht40758ZNmvvRRJ24l3roMa7MxVq4jpRdyMRc8bh9wR0TyIR
WdR9hzNXaJs3Ftif6KDWuBcBH0hErky2bNraV5E9jcBjiapE1ExHkO8iEY1OvjLTjAkugezh7ySqFUPoXHTtZAR7ncY4rRrYYgtcCtGHPUgmjEhPmiKXjXc/l4
g6HfGJT3ziEw/If86JzB/YMku9AAAAAElFTkSuQmCC" rel="icon" type="image/png" />
<style>
body {
  margin: 0px;
  font: 20px 'RobotoRegular', Arial, sans-serif;
  font-weight: 100;
```

Figure 2.13 – Example of content served by our application running in the Kubernetes Pod

Congratulations on achieving this remarkable result! You've successfully deployed your first application in Kubernetes, and the content is being correctly served, as evidenced by the successful `curl` call. This is a significant milestone in your journey with Kubernetes, showcasing your ability to not only deploy an application but also ensure its proper functioning within the cluster.

In the upcoming section, we will delve deeper into Docker, closely examining its essential components, functionalities, and practical applications. We'll build our first Docker image and demonstrate how to run it as a container locally.

Getting started with containers

Containers have become a cornerstone in cloud-native application development due to their ability to package and isolate applications with all their dependencies. This isolation ensures consistency across various environments, making them highly efficient for both development and deployment. **Container images**, which are static files containing executable code and dependencies, follow a **layered** structure for efficient modification and storage, with each layer representing changes or additions.

Despite the versatility of containers, Kubernetes does not provide a native mechanism for building these images, necessitating external tools such as **Docker**.

Docker, an open source platform, has transformed the world of containerization by simplifying the creation, deployment, and execution of applications in containers. It enables developers to encapsulate applications with their dependencies in a unified format, facilitating software development. Docker's containers offer a **semi-isolated** environment, balancing isolation with efficiency, allowing multiple containers to run concurrently on a single host. These containers are both lightweight and portable, ensuring uniform functionality across diverse platforms, from local laptops to cloud infrastructures.

Docker files are instrumental in creating these images, specifying the steps and components to be included.

The **Open Container Initiative** (**OCI**) standardizes container image formats and runtimes, further enhancing interoperability and portability across different containerization technologies.

Docker setup

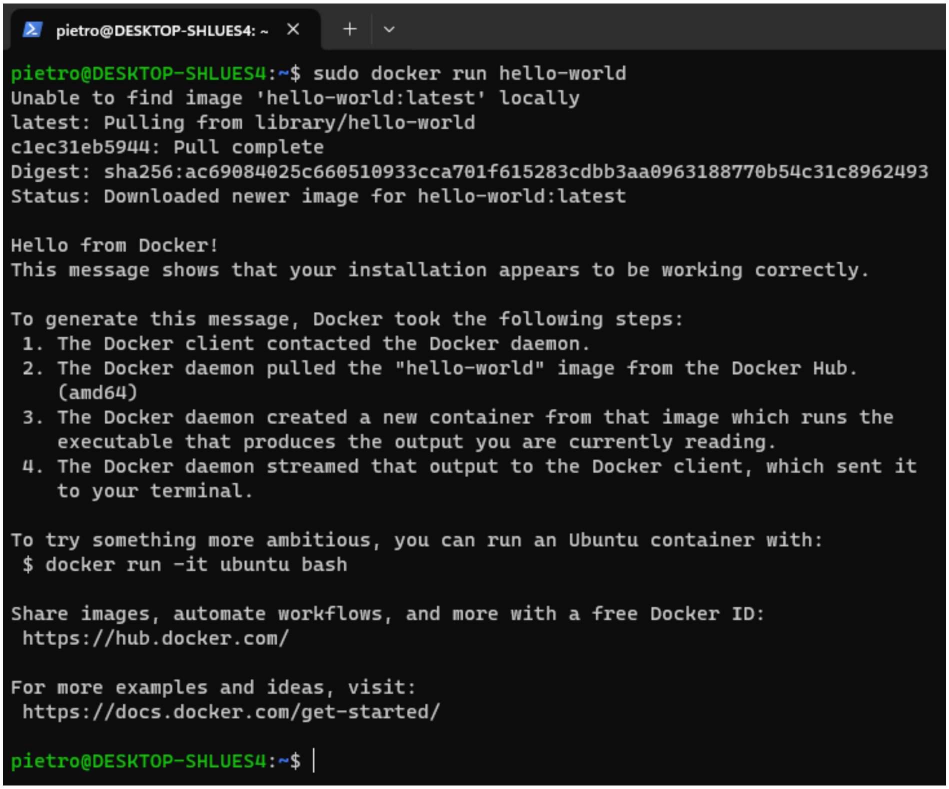
Up until now, we have focused on using Ubuntu 22.04 as an instance within WSL. While a step-by-step setup of Docker falls outside the scope of this book, you can find comprehensive installation guides and troubleshooting tips in the official Docker documentation: `https://docs.docker.com/engine/install/ubuntu/`. After successfully installing Docker, you can verify its installation and check that Docker is running correctly on your system by typing the following command in your Terminal:

```
$ sudo docker run hello-world
```

The `sudo docker run hello-world` command quickly verifies the installation and setup of Docker by running a very simple container. When executed, it does the following:

- `sudo`: Ensures the command is run with superuser privileges, which are often required for Docker commands.
- `docker run`: Tells Docker to run a container.
- `hello-world`: Specifies the image to use. In this case, it's the `hello-world` image, a minimal Docker image created by Docker, Inc. It's commonly used as a test image to validate that Docker is installed and running correctly.

If Docker has been correctly installed and configured, this command will pull the `hello-world` image from Docker Hub (if it's not already downloaded), create a new container from that image, and run it. The container simply displays a message confirming that Docker is installed correctly and then exits, as shown in *Figure 2.14*:



```
pietro@DESKTOP-SHLUES4:~$ sudo docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c1ec31eb5944: Pull complete
Digest: sha256:ac69084025c660510933cca701f615283cdbb3aa0963188770b54c31c8962493
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

pietro@DESKTOP-SHLUES4:~$
```

Figure 2.14 – Result of executing the docker run hello-world command

Docker alternatives

Although Docker is one of the most popular tools for building container images, there are several alternative tools available:

- **Podman**: An open source, daemonless container engine that can run on Linux systems. It is compatible with Docker but does not require a running daemon. Podman is known for enabling easier management of containers and pods.
- **Rancherdesktop**: An open source application that provides all the essentials to work with containers and Kubernetes on desktop.
- **containerd**: A core container runtime that adheres to industry standards, available as a service for both Linux and Windows. It is capable of managing the entire life cycle of containers on its host system.
- **CRI-O**: This is a realization of the Kubernetes Container Runtime Interface, facilitating the use of runtimes compatible with the OCI. It serves as a bridge, connecting OCI-compliant runtimes with kubelet.
- **rkt (pronounced 'rocket')**: Developed by CoreOS, it's a Pod-native container engine for Linux. It's designed for security, simplicity, and composability within modern cluster environments.
- **LXD**: A cutting-edge manager for system containers and VMs that provides a user experience akin to VMs but through the use of Linux containers.
- **OpenVZ**: This is a virtualization solution built on container technology for Linux systems that's capable of generating several secure and isolated Linux containers on a singular physical server.

To assist in choosing the most suitable containerization tool for your specific needs, the following table provides a comparison of various Docker alternatives. It highlights their key features and ideal use cases, offering insights into which tool may best align with your project's requirements or preferences:

Alternative	**Description**
Podman	Best for environments that prioritize security and for users who prefer a solution without a daemon. It's fully compatible with Docker's CLI, making it a seamless replacement.
Rancherdesktop	A user-friendly, GUI-based tool tailored for developers who want an easier way to manage containers and Kubernetes, especially on desktop environments for development and testing purposes.
Containerd	Chosen for its performance and reliability as a container runtime in production environments. Lacks Docker's image-building features but excels in running containers efficiently.

Alternative	Description
Rkt	Previously a viable alternative, but its development has ceased, potentially limiting its suitability for long-term projects.
OpenVZ	Ideal for hosting solutions or for scenarios requiring multiple, isolated Linux environments on a single host, with a focus on resource efficiency and scalability.

Table 2.1 – Comparative overview of containerization tools – evaluating alternatives to Docker for diverse development needs

Dockerfile

The first step in creating a container image involves defining a **Dockerfile**, which is essentially a **blueprint** for the image. This file contains a set of instructions and commands that tell Docker how to build the image. It starts with specifying a base image to build upon, often a minimal version of an operating system, such as Ubuntu or Alpine Linux. Then, additional layers are added by specifying dependencies, copying application files, and setting environment variables. Each command in a Dockerfile creates a new layer in the image, building up the environment that's needed to run the application. The following is an example of a Dockerfile:

```
# Use an official Python runtime as a parent image
FROM python:3.8-slim
# Set the working directory in the container
WORKDIR /usr/src/app
# Copy the current directory contents into the container at /usr/src/
app
COPY . .
# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt
# Make port 80 available to the world outside this container
EXPOSE 80
# Define environment variable
ENV NAME World
# Run app.py when the container launches
CMD ["python", "app.py"]
```

Let's take a closer look at this file:

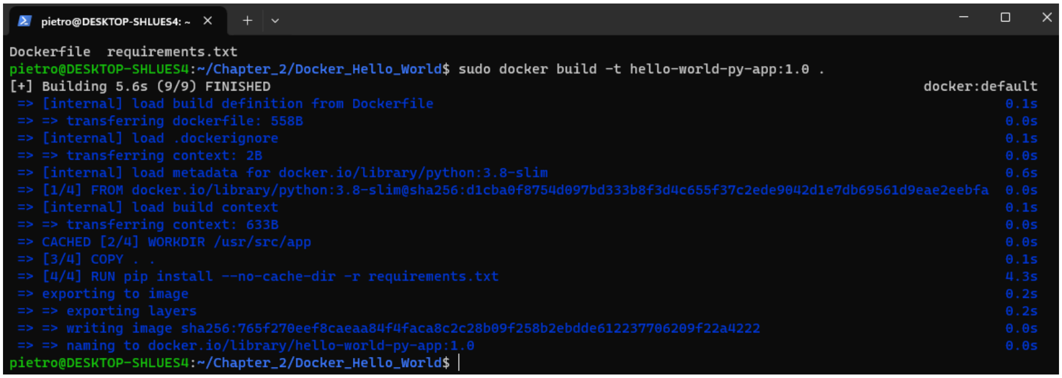
- `FROM python:3.8-slim`: This line indicates the base image from which you are building. The Dockerfile starts with the Python 3.8 image, specifically the slim variant, which is a smaller, more compact version.
- `WORKDIR /usr/src/app`: This line sets the working directory inside the container to `/usr/src/app`. Future commands will run in this directory.

- `COPY . .`: This line copies files from the Dockerfile's current directory to the working directory in the container (`/usr/src/app`).
- `RUN pip install --no-cache-dir -r requirements.txt`: This line executes a command inside the container, which in this case is installing Python dependencies listed in `requirements.txt`.
- `EXPOSE 80`: The line informs Docker that the container listens on port `80` at runtime. Note that this does not publish the port.
- `ENV NAME World`: This line sets the `NAME` environment variable to `World`. This can be used by the application running in the container.
- `CMD ["python", "app.py"]`: The default command to run when a container starts. This line runs the Python application.

This Dockerfile provides a simple example of building an image of a simple **Flask** web app application written in Python. It builds an image that includes the application and its dependencies, making it ready to run in a containerized environment. Now, imagine that you have a Dockerfile in your current directory together with a `requirements.txt` file and you want to build a Docker image from this Dockerfile. The command you would use is as follows:

```
$ sudo docker build -t hello-world-py-app:1.0 .
```

At this stage, as shown in *Figure 2.15*, your container image is in the process of being built. During this build, Docker retrieves any existing layers from **public container registries** such as DockerHub, Quay, or Red Hat Registry. The topic of container registries will be introduced in the upcoming pages. It then adds a new layer based on the instructions in your Dockerfile. If some layers are already present locally, Docker will use these from the container cache or Docker cache, speeding up the build process by avoiding redundant downloads:



```
Dockerfile  requirements.txt
pietro@DESKTOP-SHLUES4:~/Chapter_2/Docker_Hello_World$ sudo docker build -t hello-world-py-app:1.0 .
[+] Building 5.6s (9/9) FINISHED                                                                docker:default
 => [internal] load build definition from Dockerfile                                                      0.1s
 => => transferring dockerfile: 558B                                                                      0.0s
 => [internal] load .dockerignore                                                                         0.1s
 => => transferring context: 2B                                                                           0.0s
 => [internal] load metadata for docker.io/library/python:3.8-slim                                        0.6s
 => [1/4] FROM docker.io/library/python:3.8-slim@sha256:d1cba0f8754d097bd333b8f3d4c655f37c2ede9042d1e7db69561d9eae2eebfa  0.0s
 => [internal] load build context                                                                         0.1s
 => => transferring context: 633B                                                                         0.0s
 => CACHED [2/4] WORKDIR /usr/src/app                                                                     0.0s
 => [3/4] COPY . .                                                                                        0.1s
 => [4/4] RUN pip install --no-cache-dir -r requirements.txt                                              4.3s
 => exporting to image                                                                                    0.2s
 => => exporting layers                                                                                   0.2s
 => => writing image sha256:765f270eef8caeaa84f4faca8c2c28b09f258b2ebdde612237706209f22a4222              0.0s
 => => naming to docker.io/library/hello-world-py-app:1.0                                                 0.0s
pietro@DESKTOP-SHLUES4:~/Chapter_2/Docker_Hello_World$ |
```

Figure 2.15 – Result of the docker build command

The container image is now available in the local Docker cache and ready to be used. Its presence can be verified with the following command:

```
$ docker images
REPOSITORY          TAG       IMAGE ID       CREATED          SIZE
hello-world-py-app  1.0       765f270eef8c   7 minutes ago    139MB
hello-world         latest    d2c94e258dcb   8 months ago     13.3kB
```

Once the image has been created, it can be used locally or uploaded to a public container registry for external use, such as within a CI/CD pipeline. For our purposes, we'll run the container image locally. To do this, use the following command:

```
$ sudo docker run -p 8080:8080 -ti hello-world-py-app:1.0
```

The preceding command includes several options:

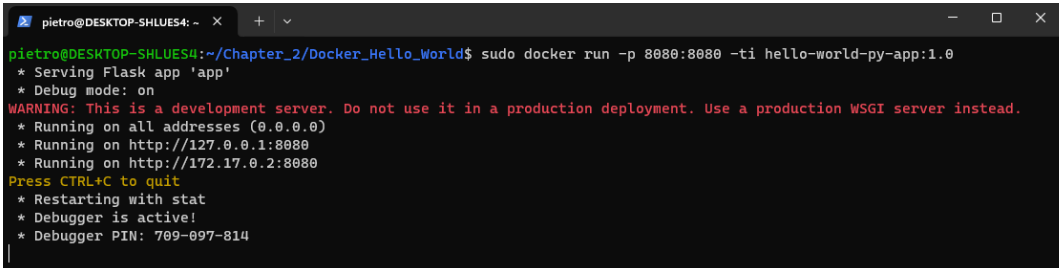
- The `-p` option binds a port on the host to a port on the container, allowing external access to the container's services
- The `-t` option allocates a pseudo-TTY, which provides a Terminal within the container
- The `-i` option enables interactive mode, allowing interaction with the container
- The `-d` option runs the container in the background and outputs a hash, which can be used for asynchronous interaction with the container

Public container registry

A public container registry is an online service where users can store and share container images. It serves as a centralized repository, facilitating the distribution of containerized applications. To upload and manage images, users typically need to create an account with the registry provider. This account allows them to publish, update, and maintain their images, making them accessible to others. Public registries such as Docker Hub, Google Container Registry, and Amazon Elastic Container Registry are popular choices, offering easy access over the internet. These platforms not only provide storage for container images but often come with additional features such as version control, cataloging, and security scanning. An account with these services enables developers to deploy applications consistently across different environments, streamline software development, and collaborate more effectively with others in the community.

Please note that the process of creating an account with a public container registry, although a crucial step for managing and distributing container images, falls outside the scope of this chapter and book. Each registry, such as Docker Hub or Google Container Registry, has its own set of guidelines and procedures for account creation and management. You are encouraged to refer to the specific documentation provided by these services for detailed instructions on setting up an account.

The preceding command will launch the application within the Docker network and bind it to port `8080` on our local machine. It will then wait for incoming requests, as illustrated in *Figure 2.16*:



```
pietro@DESKTOP-SHLUES4:~/Chapter_2/Docker_Hello_World$ sudo docker run -p 8080:8080 -ti hello-world-py-app:1.0
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:8080
 * Running on http://172.17.0.2:8080
Press CTRL+C to quit
 * Restarting with stat
 * Debugger is active!
 * Debugger PIN: 709-097-814
```

Figure 2.16 – Result of the docker run command

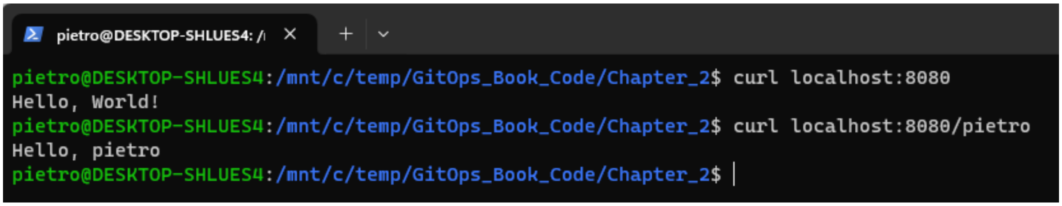
From a new Terminal, we can try to access the running container using `curl`:

```
$ curl http://localhost:8080
```

Alternatively, we can run the following command:

```
$ curl localhost:8080/[YOUR_NAME_HERE]
```

You'll receive a response similar to the one shown in *Figure 2.17*, where I used my name to obtain the output:



```
pietro@DESKTOP-SHLUES4:/mnt/c/temp/GitOps_Book_Code/Chapter_2$ curl localhost:8080
Hello, World!
pietro@DESKTOP-SHLUES4:/mnt/c/temp/GitOps_Book_Code/Chapter_2$ curl localhost:8080/pietro
Hello, pietro
pietro@DESKTOP-SHLUES4:/mnt/c/temp/GitOps_Book_Code/Chapter_2$ 
```

Figure 2.17 – Example of responses received from our Python Flask application running as a containerized image

Now that we are equipped with the necessary tools and understanding of Docker and containers, in the next section, will integrate these elements so that we can construct our first CD pipeline using Docker and K3s.

Sample workflow – effortless CD with Docker and K3s

At this point, we are ready to create a very simple CD pipeline using the tools we've explored so far. The basic idea is to simulate the operations performed by a developer who needs to update the Flask app we've used so far to add a new feature that allows the current date and time to be retrieved.

Our example will consist of performing the following steps:

1. **Local development**: We will edit the previous Python Flask app to expose a new service that returns the current date and time.
2. **Dockerizing the application and running it locally**: We will build the new version of the Docker image locally using the docker build command, as we did previously. Use the *Dockerfile* section as a reference.

   After building the image, we will run it locally using Docker to ensure the containerized application works as expected. Use the *Dockerfile* section as a reference.
3. **Publishing the image to a public container registry**: We will publish the build image to a public registry repository.
4. **Deploying to K3s**: We will write the Kubernetes manifest file to specify how our application should be deployed on K3s, including which Docker image to use and the desired number of replicas.

You will apply this configuration to your K3s cluster using the commands you learned about in the *Exploring K3s as a lightweight Kubernetes distribution* section of this chapter.

Let's get started!

Local development

Edit the `app.py` file present in this book's GitHub repository by adding the following Python code:

```
…
@app.route('/datetime')
def datetime():
    import datetime
    now = datetime.datetime.now()
    return now.strftime("%Y-%m-%d %H:%M:%S")
…
```

You're free to use whatever code editor you like to edit this file – it doesn't matter.

Dockerizing the application and running it locally

Follow these steps to Dockerize the application and run It locally:

1. In the *Dockerfile* section, we created the first version of our Docker image, tagged as `hello-world-py-app:1.0`. Now that we have added a new feature, it's time to create a new version of that image. We will use a tag of `2.0` using the following `docker build` command:

   ```
   $ sudo docker build -t hello-world-py-app:2.0 .
   ```

2. Upon typing the following command, you should be able to see both images listed:

   ```
   $ sudo docker images
   ```

   The result of this command should look like this:

   ```
   hello-world-py-app   2.0       a7d7ab4514fa   19 seconds
   ago      145MB
   hello-world-py-app   1.0       3f8f095a7b37   About an hour
   ago   145MB
   ```

3. We can run the Docker image locally with the following `docker` command:

   ```
   $ sudo docker run -p 8080:8080 -ti hello-world-py-app:2.0
   ```

   This will produce a result similar to the following:

   ```
   * Serving Flask app 'app'
   * Debug mode: on
   WARNING: This is a development server. Do not use it in a
   production deployment. Use a production WSGI server instead.
    * Running on all addresses (0.0.0.0)
    * Running on http://127.0.0.1:8080
    * Running on http://172.17.0.2:8080
   Press CTRL+C to quit
   …
   ```

4. From a new Terminal, we can try to access the running container using `curl`:

   ```
   $ curl http://localhost:8080/datetime
   ```

   We'll obtain the current response:

   ```
   2024-01-13 12:39:50
   ```

Well done – as a developer, you have tested that the new feature is working as expected! Now, we can publish our image to a public repository.

Publishing the image to a container registry

Publishing our `hello-world-py-app:2.0` Docker image to a public repository involves several steps:

1. Assuming that you have already an account on a public container registry such as Docker Hub, the first step is to open a new Terminal and log in to the registry using the Docker CLI:

   ```
   $ sudo docker login
   ```

   When requested, you need to enter your credentials – that is, the username and password you used to create an account.

2. Before pushing an image, we need to tag it with the registry's name. For Docker Hub, it's usually in `username/repository:tag` format. Run the following command to tag the previously built image:

```
$ sudo docker tag hello-world-py-app:2.0 [yourusername]/hello-
world-py-app:2.0
```

3. Then, we need to push the tagged image:

```
$ sudo docker push [yourusername]/hello-world-py-app:2.0
```

   This process will take some time because we are uploading our image to the repository.

4. To verify that the image is in the registry, log in to your Docker Hub account (or your registry's interface) and navigate to your repositories to confirm that the `hello-world-py-app:2.0` image is listed there.

Deploying to K3s

The time to deploy our image to our local Kubernetes cluster has finally arrived! We can reuse the same Kubernetes manifest file that we used in the *Our first deployment with K3s* section, but we are going to apply a couple of edits, with the most important one being to update the manifest file so that it indicates where the Kubernetes cluster has to download the image, using our container repository. So, let's get started:

1. In the `deployment` section of the manifest file, we have to change the image value from `nginxdemos/hello` to `[yourusername]/hello-world-py-app:2.0`. Then, we have to change the name (where specified in the file) from `hello-world` to `first-cd-pipeline`:

```
    spec:
      sectioners:
      - name: first-cd-pipeline
        image: [yourusername]/hello-world-py-app:2.0
```

   Here, we have also changed the name of the deployment to `first-cd-pipeline-deployment`. The deployment file can be found in the `Chapter02` folder in this book's GitHub repository.

2. Save the new file, naming it as `first-cd-pipeline-deployment.yaml`, and apply the deployment with the following command:

```
$ kubectl apply -f first-cd-pipeline-deployment.yaml
```

The response should look like this:

```
deployment.apps/first-cd-pipeline-deployment created
service/first-cd-pipeline-service created
```

3. Before establishing port forwarding, as described at the end of the *Our first deployment with K3s* section, we need to get some useful information by running the following command:

   ```
   $ kubectl get pods --namespace gitops-kubernetes & kubectl get
   services --namespace gitops-kubernetes
   ```

   This will produce an output similar to the following:

   ```
   NAME                        TYPE       CLUSTER-IP
   EXTERNAL-IP   PORT(S)        AGE
   first-cd-pipeline-service   NodePort   10.43.172.10
   <none>        80:30007/TCP   29s
   NAME                         READY   STATUS    RESTARTS
   AGE
   first-cd-pipeline-deployment-5b85cfd665-5626b   1/1
   Running   0          29s
   ```

4. At this point, we have all the information we need to perform port forwarding:

   ```
   $ kubectl port-forward first-cd-pipeline-deployment-5b85cfd665-
   5626b --namespace gitops-kubernetes 8080:80
   ```

5. Open a new Terminal and use `curl` to test that the new feature has been deployed and hosted by the K3s cluster:

   ```
   $ curl http://localhost:8080/datetime
   ```

   You should see an output similar to the following:

   ```
   2024-01-13 17:59:39
   ```

6. To delete the deployment, type the following command:

   ```
   $ kubectl delete -f first-cd-pipeline-deployment.yaml
   ```

Congratulations on reaching this milestone with a manual CD deployment!

The steps outlined here for publishing a Docker image to a public container registry should be viewed as a manual example that illustrates the basic principles of CD. In practice, however, this process is typically automated using tools such as Git Actions, which streamline and optimize the deployment cycle. While these manual steps provide a foundational understanding, real-world applications often rely on more sophisticated automation for efficiency and consistency. In the next chapter, we'll delve into how such tools can be integrated into your workflow, thereby enhancing the CD process and reducing the need for manual intervention.

Summary

In this chapter, we navigated the practical aspects of deploying cloud-native applications using Kubernetes and K3s, highlighting key techniques for efficient container management and orchestration. This chapter focused on building foundational skills that are crucial for managing cloud-native environments, including understanding Kubernetes resources and deployment methodologies.

As we move to the next chapter, the emphasis will shift to introducing Git tools. We'll explore how these tools can be leveraged to create an automated CI/CD pipeline, an essential component for seamlessly deploying and managing cloud-native applications, as well as enhancing development and operational workflows.

Further reading

To learn more about the topics that were covered in this chapter, take a look at the following resources:

- [*1*] `https://learn.microsoft.com/en-us/windows/wsl/about`
- [*2*] `https://www.virtualbox.org/`

3

Version Control and Integration with Git and GitHub

So far, our exploration has shed light on the foundational concepts of GitOps and its critical role in managing cloud-native operations. This journey has involved unpacking the principles, benefits, and transformative nature of GitOps in infrastructure management, as well as its capacity to enhance and streamline cloud-native processes.

This chapter will further expand on these concepts. It will delve into the significance of Git in **version control**, an essential component of modern software development. The chapter will also explore GitHub's contribution to collaborative development and how it can be maximized for team efficiency. Additionally, it will discuss the integration of GitOps practices with Git and GitHub, highlighting automated integration processes.

The chapter outlines the best practices for utilizing Git and GitHub within a GitOps framework, aiming to enhance workflow efficiency and promote a strong DevOps culture.

In this chapter, we'll focus on these key areas:

- Version control systems
- Why Git?
- Exploring GitHub
- Integrating GitOps and GitHub

Technical requirements

To fully engage with the concepts and practices presented in this chapter, there are a few technical requirements. You should have a basic understanding of software development processes and a familiarity with command-line tools. Access to a computer with internet connectivity is necessary to follow along with examples involving Git and GitHub. This hands-on approach will enable you to apply the theories and practices of GitOps in real-world scenarios.

The code for this chapter is available in the `Chapter03` folder of our GitHub repository at `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`.

Exploring version control systems – local, centralized, and distributed

Version control systems (**VCSs**) are essential tools in modern software development, as they are responsible for enabling efficient management of changes to files, especially code. Git, a widely used VCS, exemplifies this by tracking modifications, allowing users to view and revert to specific versions of files as needed. This functionality is crucial for collaborative projects, where it provides a comprehensive record of who made changes, when, and why, enhancing accountability and clarity.

At its core, version control is about recording changes over time. This makes it easier to recover previous states of a project, whether for fixing bugs, testing, or undoing changes. It's not limited to software code; version control can be applied to any file type, making it an asset for professionals such as graphic or web designers who might need to manage different versions of their work. The use of a VCS minimizes risks associated with file loss or errors, as it provides a safety net for easily reverting to earlier states and understanding the evolution of a project. This leads to increased efficiency and a more structured workflow with minimal overhead.

By integrating version control into their workflow, teams and individuals gain a powerful tool to navigate the complexities of project development, ensuring a smoother, more controlled process of continuous improvement and collaboration.

VCS can be categorized into three main types: local, centralized, and distributed:

- **Local VCSs** (**LVCSs**): These are the simplest forms. They involve keeping track of changes in files on a local computer, typically using a database that stores all the changes to files under revision control. This system is straightforward but has limitations, especially in the contexts of collaboration and data backup. *Figure 3.1* shows an example of LVCS on the left side.
- **Centralized VCSs** (**CVCSs**): These systems, such as **Subversion** (**SVN**), have a single server that contains all versioned files. Various clients check out files from this central place. This setup significantly enhances collaboration compared to LVCS, as everyone knows what everyone else is working on. However, the central server is also a single point of failure; if it goes down, no one can collaborate or save versioned changes.

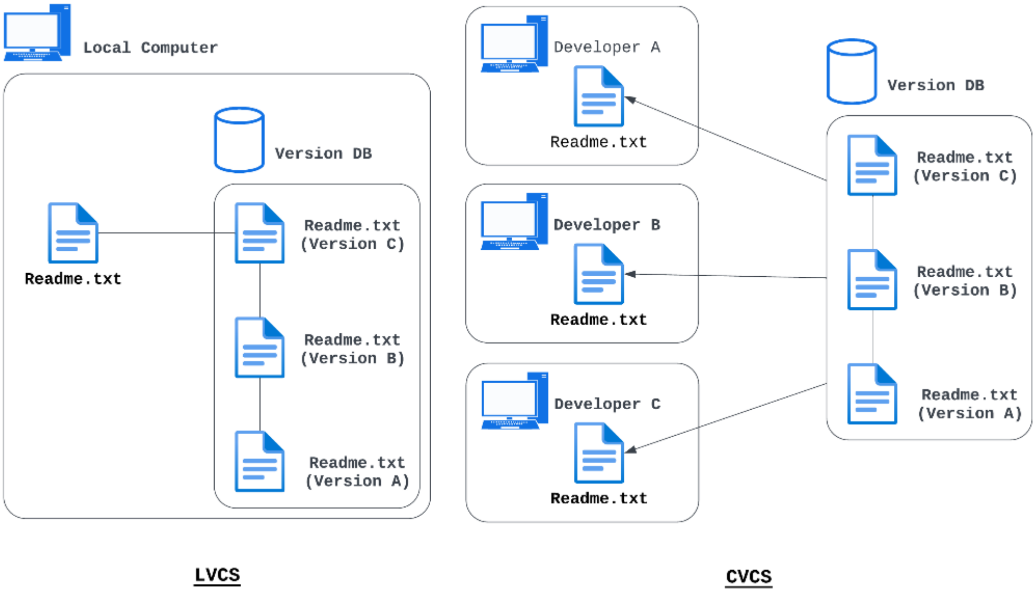


Figure 3.1 – An example of LCVS and CVCS

Concurrent version systems and SVN

The **Concurrent Versions System** (**CVS**) was an early version control system that enabled collaborative software development by allowing teams to work simultaneously on a project using a centralized repository. However, it had limitations, especially in handling binary files. It also lacked advanced features. SVN was developed as a modern alternative to CVS, addressing these shortcomings. SVN, also a centralized system, improved collaboration by allowing multiple users to work efficiently on a project with a centralized source for file changes, enhancing the tracking and management of changes made by different team members.

- **Distributed VCSs** (**DVCSs**): Systems such as Git fall into this category. Instead of a single central repository, each user has a complete copy of the entire repository. This approach allows for more complex workflows, increases redundancy and backup options, and enables offline work. It also allows for various forms of **merging** and **branching**, offering a more flexible approach to version control, especially for large-scale projects with many contributors.

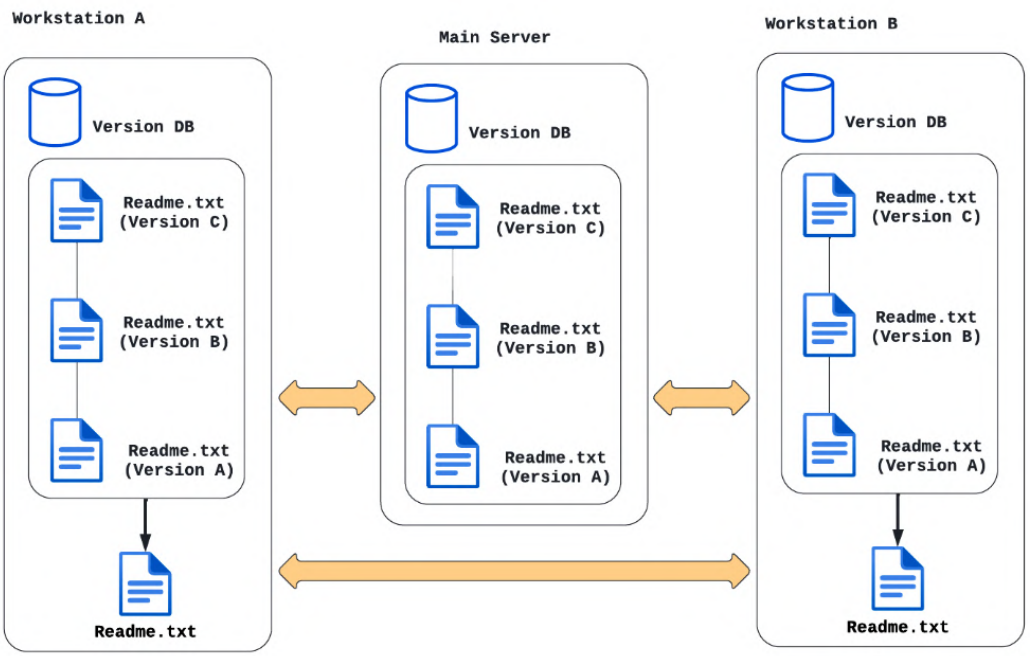


Figure 3.2 – An example of DVCS

Having explored the landscape of VCSs—including LVCSs, CVCSs, and DVCSs, each offering distinct benefits and suited to specific project needs—it's clear why Git stands out among DVCSs. Let's delve deeper into the unique advantages and capabilities that make Git an essential tool in modern software development in the next section.

Why Git?

The story of Git begins with the developmental challenges of the Linux kernel project in the early 2000s. This period saw Linus Torvalds, the creator of Linux, facing a dilemma with the existing version control tools. The limitations and restrictions of the proprietary DVCS, BitKeeper, used by the Linux community at that time, led to the need for a new system. This spurred the creation of Git in 2005, marking a significant turn in the version control system landscape.

Git was designed with specific goals in mind, drawing from the lessons learned during the use of BitKeeper. Key among these were speed, simple design, and robust support for non-linear development. The system was tailored to handle large projects such as the Linux kernel efficiently both in terms of speed and data size. This focus on efficiency and flexibility was crucial given the scale and complexity of such projects.

The structure of Git is unique and effective. It is a distributed version control system, meaning that every user has the complete history of the project stored locally. This design not only enhances speed but also enables a more flexible and secure approach to version control. **Repositories** in Git are comprehensive, containing **commits**, which are essentially snapshots of the project at various stages, as well as references to these commits (known as **heads**). Over the years, Git has evolved and matured, retaining its initial qualities while becoming more user-friendly. Its integration with platforms such as GitHub (we will delve into more details about GitHub in the *Exploring GitHub* section of this chapter), **Bitbucket**, and **GitLab** has further amplified its capabilities, providing centralized hubs for collaboration, code sharing, and project management.

To illustrate the widespread adoption of Git, it's worth noting that the GitHub community announced having surpassed 100 million developers on their platform:

`https://github.blog/2023-01-25-100-million-developers-and-counting/`

The adoption of Git in GitOps practices underscores its reliability and efficiency, making it a cornerstone in modern software development and version control.

Having understood the importance of Git from various perspectives, we will now shift our focus to a more practical approach. In the upcoming sections, we'll begin with the basics of Git commands and examples to offer a hands-on experience.

Git setup

This chapter doesn't cover the details of Git installation in depth, but you can find a thorough, step-by-step guide in the official Git documentation (refer to [1] in the *Further reading* section at the end of this chapter). For the purposes of our discussions, we'll assume that Git is already installed on a **Windows Subsystem for Linux** (**WSL**) Ubuntu 22.04 distribution. It is important to underline that Git can be natively installed on Windows (without WSL), macOS, and Linux.

For additional information on setting up WSL, please refer to *Chapter 2*, particularly the *Setup of WSL* section, which provides relevant guidance and insights.

To verify your Git installation, you can enter a specific command in your command prompt. This command will help you confirm that Git is properly installed and functioning on your system:

```
$ git version
```

The expected response should resemble the following:

```
git version 2.45.1
```

Great job! In the following sections, we will delve into the basics of Git, explore its commands, and examine the most interesting features that are essential for GitOps.

Creating and cloning a Git repository

A Git repository is a cornerstone in the Git version control system, encompassing all project files and their revision histories. It's marked by the `.git` directory containing necessary change-tracking data. Repositories exist either as local entities on a user's computer or as remote versions on servers such as GitHub, enabling both individual work and collaborative efforts. They play a crucial role in project version management and evolution, storing files and histories as commits in a structured, linked-list format. This setup supports varied interactions such as **cloning** (creating a local copy of a repository from a remote server), **branching** (diverging from the main code base to create separate versions for development or testing without affecting the original code), and **code version comparisons** (examining the differences between various versions of code files to track changes or merge updates).

A Git repository can be started using two commands:

- `git init`: This command is used to initialize a new Git repository in a directory that is not currently under version control. Executing this command creates a hidden subfolder named `.git` in the directory, setting up the necessary structure for repository management and version control.
- `git clone`: This command is used to create a local copy of an existing Git repository, typically one that's hosted remotely such as on GitHub. When you clone a repository, it downloads not only the project's files but also its complete history and all branches into your local repository. This process ensures that you have the entire development history of the project at your disposal.

Open the command prompt and activate a WSL instance, as detailed in *Chapter 2*, in the *Setup WSL* section, then follow the ensuing numbered steps to create and clone a Git repository:

1. Create a new directory named `Chapter_3`:

   ```
   $ mkdir Chapter_3
   ```

2. To set up `Chapter_3` as the current working directory, type `cd Chapter_3` into the command prompt.
3. Once in the `Chapter_3` directory, you are ready to initialize a new Git repository using the `git init` command. This step is crucial for starting version control in your new project directory:

   ```
   $ git init
   ```

   The output from the command should look something like this:

   ```
   hint: Using 'master' as the name for the initial branch. This
   default branch name
   hint: is subject to change. To configure the initial branch name
   to use in all
   …
   ```

```
hint:   git branch -m <name>
Initialized empty Git repository in /home/pietro/Chapter_3/.git/
```

4. By listing the contents of the working directory using the `ls -a` command, you should be able to see the newly created `.git` hidden directory. This directory is an indication that the Git repository has been successfully initialized:

   ```
   $ ls -a
   ```

   The response to this command should resemble the following output:

   ```
   .  ..  .git
   ```

5. To list the contents and subfolder structure of the `.git` directory, you can use the `ls -a ./.git` command. This will display all the files and subdirectories within the `.git` directory, providing insight into the structure of the initialized Git repository:

   ```
   $ ls -a ./git
   ```

   The output should look like the following:

   ```
   .  ..  HEAD  branches  config  description  hooks  info
   objects  refs
   ```

6. To initialize a repository by cloning it, you should first navigate to a different directory. Once there, execute the clone command with the appropriate URL to create a copy of an existing Git repository in that new location. This process will replicate the repository, including its history and branches, into your specified directory:

   ```
   $ git clone https://github.com/PacktPublishing/Implementing-
   GitOps-with-Kubernetes
   ```

   The output from the command should look something like this:

   ```
   Cloning into 'Implementing-GitOps-with-Kubernetes'...
   remote: Enumerating objects: 151, done.
   remote: Compressing objects: 100% (94/94), done.
   ...
   Resolving deltas: 100% (32/32), done.
   ```

We have now learned how to create a new repository from scratch or by cloning an existing one. Next, we'll focus on understanding and using basic Git commands to start recording changes in our repository. This step is crucial for managing and tracking the evolution of your project files.

The basics of Git

Understanding the basics of Git involves familiarizing yourself with a set of fundamental commands that facilitate version control in software development. These commands enable you to track and manage changes to your project's files, offering control over every aspect of its evolution. From staging and committing changes to managing branches and updating your project with the latest developments, mastering these commands is essential for the effective use of Git. This introductory overview will guide you through key commands such as `git status`, `git add`, `git commit`, `git branch`, `git merge`, `git pull`, and `git push`, each playing a critical role in the Git workflow:

- `git status`: This command shows the status of changes, indicating whether they are untracked, modified, or staged.
- `git add`: This command stages changes for inclusion in your project's history. It prepares the changes for the next step, allowing you to control what becomes part of the project's history.
- `git commit`: This command saves the staged changes to the project's history, functioning like capturing a snapshot. It finalizes what you have staged with `git add`.
- `git branch`: Use this to view the branches being worked on locally, giving insight into the various lines of development.
- `git merge`: This merges different lines of development and is commonly used to integrate changes from one branch into another (e.g., merging a `feature` branch into the `master` branch).
- `git fetch`: This command updates your local database of changes from a remote repository, which is useful for reviewing updates before integrating them into your local branch.
- `git pull`: This updates your local development line with changes from its remote counterpart. It is useful when you need to reflect changes made by others in your local environment.
- `git push`: This updates the remote repository with commits made locally, ensuring that the remote repository reflects your recent changes.

A basic Git commit-status-push loop example

Let's start with a practical example of how to use Git locally using the commands listed in the *The basics of Git* section. You are free to use the repository initialized in the *Creating and cloning a Git repository* section of this chapter or to start with a new one:

1. Set up a GIT identity:

```
$ git config --global user.email "you@example.com"
$ git config --global user.name "Your Name"
```

2. Use the `git status` command to see the current state of your files, whether they are untracked, modified, or ready to be committed:

   ```
   $ git status
   ```

   The response to this command should look something like this:

   ```
   On branch master
   No commits yet
   nothing to commit (create/copy files and use "git add" to track)
   ```

   This indicates that your working directory is clean, meaning that no tracked files have been modified and no untracked files are present. Additionally, it confirms that you're on a specific branch (`master`) that hasn't diverged from its remote counterpart.

3. Create a new non-empty file in the working directory:

   ```
   $ echo 'The content of this file will be committed.' > README
   ```

   The preceding command will create a new non-empty `README` file in the current working directory.

4. Execute the `git status` command again:

   ```
   On branch master
   No commits yet
   Untracked files:
      (use "git add <file>..." to include in what will be
   committed)
          README
   nothing added to commit but untracked files present (use "git
   add" to track)
   ```

   The `README` file appears with the `untracked` status because Git recognizes it as a file that is not present in the last commit and that hasn't been prepared for inclusion in future commits. The `untracked` status prevents Git from automatically adding files that might not be intended for version control, such as binary files. To include the `README` file in your project's version history, you need to explicitly track it with Git, ensuring it's considered for subsequent commit snapshots.

5. To track modifications, to add the new file, use the `git add` command:

   ```
   $ git add README
   ```

6. Execute the `git status` command again:

   ```
   On branch master
   No commits yet
   Changes to be committed:
   (use "git rm --cached <file>..." to unstage)
           new file:   README
   ```

`README` is now marked as `staged` because it appears under the `Changes to be committed` section, indicating that it's ready for the next commit.

7. With our staging area prepared as desired, it's now time to record our changes in the repository's history using `git commit -m "Add the README file"`, whereby we specify the modifications that we've implemented:

```
$ git commit -m "Add the README file."
```

The output from the `git commit` command should be like the following:

```
[master (root-commit) f933b54] Add the README file.
 1 file changed, 1 insertion(+)
 create mode 100644 README
```

Whenever you add, modify, or delete one or more files (or non-empty folders), you'll need to repeat *steps 5* and *7* to ensure that those changes are properly staged and committed.

Git workflows

The basic Git workflow involves a strategic approach to branching that facilitates efficient development and release processes. This strategy includes different types of branches:

- `Main (or Master)`: The foundational branch where production-ready code is maintained throughout the project's life cycle
- `Develop`: Initiated alongside the main branch, it houses pre-production code and features under testing
- `Feature`: Employed for developing new features, these branches merge back into the `develop` branch upon completion
- `Release`: Dedicated to preparing code for production release
- `Hotfix`: Addresses urgent fixes directly in the main branch

In this workflow, changes are tracked as commits. They document the history of file modifications, including what changed, who changed it, and when. This structure not only organizes work effectively but also accelerates the release cycle by clearly defining the purpose of each branch and streamlining the integration of new features, bug fixes, and releases. *Figure 3.3* illustrates an example of a Git workflow.

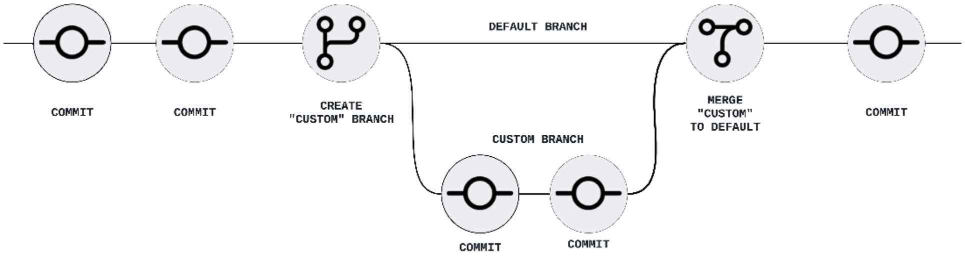


Figure 3.3 – An example of a Git workflow

Initially, the content of a repository resides in a default branch. For making modifications or additions, it's recommended to create and work within your own branch. This practice ensures that the main code base remains stable while you develop features or fixes. Delving deeply into the intricacies of Git, including branching, merging, and remote strategies, exceeds the scope of this book. However, for those seeking an in-depth exploration, Mastering Git by PACKT, *[3]* in the *Further reading* section, offers comprehensive insights into advanced Git concepts. It's an excellent resource for expanding your Git knowledge beyond the basics.

This section concludes our brief exploration of Git, marking the beginning of our journey into GitHub. The upcoming section will introduce and delve into GitHub, presenting its significance and utility in the world of GitOps.

Exploring GitHub

GitHub is a vital platform in the world of software development, serving as a hub for collaborative projects using Git. It extends the functionality of Git by providing a web-based graphical interface. GitHub facilitates the hosting of both public and private repositories, making it easier for individuals and teams to collaborate on projects regardless of their geographical location. Its importance lies not just in code storage but also in fostering a community where developers can share, contribute to, and build upon each other's work, enhancing the open source development culture.

Note

Creating a GitHub account is essential for storing source code on this widely used platform. This book will guide you through using GitHub to manage code in public and private repositories. Detailed instructions for account creation are available on GitHub's website, where you can find all the information needed to get started (`https://github.com/)`.

SSH access offers a secure way to interact with Git repositories, bypassing the need for entering a username and password with each command. It's particularly useful for cloning public projects without an account. For forking or pushing changes, however, an account is necessary. For detailed setup instructions, refer to GitHub's official SSH access guide (`https://docs.github.com/en/authentication/connecting-to-github-with-ssh`).

GitHub significantly enhances collaborative software development by enabling remote teamwork and streamlining processes such as bug tracking and continuous integration. Its pull request system bolsters collaboration, facilitating code review and merging. Moreover, GitHub serves as a comprehensive code hosting platform, supporting both public and private repositories, thus bolstering both open source and private projects. This accessibility and integration with development tools makes GitHub indispensable for developers worldwide.

GitHub's ecosystem

The GitHub ecosystem is a comprehensive and dynamic environment that extends beyond simple code hosting. It encompasses a wide range of tools, integrations, community features, and services that support the entire software development life cycle. This includes project management tools, issue tracking, continuous integration and deployment services, and code review facilities. Additionally, GitHub offers a vast community and network of developers contributing to open source projects, sharing insights, and collaborating on various initiatives. This ecosystem has become an integral part of the software development world, fostering innovation and collaboration among developers globally:

- **Version control**: At its core, GitHub offers Git-based version control for managing and tracking code changes
- **Issue tracking**: It includes tools for reporting and tracking bugs, feature requests, and tasks within projects
- **Collaboration features**: Features such as pull requests and code reviews facilitate collaborative coding and team interaction
- **GitHub actions**: For automation of workflows, such as **Continuous Integration and Continuous Deployment** (**CI/CD**)
- **Project management**: Tools for organizing and prioritizing work, such as Projects and Kanban boards
- **Community and networking**: A platform for developers to contribute to open source projects, network, and collaborate
- **Marketplace**: Offers third-party apps and integrations to extend GitHub functionality
- **GitHub Pages**: Provides hosting for static websites and documentation directly from a GitHub repository
- **Security Features**: Includes features for vulnerability scanning and managing access controls

Having gained an understanding of what GitHub is and how to access it, it's time to shift our focus toward practical applications.

GitHub enhances the software development process by integrating tools for better code quality, such as command line operations, issue discussions, pull requests, and code reviews, alongside a vast array of apps available in the GitHub Marketplace.

It fosters collaboration through the GitHub flow, enabling developers to efficiently manage work within repositories, outline project directions, and seamlessly merge updates after thorough discussion and agreement. It thereby transforms the conventional methodologies of software construction.

GitHub flow

The GitHub flow is a branch-based workflow integrating Git commands for global teams. It involves creating branches for parallel development, adding commits for reversible project points, and using pull requests to foster transparency (see the *Git workflow* section in this chapter). Code review is central, thereby promoting an open culture. Merging incorporates changes seamlessly, while deployment ensures robust code. This workflow adapts to various collaboration scales, from small teams to global contributions, emphasizing pull requests for discussion and integration. GitHub's tools replace traditional email reviews, streamlining the development process. We can summarize the GitHub Flow in the following steps:

1. Create a repository. The first thing to do is to create a repository in our GitHub space. In the upper corner of any page of each GitHub space, click on + and then on **New Repository**, as shown in *Figure 3.4*.

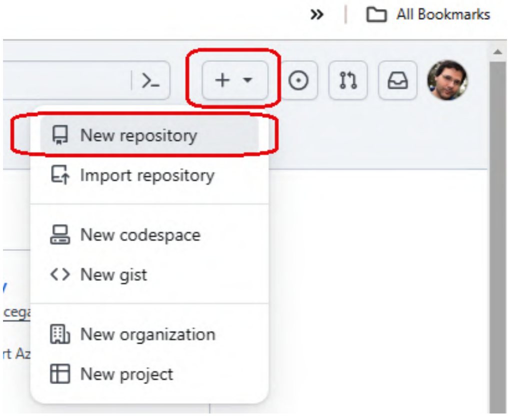
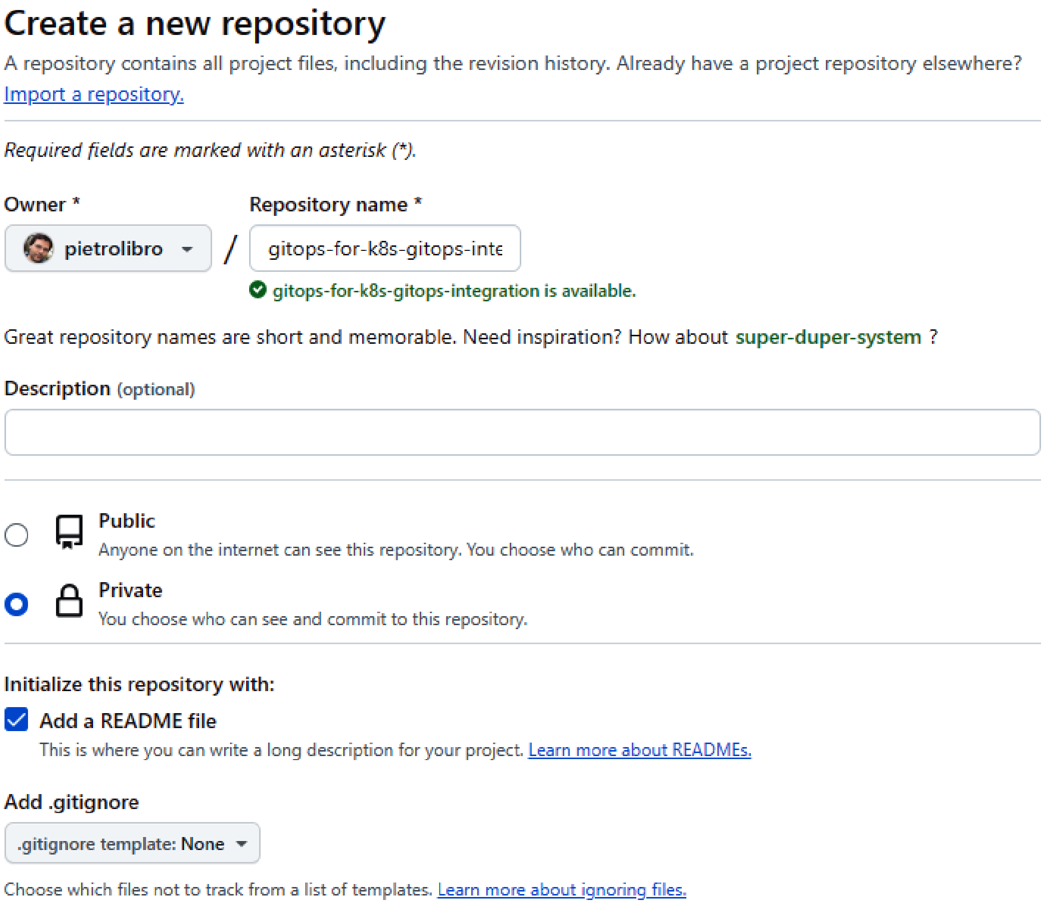


Figure 3.4 – Creating a new GitHub repository

For the repository name, you could choose something such as `gitops-for-k8s-gitops-integration`. **Description** is optional. Decide between a private or public repository based on your preference for open access or restricted visibility. Ensure that you check the **Add a README file** checkbox and then proceed by clicking the **Create repository** button, as illustrated in *Figure 3.5*.



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

Owner * pietrolibro / Repository name * gitops-for-k8s-gitops-inte

gitops-for-k8s-gitops-integration is available.

Great repository names are short and memorable. Need inspiration? How about **super-duper-system** ?

Description (optional)

○ **Public**
Anyone on the internet can see this repository. You choose who can commit.

● **Private**
You choose who can see and commit to this repository.

Initialize this repository with:

☑ Add a README file
This is where you can write a long description for your project. Learn more about READMEs.

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. Learn more about ignoring files.

Figure 3.5 – The form for creating a new repository on GitHub

2. Create a branch. By default, the created repository has one branch named `main` that is considered the one containing production-ready code. Best practices suggest using a different `feature` branch for each feature that we want to implement. To create a new branch in GitHub, click the **Code** tab of the repository that you created in the preceding step, then click the dropdown menu containing the `main` item, as shown in *Figure 3.6*. To create the new branch, click on **Create branch features/docker-file-and-ci**, as highlighted in *Figure 3.6*.

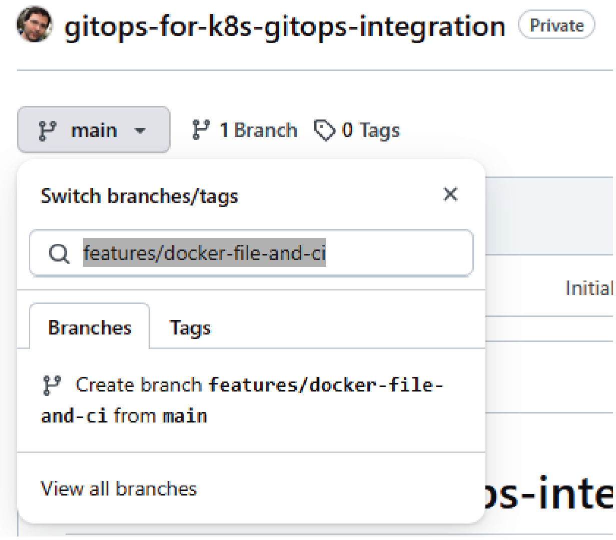


Figure 3.6 – Creating a new features/docker-file-and-ci feature branch

3. Add commits. It's time to add some useful files to our repository. For this exercise, we will use the same Docker file seen in the *Dockerfile* section of *Chapter 2* of this book. To add a Docker file, click the **Add file** button (see *Figure 3.7*). If you opt for **Create new file**, an editor will open where you can paste the Docker file contents and commit. Alternatively, selecting **Upload files** allows you to directly upload the Docker file from your computer.

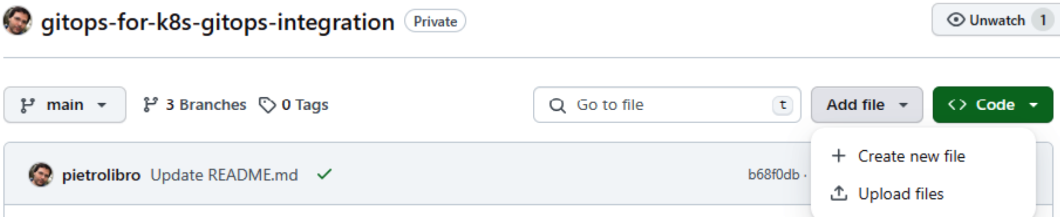
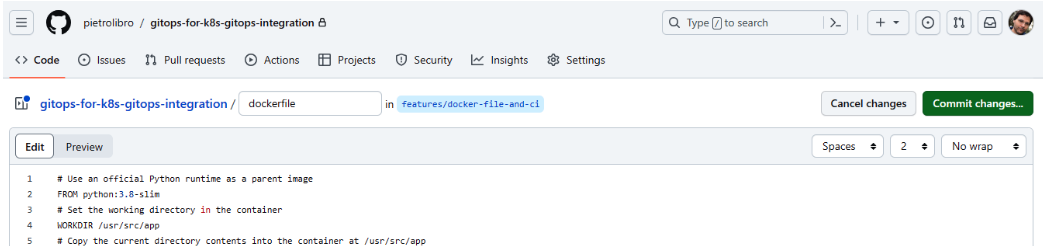


Figure 3.7 – Menu for adding a new file or uploading an existing one

4. Copy the content of the Docker file and call the file `dockerfile`, as illustrated in *Figure 3.8*. Click on **Commit changes** and accept the suggested commit message and options. Repeat the operations for the `requirements.txt` file used in the *Dockerfile* section of *Chapter 2*.



```
# Use an official Python runtime as a parent image
FROM python:3.8-slim
# Set the working directory in the container
WORKDIR /usr/src/app
# Copy the current directory contents into the container at /usr/src/app
```

Figure 3.8 – The GitHub editor interface for editing a repository file

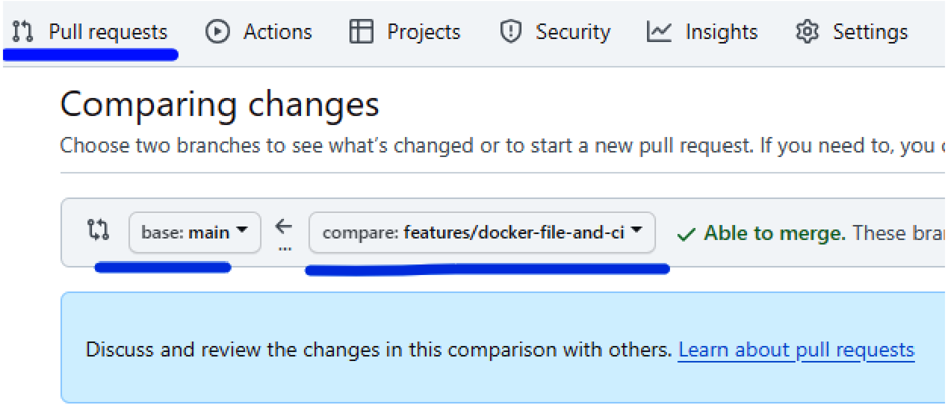
5. Adopt best practices for commit messages by specifying the action taken, such as **Add file** or **Remove API**. Use `Create dockerfile` as a suggested description and commit your changes. On the main page, select the **Edit File** icon for the `README` file and add the following line at the beginning, then commit the changes with the default description:

   ```
   ## This repository will be used to show GitOps and GitHub
   capabilities
   ```

   See *[2]* in the *Further reading* section for a useful list of commit best practices.

6. Open a pull request. After making changes in a branch off the `main` branch, you can initiate a pull request, which is a core element for collaboration in GitHub. This feature allows you to suggest changes, seek a review to merge your contributions into another branch, and highlight differences in color. You can start a discussion with a pull request immediately after committing, even if the code isn't final. Practicing this with a pull request in your repository by merging it yourself is an excellent way to familiarize yourself with GitHub's collaborative flow before moving on to bigger projects.

7. From the repository's tab, click on the **Pull requests** tab of our repository and click on **New Pull request**. In the **Example Comparison** box, select the **features/dockerfile-and-ci** branch to compare with the **main** branch as illustrated in *Figure 3.9*, then click on the **Create pull request** button.



Comparing changes

Choose two branches to see what's changed or to start a new pull request. If you need to, you

base: main ← compare: features/docker-file-and-ci ✓ Able to merge. These bra

Discuss and review the changes in this comparison with others. Learn about pull requests

Figure 3.9 – Creating a pull request and select which branches to compare

8. Review the changes in the comparison view at the bottom of the page. If they accurately reflect your intended modifications, proceed to create a pull request. Assign a title to your pull request and include a concise description of your changes, such as detailing the addition of a Docker file and the automation of Docker image builds with GitHub Actions. Feel free to enhance your description with emojis, images, or GIFs, and then finalize it by creating the pull request.
9. Discuss and review the code. Starting collaboration involves seeking reviews from your peers on pull requests. This step enables collaborators to offer feedback or suggest edits before finalizing changes to the `main` branch, ensuring a consensus-driven approach to code integration.
10. In the final step, merge your `features/dockerfile-and-ci` branch into the `main` branch to add your updates. Should any conflicts between your pull request and the `main` branch emerge, GitHub will prompt you to resolve them prior to merging. Assuming that there are no conflicts, the merge should be straightforward. To complete the process, select **Merge pull request** at the bottom of the pull request page. Confirm the merge. A notification that mentions a successful merge and request closure will appear. Lastly, delete the branch for cleanup and return to the repository's **Code** tab to view your changes in the `main` branch.

We've introduced the fundamentals of GitHub and GitFlow alongside a hands-on example. We encourage you to practice using Git command line tools for cloning repositories, as outlined in the *Git repository* section of this chapter. Armed with this essential knowledge, we're poised to delve into integrating GitOps with GitHub, specifically for automating Docker deployments with commits.

Integrating GitOps and GitHub

GitHub Actions offers a powerful automation tool within the GitHub ecosystem, enabling developers to streamline their software development workflows. Specifically, a GitHub Action can significantly enhance a GitOps workflow by automating the building and pushing of Docker images for services, followed by the deployment of new versions to Kubernetes clusters. This capability simplifies the CI/CD processes, making it easier for teams to maintain and update their applications efficiently in a cloud-native environment.

Best practices suggest that when you want to use this GitHub Action, your GitHub repository should have a `dev` and a `master` or `main` branch, and that it should use tags for releases. In the following example, for brevity, we will work directly in the `main` branch. Best practices suggest working in separate branches.

Here are the steps to automate Docker image creation with GitHub Actions:

1. Click on **Actions** in the repository's tab, as shown in *Figure 3.10*:

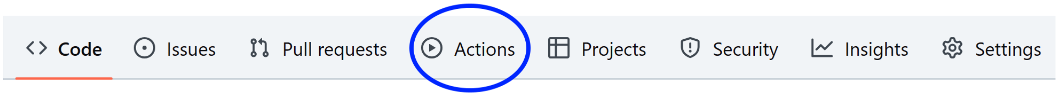
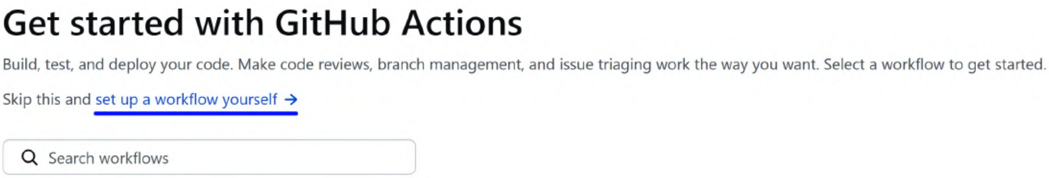


Figure 3.10 – The tab menu with the Actions button circled

2. On the new page, click on **set up a workflow yourself**, as shown in *Figure 3.11*:



Get started with GitHub Actions

Build, test, and deploy your code. Make code reviews, branch management, and issue triaging work the way you want. Select a workflow to get started.

Skip this and set up a workflow yourself →

Figure 3.11 – Manually defining a workflow in GitHub

3. In the editor, add the following content:

```
name: Create Docker Image CI
on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
```

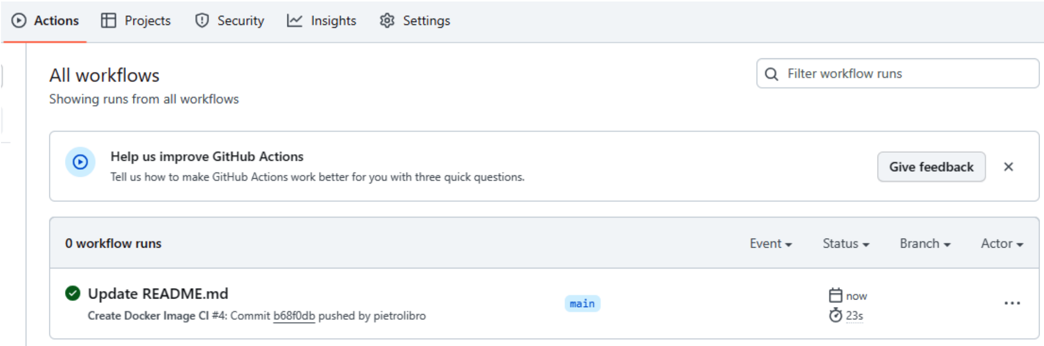
```
      - uses: actions/checkout@v4.1.1
      - name: Build the Docker image
        run: docker build . --file dockerfile --tag my-image-
name:$(date +%s)
```

The `name` field serves as the identifier for our action. In the `on` section, we specify the events that will trigger this action, primarily focusing on specific branches. This is usually set to `main`. If you happen to create a new branch and want to trigger this action, you'll need to modify this section accordingly.

Within the `jobs` | `build` section, you'll find the `runs-on` field, which specifies the type of GitHub servers that our job will run on. This is distinct from the version of our Docker image, which is determined by `Dockerfile`. In this case, our Docker image will be based on Ubuntu 20.04. The Docker image build will take place on GitHub servers running **ubuntu-latest**.

Now, let's move on to the action steps. The `uses` field instructs the workflow to perform a checkout of our repository. The `name` field provides a label for the step. Finally, the `run` field specifies the command to execute. With this configuration, we are prepared for our first build on the Ubuntu 20.04-based Docker image.

4. To initiate a new build, we need to introduce a modification to our `main` branch. You can easily accomplish this by making a simple comment change to the Docker file within the integrated editor on GitHub or editing the `README` file. In *Figure 3.12*, the build is triggered from an update to `README`.



All workflows
Showing runs from all workflows

Help us improve GitHub Actions
Tell us how to make GitHub Actions work better for you with three quick questions.

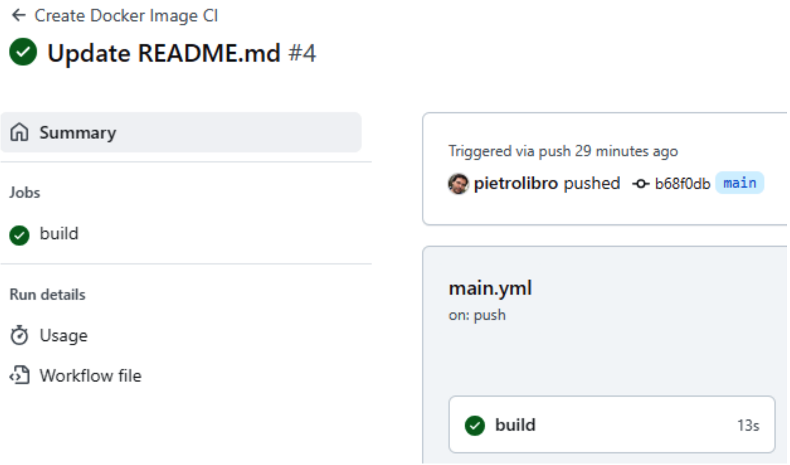
0 workflow runs

Update README.md
Create Docker Image CI #4: Commit b68f0db pushed by pietrolibro
main
now
23s

Figure 3.12 – A list of workflows triggered by commits on the Actions page

Once you've made the change in your repository, GitHub will automatically detect it and initiate a new build. You'll notice an indicator signaling that the build has been triggered.

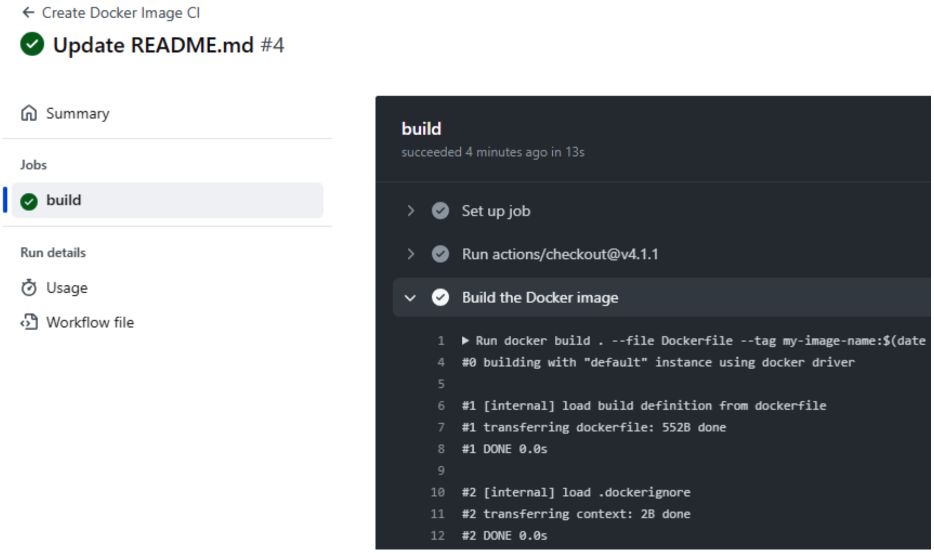
5. To access the build logs, simply click on **build**, as illustrated in *Figure 3.13*. This will lead you to the dedicated page for that build. Once on this page, you can delve into the individual steps of the build process and review the logs associated with each step, as illustrated in *Figure 3.14*. GitHub also provides email notifications in case of build failures, and you can conveniently configure your alerting preferences within the GitHub platform.



Triggered via push 29 minutes ago

pietrolibro pushed b68f0db main

Figure 3.13 – A summary of a successful build



succeeded 4 minutes ago in 13s

Set up job

Run actions/checkout@v4.1.1

Build the Docker image

```
▶ Run docker build . --file Dockerfile --tag my-image-name:$(date
#0 building with "default" instance using docker driver

#1 [internal] load build definition from dockerfile
#1 transferring dockerfile: 552B done
#1 DONE 0.0s

#2 [internal] load .dockerignore
#2 transferring context: 2B done
#2 DONE 0.0s
```

Figure 3.14 – Detailed steps of the build process with logs for each step

At this stage, you should have a comprehensive understanding of Git, GitFlow, GitHub, and how to develop a CI process that builds a Docker image for each commit.

Summary

This chapter delved into the essential role of Git for version control and GitHub for collaborative software development within the GitOps framework. It outlined the significance of these tools in enhancing team efficiency, automating deployment processes, and fostering a robust DevOps culture. Key areas covered include the technical requirements for engaging with Git and GitHub, the benefits of VCSs, the functionalities of Git and GitHub, and the integration of GitOps practices. The chapter concluded with some best practices for utilizing Git and GitHub to streamline workflow efficiency.

In the upcoming chapters, we will explore additional tools such as Argo CD and Flux CD, along with their integration into GitOps, enabling us to complete our pipeline with the deployment segment.

Further reading

- [1] `https://git-scm.com/book/en/v2/Getting-Started-Installing-Git`
- [2] `https://gist.github.com/luismts/495d982e8c5b1a0ced4a57cf3d93cf60`
- [3] `https://www.packtpub.com/product/mastering-git/9781783553754`

4

Kubernetes with GitOps Tools

Transitioning from the theoretical underpinnings of GitOps presented in the initial chapters, this chapter delves into the practical interplay between Kubernetes and GitOps tools, emphasizing their significance in modern cloud-native operations. Earlier discussions set the scene by introducing GitOps, examining its cloud-native applications, and integrating version control via Git and GitHub.

Here, we'll narrow our focus to Kubernetes, engaging with prominent tools such as Helm, Kustomize, Argo CD, and Flux CD to deepen our understanding and application insights.

This chapter aims to build on previous insights, providing a comparative look at pivotal GitOps instruments while underscoring their distinct attributes and synergy with Kubernetes. By dissecting the functionalities and applications of these tools, you'll be equipped with the acumen to make informed choices, align your GitOps strategies with specific needs, and enhance Kubernetes deployment frameworks.

In this chapter, we'll focus on the following key areas:

- Overview of popular GitOps tools
- A deep dive into Helm and Kustomize
- Argo CD integration with Kubernetes
- Flux integration with Kubernetes
- Comparing Argo CD and Flux

Technical requirements

To fully grasp and apply the content of this chapter, you must meet certain technical prerequisites. A foundational understanding of Kubernetes architecture and concepts, along with proficiency in using command-line interfaces, is essential. It is recommended that you also have prior experience with the GitOps tools discussed in previous chapters, such as Helm, Kustomize, Argo CD, and Flux. Access to a computer with internet connectivity is crucial for executing hands-on examples and integrating GitOps tools with Kubernetes clusters.

The relevant code and resource files for this chapter can be found in the `Chapter04` folder of this book's GitHub repository: `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`.

Overview of popular GitOps tools

In this section, we'll delve into a curated selection of pivotal tools that epitomize the GitOps methodology, each serving a unique role in orchestrating, managing, and deploying infrastructure and applications. This includes **Terraform** for IaC, **Argo CD** and **Flux** for continuous delivery within Kubernetes environments, and **Helm** and **Kustomize** for package management and application configuration. These tools collectively facilitate a more streamlined, secure, and scalable approach to system management, embodying the core principles of GitOps. By understanding the functionality and strategic application of each tool, you can leverage the full potential of GitOps to enhance operational efficiency and deployment precision in cloud-native environments:

- **Terraform**: Terraform (`https://www.terraform.io/`) is an open source IaC software tool created by HashiCorp. It allows users to define and provision data center infrastructure using a high-level configuration language known as **HashiCorp Configuration Language** (**HCL**), or optionally JSON. Terraform manages external resources (such as public cloud infrastructure, private cloud infrastructure, network appliances, **Software-as-a-Service** (**SaaS**), and more) with a "declarative" approach, proposing a model for what the intended end state should look like rather than the steps to get there. This approach enables developers and infrastructure teams to collaborate and manage the IT infrastructure through code, ensuring consistency and accountability.
- **Argo CD**: Argo CD (`https://argo-cd.readthedocs.io/en/stable/`) is a declarative, GitOps CD tool for Kubernetes. It automates the process of deploying applications to specified target environments within Kubernetes clusters, ensuring that the live state is always in sync with the configurations stored in Git repositories. Argo CD follows GitOps' principles, treating Git repositories as the source of truth for defining the desired application state, thereby simplifying the deployment process and enhancing security and traceability. It provides a graphical user interface for visualizing the status of applications, their version control history, and the differences between the deployed and desired states.
- **Flux**: Flux (`https://fluxcd.io/`) is another tool that embodies GitOps' principles and is designed to automate the process of deploying to Kubernetes. It ensures that the state of a cluster matches the configurations stored in a Git repository. Flux continuously monitors the repository and automatically applies any new changes to the target environment. This enables developers to manage and deploy their applications by simply updating their Git repositories. Flux supports complex deployment strategies, including canary and blue-green deployments, and integrates with Kubernetes to ensure that the actual state of live systems is always represented accurately in code.

- **Helm**: Helm (`https://helm.sh/`), a Kubernetes package manager, streamlines the deployment process by enabling developers and operators to effortlessly package, configure, and deploy applications and services on Kubernetes clusters. It employs a packaging format known as charts – collections of files that detail a set of related Kubernetes resources. These charts allow you to define, install, and upgrade complex Kubernetes applications, making updates and rollbacks simpler and thus enhancing the deployment procedure.
- **Kustomize**: Kustomize (`https://kustomize.io/`) introduces a template-free way to customize application configurations that can be applied across different environments. It is a standalone tool for customizing Kubernetes objects through a file called `kustomization.yaml`, allowing resource files, patches, and other settings to be declared without altering the original YAML files. This approach simplifies how distinct configurations for each environment (development, staging, production, and so on) can be managed without requiring multiple copies of the resource files.

In the upcoming sections, we will delve into each tool while covering practical examples.

A deep dive into Helm and Kustomize

This section delves into the nuanced world of Helm and Kustomize, aiming to provide an in-depth exploration of these powerful Kubernetes tools. By guiding you through the setup processes, practical project examples, and the integration of these tools with GitOps workflows, we endeavor to impart a solid foundation and advanced insights for effectively managing and deploying applications in cloud-native environments.

Helm

Within the sphere of managing resources for Kubernetes, Helm stands out as a pivotal instrument that's designed to enhance and simplify the orchestration process, setting itself apart from similar tools. As a blend between a management utility and a templating engine, Helm generates artifacts that can not only be deployed but also shared and capable of version control, establishing itself as a comprehensive package manager for Kubernetes environments.

By leveraging the **Go** template language in YAML configurations, Helm streamlines the setup and administration of Kubernetes applications. Helm starts by generating and deploying a Helm project into a Kubernetes cluster. A distinctive feature of Helm is its utilization of **charts**, setting it apart from tools such as Kustomize. These charts, acting as bundled artifacts, compile all necessary components, including any dependencies, facilitating their widespread distribution and reusability.

Furthermore, Helm tackles the challenges associated with the mutable nature of application settings, particularly within Kubernetes **ConfigMaps**. It recognizes that mere alterations to ConfigMaps may not trigger automatic updates to applications. To combat this, Helm introduces methods to activate rolling updates following changes in ConfigMaps, thus ensuring continuous application performance and enhanced management efficiency. This detail-oriented approach highlights the effectiveness of Helm when it comes to governance and updating Kubernetes deployments.

ConfigMaps

ConfigMaps in Kubernetes serves as a mechanism to store non-confidential data in key-value pairs, which can then be utilized by Pods and other system components. They are designed to separate configuration artifacts from image content to keep containerized applications portable. This enables users to change the configuration without having to rebuild the Docker images, and without exposing sensitive data when the configuration data is not encrypted. ConfigMaps can be used to store fine-grained values such as individual properties or coarse-grained information such as entire configuration files or JSON blobs. This functionality is particularly useful for applications that need to be configured at runtime, making it easier to deploy the same application in different environments without changing the application code.

Adding to its capabilities, Helm is recognized as an optimal package manager for Kubernetes since it aligns with GitOps methodologies. It serves as an efficient method for bundling and distributing collections of YAML files through Helm charts and enables their dissemination via a Helm repository. This streamlines the management process, offering a solution to the otherwise labor-intensive and monotonous task of handling Kubernetes manifests, illustrating why Helm is considered an invaluable tool in the Kubernetes ecosystem.

Helm setup

Setting up Helm is generally an easy task. Typically, it involves fetching a script and executing it, as shown here:

```
$ curl -fsSL -o get_helm.sh https://raw.githubusercontent.com/helm/
helm/main/scripts/get-helm-3
$ chmod 700 get_helm.sh
$ ./get_helm.sh
```

Alternatively, you can execute the following command:

```
https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-3 |
bash
```

A detailed step-by-step guide on how to install Helm is beyond the scope of this chapter. More information about Helm and instructions for its installation can be found here: `https://helm.sh/docs/intro/install/`.

Type the following command to verify that Helm has been installed correctly:

```
$ helm version
```

The expected output should look like this:

```
version.BuildInfo{Version:"v3.14.2",
GitCommit:"c309b6f0ff63856811846ce18f3bdc93d2b4d54b",
GitTreeState:"clean", GoVersion:"go1.21.7"}
```

Congratulations! Helm is correctly installed and you are ready to create your first Helm project.

Our first Helm project

First, we need to create a new **Helm chart** (or just **chart**). Helm charts are packages composed of YAML files that specify the resources needed to run an application, tool, or service inside a Kubernetes cluster. To create the Helm project, we need to enter the following command:

```
$ helm create gitops-k8s-deployments-helm
```

This command creates a new directory called `gitops-k8s-deployments-helm` containing the scaffold of the project. With this Helm's directory layout has been created. It should look like this:

```
chart.yaml  charts  templates  values.yaml
```

Let's take a closer look:

- `chart.yaml`: This is the descriptor and contains metadata related to the Helm chart
- `templates`: This directory contains all the template files that are used for installing the Helm chart. Additionally, the `tests` folder within the `templates` directory of a Helm chart repository contains test files that Helm uses to verify the chart.
- `values.yaml`: This contains the default values for the current Helm chart

Customization of the **Helm chart** passes through the edits of the files under the `templates` subdirectory. The content of the `deployment.yaml` file is available in this book's GitHub repository. At this point, we are ready to view a local preview of the Helm chart. We can do this by entering the following command:

```
$ helm template .\gitops-k8s-deployments-helm\
```

The following is an extract of the expected output:

```
---
# Source: gitops-k8s-deployments-helm/templates/service.yaml
apiVersion: v1
kind: Service
```

```
metadata:
  name: release-name-gitops-k8s-deployments-helm
  labels:
    helm.sh/chart: gitops-k8s-deployments-helm-0.1.0
    app.kubernetes.io/name: gitops-k8s-deployments-helm
    app.kubernetes.io/instance: release-name
    app.kubernetes.io/version: "1.16.0"
    app.kubernetes.io/managed-by: Helm
spec:
  type: ClusterIP
  ports:
    - port: 80
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/name: gitops-k8s-deployments-helm
    app.kubernetes.io/instance: release-name
---
```

Here, the values for `name`, `type`, `port`, and others are taken from the `values.yaml` file. If possible, override them using the `--set` parameter in the Helm command. For instance, we can override the port value to `8080` like so:

```
$ helm template --set service.port=8080 .
```

Using the local Kubernetes cluster, `up&running` (see *Chapter 2*), we are now able to install our chart and upgrade it. Instead of using the default namespaces, we want to use the `gitops-kubernetes` one:

```
$ helm install gitops-k8s-deployments-helm --namespace gitops-
kubernetes .
```

The output of this command should be as follows:

```
NAME: gitops-k8s-deployments-helm
LAST DEPLOYED: Sat Mar 2 12:23:21 2024
NAMESPACE: gitops-kubernetes
STATUS: deployed
REVISION: 1
NOTES:
…
```

We can list the installed Helm charts using the `helm list` command:

```
$ helm list --namespace gitops-kubernetes
```

Here's the output:

```
NAME                            NAMESPACE
REVISION        UPDATED
STATUS          CHART
APP VERSION
gitops-k8s-deployments-helm     gitops-kubernetes
1               2024-03-02 12:23:21.29488893 +0100 CET
deployed        gitops-k8s-deployments-helm-0.1.0       1.16.0
```

We can retrieve the running Pods with the following command:

```
$ kubectl get pods --namespace gitops-kubernetes
```

Here's the expected output:

```
NAME                                                READY
STATUS    RESTARTS       AGE
first-cd-pipeline-deployment-5b85cfd665-wlzq4   1/1
Running   3 (106m ago)   48d
gitops-k8s-deployments-helm-7464b9b75d-mh4xx    1/1
Running   0              4m41s
```

We can grab history information about the installed Helm chart using the `history` command:

```
$ helm history gitops-k8s-deployments-helm --namespace gitops-
kubernetes
```

At this point, we can imagine upgrading the image (or just the tag) of the images used by our deployment from `nginx` to `nginxdemos/hello`, as specified by the `values.yaml` file:

```
image:
  repository: nginxdemos/hello
```

We need to change the `appVersion` value in the `chart.yaml` file (for instance, from `1.16.0` to `2.0.0`):

```
appVersion: "2.0.0"
```

At this point, we can upgrade our deployment using the `helm upgrade` command:

```
$ helm upgrade gitops-k8s-deployments-helm --namespace gitops-
kubernetes .
```

We can grab useful information with the `history` command:

```
1               Sat Mar  2 12:23:21 2024        superseded
gitops-k8s-deployments-helm-0.1.0       1.16.0          Install
complete
2               Sat Mar  2 12:37:42 2024        deployed
gitops-k8s-deployments-helm-0.2.0       2.0.0           Upgrade
complete
```

To remove the deployment, we can use the `helm uninstall` command:

```
$ helm uninstall gitops-k8s-deployments-helm --namespace gitops-
kubernetes
```

Integrating Helm charts with GitOps

We can effortlessly integrate Helm charts with Git and GitOps practices to establish a single source of truth for our application's desired state. This integration is fundamental for modern DevOps workflows as it leverages the versioning capabilities of Git alongside the automated, declarative nature of GitOps. Here's an example:

```
git init
git add .
git commit -m "Initial commit of gitops-k8s-deployments-helm."
git remote add origin <your-repository-url>
git push -u origin master
```

Now that we have a good understanding of Helm and how it works, it's time to move on to the next section, which is dedicated to another important tool: Kustomize.

Kustomize

Kustomize is a standalone tool that's designed to customize Kubernetes configurations. It's part of Kubernetes itself and was introduced as a feature within **kubectl** in version 1.14. Kustomize introduces a **template-free** way to customize application configuration. This differs significantly from other configuration management tools, which rely on templates or scripting. It enables users to declare configuration changes through files that Kubernetes can understand without altering the original YAML files. This method avoids the pitfalls of templating languages and maintains the declarative nature of Kubernetes objects.

Kustomize works by using a file called `kustomization.yaml`, which contains customization instructions. These instructions can include adding labels and annotations, changing the number of replicas, altering container images, and more. Users can build an overlay structure with Kustomize, where a base configuration can serve as a foundation, and overlays can modify this base for specific environments, such as development, staging, or production.

In a GitOps workflow, Kustomize helps manage and apply different configurations for different environments by maintaining them in separate directories within a Git repository. Kustomize enables your team to update base files for core components without disrupting customizations specific to your use case. Additionally, using patch overlays offers the advantage of adding layers to your configuration settings. This allows for easier isolation when troubleshooting or layering configurations from the most general to the most specific. For example, *Figure 4.1* illustrates how to integrate Kustomize and Helm within a **continuous deployment** (**CD**) pipeline.

Kustomize uses a layered approach to configuration management to facilitate reuse:

- **Base layer**: Defines the most widely used resources
- **Patch layers**: Tailors resources to specific use cases:

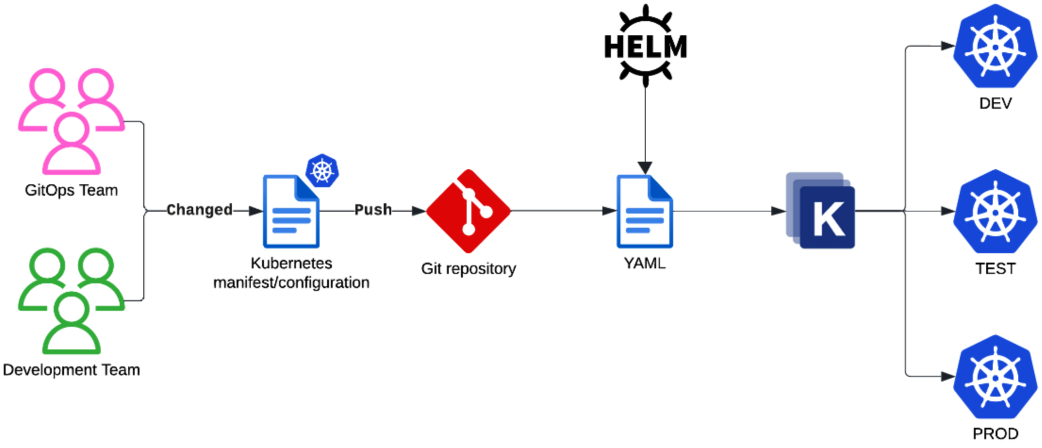


Figure 4.1 – Example of Kustomize and Helm placed in a CD pipeline

When changes are committed to the repository, a GitOps tool such as **Argo CD** or **Flux**, both of which will be covered in the *Argo CD integration with Kubernetes* and *Flux integration with Kubernetes* sections of this chapter, respectively, can automatically apply the changes to the appropriate Kubernetes clusters based on the Kustomize configurations. This approach enhances automation, improves traceability, and ensures consistency across environments.

Kustomize setup

As previously mentioned for Helm, setting up Kustomize is an easy task. Execute the following script to detect the corresponding operating system and install the necessary libraries:

```
$ curl -s "https://raw.githubusercontent.com/kubernetes-sigs/
kustomize/master/hack/install_kustomize.sh" | bash
```

For a detailed step-by-step guide on installing Kustomize, please refer to the Kustomize section at `Kustomize | SIG CLI (kubernetes.io)`.

As a standard step to verify the correctness of the Kustomize setup, you can type the following command in a Terminal:

```
$ kustomize version
```

The expected output should resemble the following:

```
v5.3.0
```

A Kustomize example for Kubernetes deployment

Kustomize is a tool for customizing Kubernetes configurations. It lets you maintain configuration changes separately from the base configuration files, which is particularly useful for keeping track of environment-specific changes without duplication. Let's dive into a step-by-step example of deploying an application on Kubernetes using GitOps and Kustomize, bringing all the elements together for a cohesive process:

1. The first step is creating the base configuration where we will add the default Kubernetes resource files:

   ```
   $ mkdir gitops-k8s-deployments-kustomize/base
   ```

2. Inside the `base` directory, we need to create a Kubernetes deployment file named `deployment.yaml`:

   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: gitops-k8s-kustomize
     labels:
       app: gitops-k8s-kustomize
   spec:
     replicas: 1
     selector:
       matchLabels:
         app: gitops-k8s-kustomize
     template:
       metadata:
         labels:
           app: gitops-k8s-kustomize
   ```

```
    spec:
      containers:
      - name: gitops-k8s-kustomize
        image: k8s.gcr.io/echoserver:1.10
        ports:
        - containerPort: 8080
```

3. In the same base directory, we need to create another Kubernetes service file named `service.yaml`:

```
apiVersion: v1
kind: Service
metadata:
  name: gitops-k8s-kustomize
spec:
  selector:
    app: gitops-k8s-kustomize
  ports:
    - protocol: TCP
      port: 80
      targetPort: 8080
```

4. Lastly, within the base directory, create a `kustomization.yaml` file that references your Kubernetes resources:

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

5. Now, we need to create the overlay configuration so that we can maintain `patches` separately from the base configuration. You might have different environments, such as `development`, `staging`, and `production`. For instance, we can create an overlay directory and subdirectory for the development environment like so:

```
$ mkdir -p overlays/development
```

6. Inside the `overlays/development` directory, create a `kustomization.yaml` file to specify `namespace`, `patches`, and any other environment-specific changes:

```
namespace: gitops-k8s-kustomize-dev
resources:
  - ../../base
patches:
  - path: ./patches/deployment_patch.yaml
```

7. Also, in the `overlays/development` directory, inside the `patches` subdirectory, create a `deployment_patch.yaml` file to update the image or any other properties specific to the development environment. In this case, we are increasing the number of replicas in development:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gitops-k8s-kustomize
  labels:
    environment: development
spec:
  replicas: 2  # Increase the number of replicas in development.
```

8. At this point, navigate to your overlay directory and use Kustomize to build the final configuration. Then, you can apply it to your Kubernetes cluster. To build the overlay configuration, run the following command:

```
$ kustomize build overlays/development
```

9. Before applying the configuration to our local cluster, we need to create the proper namespace:

```
$ kubectl create namespace gitops-k8s-kustomize-dev
```

10. At this point, we can apply the configuration by entering the proper command:

```
$ kustomize build overlays/development | kubectl apply -f -
```

11. The expectation is to have three replicas running in the specified namespace. We can confirm this by using the following command:

```
$ kubectl get pods --namespace gitops-k8s-kustomize-dev
```

This will produce an output similar to the following:

```
NAME                         READY   STATUS    RESTARTS   AGE
gitops-k8s-kustomize-…       1/1     Running   0          114s
gitops-k8s-kustomize-…       1/1     Running   0          114s
```

12. If we want to deploy to a `staging` namespace using three replicas, we need to create an `overlays/staging` folder and add a `kustomization.yaml` file with the following content:

```
namespace: gitops-k8s-kustomize-stg
resources:
  - ../../base
patches:
  - path: ./patches/deployment_patch.yaml
```

13. The content of the `deployment_path.yaml` file will be as follows:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gitops-k8s-kustomize
  labels:
    environment: staging
spec:
  replicas: 3  # Increase the number of replicas in staging.
```

14. As before, to apply the configuration to our local cluster, we need to create the proper namespace:

```
$ kubectl create namespace gitops-k8s-kustomize-stg
```

15. At this point, we can apply the configuration by entering the proper command:

```
$ kustomize build overlays/staging | kubectl apply -f -
```

16. The expectation is to have three replicas running in the specified namespace. We can confirm that using the following command:

```
$ kubectl get pods --namespace gitops-k8s-kustomize-stg
```

This will produce an output similar to the following:

```
NAME                         READY   STATUS    RESTARTS   AGE
gitops-k8s-kustomize-…       1/1     Running   0          14s
gitops-k8s-kustomize-…       1/1     Running   0          14s
gitops-k8s-kustomize-…       1/1     Running   0          14s
```

Integrating Kustomize with GitOps

As we did for Helm charts, we can effortlessly integrate Kustomize with Git and GitOps practices to establish a single source of truth for our application's desired state. This integration is fundamental for modern DevOps workflows as it leverages the versioning capabilities of Git alongside the automated, declarative nature of GitOps:

```
git init
git add .
git commit -m "Initial commit of gitops-k8s-deployments-kustomize."
git remote add origin <your-repository-url>
git push -u origin master
```

Helm versus Kustomize – configuration tools compared

Kustomize and Helm are both powerful tools that can be used for managing Kubernetes configurations, but they approach configuration management differently and serve slightly different purposes within the Kubernetes ecosystem.

Helm is often described as a package manager for Kubernetes. It uses a packaging format called charts, which are collections of files that describe a related set of Kubernetes resources. Helm charts are templates that can be dynamically configured through the use of values that are passed at runtime. This feature allows complex applications and their dependencies to be deployed using a single command, making Helm particularly suitable for managing packaged applications and sharing them across a wide user base. Helm also supports versioning, which means you can manage and track releases of your applications, roll back to previous versions, and manage application deployment in a structured manner.

Kustomize, on the other hand, introduces a template-free way of customizing Kubernetes applications. It uses a base and overlay model where the base configuration can be overridden or extended by overlays without altering the original files. This approach is particularly powerful for maintaining slight variations of the same application (for example, across different environments, such as staging and production) without duplicating effort. Kustomize has been integrated directly into kubectl since Kubernetes v1.14, allowing users to apply configurations using the `kubectl apply -k` command. Unlike Helm, Kustomize does not manage packages or dependencies; it focuses purely on configuration customization.

In summary, while both Helm and Kustomize enhance Kubernetes' native capabilities, they cater to different aspects of application deployment and management. Helm is ideal for defining, installing, and upgrading complex Kubernetes applications as discrete packages, while Kustomize excels at customizing and managing the configurations of these applications across various environments without the need to maintain separate files for each environment. Choosing between them – or using them in tandem – depends on your specific needs and the complexity of your deployment scenarios.

Having explored Helm and Kustomize, we'll now transition to discussing two other significant tools in the DevOps landscape: Argo CD and Flux.

Argo CD integration with Kubernetes

Argo CD is a prominent GitOps tool that's designed for Kubernetes and enables automated, CD by syncing application definitions, configurations, and environments directly from a Git repository. This section aims to shed more light on the architecture of Argo CD and its synchronization mechanism. Argo CD follows a client-server architecture:

- **Argo CD API server**: The core service that provides the primary API for Argo CD, it handles user requests and performs application state analyses and reporting.

- **Repository server**: This service is responsible for fetching repository contents and returning manifest files. It understands various configuration management tools, such as Helm, Kustomize, Jsonnet, and others, enabling Argo CD to work with multiple types of manifests.
- **Application controller**: This is the key component that continuously monitors application definitions and configurations in the Git repository. It compares the current state of the cluster with the desired state defined in Git and applies any necessary changes to align the two.
- **Argo CD UI and CLI**: The services provide a visual interface and command-line tooling for users to manage applications, view their states, and perform various operations.
- **Dex server**: An optional component for integrating with external identity providers for authentication.

Argo CD employs a declarative approach to ensure the desired application state defined in a Git repository matches the actual state deployed in the Kubernetes cluster. Here's how the synchronization process works:

- **Monitoring**: Argo CD continuously monitors the specified Git repository for changes to the application definitions and configurations.
- **Comparison**: When changes are detected, Argo CD compares the new desired state defined in Git with the current state of the Kubernetes cluster.
- **Convergence**: If discrepancies are found, Argo CD undertakes actions to reconcile the differences. This could involve creating, updating, or deleting Kubernetes resources to ensure the cluster's state aligns with the configuration specified in the Git repository.
- **Self-healing**: If the cluster state deviates from the Git-defined state due to manual changes or other factors, Argo CD automatically corrects these deviations to maintain consistency.
- **Manual or automatic sync**: Users can configure Argo CD to synchronize changes either automatically as they occur in the Git repository or require manual intervention for synchronization, providing flexibility based on the team's workflow and policies.

Argo CD setup

Setting up and configuring Argo CD is beyond the scope of this book. For comprehensive and detailed instructions on how to set up Argo CD, we highly recommend visiting the Argo CD website at `https://argo-cd.readthedocs.io/en/stable/getting_started/`. There, you will find the most up-to-date documentation, including step-by-step guides, troubleshooting tips, and best practices specifically designed to facilitate a smooth setup process. For the remainder of this chapter, we will assume that Argo CD has been successfully installed and configured.

If Argo CD has been correctly set up on your local laptop, should be possible to navigate the Argo CD UI, which is normally available at `https://localhost:8080`, as shown in *Figure 4.2*:

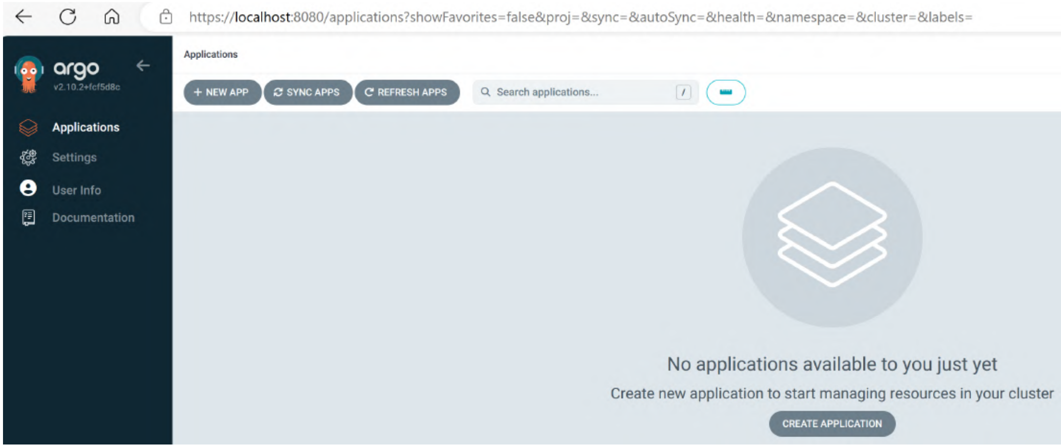


Figure 4.2 – Argo CD UI

Deploying to Kubernetes with Argo CD

When deploying to Kubernetes with Argo CD, a crucial step is to create an Argo CD application, which is fundamental to the GitOps-driven deployment strategy. This object connects the version-controlled repository, housing the Kubernetes manifests, to the actual state of applications operating within the Kubernetes cluster, whether local or remote. By establishing an Argo CD application, we dictate to Argo CD what to deploy, the deployment target, and the method of synchronizing the live state with the desired state outlined in the Git repository. To demonstrate the effectiveness of this deployment approach, we will introduce a straightforward application, as detailed in the upcoming section.

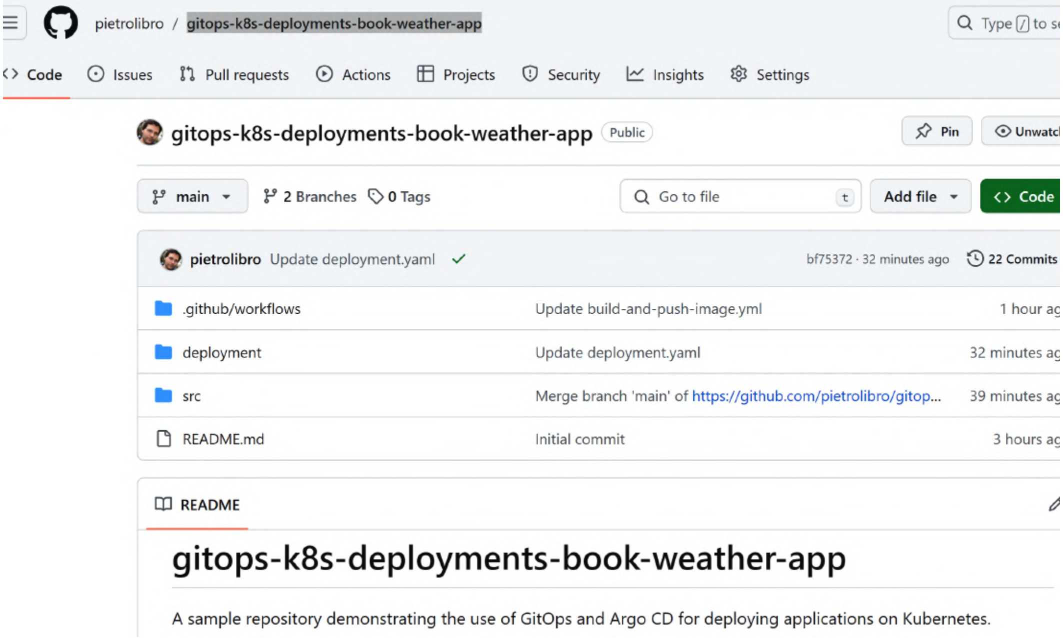
The My City Weather app

The weather app we are deploying provides simulated temperature forecasts and weather updates while utilizing a CSV file as its data source. By deploying on Kubernetes with Argo CD, we can adopt GitOps principles for streamlined, error-free updates, ensuring that our application consistently aligns with the most recent configurations and code changes maintained in the Git repository.

Follow these steps to deploy the weather app on Kubernetes via Argo CD:

1. **Create a new GitHub repository**: The first step is to create a new GitHub repository or use an existing one. I suggest that you start from scratch with a new one, as detailed in the *GitHub Flow* section in *Chapter 3*. In the upcoming section, we will refer to the `gitops-k8s-deployments-book-weather-app` repository.
2. **Populate the GitHub repository**: The second step is populating the repository we created in *Step 1* with the files located in the `Chapter_4/argocd_gitops` directory of the repository accompanying this book.

The outcome should resemble what's depicted in *Figure 4.3*. The `src` folder will contain the necessary files to construct a chart, which will display temperatures retrieved from a CSV file that is deployed alongside the application to the end user. The deployment folder will contain the Kubernetes manifest files (such as `deployment.yaml` and `service.yaml`) that are used to ensure synchronization between the Argo CD application's state and the repository's state. Additionally, the `.github/workflows` directory will house the definitions of the CI/CD pipeline used by GitHub Actions. This pipeline is responsible for building and pushing a new Docker image to a container registry each time changes are committed to the GitHub repository, thereby automating the deployment process and ensuring that updates are seamlessly reflected in the live application. To complete this section, you need to have a valid Docker Hub account, as mentioned in the *Publishing the image to a container registry* section in *Chapter 2*:



pietrolibro Update deployment.yaml ✓ bf75372 · 32 minutes ago 22 Commits

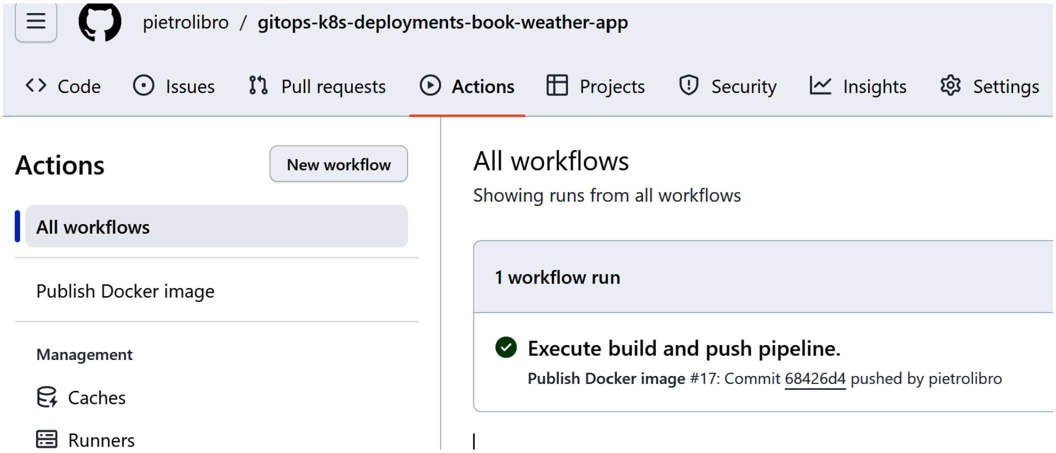
.github/workflows	Update build-and-push-image.yml	1 hour ag
deployment	Update deployment.yaml	32 minutes ag
src	Merge branch 'main' of https://github.com/pietrolibro/gitop...	39 minutes ag
README.md	Initial commit	3 hours ag

gitops-k8s-deployments-book-weather-app

A sample repository demonstrating the use of GitOps and Argo CD for deploying applications on Kubernetes.

Figure 4.3 – The new GitHub repository is now populated with Kubernetes manifest files and application files

3. **Deployment status**: By clicking on the **Actions** tab of our repository, we should be able to see at least one completed workflow, as illustrated in *Figure 4.4*:



All workflows

Showing runs from all workflows

1 workflow run

Execute build and push pipeline.

Publish Docker image #17: Commit 68426d4 pushed by pietrolibro

Figure 4.4 – Our completed the build and push pipeline

4. **Argo CD application**: At this point, we are ready to define the Argo CD application. As usual, we must define a YAML file containing the definition of the application we want to deploy via Argo CD. Let's create a file named `argco_cd_deployment.yaml` (the name isn't important) with the following content:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-city-weahter-app
  namespace: argocd
spec:
  destination:
    namespace: gitops-kubernetes
    server: https://kubernetes.default.svc
  project: default
  source:
    repoURL: https://github.com/[USERNAME]/[REPOSITORY].git
    path: deployment
    targetRevision: main
```

In this context, `name` defines the name of the Argo CD's application, `namespace` defines where Argo CD is installed, and `destination` specifies the target Kubernetes cluster and namespace (in our case, `gitops-kubernetes`). The `project` parameter indicates that the application should be installed within Argo CD's default project. The `repoURL` parameter denotes the repository where the Kubernetes manifest YAML files are located, `path` specifies where to look for these manifests, and `targetRevision` identifies the repository branch to be checked out.

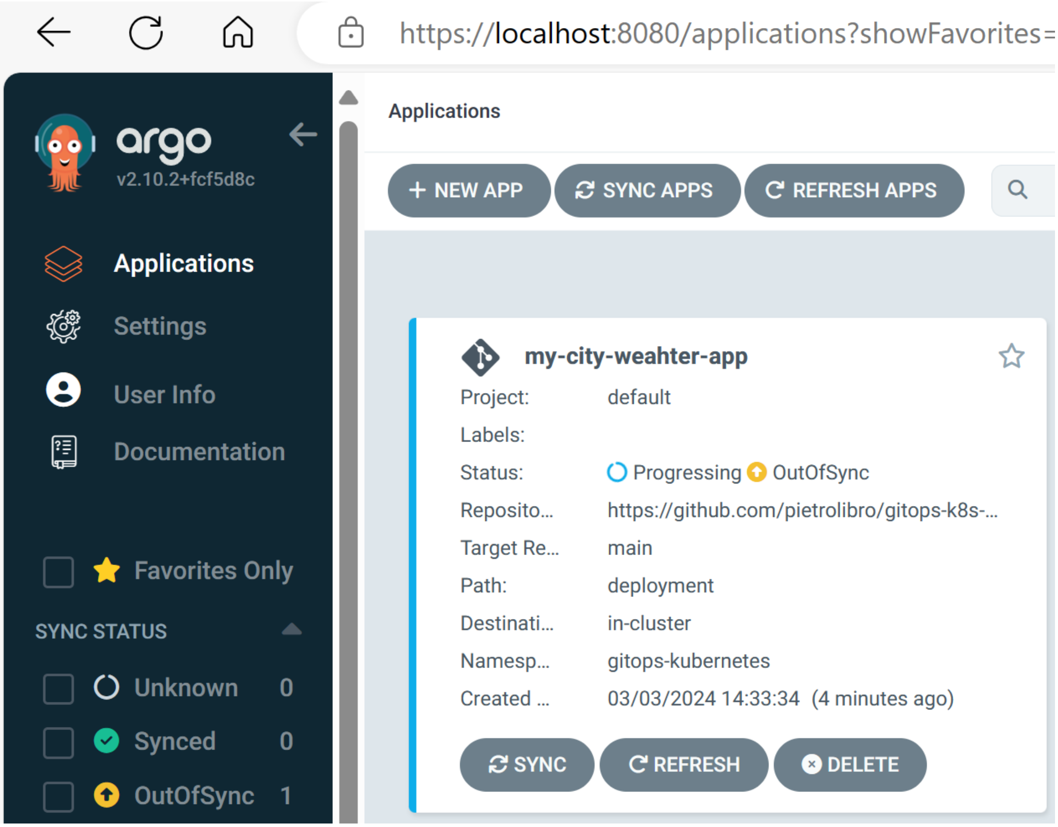
To create the defined application in Argo CD, enter the following command:

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

The expected output is shown here:

```
application.argoproj.io/my-city-weahter-app created
```

The crucial field to observe, as shown in *Figure 4.5*, is **Status**, which is labeled as **OutOfSync**. This status indicates that although the application has been registered, there is a **mismatch** between its current state (in this case, the application has not been deployed yet) and the information in the Git repository (which contains the application's deployment files):



https://localhost:8080/applications?showFavorites=

my-city-weahter-app

Project: default

Labels:

Status: Progressing OutOfSync

Reposito... https://github.com/pietrolibro/gitops-k8s-...

Target Re... main

Path: deployment

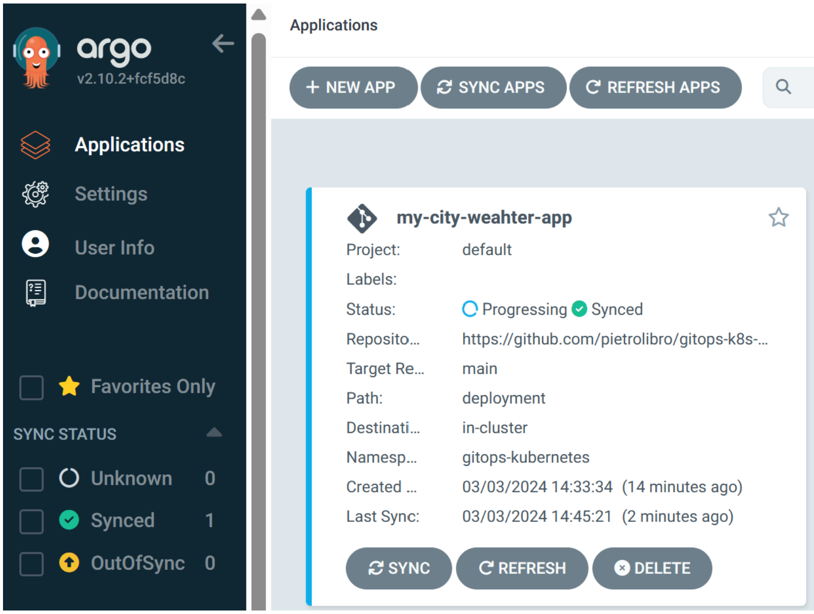
Destinati... in-cluster

Namesp... gitops-kubernetes

Created ... 03/03/2024 14:33:34 (4 minutes ago)

Figure 4.5 – The new Argo CD application has been successfully added to the default project

Click the **Sync** button to start the synchronization process. After a few seconds, **Status** should be **Synced**, as shown in *Figure 4.6*:



my-city-weahter-app

Project: default

Labels:

Status: Progressing Synced

Reposito… https://github.com/pietrolibro/gitops-k8s-…

Target Re… main

Path: deployment

Destinati… in-cluster

Namesp… gitops-kubernetes

Created … 03/03/2024 14:33:34 (14 minutes ago)

Last Sync: 03/03/2024 14:45:21 (2 minutes ago)

Figure 4.6 – The Argo CD application has been successfully synchronized with the repository

Clicking on the `my-city-weather` app provides additional details about the Argo CD application, as illustrated in *Figure 4.7*:

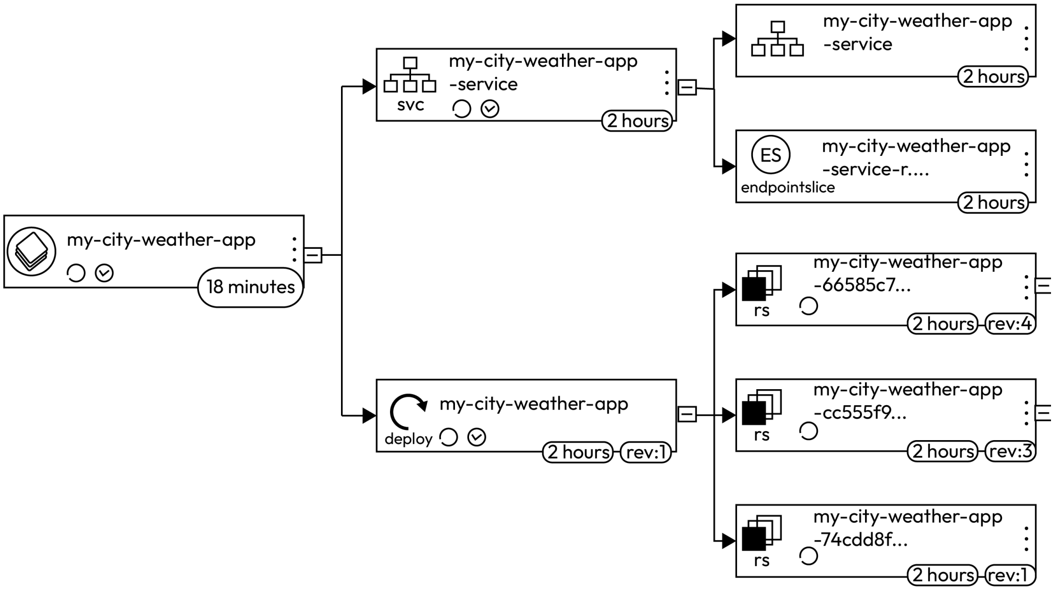
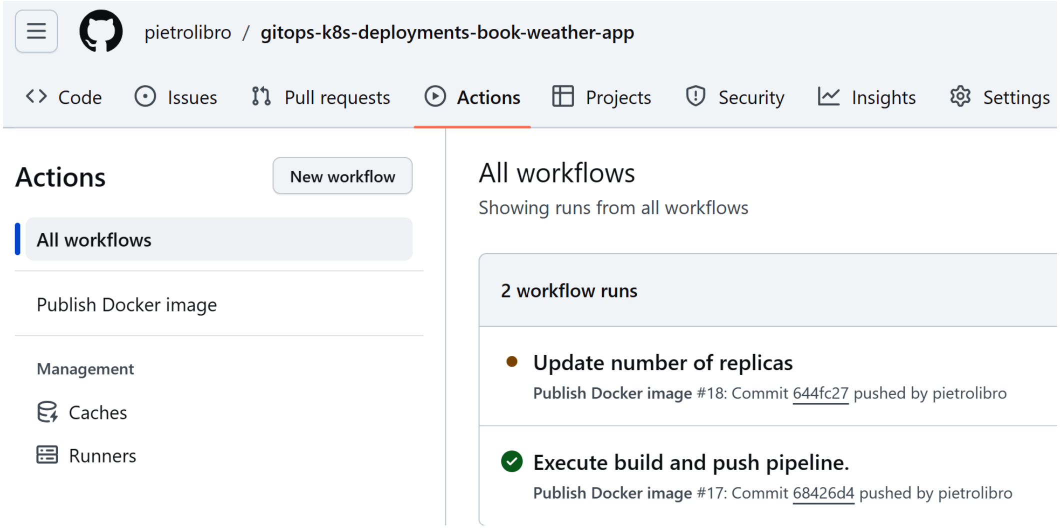


Figure 4.7 – Graphical view of the resources deployed by Argo CD to the Kubernetes cluster

5. **Modify the application**: Make changes to your application's code or Kubernetes manifests and push these changes to your GitHub repository. For instance, we can change the number of replicas in the `deployment\deployment.yaml` file from `1` to `2`:

```
spec:
  replicas: 2
  selector:
    matchLabels:
```

This change will trigger the proper action and a new run of the build-and-push pipeline will be performed, as shown in *Figure 4.8*:

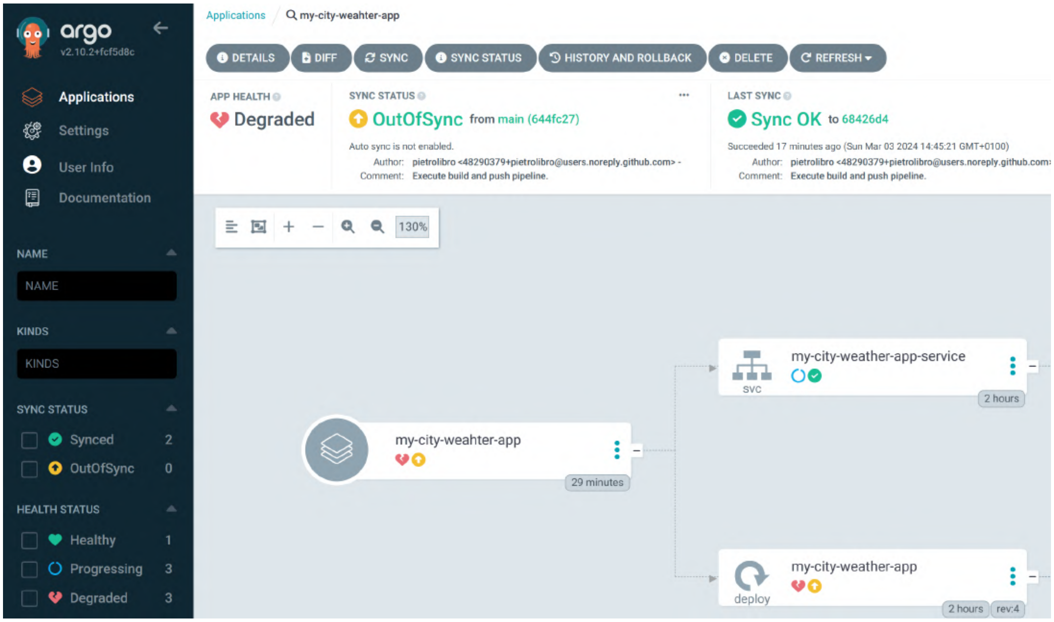


Update number of replicas
Publish Docker image #18: Commit 644fc27 pushed by pietrolibro

Execute build and push pipeline.
Publish Docker image #17: Commit 68426d4 pushed by pietrolibro

Figure 4.8 – Executing a new run of the defined pipeline

Now, observe how Argo CD detects that the live state (current deployment in Kubernetes) is out of sync with the desired state (the latest changes in your Git repository), as shown in *Figure 4.9*:



APP HEALTH
Degraded

SYNC STATUS
OutOfSync from main (644fc27)
Auto sync is not enabled.
Author: pietrolibro <48290379+pietrolibro@users.noreply.github.com> -
Comment: Execute build and push pipeline.

LAST SYNC
Sync OK to 68426d4
Succeeded 17 minutes ago (Sun Mar 03 2024 14:45:21 GMT+0100)
Author: pietrolibro <48290379+pietrolibro@users.noreply.github.com
Comment: Execute build and push pipeline.

Figure 4.9 – Argo CD's status changes to OutOfSync after modifying the number of replicas

6. **Sync Argo CD**: Use the Argo CD UI or CLI to synchronize the changes, updating your Kubernetes deployment so that it matches the latest state defined in your GitHub repository.

Congratulations on successfully navigating Argo CD! You've made significant strides in mastering GitOps principles and applying them effectively in a Kubernetes environment. As we transition from Argo CD, we'll embark on exploring another pivotal GitOps tool: Flux. The next section will guide you through the fundamentals of Flux, offering you another perspective on automated deployment strategies within Kubernetes. Let's continue our journey into the world of GitOps with an in-depth look at Flux and its unique features.

Flux integration with Kubernetes

Flux is a powerful, open source tool that automates the deployment of applications within Kubernetes, embodying the principles of GitOps. As a next-generation deployment solution, Flux continuously monitors your Git repositories for changes and automatically applies those changes to your Kubernetes clusters, ensuring that the state of your deployments always matches the source of truth in your version control system. This approach not only simplifies deployment workflows but also enhances security, traceability, and reliability across your infrastructure. In this section, we will dive into how Flux can transform your deployment strategy, making your operations more efficient and aligned with modern DevOps practices.

Flux consists of several key components that work together to monitor, pull, and apply configurations from a Git repository to a Kubernetes cluster:

- **Flux daemon (Fluxd)**: This is the core component that continuously monitors the configured Git repository. It checks for new commits to branches, tags, or specific paths and applies the changes to the Kubernetes cluster. It acts as the agent that ensures the desired state from the repository is accurately reflected in the cluster.
- **Source controller**: This component acquires source materials from repositories (such as Git) and detects if there are any changes. It ensures that the latest configurations are always used for deployments.
- **Kustomize controller (for Flux v2)**: Applies Kustomize configurations, allowing raw Kubernetes manifests to be customized without the need to alter the original YAML files.
- **Helm controller (for Flux v2)**: Provides declarative management for Helm chart releases. This component allows Flux to manage Helm chart dependencies directly from Git.
- **Git repository**: This is where the Kubernetes resource files (such as YAML manifests, Kustomize configurations, or Helm charts) are stored. Flux relies on this repository as the single source of truth for the state of resources in the Kubernetes cluster.
- **Synchronization**: Flux continuously synchronizes the configuration from the Git repository with the Kubernetes cluster. If it detects any discrepancies between the configuration in Git and the current state of the cluster, it updates the cluster so that it matches the desired state defined in Git.
- **Automation and rollback**: Flux supports automated deployments and rollbacks. If a new configuration is detected in the Git repository, Flux automatically applies it to the Kubernetes cluster. Similarly, if a configuration change leads to an error, Flux can revert to the previous stable version, aligning with the rollback capabilities of Git.
- **Security and secrets management**: Flux provides mechanisms to handle secrets securely and supports integration with Kubernetes secrets management tools. This ensures that sensitive information is not exposed in the Git repository.
- **Observability and notifications**: Flux can be configured to send notifications (for example, via Slack, email, or other messaging platforms) about the status of deployments and sync operations. Additionally, it provides observability features for monitoring the health and performance of the GitOps pipeline.

Flux setup

Setting up and configuring Flux is beyond the scope of this book. For comprehensive and detailed instructions on how to set up Flux, we highly recommend visiting the Flux website at `https://fluxcd.io/flux/installation/`. There, you will find the most up-to-date documentation, including step-by-step guides, troubleshooting tips, and best practices specifically designed to facilitate a smooth setup process.

You can enter the following command to check you have everything you need to run Flux:

```
$ flux check --pre
```

Before continuing, note the output of the previous command must resemble the following:

```
► checking prerequisites
✓ Kubernetes 1.28.5+k3s1 >=1.26.0-0
✓ prerequisites checks passed
```

In the upcoming section, we will refer to the same weather app described in the *The My City Weather app* section of this chapter.

Deploying to Kubernetes with Flux

First, we need to export our GitHub credentials (username and access token):

```
export GITHUB_TOKEN=ghp_jlFEXvI…
export GITHUB_USER=[GITHUB_USERNAME]
```

GitHub personal access token (PAT)

A GitHub **PAT** is a secure way to provide authentication to your GitHub account without the need to use a password. This token acts as a key, granting access to GitHub services through the API, command-line tools, and other third-party applications. This token can be customized with varying scopes of access, from read-only to full repository control, depending on what tasks need to be performed. This flexibility makes PATs particularly useful for automation, scripting, and integrating with third-party services or CI/CD pipelines, ensuring that operations that require GitHub access can be executed securely and efficiently. It's essential to keep your PATs confidential and treat them with the same level of security as your password to prevent unauthorized access to your GitHub account.

After installing Flux, you need to bootstrap it to your repository. This connects your Kubernetes cluster to the repository and sets up the necessary components for continuous synchronization:

```
flux bootstrap github \
--owner=[GITHUB_USERNAME]\
```

```
--repository=gitops-k8s-deployments-book-weather-app \
--path= ./deployment/base
--personal
```

The output should look like this:

```
► connecting to github.com
► cloning branch "main" from Git repository "https://github.com/
[GITHUB_USERNAME]/gitops-k8s-deployments-book-weather-app.git"
✓ cloned repository
► generating component manifests
✓ generated component manifests
…
► pushing sync manifests to "https://github.com/[GITHUB_USERNAME]/
gitops-k8s-deployments-book-weather-app.git"
► applying sync manifests
✓ reconciled sync configuration
✔ waiting for GitRepository "flux-system/flux-system" to be
reconciled
✓ GitRepository reconciled successfully
```

By executing the following command, we can verify how the resources of the `flux-system` namespace have been deployed within the respective namespace:

```
$ kubectl get all -n flux-system
```

Now, we need to create a file in the repository that outlines how Flux should apply the manifests. If you bootstrapped Flux with a path such as `./deployment/base`, you should place the `kustomization.yaml` file inside the deployment directory:

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

Commit and sync the changes with a comment like `Define Flux Kustomization`. After a few minutes, Flux should start synchronizing the repository's state with the local Kubernetes cluster. We can check the status of the synchronization by running the following command:

```
$ flux get kustomizations
```

Once everything is up and running, any changes you make to the Kubernetes manifests in your GitHub repository will be automatically applied to your cluster. Try changing the number of replicas in your `deployment.yaml` file, commit them, and push the changes. Flux will detect the update and apply the change to your cluster.

By following these steps, you have successfully connected your weather app, which is in a GitHub repository, to your Kubernetes cluster using Flux, achieving a GitOps workflow.

Now that we've learned how to deploy to Kubernetes with Flux, we'll compare Argo CD and Flux. This comparison will shed light on their distinctive capabilities and how each tool positions itself within the Kubernetes deployment ecosystem.

Comparing Argo CD and Flux

The following table compares Argo CD and Flux, two leading GitOps CD tools that are used in the Kubernetes ecosystem that have been introduced in depth in this chapter. Both tools are designed to automate the deployment process by syncing your Kubernetes cluster state with configurations stored in a Git repository, enabling a declarative approach to infrastructure management. This table outlines key features and differences between Argo CD and Flux to help you understand their capabilities, installation processes, synchronization mechanisms, and other critical aspects, all of which should help you select the right tool for your needs:

Feature	**Argo CD**	**Flux**
Project Maturity	Mature, widely adopted	Mature, widely adopted
Installation	**Custom resource definitions** (**CRDs**) and a dedicated namespace	Uses Helm or manifests for installation
Configuration	Declarative setup through YAML files	Declarative setup through YAML files
Sync Mechanism	Pull-based from Git repository	Pull-based from Git repository
Sync Frequency	Configurable, default every 3 minutes	Configurable, can be set as low as a few seconds
Auto-Sync	Supported, can automatically apply changes from Git	Supported, can automatically apply changes from Git
Secret Management	Integrates with external secret management tools	Has built-in secret management via Mozilla SOPS
Customization	Highly customizable through plugins and hooks	Extensive customization with Kustomize and Helm
User Interface	Provides a web-based UI for monitoring and management	Primarily a command line with an optional web UI component

Feature	**Argo CD**	**Flux**
Multi-Tenancy	Supported through namespaced installations	Supported using multiple instances
Community and Support	Large community, active development	Large community, active development
Rollback Features	Supports automatic and manual rollbacks	Supports rollbacks, typically managed through Git

Table 4.1 – Argo CD versus Flux – key differences in Kubernetes deployment

This section concludes our exploration of the most popular tools that are used in the context of GitOps and Kubernetes. We've delved into the intricacies of various tools that facilitate the GitOps workflow, emphasizing their roles, features, and how they integrate within the Kubernetes ecosystem. By understanding these tools, we equip ourselves with the knowledge necessary to implement efficient, automated, and scalable DevOps practices in cloud-native environments.

Summary

This chapter transitioned from theoretical aspects of GitOps to practical applications within Kubernetes by focusing on tools such as Helm, Kustomize, Argo CD, and Flux. It provided a comparative analysis of these tools, highlighting their unique features and how they complement Kubernetes deployments. This chapter was designed for those with foundational Kubernetes knowledge and covered technical prerequisites, how to integrate GitOps tools with Kubernetes clusters, and practical examples for real-world application. Additionally, it offered step-by-step guides for deploying applications using Argo CD and Flux, emphasizing GitOps methodologies for efficient and secure application management.

As we conclude this chapter on orchestrating and managing Kubernetes environments, our exploration transitions to the next frontier: diving into advanced Git repository strategies, service catalog construction, and the scalability of GitOps through ApplicationSet generators. These topics will further refine our understanding of efficient operations and the multifaceted approach required for successful multitenancy and cloud-native scalability.

Part 2:

Harnessing Advanced Orchestrations, Culture, and Control in GitOps Practices

In this part, you will delve into advanced GitOps practices. You will learn about scaling GitOps across multiple clusters, architectural designs for efficient operational control, and the cultural transformation required to embrace GitOps fully within IT organizations. This section emphasizes not only the technical implementation but also the strategic and cultural shifts needed to support and sustain GitOps at an enterprise level, providing a holistic view of GitOps adoption and integration.

This part includes the following chapters:

- *Chapter 5*, GitOps at Scale and Multitenancy
- *Chapter 6*, GitOps Architectural Designs and Operational Control
- *Chapter 7*, Cultural Transformation in IT for Embracing GitOps

5

GitOps at Scale and Multitenancy

This chapter delves into advanced **GitOps** applications, focusing on scaling and **multitenancy** within Kubernetes environments. It's tailored for those with a foundational understanding of tools such as **Argo CD** and who are looking to expand their knowledge in more complex scenarios.

We'll start by discussing strategies to build scalable cluster infrastructures using GitOps. This includes designing Kubernetes clusters that can adapt to increasing demands, all managed through GitOps methodologies. A significant part of this discussion involves deploying applications across multiple clusters efficiently, focusing on scalability and customization.

A critical aspect we'll address is enforcing multitenancy in shared Kubernetes environments. We'll explore how to achieve this using GitOps tools such as Argo CD, adhering to their operational philosophies. Complementary to this, we'll introduce tools such as **vCluster** that simplify multitenancy enforcement while maintaining GitOps principles.

The emphasis throughout this chapter is on concepts over tools. While tools may evolve, the underlying principles remain constant, providing a stable foundation for understanding these technologies.

We'll also cover the implementation of a lightweight **internal developer platform** (**IDP**) to facilitate the **deployment** of third-party tools through **Kubernetes Service Catalog** (**KSC**). This approach simplifies application management within Kubernetes.

Real-world insights form a significant part of this chapter, drawing from experiences and lessons learned in diverse project environments. This practical perspective is invaluable for understanding the real-world application of these strategies.

This chapter, which is aimed at **intermediate learners**, won't delve into the setup or basic operations of Argo CD. Instead, it will focus on their application in complex, real-world scenarios, demonstrating the practical use of the most suitable tools for each case. The goal is to equip you with a comprehensive understanding of scaling and managing multitenancy in Kubernetes using GitOps, enriched with real-world applications and insights, by staying focused on the approaches and not the tools.

The chapter is tough, but I hope you'll have learned a lot by the time you finish it. I've tried to share all the insights from projects in a compact way with you. This chapter can be logically divided into two sections. The first section covers approaches to using GitOps at scale and the necessary setup via KSC. The second section, starting on page 45, focuses on multitenancy with GitOps to get the most out of the setup.

We will cover the following main topics in this chapter:

- Understanding the App of Apps approach
- Understanding multi-cluster management
- Understanding effective Git repository strategies
- Building a service catalog for Kubernetes
- Exploring native multitenancy with Argo CD
- Exploring multitenancy with vCluster and Argo CD

Technical requirements

Due to the limited space available, many examples have been shortened or are incomplete. Therefore, we have a repository with complete examples. This follows the `chapter05/section` pattern – that is, `chapter05/chapter-5-building-a-service-catalog-for-kubernetes`. For all the code examples discussed, along with additional resources, please refer to the `Chapter05` folder in the book's GitHub repository at `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`.

Traditional CI/CD versus GitOps CD

The main difference between traditional CI/CD and Argo CD is in how deployments are handled

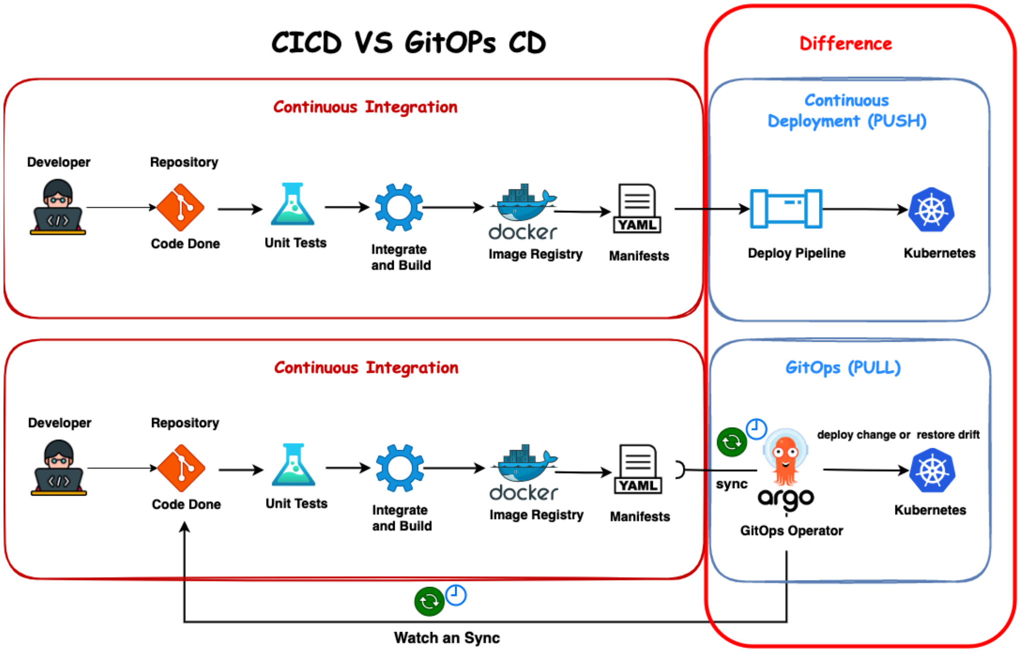


Figure 5.1 – Traditional CI/CD versus GitOps CD

Traditional CI/CD follows a workflow where changes are automatically integrated, tested, and deployed, while Argo CD uses a synchronization mechanism to ensure the actual state matches the desired state in the Git repository. Argo CD relies on a CI step, meaning any change must go through a CI process before Argo CD can detect and act on it. This ensures that only verified changes are deployed. Unlike CI/CD, which might require manual interventions that can lead to discrepancies, Argo CD continuously monitors and synchronizes the system state with the Git repository, reducing the risk of drifts and maintaining consistency.

In the following section, we'll look at the difference between platform engineering and IDPs.

Platform engineering versus IDPs

In my opinion, a distinction between **internal developer platforms** (**IDPs**) and **internal developer portals** (**IDPOs**) is becoming increasingly relevant and is already a topic of much debate. To clarify the different approaches, here's how I differentiate these terms:

- **Platform engineering**: This is the traditional approach where a dedicated team owns, operates, and continuously improves the Kubernetes platform itself. Their focus is on the underlying infrastructure, ensuring its stability, scalability, and security. Developers working on applications typically consume this platform and concentrate solely on their software or third-party tools that aren't offered by the platform team. This is a common approach in many projects today.
- **Internal developer platform (IDP)**: This approach introduces an IDP solution, such as backstage.io from Spotify. The IDP team is responsible for providing and maintaining the IDP itself, which includes extensive documentation, building blocks, and templates for developers and acts like a portal (IDPO) for developers. Developers can then leverage these resources to self-deploy their applications within a defined Kubernetes environment. This allows developers to have some influence on the content and functionality of the IDP through contributions or requests. The engineering focus of the platform team shifts from core Kubernetes operations to managing and evolving the IDP. However, developers are still responsible for the day-to-day operations of their deployed Kubernetes environments, including updates and troubleshooting.

Now, let's explore the App of Apps approach.

Understanding the App of Apps approach

In managing multiple applications, there are primarily two established strategies: the **App of Apps** approach and **ApplicationSets**. This section will address several key questions:

- What challenges does the App of Apps approach overcome?
- In which situations is the App of Apps approach most beneficial?
- How does the App of Apps approach enhance GitOps practices?

An *application* in this context refers to the Git repository and folder where manifests, which are essential definitions that allow your app to run in Kubernetes, are stored. Argo CD is versatile as it supports raw **YAML** manifests, custom configuration management, and popular tools such as **Kustomize**, **Helm**, and **Jsonnet**.

But what about scenarios where we must deploy multiple applications? How should we manage these manifests? Each application being deployed requires an application definition. However, when these applications are a collection of related entities, it would be beneficial if Argo CD could recognize this relationship. To address this need, the Argo community developed the App of Apps approach. Essentially, this approach allows a root Argo CD *application* to be defined, which, in turn, defines and synchronizes multiple child applications. This method streamlines the management process, especially in complex deployments, by leveraging a hierarchical folder structure.

Figure 5.2 illustrates an instance of the App of Apps approach. In this example, a single Argo CD *application* corresponds to just one specific web application:

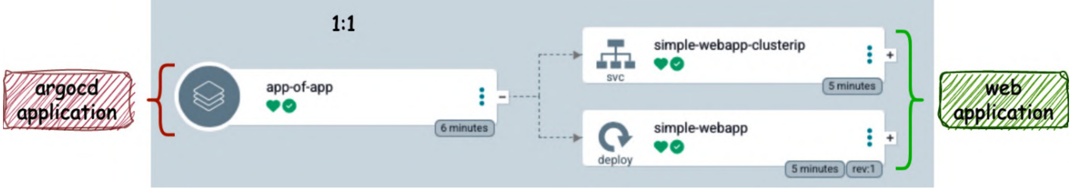


Figure 5.2 – The App of Apps approach

Examining the application manifest reveals key details:

```
project: default
source:
  repoURL: 'git@github.com:PacktPublishing/Implementing-GitOps-with-Kubernetes.git'
  path: ./chapter05/chapter-5-the-app-of-apps-approach/app-of-app/simple-webapp
  targetRevision: main
destination:
  server: 'https://kubernetes.default.svc'
  namespace: app-of-app
syncPolicy:
  automated:
    prune: true
    selfHeal: true
```

Here, `path` is set to directly point to the specific application. Next, let's explore the App of Apps approach shown in *Figure 5.3* for a more comprehensive understanding:

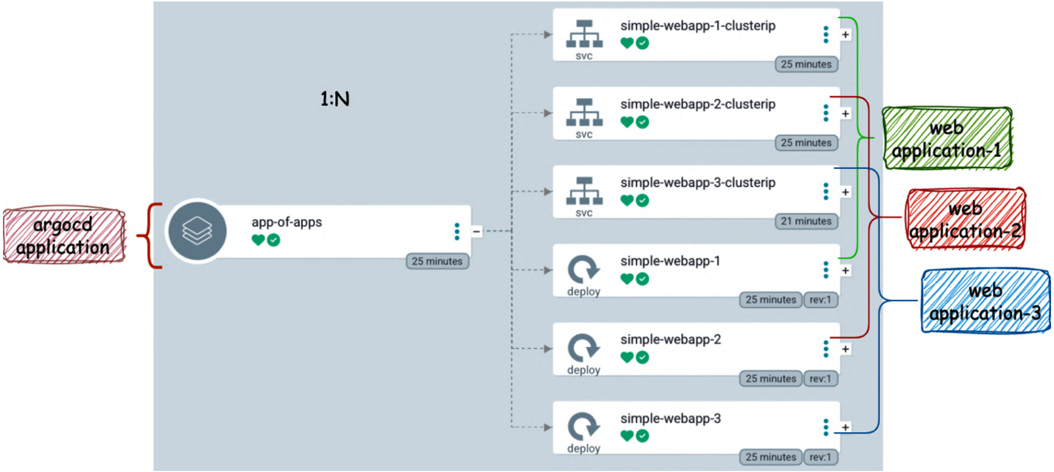


Figure 5.3 – App of Apps

Rather than directing toward a singular application manifest, as it did previously, the root app now references a specific folder within a Git repository. This folder contains all the individual application manifests that define and facilitate the creation and deployment of each application. By adopting this approach, it's possible to declare all your applications within a unified YAML manifest. The following example demonstrates this pattern for enhanced comprehension:

```
project: default
source:
  repoURL: 'git@github.com:PacktPublishing/Implementing-GitOps-with-
Kubernetes.git'
  path: ./chapter05/chapter-5-the-app-of-apps-approach app-of-apps/
simple-webapps
  targetRevision: main
  directory:
    recurse: true
 destination:
  server: 'https://kubernetes.default.svc'
  namespace: app-of-apps
syncPolicy:
  automated:
    prune: true
    selfHeal: true
```

In the provided definition, the `path` attribute instructs Argo CD to target a specific directory – in this case, named `simple-webapps` – located within the repository. This directory contains Kubernetes manifests that define the applications, as well as supporting various formats such as Helm, Kustomize, or plain YAML files. In the provided configuration, there are two notable attributes worth highlighting: `selfHeal: true` and `directory.recurse: true`. The `selfHeal` feature ensures automatic updates of the child applications in response to any changes detected, maintaining consistent deployment states. Additionally, the `recurse` setting enables the iteration through the `webapps` folders, facilitating the deployment of all applications contained within.

Therefore, the App of Apps approach enables you to administer your *application* resources by simply updating your manifests in the Git repository – adding or removing application resources as needed. This approach reduces reliance on direct interactions with Argo CD *applications* through the web UI or **CLI**.

Use cases of App of Apps combined with examples

The App of Apps approach in Argo CD is highly advantageous for managing multiple applications as a single entity while ensuring their isolation during deployment. This is particularly useful in scenarios such as cluster bootstrapping and managing Argo CD applications without relying on the CLI or UI. Let's explore these use cases with relevant examples:

- **Cluster bootstrapping**: Imagine that you have a standard set of applications that need to be installed in every new Kubernetes cluster. Rather than deploying each application individually, you can group them into a single "root" application. This simplifies the process, allowing you to deploy the entire set of applications simultaneously, enhancing efficiency and consistency across different deployments.

  Example for developers and service providers: Let's say you're developing or providing services that involve deploying a custom yet similar stack for each deployment, such as the following:

  - Frontend
  - Backend for frontend
  - Database

  The App of Apps approach allows you to encapsulate these components into a single deployment entity, streamlining the process.

- **Managing Argo CD applications without the CLI or GUI**: You can modify an existing root application using Git operations, such as adding new folders to the paths it monitors. This capability lets Argo CD automatically deploy new applications or update existing ones without needing to interact through the CLI or web UI, aligning with GitOps principles of version control and auditability.

  Example for platform engineers: As a platform engineer, let's say you're providing a similar stack on Kubernetes for each customer, such as the following:

  - **Ingress-Controller**
  - **Cert-Manager**
  - **External-DNS**

  The App of Apps approach is beneficial here as it allows you to manage these components effectively, ensuring that each customer's environment is consistently configured with the necessary tools.

In both use cases, the App of Apps approach facilitates a more streamlined, efficient, and consistent deployment process, whether you're dealing with different client requirements as a service provider or ensuring uniformity across various Kubernetes clusters as a platform engineer.

The ApplicationSets approach

In this section, you'll discover the process of creating, updating, managing, and removing numerous Argo CD *applications* through the use of an *ApplicationSet* [*1*] controller.

We will delve into the concept of an *ApplicationSet* and address key questions such as the following:

- What exactly constitutes an *ApplicationSet*?
- What are the functionalities and advantages of an *ApplicationSet*?
- Why is a generator necessary, and what varieties exist?

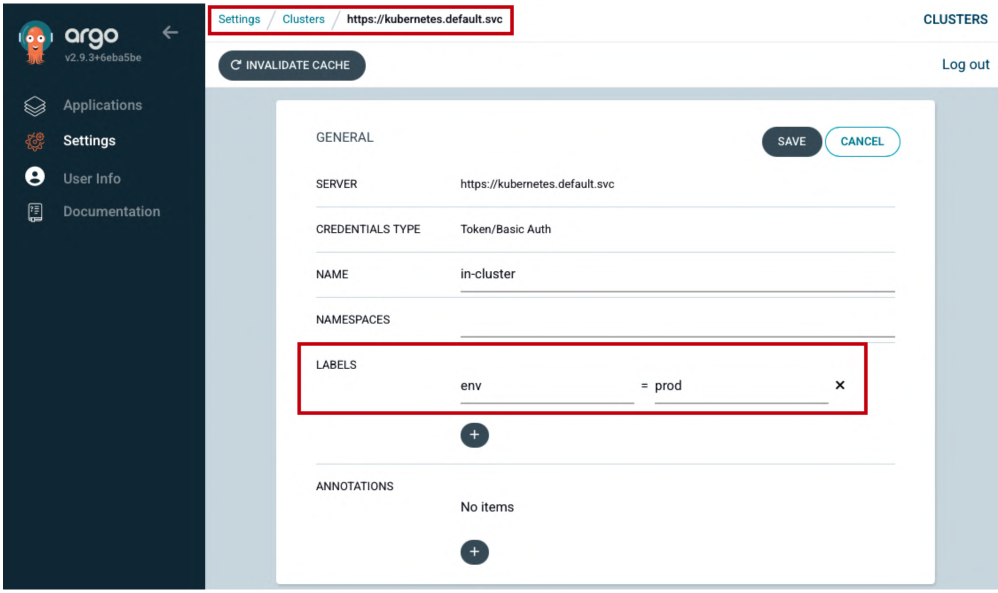
In the Argo CD framework, an *ApplicationSet* [*2*] significantly enhances the GitOps strategy for **continuous deployment** (**CD**) within Kubernetes. This tool adeptly handles the complexity involved in managing a variety of Kubernetes manifests, such as deployments, services, secrets, and configuration files, all within a Git repository. Unlike the Argo CD *application* resource, which is limited to deploying resources from a single git repository to one cluster or namespace, the ApplicationSet extends this functionality. It utilizes templated automation to concurrently create, modify, and oversee multiple Argo CD applications, thereby broadening its operational scope to encompass several clusters and namespaces.

The ApplicationSet controller, which is installed in the same namespace as Argo CD, plays a crucial role. It generates a multitude of Argo CD applications from the ApplicationSet's **custom resource** (**CR**). This arrangement ensures that your Argo CD applications are in sync with your specified resources, effectively transforming the ApplicationSet into one or more Argo CD applications, thus enhancing overall deployment efficiency and scalability.

Generate refers to the process employed by the controller using various **generators**, but what exactly are these generators? Generators in the ApplicationSet resource play a crucial role by creating parameters that are incorporated into the template fields, ultimately generating Argo CD Applications. For a practical example of this process, refer to this chapter's introduction. The functionality of generators is determined by their data sources. For instance, the List generator derives parameters from a predefined list, the Cluster generator utilizes the Argo CD cluster list, and the Git generator gets sources from files or directories in a Git repository.

There are numerous generators for different use cases and roles within ApplicationSets. For instance, there's the Cluster generator, which is ideal for **platform engineers** to scale their platforms, and the Pull Request generator, which allows developers to deploy features for QA through GitOps. Additionally, the **Matrix generator** allows you to combine up to two generators to meet more specific requirements.

We'll be utilizing the Cluster generator to implement GitOps at scale. Moving beyond theory, let's dive into its practical application. We'll start with a single cluster, tagging it with a label such as *env=prod*. This can be done in the Argo CD UI by navigating to **Settings** | **Clusters** | **Select In-Cluster** |**Edit** | **Add Labels**:



SERVER https://kubernetes.default.svc

CREDENTIALS TYPE Token/Basic Auth

NAME in-cluster

NAMESPACES

LABELS

env = prod

ANNOTATIONS

No items

Figure 5.4 – Adding labels

Now, create an `ApplicationSet` manifest, like so:

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: simple-webapp
  namespace: argocd
spec:
  generators:
    - clusters:
        selector:
          matchLabels:
            env: prod
        values:
          branch: main
  template:
    metadata:
      name: "{{name}}-simple-webapp"
      annotations:
        argocd.argoproj.io/manifest-generate-paths: ".;.."
    spec:
      project: default
      sources:
        - repoURL: git@github.com:PacktPublishing/Implementing-GitOps-
with-Kubernetes.git
          targetRevision: "{{values.branch}}"
          path: ./chapter05/chapter-5-the-app-of-apps-approach/
applicationsets/simple-webapp
      destination:
        name: "{{name}}"
        namespace: "argocd"
        syncOptions:
          - CreateNamespace=true
```

Here, we're employing the Cluster generator, which is designed to deploy our `simple-webapp` across various Kubernetes clusters. By using the `env=prod` selector, the *ApplicationController* will create several applications corresponding to the count of your clusters. Each application's name will be modified to include the cluster name – for example, `in-cluster-simple-webapp`.

To apply `ApplicationSet`, use the following command:

```
kubectl apply -f simple-webapp-applicationset.yaml
```

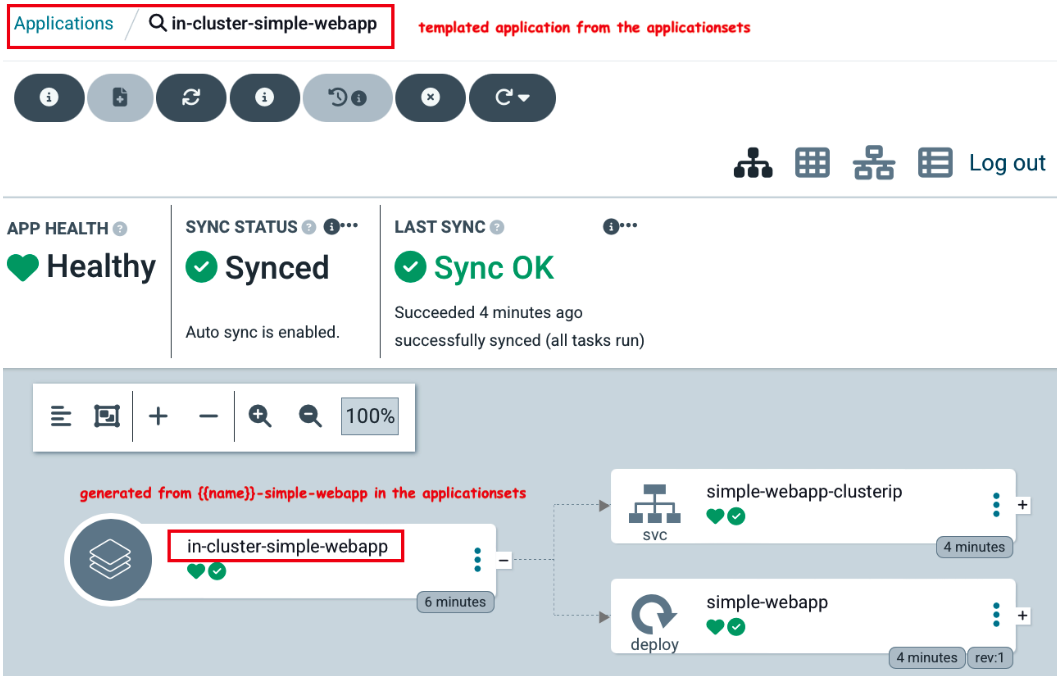
To view `ApplicationSet`, run the following command:

```
kubectl get applicationsets -n argocd
```

You can also view a templated application like so:

```
kubectl get application -n argocd
```

When you check the Argo CD UI, you won't find the ApplicationSet directly, but you will see the templated application, named `in-cluster-simple-webapp`. This application is managed by the application controller in the following manner:



templated application from the applicationsets

APP HEALTH: Healthy

SYNC STATUS: Synced

Auto sync is enabled.

LAST SYNC: Sync OK

Succeeded 4 minutes ago

successfully synced (all tasks run)

generated from {{name}}-simple-webapp in the applicationsets

Figure 5.5 – The application templated through an ApplicationSet, managed by the application controller, and informed over the Cluster generator

The templated application is visible through the `ApplicationSet` manifest, where the application controller uses the Cluster generator to set the necessary parameters.

Which approach should be used?

In most projects I've been involved in, teams prefer using individual applications and ApplicationSets over the App of Apps approach. From a platform engineer's [3] perspective, for creating scalable infrastructure with GitOps, the ApplicationSets approach seems to be the most logical choice.

The App of Apps pattern in Argo CD is suitable for the following aspects:

- **Bootstrapping multiple applications**: Efficiently deploy numerous applications simultaneously.
- **Managing applications as a single unit**: Simplify the management of multiple applications.
- **Enhancing the deployment workflow**: Streamline the process of deploying and updating applications.

ApplicationSets can be particularly beneficial in the following scenarios:

- **Creating flexible deployment strategies for diverse environments**: Deploy to multiple Kubernetes clusters, to different namespaces in different clusters, or to different namespaces on a single cluster (developer, DevOps, and platform engineer).
- **Managing Applications in Monorepos**: Deploy from different Git repositories, SCM providers, or folders (developer, DevOps, platform engineer).
- **Enabling self-service for multi-tenant clusters**: ApplicationSets can facilitate a self-service model, particularly with the Pull Request generator. This allows developers to deploy applications in multi-tenant clusters with greater autonomy, without needing cluster-level permissions (collaboration between developers and platform engineers).
- **Deploying cluster add-ons across multiple clusters**: Using the Cluster generator, you can target add-ons to specific clusters managed within Argo CD, which is useful for large-scale, multi-cluster environments (platform engineers, **Site Reliability Engineers** (**SREs**) and DevSecOps engineering).

Contrasting this with the App of Apps approach, which is more suited for managing a collection of related applications within a single repository or cluster, ApplicationSets offer more flexibility and scalability, especially in environments with diverse deployment needs. They allow for more granular and distributed control, aligning with complex infrastructure requirements and multi-cluster strategies.

Later, we will adopt an ApplicationSet to develop a scalable model for an IDP, referred to as KSC. This platform aims to efficiently manage a large number of Kubernetes clusters while ensuring up-to-date security measures are maintained.

Next, we'll take a deep dive to understand what multi-cluster management means in the context of GitOps with Argo CD and what possibilities it opens.

Understanding multi-cluster management

In this section, we'll delve into our experiences with two different approaches to managing multi-cluster environments within the GitOps framework. Our focus is not on the tools themselves, but rather on the overarching strategy of orchestrating and managing these clusters as though they were a singular platform. Through this exploration, we aim to impart a deeper understanding of these methodologies, emphasizing that the key lies in effective orchestration rather than a comparison of specific tools. We'll discuss two distinct concepts:

- **One cockpit to rule them all**: This concept emphasizes centralized management and orchestration of multiple clusters as a unified platform
- **One cockpit – diverse fleet management**: This concept focuses on managing a diverse range of Argo CD clusters from a single control point while considering security aspects of inter-cluster communication

In terms of tool selection, our experience suggests that Argo CD, with its support for clusters, ApplicationSets, and generators, is better suited for scaling with GitOps in multi-cluster environments compared to Flux, which lacks a dedicated multi-cluster management concept. This differentiation becomes clearer in contexts where management extends beyond dedicated clusters to scenarios such as **vCluster** approaches within a host cluster.

Here are some additional considerations for multi-cluster management in a GitOps framework:

- **High availability and disaster recovery**: This includes deploying across multiple regions or even using multiple providers to ensure robustness and resilience
- **Existing expertise**: The level of knowledge in Kubernetes, the cloud, or GitOps within the organization plays a critical role
- **Budget constraints**: The costs involved in multi-cluster management should not exceed a predetermined amount
- **Compliance and regulatory requirements**: Ensuring adherence to industry standards and legal regulations in different regions or sectors
- **Network infrastructure and latency**: Optimizing for network performance and reducing latency, something that's especially important in geographically dispersed clusters
- **Cloud or service provider implementation**: For example, AKS works with Flux CD for its GitOps implementation, while OpenShift uses Argo CD for GitOps deployments

Overall, the choice between Argo CD and Flux CD hinges on their respective capabilities in orchestrating and managing multi-cluster environments and specific use case requirements.

One cockpit to rule them all

In this approach, there is a single Argo CD instance that's shared by both developers and platform engineers (*Figure 5.6*). This shared instance allows platform engineers to offer centralized management functions and comprehensive control over various Kubernetes clusters. They manage and monitor all deployments, ensuring developers have access to the required resources while upholding company policies and security standards. This method promotes collaboration and provides a unified view of all clusters, effectively reducing complexity in large, distributed organizations:

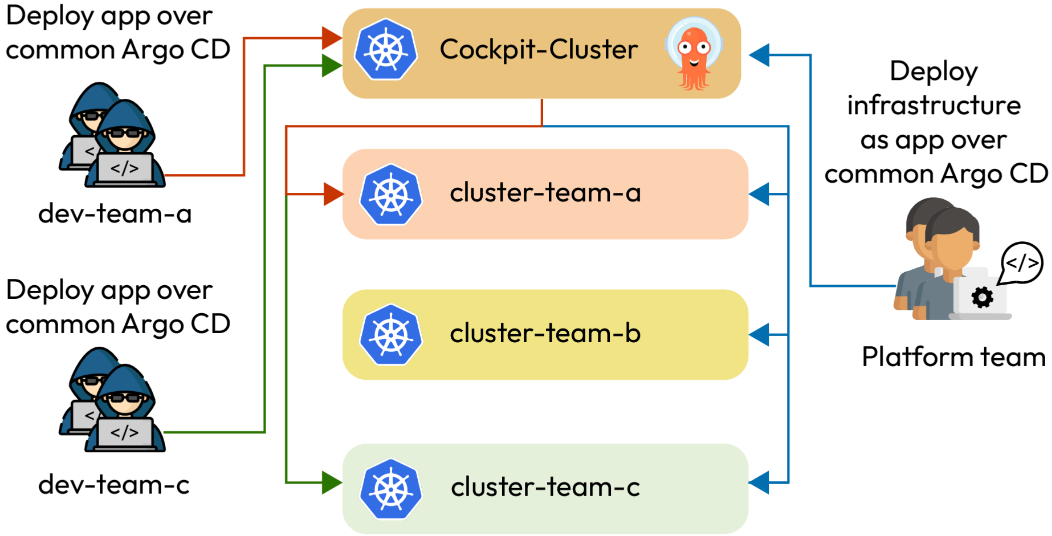


Figure 5.6 – One cockpit to rule them all – with a common Argo CD instance

In the *one cockpit to rule them all* approach for GitOps with Argo CD, the following are some crucial considerations:

- **Developer access**: Determining developer access to the Argo CD UI involves establishing **role-based access control** (**RBAC**) associated with groups, projects, and roles, potentially including Dex and OIDC provider integration.
- **Project access control**: Limiting team access to specific projects to prevent unauthorized deployments across clusters.
- **Resources allocation**: Are savings being made in the right areas? This question prompts a critical evaluation of resource allocation, questioning whether cost savings are effectively targeted in areas that maximize efficiency and overall value.

- **Version updates**: Coordinating updates for Argo CD across teams on a shared instance used by multiple teams is crucial. This coordination ensures version compatibility and prevents issues related to API deprecations, maintaining a stable and functional GitOps environment for all teams.

Working with multiple teams on this approach brings a multitude of questions and considerations that need to be addressed, emphasizing the complexity and planning required for effective implementation.

Here are some points based on our experience with this approach:

- Shared Argo CD usage often led to increased system load, occasionally causing outages
- Integration of features such as Argo CD's PR-Generator added complexity
- It necessitated robust multitenancy frameworks and tools
- Argo CD, as the sole point of control, introduced a heightened risk of system failure
- The increase in teams significantly raised the requirements for communication and system maintenance
- While there were savings in hardware resources, these were counterbalanced by greater demands on human resources for management and coordination

We employ this approach for shared clusters to facilitate cold starts, save money and resources, provide a learning platform, and offer shared services such as documentation and runners. A shared cluster is more relevant for incubating projects, and when a project is mature or needs to head for production, it should be moved onto its own cluster.

One cockpit – multiple fleet and commander concept

In this concept, one Argo CD instance will be used by the platform team to deploy and manage the whole infrastructure that's needed by developer teams. The platform team also deploys dedicated Argo CD instances with a **dedicated UI** for every team. The focus is the same, but it's more so about the orchestration process and managing those clusters as if they were a single platform. The developer will get their dedicated Argo CD instance:

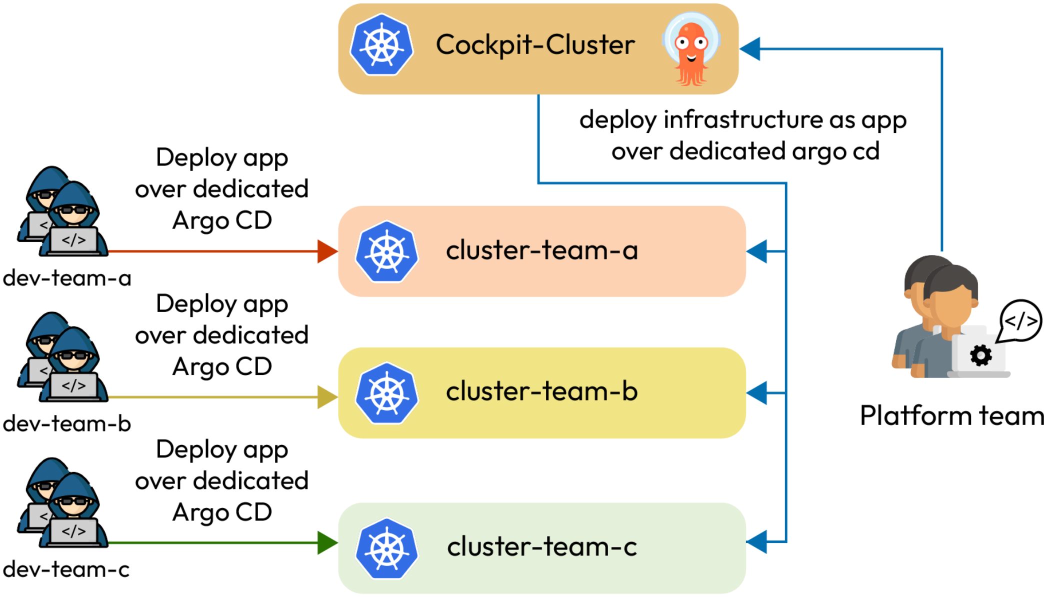


Figure 5.7: One cockpit – diverse fleet management with dedicated Argo CD instances per cluster

In contrast to the *one cockpit to rule them all* approach, this approach involves the following aspects:

- **Dedicated Argo CD instances**: Each cluster is managed by its own Argo CD instance, enhancing individual cluster autonomy and reducing risks associated with a single point of failure.
- **Autonomous project management**: Each team manages projects within their designated clusters, allowing for greater control and customization.
- **Resource allocation**: Instead of leveraging a shared Argo CD instance, each cluster operates its own stack. While this may offer focused resource management within individual Argo CD instances per cluster, it potentially leads to higher overall resource consumption.
- **Version control autonomy**: Each team controls the version updates of their Argo CD instance, ensuring smooth operation and compatibility within their cluster environment.

Here are some insights from our experience with the *one cockpit – diverse fleet management* approach:

- **Increased autonomy**: Dedicated resources and UI access grant more **freedom**, **flexibility**, and **responsibility**. This setup allows for self-service and extended control for developers, enabling them to focus on development.
- **Ease of onboarding**: This model simplifies GitOps adoption, making it suitable for both newcomers and experienced teams. Onboarding new colleagues requires less time with this approach.
- **Resource consumption**: While offering clear team separation, this approach can lead to greater resource usage.
- **Enhanced security**: The compromise of one Argo CD instance doesn't impact others, increasing overall security.
- **Platform team focus**: Platform teams concentrate on scaling and engineering services for a scalable, self-service approach. Less time is needed for interactions with development teams and for creating and hardening multitenancy aspects related to the shared Argo CD instance.
- **Resilience and isolation**: This approach enhances resilience and isolation for each project or development team. It eliminates issues with noisy neighbors, and upgrades to Kubernetes or infrastructure by other teams or projects do not affect individual teams, ensuring smoother operations and reduced disruptions.

This approach is adopted for handling **critical workloads**, ensuring strict tenant separation, enhancing developers' self-service capabilities, and bolstering security in compliance with stringent requirements for security, governance, and compliance. In the realm of GitOps, which is central to continuously delivering software into clusters, this approach becomes even more crucial when scaling, particularly from a security perspective in DevOps practices.

In managing multi-cluster environments at scale with GitOps, the emphasis extends beyond just overseeing dedicated Kubernetes clusters. It involves leveraging Argo CD to address varied needs and deploy diverse workloads, including both customer applications and infrastructure components, while maintaining security compliance across different clusters. This approach aims to bridge the gap between developers and platform engineers, fostering scalability and boosting productivity in both domains.

However, there is another gap with GitOps that relates to the best way to set up a staging concept. This will be explained in more detail in the next section.

Understanding effective Git repository strategies

Understanding how to effectively promote applications between stages in a GitOps framework, especially at scale, is a crucial challenge for both developers and platform engineers. Deploying an application to various environments involves navigating complexities beyond a single deployment scenario. With Argo CD, the process becomes more manageable, allowing deployment across multiple clusters without needing multiple CD pipelines.

This section delves into various approaches for managing environments: environment branches, environment per Git, and folders for environments. Each has its pros and cons, and the choice largely depends on the specific needs of the project and the team's expertise. Companies such as **Codefresh** [4] have developed solutions to facilitate stage propagation with Argo CD. However, this book focuses more on understanding these approaches rather than specific tools, guiding you to choose the most suitable strategy for your environment.

Environment branches

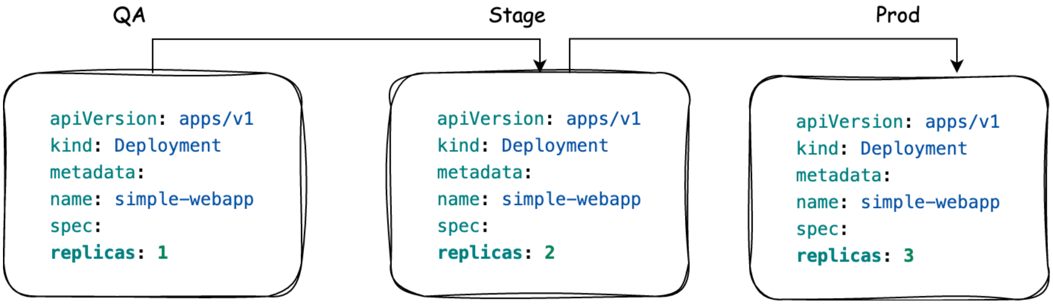
The environment-per-branch [4] approach in GitOps, which involves using branches to represent different environments such as staging or production, is often considered an anti-pattern. This approach can complicate pull requests and merges between branches, create configuration drifts, and increase maintenance challenges with a large number of environments. It also contrasts with the Kubernetes ecosystem and is generally better suited for legacy applications. In GitOps, it's recommended to separate application source code and environment configuration into different repositories, avoiding the branch-per-environment model. For deployment promotions, Git merges can be problematic due to conflicts and unintended changes, making promotion management more complex than it appears.

Advantages of the environment-per-branch approach include familiarity for many developers and the theoretical simplicity of promoting releases through git merges.

However, the disadvantages are significant if you're working with GitOps and Kubernetes:

- **Complex merges**: Promotion through Git merges can lead to conflicts and unintended changes
- **Configuration drift**: Different branches might lead to environment-specific code, causing configuration drift
- **Maintenance challenges**: Managing a large number of branches can become unwieldy
- **Dependent changes**: Difficulties arise in managing changes that have dependencies as not all commits can be cleanly cherry-picked

Now, let's explore a straightforward deployment manifest as an illustrative example.



```
apiVersion: apps/v1
kind: Deployment
metadata:
name: simple-webapp
spec:
replicas: 1
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: simple-webapp
spec:
replicas: 2
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: simple-webapp
spec:
replicas: 3
```

Figure 5.8: How to propagate between stages

This basic example illustrates the need to incorporate propagation logic in deployment manifests, ensuring that specific values, such as the number of replicas, are appropriately scaled for each stage according to its unique requirements. Using tools such as Helm or Kustomize with GitOps can present challenges as Helm uses different `values.yaml` files for stages, and Kustomize relies on overlays. Additionally, GitOps, based on the Kubernetes ecosystem, brings its own complexities that must be managed.

While this approach might suit legacy applications, it's less ideal for modern Kubernetes environments. Branches can still be used for features and PRs for testing changes. However, you still need pipelines or workflows to commit the changes into Git. There is an option for writing back to Git outside of Argo CD. Tools like Argo CD and its PR-Generator can help manage these processes, but note that the PR generator is used to manifest Git content into the cluster, not the other way around.

Environment per Git

The **environment per Git** approach arises from concerns, especially in security teams, about a single Git repository branch containing both production and non-production environments. While securing individual branches can be easier, issues in the folder approach can be addressed through automation, validation checks, owner of code, or manual approvals. Promotions should ideally be automated, with no direct commits to `main` but through pull requests and review workflows. For heightened security needs, organizations can use two git repositories: one for all production-related configurations and environments, and another for non-production elements, balancing security with a manageable number of repositories.

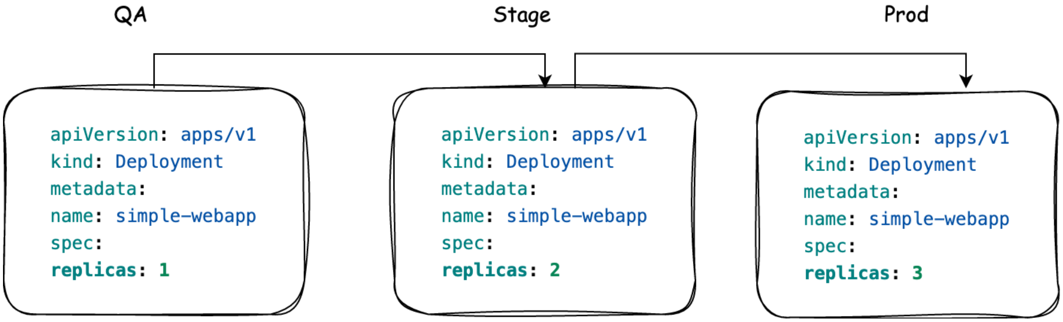
Throughout my career, I've only encountered one team that adopted this approach specifically to prevent junior developers from accidentally accessing and exposing sensitive data.

Folders for environments

Adopting the **environment-per-folder** approach in GitOps and Kubernetes involves organizing different environment configurations within separate folders of a single Git repository. Each folder represents a specific environment such as **development**, **staging**, or **production**. This structure allows for clear separation and management of configurations for each environment, facilitating easier updates and maintenance. It streamlines the deployment process in Kubernetes, aligning with the principles of GitOps by keeping all environment configurations in a unified repository, ensuring consistency and traceability of changes.

To effectively create a folder structure for Kubernetes, start by understanding your business needs, such as developing highly available portals for different countries and their specific version requirements. Then, integrate business-related values, such as a company logo, into all environments. Also, consider dynamic customer-related values, such as customer status levels such as *silver*, *gold*, or *platinum*. This approach ensures that your Kubernetes settings, such as the minimum replica count, are aligned with both general and specific business requirements.

We will use the same example we did previously for the environment-per-folder approach:



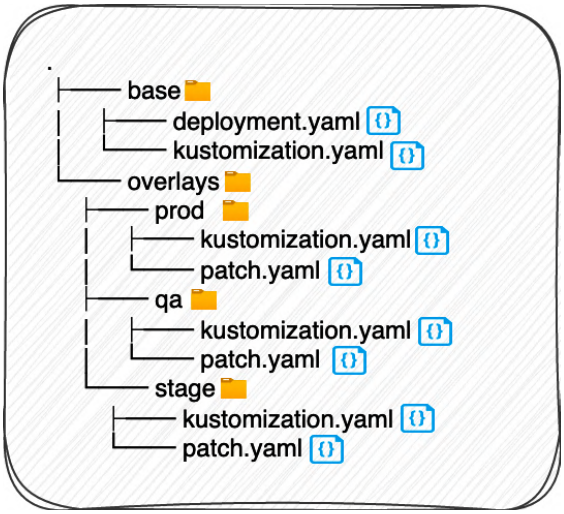
```
apiVersion: apps/v1
kind: Deployment
metadata:
name: simple-webapp
spec:
replicas: 1
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: simple-webapp
spec:
replicas: 2
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
name: simple-webapp
spec:
replicas: 3
```

Figure 5.9: How to propagate between stages with folders for environments

We will showcase the ease of change propagation across environments using **Kustomize**. The first step involves setting up a specific folder structure to facilitate this process:



```
.
├── base
│   ├── deployment.yaml
│   └── kustomization.yaml
└── overlays
    ├── prod
    │   ├── kustomization.yaml
    │   └── patch.yaml
    ├── qa
    │   ├── kustomization.yaml
    │   └── patch.yaml
    └── stage
        ├── kustomization.yaml
        └── patch.yaml
```

Figure 5.10: Example stages with folders for environments

Exploring the file structure, the base directory contains configurations that are shared across all environments and typically undergoes infrequent changes. For simultaneous modifications across multiple environments, the `base` folder is the ideal location to manage these changes as it provides a centralized point for updates affecting various deployment settings:

./base/deployment.yaml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simple-webapp
spec:
  replicas: 2
  selector:
    matchLabels:
      app: simple-webapp
  template:
    metadata:
      labels:
        app: simple-webapp
    spec:
      containers:
        - image: "ghcr.io/la-cc/simple-webapp:1.0.1-stable" #specific
version
          name: simple-webapp
          env:
            - name: UI_X_COLOR #business-related values
              value: darkblue
            - name: SUBSCRIPTION_TIER
              value: silver
```

In this simple example, you can see that we cover all the points, such as the specific version, business-related values, and more from the previous description:

./base/kustomization.yaml:

```
resources:
  - deployment.yaml
```

The `base` folder, also known as mixins or components, contains configurations common to different environments. Its contents are defined based on what you consider to be shared characteristics across your environments, a decision guided by your application's specific needs. In our example, this folder includes configurations for the QA, staging, and production environments.

In this section, we'll modify the base deployment by applying QA-specific configurations using a `patch.yaml` file. This approach allows us to customize the base setup for the QA environment without altering the common deployment settings:

./overlays/qa/kustomization.yaml:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - ../../base
patches:
  - path: patch.yaml
namePrefix: qa-
commonLabels:
  variant: qa
```

The following code shows how to overwrite the color in the base with a patch and thus allow a stage that's specific for `qa` configuration:

./overlays/qa/patch.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simple-webapp
spec:
  replicas: 1
  template:
    spec:
      containers:
        - image: "ghcr.io/la-cc/simple-webapp:1.1.5-new-ui"
          name: simple-webapp
          env:
            - name: UI_X_COLOR
              value: aqua
```

Important note

The `kustomize` tool in Kubernetes allows you to customize your YAML configurations without having to modify the original files. It enables you to manage configuration variations in a more structured and scalable way by using patches, overlays, and other techniques to generate final configuration manifests.

You can easily check the patching changes by executing the `kustomize` CLI tool from inside the `..chapter05/chapter-5-effective-git-repository-strategies/folders-for-environments` folder with the following command, which will show that the base overlay has been patched:

```
kustomize build overlays/qa
```

In this stage, we will conduct tests on a specific version that incorporates business-related values. This version is intended to eventually become the new release for our "gold" customers in the future.

The following code block shows how to reference the base, link it to the correct patch, and set a corresponding name prefix for all generated resources:

./overlays/stage/kustomization.yaml:

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
  - ../../base
patches:
  - path: patch.yaml
namePrefix: stage-
commonLabels:
  variant: stage
```

This process follows a similar patching method to what's used for QA.

The following code block shows how to overwrite the color in the base with a patch and thus allow a stage that's specific for stage configuration:

./overlays/stage/patch.yaml:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: simple-webapp
spec:
  replicas: 2
  template:
    spec:
      containers:
        - image: "ghcr.io/la-cc/simple-webapp:1.1.4-feature-login"
          name: simple-webapp
```

The folder-based approach in GitOps offers a streamlined overview of all stages, eliminating the need to switch and compare across different branches. This simpler example, however, only scratches the surface. In practice, you would manage a variety of files and environment-specific configurations. The structure's adaptability also allows for expansion into region-specific environments such as `qa-europe` or `qa-asia`, enabling customization based on unique regional requirements and business objectives.

The following scenarios showcase how mighty this approach is:

Important note

This process is an external workflow to Argo CD that greatly simplifies deployment management by enabling easy comparison and propagation between stages.

- **Use case 1**: Find a specific diff between QA and staging:

  The `diff` command in Linux compares two files or directories, showing the differences in their content in a line-by-line format.

  You can see the difference between the QA and staging environments by running `diff qa/patch.yaml stage/patch.yaml`.

- **Use case 2**: Promote the application version from QA to staging:

Important note

The `cp` command in Linux is used to copy files and directories from one location to another.

  I. First, copy the new version file with `cp qa/version.yaml staging-us/version.yaml`.
  II. Then, commit and push the changes to Git.

- **Use case 3**: Initially test new business-related values, and upon successful testing, incorporate them into the common settings:

  I. First, add the new specific config to QA.
  II. Then, promote the change from QA to staging.
  III. After that, commit and push the changes to Git and test the changes.
  IV. After testing, you can promote the change from staging to production.
  V. Again, commit and push the changes to Git.
  VI. Now, all stages have the same specific config. It's no longer a specific config to a specific stage. This means you can promote the change to `base`.
  VII. Now, again, commit and push the changes to Git.

VIII. At this point, you can remove additional config from QA, staging, and production because the config already exists in `base`.

IX. For the final time, commit and push the changes to Git.

This approach greatly simplifies managing deployments by allowing you to easily compare stages. You can quickly compare differences by selecting files or folders, eliminating the need for complex Git operations such as **cherry-picking**. Changes between stages, such as moving from QA to staging, are efficiently handled by copying and pasting files. The flexibility of this method extends to different regions, countries, Kubernetes distributions, and tools, limited only by your own requirements. It offers a customizable and adaptable solution for a variety of deployment scenarios and can be automated with any workflow mechanism.

What makes this approach particularly effective for Kubernetes and especially for scaling GitOps operations?

The suitability of this approach for Kubernetes and GitOps at scale is inherent in Kubernetes itself. Designed to be declarative and configuration-centric, Kubernetes seamlessly integrates with the structured, folder-based approach. This method supports GitOps' focus on version control and traceability, which is crucial for managing configurations effectively in large-scale Kubernetes environments. The approach's simplicity and organizational clarity make it an ideal match for the scalable and systematic deployment needs of Kubernetes and GitOps frameworks.

Scaling with ApplicationSet generators

We've already delved into the distinction between the App of Apps [5] approach and *ApplicationSets* in GitOps at scale with Kubernetes. Now, we will explore how to use *ApplicationSets* with generators to develop a streamlined developer platform, known as the KSC in this book. Specifically, we will demonstrate deploying an ingress controller across different clusters using *ApplicationSet*, each tailored with cluster-specific values. Our focus will be on using **Helm**, a prevalent tool for delivering configurable software. Helm's flexibility is showcased in the umbrella chart approach, where it can deploy either a single chart or a group of related charts. This can be integrated with tools such as **Helmfile** or Argo CD or used independently in a CD pipeline. An example of this is `kube-prometheus-stack`, which employs an **umbrella chart** with **subcharts**, such as Grafana.

Here's a brief overview of using an umbrella `ingress-nginx Chart.yaml` file:

```
apiVersion: v2
name: ingress-nginx #umbrella chart
version: 1.0.0
description: This Chart deploys ingress-nginx.
dependencies:
  - name: ingress-nginx #subchart
    version: 4.8.0
    repository: https://kubernetes.github.io/ingress-nginx
```

In the case of the `ingress-nginx` umbrella chart, using the same name for both the **umbrella** and **subchart** can lead to confusion, especially when overriding values. To address this, you should use global values for the umbrella chart and specify the subchart's name – for example, `ingress-nginx.controller.resources` – for specific overrides. This distinction is crucial for those unfamiliar with umbrella charts as many teams struggle with values not being applied as expected due to this naming overlap.

Now, let's understand why and which part allows us to build scalable deployments with Argo CD by looking at the following extract with `nginx-ingress-applicationset.yaml`:

```
spec:
  generators:
    - clusters:
        selector:
          matchLabels:
            env: prod
        values:
          branch: main
  template:
    metadata:
      name: "{{name}}-ingress-nginx "
    spec:
      sources:
        - repoURL: git@github.com:PacktPublishing/Implementing-GitOps-with-Kubernetes.git
          targetRevision: main
          ref: valuesRepo
        - repoURL: git@github.com:PacktPublishing/Implementing-GitOps-with-Kubernetes.git
          targetRevision: "{{values.branch}}"
          path: "./chapter05/chapter-5-building-a-service-catalog-for-kubernetes/networking/ingress-nginx"
          helm:
            releaseName: "ingress-nginx"
            valueFiles:
              - "values.yaml"
              - "$valuesRepo/chapter05/chapter-5-building-a-service-catalog-for-kubernetes/cluster/{{name}}/networking/ingress-nginx/values.yaml"
```

You can find the complete example in this book's GitHub repository via the name provided.

The concept of the generator, as previously explained, remains the same. The change lies in how the ApplicationController sources the umbrella chart (see **1** in *Figure. 5.11*), retrieves custom values for the specific target cluster (see **2** in *Figure. 5.11*), then uses Helm to template these values and deploys them to the target cluster (see **3** in *Figure. 5.11*). The following figure will further clarify the underlying processes, enhancing understanding of the operations taking place behind the scenes:

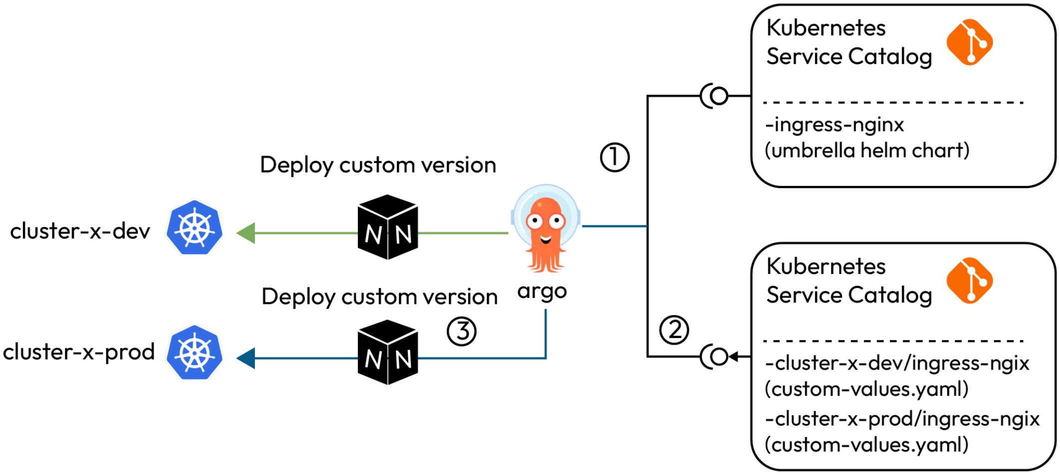


Figure 5.11: Example stages with folders for environments

This ApplicationSet, utilized by the application controller, employs generator clusters to modify the Helm charts, including both umbrella and subcharts. This methodology facilitates the implementation of GitOps at scale. Argo CD, a widely used tool in the market, plays a crucial role in enabling this approach, supporting the dynamic and scalable management of application deployments.

In the next section, we will see how to build KSG, which allows you to deploy different services from this catalog distributed across the clusters with a scaling approach.

Building a service catalog for Kubernetes

In this section, we will develop a service catalog for Kubernetes as a lightweight IDP. This platform will streamline providing necessary infrastructure through Kubernetes deployments. The lightweight IDP is designed to scale Kubernetes cluster flexibly via GitOps as projects grow, ensuring rapid time to market. Additionally, it will facilitate the extension of services such as **security**, **FinOps**, and **service-mesh** as needed, while ensuring that clusters are up to date and simplifying their management, regardless of the number of clusters involved. This approach underscores the synergy between Kubernetes, GitOps, and scalable infrastructure management.

Before proceeding, it's important to establish your labeling or tagging strategy, especially if you prefer not to deploy the entire stack to every cluster. Additionally, providing the opportunity to create and expand a service catalog can be beneficial. In such cases, it might be advisable to deploy a basic Kubernetes cluster tailored for various specific purposes. One potential approach could be as follows:

Labels	Stack will be Deployed	Notes
`env: dev`, `staging`, `prod`	`none`, `plain cluster`	No stack will be deployed. Dev should only be used for testing purposes. It is not stable.
`core-basic: enabled`	`argocd`, `external-dns`, and `ingress-nginx`	Should be deployed with security-basic if you want to issue certificates over cert-manager.
`security-basic: enabled`	`cert-manager`, `acme-issuer`, `falco`, `kyverno`, `sealed-secrets`, `external-secrets`, and `rbac`	Should be deployed on every cluster to keep the cluster secure.
`monitoring-basic: enabled`	`grafana`, `victoria metrics`, `msteams-proxy`, `mailhog`, `stunnel`, `prometheus-node-exporter`, `prometheus-alertmanager`, `falco-exporter`, and `cluster-alerting`	Will deploy a monitoring stack without the logging stack.
`monitoring-medium: enabled`	`loki`, `promtail`, and `minio-loki-tenant`	This stack requires storage and should be deployed with `storage-basic: enabled`.
`storage-basic: enabled`	`minio-operator` and `nfs-subdir-provisioner`	This stack is needed for the `monitoring-medium` stack. You can also deploy the stack without the `monitoring-medium` stack to use object storage or NFS for other purposes.

Table 5.1: One potential approach of using labeling to manage different tech stacks

For instance, using an operator or a CI/CD pipeline based on your workflow, you can efficiently map and transfer labels from your Kubernetes cluster to your Argo CD cluster.

Building the service catalog

To avoid exceeding the scope, we will focus on a handful of services when building the service catalog, and we'll be using the following folder structure:

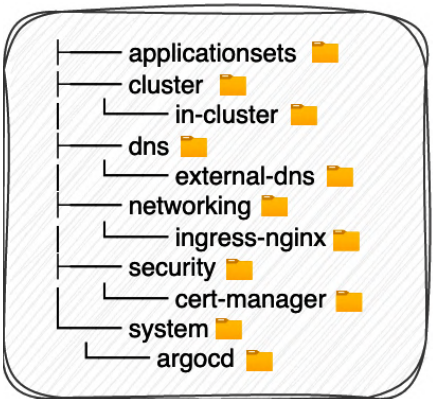


Figure 5.12: Example of possible services in the catalog

Here, `applicationsets` is a directory dedicated to deploying various services across multiple Kubernetes clusters using labels. Inside, the `cluster` directory contains multiple clusters, each iterated within the *ApplicationSet* and enhanced with parameters from the Cluster generator. Other directories, such as `dns`, `networking`, and others, group services logically. They comprise Helm umbrella charts along with subcharts for specific services, such as `external-dns` or `ingress-nginx`.

We will now review the well-known `nginx-ingress-applicationset.yaml` file, which has been updated to include the labels approach:

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: ingress-nginx
  namespace: argocd
spec:
  generators:
    - clusters:
        selector:
          matchLabels:
            env: dev
            core-basic: enabled
```

```
          values:
            branch: main
      - clusters:
          selector:
            matchLabels:
              env: prod
              core-basic: enabled
          values:
            branch: main
....
```

Now, `ApplicationController` deploys exclusively to clusters matching the `env = dev | prod` and `core-basic = enabled` labels. Structurally akin to other ApplicationSets for services shown in *Figure 5.11*, there is a unique aspect in cert-manager from the security folder, incorporating an additional label, `security-basic: enabled`, in its `ApplicationSet`.

The following code block shows how to implement it:

```
apiVersion: argoproj.io/v1alpha1
kind: ApplicationSet
metadata:
  name: cert-manager
  namespace: argocd
spec:
  generators:
    - clusters:
        selector:
          matchLabels:
            env: dev
            core-basic: enabled
            security-basic: enabled
        values:
          branch: main
    - clusters:
        selector:
          matchLabels:
            env: prod
            core-basic: enabled
            security-basic: enabled
        values:
          branch: main
...
```

Now, Argo CD's application controller only deploys `cert-manager` to clusters tagged with the specific additional label.

Labels are powerful tools that allow you to manage different stacks for on-premises and public cloud environments through a single central Argo CD unit. This versatility allows for seamless management across multiple environments. Integrating this with the varied perspectives of different roles, labels facilitate the following:

- Platform engineers can employ this method to deliver scalable Cluster-as-a-Service while incorporating SRE principles
- DevSecOps engineers can implement policies across all clusters, ensuring governance and compliance
- Developers can utilize this approach to deploy scalable applications across various customer clusters with customized values

This unified approach streamlines operations and ensures consistency across diverse environments and roles.

Should you possess extra manifests tailored to your specific requirements where the service needs to be extended – for example, if Argo CD requires an optional ingress during deployment to a target cluster for external access over ingress, and not direct via the Kubernetes API – you can introduce an additional `templates` folder under `system.argocd`. In this folder, you can include a manifest similar to the following extract from `templates.ingress.yaml`:

```
{{- if .Values.ingress -}}
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: argocd-server-ingress
  annotations:
    kubernetes.io/ingress.class: {{ .Values.ingress.className }}
    ingress.kubernetes.io/force-ssl-redirect: "true"
    nginx.ingress.kubernetes.io/ssl-passthrough: {{ .Values.ingress.
sslPassthrough | default "false" | quote }}
    cert-manager.io/cluster-issuer: {{ .Values.ingress.issuer }}
    cert-manager.io/renew-before: 360h #15 days
    cert-manager.io/common-name: {{ .Values.ingress.host }}
    kubernetes.io/tls-acme: "true"
    nginx.ingress.kubernetes.io/backend-protocol: "HTTPS"
spec:
  rules:
    - host: {{ .Values.ingress.host }}
…
```

This approach enables you to override the values in the umbrella chart and, if necessary, deploy an additional ingress, like so:

```
#overwrite helm umbrella chart values:
ingress:
  host: argocd.your-domain.com
  issuer: "letsencrypt"
  className: "nginx"
```

If your organization uses common certificates from a root CA required by all external DNS, you could utilize `kustomize` to establish a folder structure. For instance, you could create `kustomize.dns.external-dns-secrets` with your `root-ca.yaml` file. Then, you could integrate this into your `external-dns` ApplicationSet using just a few lines, like so:

```
...
      sources:
        - repoURL: git@github.com:PacktPublishing/Implementing-GitOps-
with-Kubernetes.git
          targetRevision: "{{values.branch}}"
          path: "./kustomize/dns/external-dns-secrets"
        - repoURL: git@github.com:PacktPublishing/Implementing-GitOps-
with-Kubernetes.git
...
```

With that, we have successfully created a catalog that will broaden the scope of our services, policies, and application portfolio.

A crucial point is focusing on the overarching approach to achieving GitOps at scale, rather than fixating on specific tools such as Argo CD. Tools may come and go, such as Flux or successors to Argo CD, but what remains vital is an adaptable strategy. This is akin to development practices evolving across different frameworks and languages while maintaining their foundational methods. By carefully organizing folders for various environments and merging this structure with an *ApplicationSet*, an efficient and scalable GitOps solution is formed. Such a strategy not only streamlines management but also significantly reduces the maintenance required for any given environment. KSC describes the power of joining CNCF/OSS projects together to create a secured, self-managed serving platform for developers, platform engineers, and SREs.

Bonus – maintenance with GitOps at scale and KSC

Let's say that you're managing hundreds of Kubernetes clusters, each with different stacks, and in every cluster, a core service is deployed. This results in approximately 2,000 applications being distributed across the clusters. The key requirement is to keep everything up-to-date and secure. You've scaled up brilliantly and kept your applications in sync. Suddenly, on the same day, two critical **Common**

Vulnerabilities and Exposures (**CVEs**) for the Ingress-NGINX image you're using are exposed, along with a bug fix for Cert-Manager and an extra configuration for External-DNS to enhance its resilience. All your clusters are affected. How do you maintain all clusters at once, considering everything is managed through Helm charts? Here's how:

- You could manually check all the Helm chart versions, look at the deltas, then upgrade the versions and deploy them across all clusters using Argo CD. However, CVEs and changes don't always get published at a specific time; they're released when a vulnerability is discovered.
- You have a pipeline or a job in your Kubernetes cluster that runs against your service catalog at set intervals to check for newer versions of a Helm chart. If a new version is found, it creates a pull request, displays all the changed values, and automatically merges if all criteria are met or after a review.

The second option sounds much better, and there is a script you can run in a pipeline that does precisely this. Alternatively, you can use a GitHub Action for this purpose.

This is what a pull request looks like:

```
Name: External DNS
Version in Chart.yaml: 6.23.6
Latest Version in Repository: 6.26.4
between     /tmp/tmp.aJBFfh/diff_value.yaml
and /tmp/tmp.aJBFfh/diff_latest_value.yaml


     _        __  __
   _| |_   _ / _|/ _|
 / _' | | | | |_| |_
| (_| | |_| |  _|  _|
 \__,_|\__, |_| |_|
|___/
returned three differences
image.tag
  ± value change
    - 0.13.5-debian-11-r79
    + 0.13.6-debian-11-r28
```

Afterward, the GitOps approach takes over, rolling out the changes across all clusters. This way, you minimize maintenance by centrally managing changes and deploying them via GitOps, regardless of the number of clusters involved.

The following section is about how to implement a multi-tenancy concept using only the board resources that Argo CD provides.

Exploring native multitenancy with Argo CD

This chapter is not about setting up the most secure, optimal multitenancy environment with Argo CD. That's because *best* is based on your specific SRE motivations, security team constraints, governance, the compliance policies of your company, your industry, and the skill level of your team. Additionally, tools change frequently, often multiple times a year, with minor releases introducing new features. Therefore, our focus here is on the approach to creating multitenancy with Argo CD while considering the aspects you should pay attention to, what you can address now, and future considerations.

But why opt for a multitenancy setup with Argo CD instead of using a dedicated cluster for each project?

There are several reasons:

- Reducing the count of clusters makes it easy to maintain aspects such as upgrading Kubernetes versions.
- To maximize resource utilization and efficiency, multitenancy is essential.
- In non-dynamic, ticket-based cold start environments, especially when not every Kubernetes cluster operates in the cloud, multitenancy becomes a necessity.
- By implementing FinOps practices, each machine's cost is closely monitored and optimized. Multitenancy helps reduce overall costs by maximizing the utilization of resources across multiple teams or projects, ensuring efficient spending and minimizing waste.

In this section, you will learn how to implement a multitenancy setup using Argo CD as the central management tool for workloads from different teams.

The setting for this setup is a Kubernetes platform in a company-owned data center, constrained by resources such as storage. A core Argo CD instance, provided by the platform team, is employed for the GitOps approach. This core instance delivers platform context (such as **ExternalDNS**, **Cert Manager**, and more) declaratively. Developer teams maintain their own Git repositories, with access to the Kubernetes cluster facilitated through **Active Directory** (**AD**) groups.

The challenge lies in deploying over the same core Argo CD instance without allowing teams to misuse it and break out of their isolated environments.

The following are the requirements to enable secure multi-client operation:

- Teams should work autonomously
- Each team can only access its designated namespace
- Teams must not negatively impact others, for instance, through improper resource provisioning
- A declarative GitOps approach is maintained
- GitOps at scale is managed declaratively for new projects and teams

Additionally, there needs to be a clear understanding of where the platform team's responsibilities begin and end. For this, you can consider the following approaches:

- The platform team provides the platform and context but stops at the initial repository setup:
    - **Pros**: No need to manage credentials such as **personal access tokens** (**PATs**) or **SSH keys**.
    - **Cons**: **Disaster recovery** is more challenging; developers must reinitialize access to the repo after, for instance, rotating a cluster
- The platform team provides the platform, context, and the initial repository:
    - **Pros**: Disaster recovery is simplified. A stateless cluster can be discarded or migrated, and the GitOps approach will restore everything correctly.
    - **Cons**: Managing credentials for different team repos becomes necessary.

By understanding and implementing these strategies, you can effectively manage a multitenancy environment in a resource-limited, Kubernetes-based infrastructure using Argo CD. The implementation of those strategies is based on the GitOps experience of the development and platform teams.

We will use the second approach from the view of the platform team and create a folder in our Git repository to be aggregated by Argo CD, with a specific folder structure. But first, let's look at *Figure 5.13*. Here, you can see that the different teams have access to a specific namespace, which is regulated by **RBAC**, **quotas**, and **network policies** and managed over a common Argo CD instance with projects and applications. The Argo CD instance is also used by the platform team to provide a new namespace as a service for new projects of the developer teams:

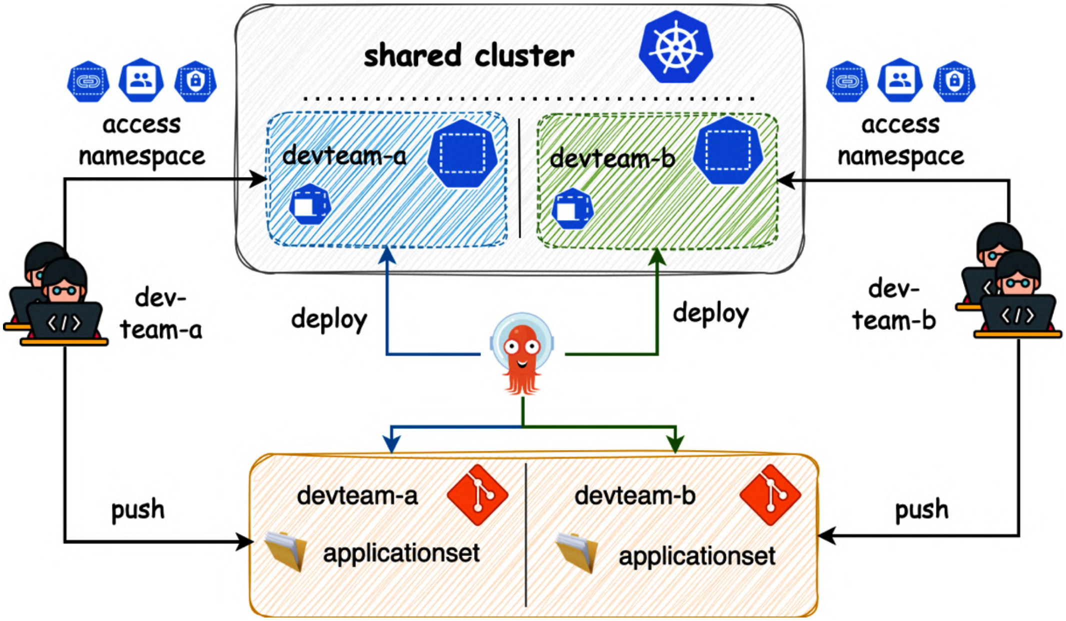
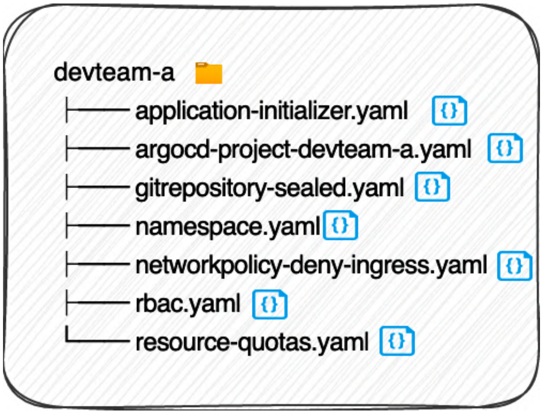


Figure 5.13: Multitenancy with GitOps and Argo CD

The folder structure that will be created for every team looks like this:



```
devteam-a
├── application-initializer.yaml
├── argocd-project-devteam-a.yaml
├── gitrepository-sealed.yaml
├── namespace.yaml
├── networkpolicy-deny-ingress.yaml
├── rbac.yaml
└── resource-quotas.yaml
```

Figure 5.14: Multitenancy with GitOps and Argo CD

A project on Argo CD represents the team's dos and don'ts. You can define the relevant set of rules by setting a project per development team. With this, the platform team gives just enough of a playground for the developers.

Let’s take a look at `argocd-project-devteam-a.yaml`:

```
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: devteam-a
  namespace: argocd  finalizers:
    - resources-finalizer.argocd.argoproj.io
spec:  description: Enable DevTeam-A Project to deploy new
applications  sourceRepos:
    - "*"
  destinations:
    - namespace: "devteam-a"
      server: https://kubernetes.default.svc
  # Restrict Namespace cluster-scoped resources from being created
  clusterResourceBlacklist:
    - group: ""
      kind: "Namespace"
  # Restrict namespaced-scoped resources from being created
  namespaceResourceBlacklist:
    - group: "argoproj.io"
      kind: "AppProject"
    - group: "argoproj.io"
      kind: "Application"
    - group: ""
      kind: "ResourceQuota"
    - group: "networking.k8s.io"
      kind: "NetworkPolicy"
```

In this context, `AppProject` is structured simply, encompassing elements such as `sourceRepos`, `destinations`, `roles`, `clusterResourcesBlackList`, or `WhiteList`, and `namespaceResourceBlacklist` or `WhiteList`.

The operational flow involves determining the origins of resources, the clusters they can be deployed to, and the namespaces within a project where deployment is permitted. It defines which resources and namespace resources can be created within the cluster by the project, with options to either allow or explicitly disallow (`!`) actions.

Global projects can be set up to distribute configurations to other projects. This means you can implement cluster-wide restrictions that are inherited by other *AppProjects*, eliminating the need to replicate the entire configuration block for each project.

Project roles can be used to grant or deny access to specific resources within the project. Global roles (`argo-rbac-cm`) and team or project roles are also used here. However, if a user's Kubernetes RBAC permissions are more extensive than those defined by the project, the project settings won't limit their ability to access or modify resources. Therefore, it's essential to constrain user rights at the RBAC cluster level.

Consider the following example:

```
destinations:
    - namespace: "!kube-system"
      server: https://kubernetes.default.svc
    - namespace: "devteam-a-*"
      server: https://kubernetes.default.svc
    - namespace: "*"
      server: https://kubernetes.default.svc
```

Important note

Here, you have explicitly forbidden deployment to `kube-system` while allowing deployments to the `devteam-a` namespace. However, this also inadvertently permits deployments to other namespaces, such as `devteam-b`, because they don't match the deny pattern. This scenario underscores the importance of a deep understanding of Kubernetes and the relevant tools for implementing a multitenancy approach with GitOps at scale.

Now, let's examine how you would set up an application in this environment:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: application-initializer-devteam-a
  namespace: argocd
spec:
  project: devteam-a
  source:
    repoURL: https://dev.azure.com/ORGA-X/devteam-a/_git/application
    targetRevision: main
    path: ./applicationset
  destination:
    server: https://kubernetes.default.svc
    namespace: devteam-a
…
```

The resources are collected by the Argo CD core instance and deployed via the GitOps approach. At this point, the platform team takes care of applications and doesn't allow the developer to deploy their own applications (see the limitations). You should be careful because restriction to create resources is not the same as manipulating resources. So, you might have to configure additional policies and **WebHook** validations, depending on your environment, corporate governance, and security guidelines.

`devteam-a` has a folder named `applicationset` in their repository. The folder represents the start of Argo CD. The developers can determine the order and deploy their application over `kustomize`, Helm, or a direct Kubernetes manifest file. In this configuration, developers are unable to create custom resources such as applications, ApplicationSets, and AppProjects.

This approach to multitenancy with GitOps presents certain limitations:

- **Limited team access to the Argo CD UI/CLI**: Teams typically can't create and manage applications through Argo CD's UI or CLI. If you wish to provide such access, you will need to use a Dex server, create policies in the **RBAC ConfigMap**, map groups to roles, or utilize project roles.
- **Restricted access to monitoring stacks**: Teams may not have access to a monitoring stack. Implementing multitenancy at this level is also necessary.
- **Potential conflicts with CRD versions**: If two teams opt to deploy different versions of CRDs for the same service, these conflicts must be mitigated. One way to do this is by blocking such actions and handling them through pull requests.
- **Challenges with declarative management**: The concept of declarative management can conflict with multitenancy principles. This is because Argo CD requires that custom resources, such as applications, be in the same namespace as Argo CD itself. A beta feature currently in development may allow applications to be deployed in different namespaces.

When opting for multitenancy with GitOps, you can conserve hardware resources. However, it's important to consider that the engineering resources required to maintain and enforce multitenancy setups might increase. GitOps can simplify management and enforcement, but it also comes with its own set of constraints, depending on the tools used and the underlying Kubernetes version and core approach.

This is why *AppProject* is not enough for multi-tenancy. To get the full GitOps experience for development teams, Argo CD by itself needs some enforcement. Due to the limits described here, some tools can be used to reduce them. We'll cover this in the next section.

Exploring multitenancy with vCluster and Argo CD

In the previous section, we explored how multitenancy is implemented natively with Argo CD, along with its current limitations at the time of writing. While these limitations may be addressed in the future, there's no certainty.

In this section, we'll expand the multitenancy approach by introducing a tool called vCluster. This tool addresses most of the limitations discussed earlier by enabling scaling through GitOps in a declarative manner, facilitated by Argo CD. This method shifts from a Namespaces-as-a-Service model to a more comprehensive **Virtual-Kubernetes-as-a-Service** (**VKaaS**) or **Kubernetes-as-a-Service** (**KaaS**) approach.

vCluster, an open source solution for creating and managing virtual Kubernetes clusters, presents a novel approach to Kubernetes cluster management. It functions by establishing a host cluster, on top of which vClusters – akin to seed clusters – operate within namespaces. This setup allows multiple clusters to be deployed. Necessary Kubernetes components and an additional **syncer** operate as pods within the namespace, offering a unique virtual cluster with its dedicated API. These virtual clusters, although running within a single namespace of a host cluster, provide the illusion of being independent, full-fledged clusters. This is particularly useful in scenarios where namespace limitations are a concern, and specific configurations incompatible with the host cluster's multitenancy setup are needed.

The focus here is not just on the tool itself, but on how it enables us to meet the requirements outlined in the previous section. We will maintain the same requirements and explore how vCluster can overcome the previously discussed limitations.

In our context, a **tenant** (*Figure 5.15*) refers to a project managed by vCluster on dedicated nodes, each with its own Argo CD instance. Thus, in our framework, every project is a tenant, and every tenant corresponds to a namespace, aligning with vCluster's namespace-based approach within the shared cluster:

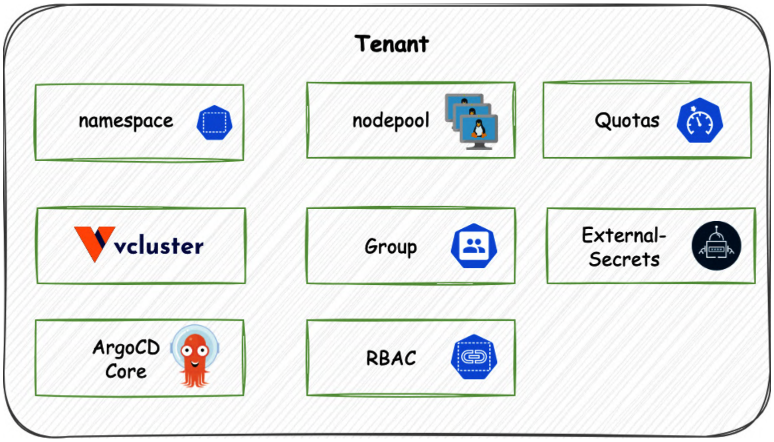


Figure 5.15: Example of how a tenant can be grouped by services

After this setup process, as illustrated in *Figure 5.16*, your system will feature a configuration where shared and isolated workloads exist simultaneously on the host cluster. The degree of this simultaneous existence hinges on how you apply taints and tolerations, along with other strategies, to guide Kubernetes' deployment decisions. This ability is key to distributing shared workloads throughout the cluster effectively.

The method we're about to delve into shows how to distinguish between the workloads of customer seed clusters, which operate on dedicated node pools, and those of the default node pool and host cluster. It's important to note that these host cluster workloads are vital for either Kubernetes or your platform team's operations, such as deploying the seed clusters:

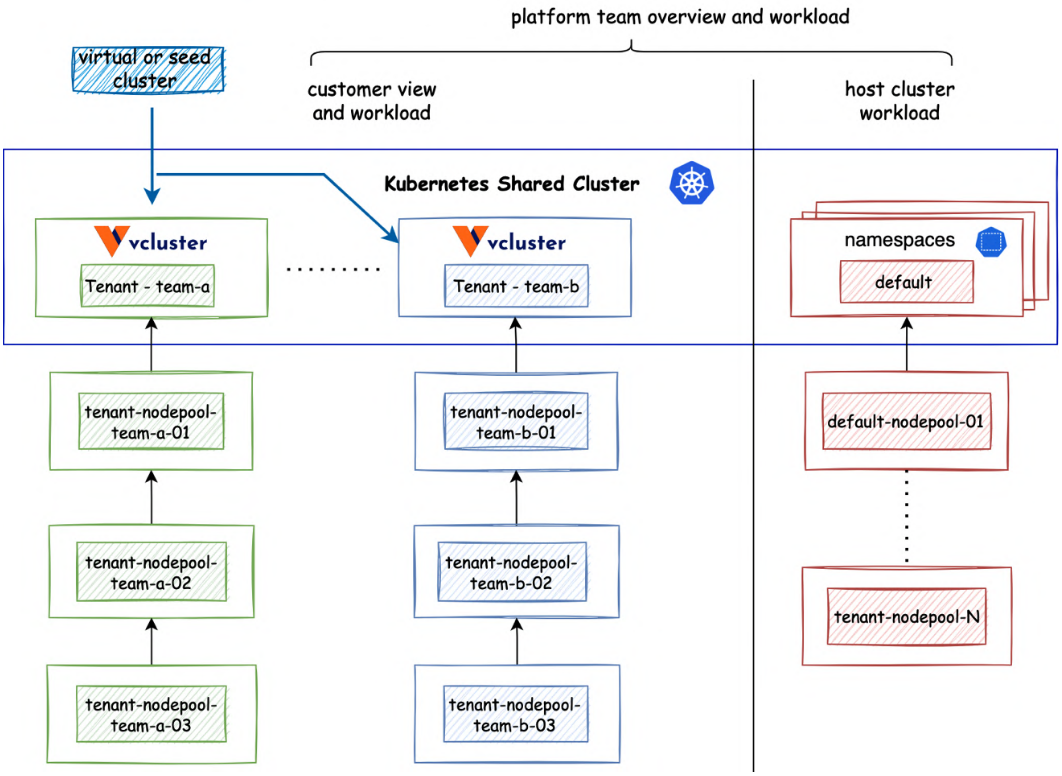


Figure 5.16: Multitenancy setup with vCluster

We will split the workflow into four steps. However, only the first two steps are necessary to build a multitenancy setup with vCluster:

1. **Step 1 – deploy vCluster**:

> **Important note**
>
> For this part, KSC is extended by optimization/vCluster.

The following code is an extract from the application and can be found on GitHub in the `vcluster-application.yaml` file, which focuses on the essentials. It is important to understand that several repos are referenced as sources, and a Helm release is created and overwritten with specific values:

```
  sources:
    - repoURL: git@github.com:PacktPublishing/Implementing-
GitOps-with-Kubernetes.git
      targetRevision: main
      ref: valuesRepo
  # - repoURL: git@github.com:PacktPublishing/Implementing-
GitOps-with-Kubernetes.git
  #   path: "./chapter05chapter-5-building-a-service-catalog-
for-kubernetes/cluster/vcluster-team-a/" #this part allows to
deploy additional manifest like rbac.yaml
  #   targetRevision: main
    - repoURL: git@github.com:PacktPublishing/Implementing-
GitOps-with-Kubernetes.git
      targetRevision: main
      path: "./chapter05/chapter-5-building-a-service-catalog-
for-kubernetes/optimization/vcluster"
      helm:
        releaseName: "vcluster-team-a"
        valueFiles:
          - "values.yaml"
          - "$valuesRepo/chapter05/chapter-5-building-a-service-
catalog-for-kubernetes/cluster/vcluster-team-a/optimization/
vcluster/values.yaml"
```

Additionally, we'll deploy specific Kubernetes resources, such as `networkpolicy-deny-ingress.yaml` and `rbac.yaml`, to effectively manage access control at the namespace level. These resources are crucial for ensuring proper security and access protocols within the multitenancy environment.

2. **Step 2 – connect Argo CD running on the host cluster to the vCluster**:

    I. This step involves setting up a multitenancy environment using vCluster and Argo CD. The focus is on establishing a connection between Argo CD, which operates on the host cluster, and the vCluster running on the host cluster. This process begins with connecting to the vCluster. You can achieve this with a single command. For a port forwarding connection, the following command can be used:

```
vcluster connect vcluster-team-a -n vcluster-team-a
```

    II. When utilizing an ingress for SSL passthrough, you can establish a connection using the following command:

```
vcluster connect vcluster-team-a \
-n vcluster-team-a \
--server=https://vcluster-team-a.example.com
```

Remember to configure certain vCluster and ingress-controller-specific parameters to enable SSL passthrough.

    III. Once this is set up, connecting Argo CD on the host system to the vCluster is a straightforward process and can be achieved with a single command line:

```
argocd cluster add vcluster_vcluster-team-a \
--label env=vdev \
--upsert \
--name vcuster-team-a
```

Important note

At this stage, we have effectively addressed the challenge of multitenancy. Teams are now equipped to connect to the virtual cluster using the familiar `vcluster connect` command. This capability is crucial in fostering self-sufficiency among teams.

3. **Step 3 – point to KSC**:

    Here, the focus shifts to integrating the virtual Kubernetes environment with a service catalog. This step involves creating an application that points to a repository containing a suite of applications specifically designed for the virtual cluster. This is like the application example provided earlier. By establishing a **virtual KSC** (**vKSC**), you can delineate the difference between various environments and adopt your approach with labels to deploy in a manner akin to how you manage dedicated Kubernetes clusters.

    To implement this, you can follow the step-by-step guide available on my blog [*7*]. This guide provides detailed instructions and insights into building a vKSC. This resource is particularly useful for understanding how to effectively manage and deploy applications in a multi-tenant setup using vCluster, ensuring a smooth and scalable operation within your Kubernetes environment.

4. **Step 4 – deploy Argo CD into vCluster while running Argo CD on the host cluster**:

   While setting up a multitenancy environment with vCluster, we have already established a folder structure that represents our vKSC. This structure is crucial for organizing various services and applications within the vCluster. After integrating Argo CD into this service catalog, similarly to other services in the KSC, we must now use Argo CD running on the host cluster to deploy Argo CD across all clusters using GitOps. This approach allows for an automated and consistent deployment process. It helps in avoiding code redundancy and enables individual patching of manifests for each cluster. This setup ensures efficient management and deployment in a multi-tenant Kubernetes environment by uniformly deploying Argo CD across all clusters.

Bonus – simplified connection to multiple vClusters – a handy bash script

Managing multiple vClusters can often become a cumbersome and noisy process, particularly when it involves connecting to each cluster individually. To address this challenge and streamline the workflow, I have developed a straightforward bash script. This script simplifies the process significantly. It operates based on the context of the host cluster and utilizes a consistent naming pattern, such as a namespace named `vcluster-demo` and an ingress formatted like `vcluster-team-a.example.com`. With this script, you can efficiently iterate over and establish connections to all vClusters, thereby saving considerable time and effort in managing your multi-cluster environment

Limitations solved in multitenancy with GitOps – a review

The implementation of multitenancy using vCluster and Argo CD, as outlined in the earlier steps, addresses several key limitations:

- **Limited team access to Argo CD UI/CLI**: This can be resolved by implementing *Steps 3* and *4*. Each team receives an Argo CD instance, complete with UI and CLI access, enhancing autonomy and operational efficiency.
- **Restricted access to monitoring stacks**: This can be solved by deploying an additional monitoring stack within vCluster. This step ensures that teams have the necessary monitoring tools at their disposal for effective cluster management.
- **Potential conflicts with CRD versions**: This is no longer an issue because each virtual Kubernetes cluster maintains a key-value store with distinct CRDs. This separation effectively eliminates conflicts arising from CRD version discrepancies.
- **Challenges with declarative management**: This is addressed through the platform team's ability to provision virtual clusters over Argo CD, enabling teams to manage their dedicated vClusters effectively. From a developer's perspective, following the implementation of *Steps 3* and *4* (refer to the Exploring multitenancy with vCluster and Argo CD section) ensures a smooth declarative management process.

However, it's important to note the inherent trade-offs in this approach. While this concept appears resource-efficient, especially compared to the dedicated approach of one cluster per project/team, vCluster does consume additional resources compared to the native approach. Most resources from the virtual cluster can be synced into the host cluster, and **loft.sh**, the creator behind vCluster, is working on expanding this bidirectional synchronization. There's also a vCluster Pro enterprise version offering further enhancements such as custom syncs between the host and vCluster, hibernating vClusters, creating templates, and more. Nevertheless, the focus here is not strictly on the tool but on the approach – how to effectively implement a multitenancy strategy using GitOps at scale.

As this chapter draws to a close, the following section summarizes the key points.

Wrapping up – insights and lessons from multitenancy experiences

As we wrap up this chapter, it's insightful to revisit the key themes and lessons that have emerged. Beginning with the *App of Apps approach*, we set the stage for understanding the complexities of managing applications in a Kubernetes environment. This approach emphasized the importance of a structured and scalable method to handle application deployment and orchestration.

Moving on to *multi-cluster management*, we explored the intricacies of managing numerous Kubernetes clusters, a critical aspect for organizations operating at scale. This exploration was complemented by the section on effective Git repository strategies, where the focus was on optimizing the management of repositories to enhance operational efficiency in a GitOps-centric environment.

The journey further unfolded with *ApplicationSet generators* and *building a service catalog for Kubernetes*. These sections delved into the techniques and tools necessary for effectively scaling applications and services across multiple Kubernetes clusters, underscoring Kubernetes' inherent scalability and flexibility.

The *Native multitenancy with Argo CD* and *Multitenancy with vCluster and Argo CD* sections provided a thorough understanding of different methodologies and tools for achieving efficient multitenancy. They highlighted how multitenancy can be implemented and managed using Argo CD and vCluster, offering insights into creating isolated, efficient multi-tenant environments.

Throughout this exploration, several key lessons and insights emerged:

- The strategy often outweighs the choice of tools, reinforcing the importance of approach over specific technologies
- Implementing multitenancy natively can save hardware resources but may increase the demand for engineering resources
- The use of patterns such as App of Apps and ApplicationSets can greatly aid in building scalable deployment strategies for different roles and use cases

- Managing multiple clusters with Argo CD is simplified, especially when combined with the effective use of labels and a GitOps approach
- Labels, when used with ApplicationSets and Cluster generators, can facilitate selective and flexible deployment strategies across clusters
- While native multitenancy approaches appear resource-efficient, they can introduce complexities and necessitate more engineering resources
- Tools such as vCluster offer a more isolated approach to multitenancy while maintaining declarative management and utilizing GitOps
- The balance between conserving hardware resources and the increase in engineering and developer onboarding efforts needs to be carefully managed
- GitOps at scale empowers platform engineering teams and developers, allowing them to focus on application development rather than operational burdens

In concluding this chapter, it becomes evident that understanding the underlying approach opens a myriad of possibilities for creating diverse and impactful real-world solutions and products.

These include products:

- **Platform as a Service (PaaS) products**: As a platform engineering team, understanding these concepts enables the creation of a PaaS product. This platform offers a suite of tools and services that are essential for streamlined application development and deployment.
- **Software as a Service (SaaS) solutions development for diverse clients**: By leveraging GitOps and Argo CD, developers can create customizable SaaS solutions that are easy to deploy across different Kubernetes clusters in various regions and versions. This approach ensures automated, consistent deployment, allowing developers to efficiently cater to the unique requirements of a diverse client base.
- **Deployment solutions for SRE teams**: SRE teams can leverage this knowledge to improve their deployment strategies, ensuring high availability and efficiency in their operations.
- **Governance and compliance tools for security teams**: Security teams can use these strategies to implement robust governance and compliance measures across all clusters. By utilizing labels effectively, they can establish comprehensive service packages that ensure adherence to security standards and regulatory requirements.

These applications demonstrate the versatility and real-world impact of the strategies discussed in this chapter, highlighting how a deep understanding of GitOps and multitenancy can lead to the creation of diverse, scalable, and secure technological solutions.

Summary

In conclusion, this chapter has not only illuminated various strategies and tools for achieving multitenancy and scaling in Kubernetes but also highlighted the crucial role of understanding these concepts deeply. The journey through GitOps at scale and multitenancy reveals that while there are multiple approaches to achieving efficiency in Kubernetes, each comes with its trade-offs. The ultimate choice should be aligned with the organization's specific needs and the goals of its development teams.

In the next chapter, we will introduce different architectures that have already been partially utilized in this chapter to illustrate concepts such as *One cockpit rule them all* approach for multi-cluster management, as well as discuss their advantages, disadvantages, use cases, and insights gathered from various projects.

References

- [*1*] `https://docs.akuity.io/tutorials/cluster-addons-with-applicationsets/`
- [*2*] `https://argocd-applicationset.readthedocs.io/en/stable/Generators/`
- [*3*] `https://codefresh.io/blog/argo-cd-best-practices/`
- [*4*] `https://codefresh.io/docs/docs/pipelines/deployment-environments/`
- [*5*] `https://codefresh.io/blog/codefresh-gitops-app-of-apps/`
- [*6*] `https://www.vcluster.com/docs/using-vclusters/access`
- [*7*] `https://medium.com/devops-dev/multi-tenancy-with-vcluster-794de061fff1`

6

GitOps Architectural Designs and Operational Control

In the rapidly evolving landscape of cloud-native technologies, understanding and effectively implementing various architectural frameworks becomes crucial for organizations seeking to harness the full potential of Kubernetes. As we dive deeper into this topic in this chapter, we will explore diverse architectures that not only enable multi-cluster management – a theme partially introduced in the previous chapter – but also facilitate effective **GitOps** implementations for service and product deployments utilized by various companies.

Our journey will take us through real-world scenarios and practical insights from projects that have employed different architectural approaches. By examining how various companies have successfully integrated GitOps methodologies to deploy and manage their services and products, we gain valuable perspectives on what works in different contexts. This chapter will particularly benefit **platform engineers**, **SREs**, and **internal developer platform** builders as it focuses on deploying various operational models used by teams to provide their workloads or the platform context.

We will delve into the nuances of managing Kubernetes clusters and workloads using tools such as **Argo CD**, **Flux CD**, and **Cluster API**. These tools are at the forefront of enabling efficient and scalable management of Kubernetes environments.

By the end of this chapter, you will have a comprehensive understanding of how different architectural choices impact the effectiveness and efficiency of Kubernetes deployments, particularly in the context of GitOps. Whether you're a platform engineer crafting the infrastructure, an SRE ensuring its reliability, or a developer building internal platforms, the insights shared here will be invaluable in your pursuit of operational excellence in cloud-native environments.

As such, the following topics will be covered in this chapter:

- Exploring diverse GitOps architectural frameworks for Kubernetes environments
- Examining the impact of architectural choices on GitOps' effectiveness

- Tailoring designs for scalability, resilience, and efficiency in cloud-native deployments
- Centralized control – managing clusters with a solo Argo instance
- Dedicated instances – instance per cluster with Argo CD
- Dedicated instances – instance per cluster with Flux CD
- The middle way – instance per logical group with Argo CD
- The cockpit and fleet approach with Argo CD
- Centralized Kubernetes cluster creation – leveraging Cluster API and Argo CD for streamlined cluster deployment
- A deep dive into Cluster API and GitOps – hands-on

Exploring diverse GitOps architectural frameworks for Kubernetes environments

Exploring diverse GitOps architectural frameworks for Kubernetes environments is crucial for organizations looking to streamline their deployment pipelines and operational workflows. GitOps, a term coined by **Weaveworks**, emphasizes the use of Git as the single source of truth for declarative infrastructure and applications. In Kubernetes environments, this translates to a series of best practices and patterns that guide the management and automation of container orchestration.

Several architectural frameworks within GitOps cater to different organizational needs and technical contexts. The choice of framework often depends on the complexity of the environment, the scale of the operations, and the governance requirements.

The adoption of GitOps influences architectural decisions in Kubernetes in several ways:

- **Infrastructure as Code (IaC)**: With GitOps, the entire Kubernetes architecture is defined as code – typically YAML files that describe the desired state of the system. This approach enables developers and operations teams to collaborate on infrastructure changes, which can be versioned, reviewed, and audited just like application code.
- **Immutable infrastructure**: The architectural frameworks that embrace GitOps often prioritize immutability. Once a resource is deployed, it should not be changed manually in the running environment. Instead, any modifications are made in the Git repository, which triggers a deployment process to update the infrastructure.
- **Modular design**: GitOps encourages a modular approach to infrastructure. Each module, or set of Kubernetes resources, can be managed as a separate project within Git. This modularization aligns with Kubernetes' architectural philosophy of microservices, where each service can be deployed, scaled, and managed independently.

- **Automated deployment strategies**: Architectural frameworks under GitOps often incorporate advanced deployment strategies such as canary releases, blue-green deployments, and A/B testing. GitOps tooling automates the rollout and monitoring of these strategies, making it easier to implement them in a controlled manner.
- **Environment parity**: GitOps ensures that each environment – from development to staging to production – can be replicated with a high degree of fidelity. This is achieved by using the same declarative configurations across environments, reducing the "works on my machine" syndrome.
- **Security and compliance**: By defining architectural elements as code in a Git repository, GitOps enables the application of security policies and compliance checks as part of the **continuous integration/continuous deployment** (**CI/CD**) pipeline. This means that security becomes a part of the architecture by design, not an afterthought.
- **Single repository versus multiple repositories**: Some organizations opt for a single repository containing all configurations and applications, which simplifies management but may not scale well with large teams or complex applications. Others prefer multiple repositories and separating configurations and applications to provide finer-grained access control and clearer separation of concerns.
- **Push versus pull deployment models**: In a push-based model, changes are pushed from the repository to the Kubernetes clusters, often through a CI/CD pipeline. The pull-based model, conversely, involves a Kubernetes operator within the cluster monitoring the repository and pulling in changes when they're detected. While the push model offers immediacy, the pull model is praised for its alignment with the Kubernetes declarative philosophy and enhanced security posture.
- **Monolithic versus microservices architectures**: When it comes to application architectures within Kubernetes, GitOps can be applied to both monolithic and microservices patterns. Monolithic architectures may be easier to manage through GitOps due to their singular nature, but microservices architectures benefit from GitOps through the ability to independently deploy and scale services.

The architectural frameworks for Kubernetes, empowered by GitOps, are evolving to facilitate more robust, scalable, and secure application deployments. Embracing GitOps not only streamlines the operational workflow but also enforces best practices in software architecture. As organizations adopt these frameworks, they must remain flexible and willing to adapt to the rapidly changing landscape of cloud-native technologies.

In the next section, we will look at the effects the choice of architecture has on working with GitOps.

Examining the impact of architectural choices on GitOps' effectiveness

GitOps is inherently aligned with Kubernetes' declarative approach to managing infrastructure, where the desired state of the system is described in code. This state is checked into a Git repository, which then serves as the single source of truth. The effectiveness of GitOps is contingent on how well the architectural choices support a declarative model that enables the following aspects:

- **Version control**: Tracking changes over time, providing a historical context, and enabling rollback to previous states
- **Change management**: Facilitating peer reviews and approvals for changes to infrastructure code, enhancing the quality and security of deployments
- **Automated synchronization**: Ensuring that the actual state of the system automatically converges to the desired state defined in the repository

Architectural choices impacting GitOps

When considering the impact of architectural choices on GitOps, several key factors come into play:

- **Repository structure**: Choosing between a monolithic (single repository) versus a multi-repository (one per service or team) approach can significantly affect the manageability and scalability of applications. A monolithic repository might simplify dependency tracking and versioning but could become unwieldy with scale. Multi-repository strategies enhance modularity and separation of concerns but require more sophisticated synchronization mechanisms.
- **Deployment strategies**: The architecture must support a variety of deployment strategies, such as canary, blue-green, or rolling updates. GitOps tools automate the execution of these strategies, and the choice of strategy can impact resource utilization, downtime during deployments, and the ability to test changes in production-like environments.
- **Environment isolation**: Architectural decisions on how to isolate and manage environments (development, staging, and production) will affect the GitOps workflow. Environment-specific configurations can be handled via separate branches, directories, or even separate repositories, each with implications for access control, traceability, and complexity.
- **Scalability**: As organizations scale, the architecture should facilitate a GitOps approach that can handle increased workloads, more complex deployments, and a growing number of services. This may involve partitioning clusters, adopting multi-cluster strategies, or leveraging cloud-native tools that specifically address scalability.
- **Security considerations**: Architectural choices must ensure that security is embedded in the GitOps workflow. This includes everything from securing access to Git repositories to encrypting sensitive data and automatically enforcing policies throughout the CI/CD pipeline.

Making informed architectural decisions

To ensure GitOps effectiveness, organizations must make informed architectural decisions:

- **Assess organizational needs**: Understand the organization's requirements in terms of scale, complexity, compliance, and team workflows
- **Evaluate tooling compatibility**: Select GitOps tooling that is compatible with the chosen architecture and can support the required deployment strategies
- **Promote collaboration**: Architectures should encourage collaboration between development, operations, and security teams to leverage the collective expertise in support of GitOps workflows
- **Continuously refine**: Architectural choices should be revisited and refined based on feedback from ongoing operations so that they can adapt to new challenges and opportunities

In conclusion, the architectural choices that are made when setting up Kubernetes environments have far-reaching implications for the success of a GitOps approach. By fostering an architecture that embraces version control, change management, and automated synchronization, organizations can leverage GitOps to enhance the agility and stability of their infrastructure. Making informed decisions about repository structures, deployment strategies, environment isolation, scalability, and security will position teams to harness the full potential of GitOps, leading to a more resilient and responsive infrastructure.

Tailoring designs for scalability, resilience, and efficiency in cloud-native deployments

Tailoring architectural designs to achieve scalability, resilience, and efficiency is fundamental for cloud-native deployments, where the dynamic nature of the cloud environment can present both opportunities and challenges. Cloud-native architectures enable systems to be resilient to failures, adaptable to changing loads, and efficient in resource utilization. When incorporating GitOps practices, these designs can be systematically enforced and continuously improved.

Scalability in cloud-native architectures

Cloud-native deployments are expected to handle varying loads gracefully. This flexibility is crucial for maintaining performance during demand spikes and optimizing costs during quieter periods. Here are a few ways you can achieve high scalability:

- **Horizontal scaling**: Architectures should be designed to allow for horizontal scaling, which involves adding more instances of an application to handle increased load
- **Microservices**: Breaking down applications into microservices enables individual components to scale independently, providing granular control over resource allocation

- **Stateless applications**: Stateless applications are inherently more scalable since any instance can handle any request, allowing for straightforward horizontal scaling

GitOps can manage the deployment and scaling of these services by automatically adjusting the number of instances based on the load, as defined in the Git repository.

Resilience through redundancy and isolation

A resilient system can withstand and recover from failures without significant downtime or data loss. Here are a few ways you can achieve stronger resilience:

- **High availability**: Architectures must be designed for high availability, with redundant components that can take over in case of failure.
- **Fault isolation**: Microservices architectures naturally lend themselves to fault isolation. A problem in one service should not cascade and cause system-wide failures.
- **Disaster recovery**: A robust backup and recovery strategy, along with multi-region deployments, can ensure that applications survive even catastrophic events.

In GitOps workflows, the desired state in the repository reflects these high-availability configurations, enabling the system to self-heal by automatically re-deploying failed components.

Efficiency with proactive optimization

Efficiency in cloud-native deployments is about doing more with less – less time, less resources, and less manual intervention. You can achieve early efficiency if the following aspects are optimized in advance:

- **Auto-scaling**: Implement auto-scaling policies to adjust resources in response to real-time metrics, ensuring efficient use of infrastructure
- **Load balancing**: Effective load balancing distributes traffic across instances to optimize resource utilization and ensure consistent performance
- **Resource limits**: Setting appropriate resource limits and requests in Kubernetes helps prevent any single service from consuming more than its fair share of resources, leading to a more efficient system

GitOps automates the process of enforcing these policies by triggering actions based on the configurations defined in the Git repository.

Tailoring designs with GitOps

Designing architectures with scalability, resilience, and efficiency in mind requires careful planning and the right set of tools to manage the deployment and operation of cloud-native applications. Consider the following aspects when tailoring the design:

- **IaC**: Define your infrastructure and policies as code to maintain a clear and auditable trail of how resources are allocated and managed
- **Observability**: Implement comprehensive logging, monitoring, and alerting to gain insights into the system's performance and health, informing decisions about design adjustments
- **Continuous improvement**: Use GitOps to continuously deploy updates and improvements to the architecture, ensuring it evolves to meet changing needs and challenges

Tailoring designs for scalability, resilience, and efficiency is vital for cloud-native deployments to thrive in the elastic and often unpredictable cloud environment. By leveraging GitOps, teams can ensure that these design principles are consistently applied across all environments, enabling them to respond quickly to changes and maintain robust, efficient systems. As cloud technologies continue to evolve, so too must the architectures and practices that support them, with GitOps providing a framework for that ongoing evolution.

In *Chapter 11*, we'll explore the practical use of different GitOps approaches to deploy real-world applications. The next section delves into the significance of application design in GitOps, emphasizing the importance of the operational setup. A team's focus shouldn't just be on deploying applications, but also on choosing the right GitOps instance strategy for effective deployment. Before implementing GitOps with tools such as Argo CD, you must carefully consider the required number of clusters. We briefly introduced single instance approaches in the previous chapter while focusing on scalability. The next section will examine various approaches, their real-world application by organizations, and the advantages and disadvantages that are experienced in these scenarios, with a particular emphasis on the operational control of GitOps instances.

Centralized control – managing clusters with a solo Argo instance

As organizations grow and evolve, the demand for scalable, secure, and efficient deployment strategies becomes increasingly critical. **Argo CD**, a key player in the GitOps field [*1*, *2*], stands out for its comprehensive capabilities in scaling across various aspects, such as performance, security, usability, and failover processes. In this section, we'll delve into the nuances of different architectural models and the intricacies of operational management. However, before we dive into these topics, it's essential to address a few preliminary points that significantly influence decision-making in this realm:

- **Security considerations**: Security in scaling with Argo CD involves a robust combination of **role-based access control** (**RBAC**) and **single sign-on** (**SSO**) mechanisms. A key security feature is the client-side rendering of manifests, which reduces the threat landscape. However, the power of tools such as **Helm**, **Kustomize**, and **Jsonnet** in manifest generation introduces potential risks as they allow arbitrary code to be executed. This necessitates careful consideration, especially in larger instances, to prevent abuse in manifest generation.

- **Usability at scale**: Argo CD's reputation for ease of use and extensibility remains intact even as it scales. However, managing applications across numerous Kubernetes instances can lead to complexities, such as the need for unique application naming conventions and the challenge of managing a large number of applications within a single UI instance. The tool's UI filters, while powerful, are not savable, leading teams to find creative solutions such as using bookmarks for saved filters.
- **Failover strategies**: The GitOps approach, embodied by Argo CD, facilitates rapid changes and recovery through simple git commits. However, this ease of setup and teardown also brings to light the potential for significant impact due to misconfigurations. For instance, a minor error in updating a config management plugin could lead to widespread application failures. This raises the question of the "blast radius" – the extent of impact that a single misconfiguration could have.
- **Performance scaling**: Argo CD is available in two versions: the standard and the **high-availability** (**HA**) version. The latter is specifically designed for scalability, deploying multiple replicas of key components such as the repo-server. Argo CD's scalability is evident in its capacity to support, without major adjustments, up to 1,500 applications, 14,000 objects, 50 clusters, and 200 developers. This benchmark, though conservative, accounts for variations in applications, such as their object count, manifest complexity, and update frequency. These figures, sourced from a KubeCon talk by *Joseph Sandoval* and from *Adobe and Dan Garfield from Codefresh* [*3*], provide a foundational guideline for planning scaling needs.

In summary, while Argo CD is a robust tool that's capable of handling the complexities of scaling in modern software environments, it requires careful consideration in terms of its performance capabilities, security risks, usability challenges, and failover strategies. The correct architectural choices, tailored to an organization's specific needs, can leverage Argo CD's strengths while mitigating potential risks associated with scaling.

The approach – centralized control

This approach may be familiar to you. In the managed cluster approach, a single Argo CD instance is utilized by various teams for different purposes. The platform team employs this shared instance to deploy the necessary platform context for other teams.

Important note

The term platform context, tailored to individual company needs, includes essential tools such as Ingress Controllers and Cert-Managers for effective Kubernetes platform operations.

Developers use the same Argo CD instance to deploy their applications. Similarly, **SRE teams** leverage it to deploy tools that help in identifying bottlenecks, analyzing performance issues, and more. The security team also makes use of this shared Argo CD instance to deploy its policies and enforce them through a policy engine. Additionally, they deploy security tools such as **kubeclarity** [4] or the **trivy operator** [5] to monitor vulnerabilities in images running in the cluster, track used packages, and check licenses. However, the significant change now is that instead of one Argo CD instance managing the platform and applications on a shared control and workload cluster, there is one Argo CD instance managing multiple clusters (*Figure 6.1*):

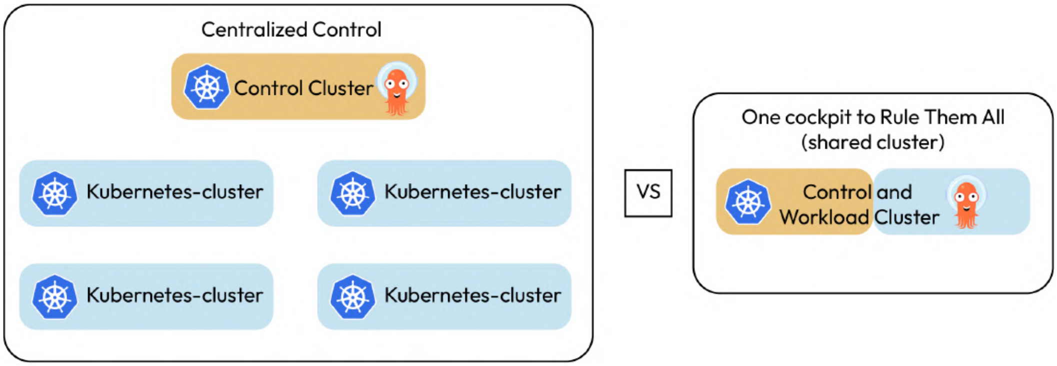


Figure 6.1 – Difference between centralized control and one cockpit to rule them all

The following table outlines the advantages and disadvantages:

Advantages	**Disadvantages**
Centralized view and control: Unified view for deployment activities across all clusters	**Scaling and performance**: Scaling necessitates tuning individual components
Simplified management: Managing a single Argo CD instance for multiple Kubernetes clusters eases administrative tasks	**Single point of failure**: Potential single point of failure for deployments
API/CLI integration: With only one server URL, API and CLI integration becomes more straightforward	**Security implications**: Centralization of admin credentials for all clusters

Advantages	Disadvantages
	Network traffic and cost implications: The application controller, responsible for Kubernetes watches, can incur significant network costs, especially if clusters are located in different regions

Table 6.1 – Advantages and disadvantages of the centralized control approach

The single control plane approach involves one Argo CD instance managing all clusters, a popular approach for offering a unified application view and enhancing developer experience.

For organizations that delegate access based on environment and are concerned about managing all applications under one instance, **RBAC** policies and **AppProjects** can establish necessary boundaries, defining deployment locations and access controls.

This architecture also mandates establishing and maintaining a dedicated management cluster to host the Argo CD control plane, with direct access to all other clusters. The location of this management cluster could lead to security concerns, especially if it involves public exposure.

The key question here is how to ensure tenant separation while maintaining collaboration. In the GitOps approach with Argo CD, built-in **custom resources** such as **projects**, **roles**, and **groups** are utilized to implement a multitenancy framework. However, determining who is responsible for implementing, maintaining, and extending this approach to optimize the shared Argo CD instance is crucial.

If the platform team is in charge, they must also manage the security, governance, and compliance aspects, especially if they have admin rights over the cluster. The feasibility of this depends on the team's resources and expertise. For instance, a well-resourced platform team with core knowledge of the platform and specialized skills in areas such as security and **FinOps** can manage this effectively. However, smaller teams may find it challenging to maintain security.

A solution some companies adopt involves collaboration between different teams:

- Platform team ↔ security team
- Platform team ↔ developers
- Platform team ↔ FinOps team
- Platform team ↔ SRE teams

The platform team is responsible for liaising with these teams and implementing commitments. They must also justify necessary changes enforced by the security team. This model works well for smaller companies with up to 30-50 mixed IT employees. Beyond 50 employees, the increased interaction between the platform team and developer/SRE teams can slow down development.

When to use the centralized control approach

The centralized control approach is often the initial choice for teams operationalizing Argo CD across many clusters. It's particularly effective for managing dev, staging, and production environments within a small team framework. The model supports high availability, scalable components, RBAC, and SSO, making it suitable for smaller-scale operations and straightforward network configurations.

When to avoid the centralized control approach

Larger organizations with multiple independent teams, extensive networks, or a need for high flexibility should be cautious. The model's potential for a large "blast radius" during critical failures and its limited flexibility with large numbers of users can be detrimental. For networks hosting Kubernetes in virtual private clouds or behind firewalls, while possible, the addition of network tunnels can add complexity.

In summary, the single control plane approach of the centralized control approach, while offering numerous advantages in terms of simplicity and ease of management, carries risks related to security, scalability, and performance. Organizations must weigh these factors carefully while considering their specific needs, team size, and network architecture before adopting this model.

The next section deals with dedicated Argo CD instances per Kubernetes cluster and the associated challenges.

Dedicated instances – instance per cluster with Argo CD

An Argo CD instance is installed and co-located with the cluster it manages, meaning each cluster has its own dedicated Argo CD instance. This approach provides several advantages and challenges. This section aims to help you understand and implement standalone Argo CD instances in your Kubernetes environment. Each Argo CD instance will be installed and co-located with the cluster it manages, providing a dedicated instance per cluster, as shown in *Figure 6.2*.

Each cluster and each team benefits from having a dedicated Argo CD instance. This means that every cluster can be configured, managed, and monitored independently, allowing for tailored management strategies that align with the specific needs of each cluster. But this approach also means that each Argo CD instance necessitates its own set of resources. Ensuring that each cluster has the necessary resources to support its Argo CD instance is crucial. This requirement calls for detailed planning and assessment to allocate resources efficiently and avoid potential shortages or imbalances:

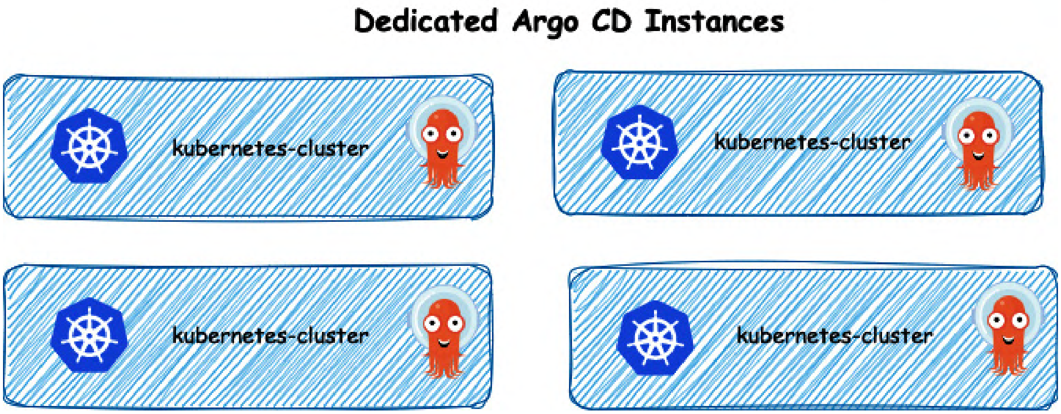


Figure 6.2 – Example of dedicated Argo CD instances per cluster

Let's take a look at some pros and cons to better understand this approach:

Advantages	**Disadvantages**
Reliability improvement: Each cluster operates independently, enhancing overall reliability	**Management complexity**: Each instance requires individual management and updates, increasing the overall complexity
Isolation of concerns: This setup offers better security as each cluster is self-contained	**Access complexity**: Providing access to users across multiple instances can be challenging
No external networking access required: Standalone Argo CD instances operate independently of external network access, which is crucial for edge deployments and even air-gapped environments where updates might occur via a USB drive	**Disaster recovery considerations**: Special planning is needed for disaster recovery due to the decentralized nature of the setup
Suitability for Edge deployments: Its standalone nature is ideal for clusters at the Edge, ensuring each operates completely independently	

Table 6.2 – Advantages and disadvantages of the dedicated instances approach

At first glance, it appears that the advantages would outweigh the disadvantages. Two significant benefits have been proven in practice over the years.

Heightened security is perhaps the most standout benefit. By employing individual Argo CD instances, security measures can be precisely tailored to meet the unique requirements and vulnerabilities of each cluster. This level of customization ensures that the security protocols are not only robust but also intricately designed to address specific threats, providing a fortified defense for each unique environment.

Another critical advantage is **isolated risk management**. In conventional setups, a single failure or breach could potentially escalate into a system-wide crisis. However, with dedicated Argo CD instances, such risks are contained within the affected cluster, significantly reducing the likelihood of widespread issues. This isolation of risk is vital in a landscape where a single vulnerability can lead to significant operational disruptions.

However, from practical experience, there is a disadvantage that outweighs many advantages, especially as the number of clusters and Argo CD instances increases and maintenance effort escalates significantly.

The maintenance overhead is another significant factor. Managing multiple Argo CD instances means that each one demands individual attention – from updates and configuration tweaks to regular monitoring. This increased workload can place a strain on IT teams, necessitating more robust and efficient management strategies to handle the additional administrative tasks.

When to use dedicated Argo CD instances

Standalone Argo CD instances are most beneficial in scenarios where the greatest reliability and accessibility are required, especially in situations where external networking is limited or non-existent. These instances are the default choice for deploying to clusters at the edge, given their complete operational independence. To scale these deployments, integration with infrastructure management tools such as Crossplane or Terraform is often employed to streamline setup and teardown. Another variation involves using a hub-and-spoke model to manage multiple standalone Argo CD instances across numerous clusters.

When to avoid dedicated Argo CD instances

However, standalone instances come with significant management overhead. In scenarios where a single team manages a simple staging-to-production workflow, it might not be efficient to separate Argo CD for each cluster. Testing and implementing changes across multiple instances also present challenges. Adopting a canary release approach to update a fleet of instances, while effective, adds another layer of complexity to the process.

In conclusion, the decision to use standalone Argo CD instances should be based on the specific needs and capabilities of the organization. While they offer improved reliability and security, the complexities in management, updates, and disaster recovery planning must be carefully considered. For certain environments, especially those at the Edge, the standalone approach is ideal, but for simpler setups or smaller teams, this approach might introduce unnecessary complexity.

Deciding to use an Argo CD instance per cluster depends on your organization's specific needs and capabilities. While they offer greater reliability and security, this approach requires careful consideration in terms of the complexities in management, updates, and disaster recovery. This strategy is highly effective for Edge deployments and environments with limited external networking but may be too complex for simpler setups or smaller teams.

The next section deals with dedicated Flux CD instances per Kubernetes cluster and the associated challenges.

Dedicated instances – instance per cluster with Flux CD

In the realm of GitOps tools, the distinction between Argo CD and **Flux CD** is crucial, particularly when considering their application in dedicated instances per cluster. While Argo CD is a well-known entity in the GitOps conversation, Flux CD holds a significant place, with a robust community and a substantial user base. This diversity of tools is essential to understanding the range of options available for Kubernetes cluster management.

My journey in GitOps began with Flux CD, a tool that served effectively over a long period, especially in projects where scaling and managing multiple clusters wasn't a requirement. This context-specific suitability of Flux CD stems from its distinct approach and capabilities compared to Argo CD.

At first glance, the use of Flux CD might appear like that of Argo CD, almost as if it's a simple icon swap, as visually represented in *Figure 6.3*, where there's a dedicated Flux CD instance per cluster. However, practical experience with Flux CD reveals a deeper layer of complexity. Unlike Argo CD, Flux CD requires a higher level of expertise in Kubernetes and Helm, demanding proficiency from teams in tools such as *Helm*, *Kustomize*, and *Kubernetes*:

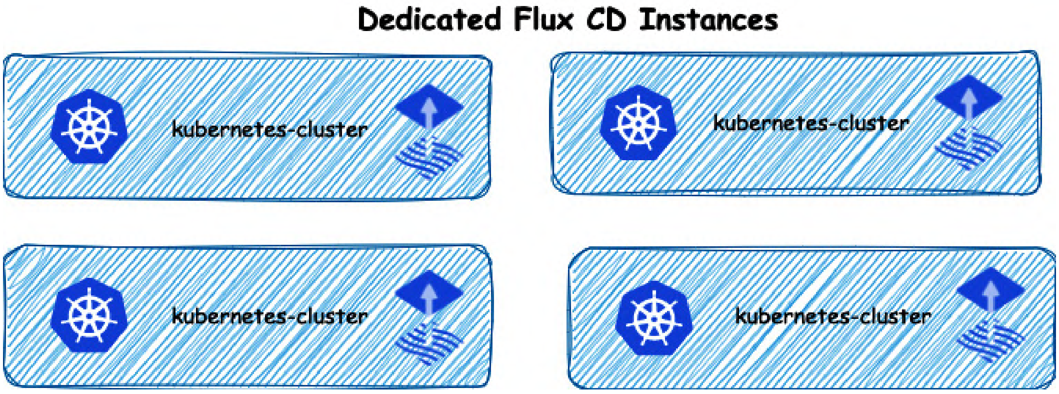


Figure 6.3: Example of dedicated Flux CD instances per cluster

Flux CD's approach to managing deployments revolves around Helm releases and the Helm controller. These elements are crucial for handling package deployments and life cycle management in Kubernetes. The Helm controller in Flux CD offers a declarative way to install, upgrade, and manage Helm charts in a Kubernetes environment, aligning with the GitOps principles. This requires teams to have a solid understanding of Helm charts and their management.

Additionally, Flux CD utilizes **Kustomizations** for applying Kubernetes manifests. This feature allows resources to be customized before they are applied to the cluster, providing a powerful tool for managing complex deployments. Understanding and effectively using Flux Kustomization requires a deep knowledge of how Kubernetes manifests work and how they can be customized for specific deployment needs.

In terms of organizational structure, adopting Flux CD effectively usually involves small, autonomous teams, typically consisting of 5-7 members. These teams handle projects as independent units within a larger organization, delivering specific products or services. This structure, while beneficial for focused and efficient delivery, poses challenges in onboarding due to the complexity of Flux CD and the advanced skills it demands.

The key advantages of using Flux CD for individual cluster management per team lie in its flexibility and adaptability, making it an ideal choice for complex deployment scenarios. Flux CD's support for advanced customization options is particularly beneficial for teams with comprehensive knowledge of Kubernetes and Helm.

However, this sophistication comes with a steep learning curve. The advanced functionalities and customization options of Flux CD add to the complexity of its setup and operation. As a result, integrating teams into a Flux CD workflow can be time-consuming, demanding a higher level of technical proficiency.

This discussion isn't centered around comparing Flux CD and Argo CD, but rather on the fact that the "instance per cluster" approach works effectively with both. The previously mentioned pros and cons apply to Flux CD as well since both tools operate on Kubernetes and employ native Kubernetes methods to facilitate deployments into clusters. However, it's important to note that while Flux CD demands a higher skill level and prior experience with Helm charts or Kustomization, Argo CD can also be utilized directly with plain Kubernetes manifests.

This distinction highlights that both Argo CD and Flux CD, despite their different complexities and requirements, can be effectively integrated into the instance per cluster model. While Argo CD offers a more user-friendly approach suitable for less complex scenarios, Flux CD's adaptability and technical demands make it ideal for more intricate deployments, especially for teams well-versed in Kubernetes and Helm.

Understanding the operational nuances, strengths, and requirements of each tool is crucial for organizations looking to optimize their Kubernetes management strategies. The choice between Argo CD and Flux CD in the instance per cluster approach should be informed by the specific needs of the deployment scenario and the skill level of the managing team. By aligning the tool's capabilities with the team's expertise and the project's requirements, organizations can achieve efficient and effective management of their Kubernetes clusters.

The next section will be an exciting one as various concepts that have already been presented will be combined to help you find a middle way.

The middle way – instance per logical group with Argo CD

The middle way – instance per logical group with Argo CD is an approach that centers around using a control cluster with Argo CD to manage a group of clusters. This approach presents a refined architecture that seeks to balance scalability, manageability, and efficiency in Kubernetes cluster management. It involves running one Argo CD instance per logical group of clusters, such as per team, region, or environment, depending on the organizational structure and requirements.

In this model, Argo CD is deployed on a control cluster that belongs to a specific group. From this central point, Argo CD manages all clusters within that group. This arrangement aims to streamline the management process by consolidating control, yet it still maintains a level of separation between different groups of clusters (*Figure 6.4*):

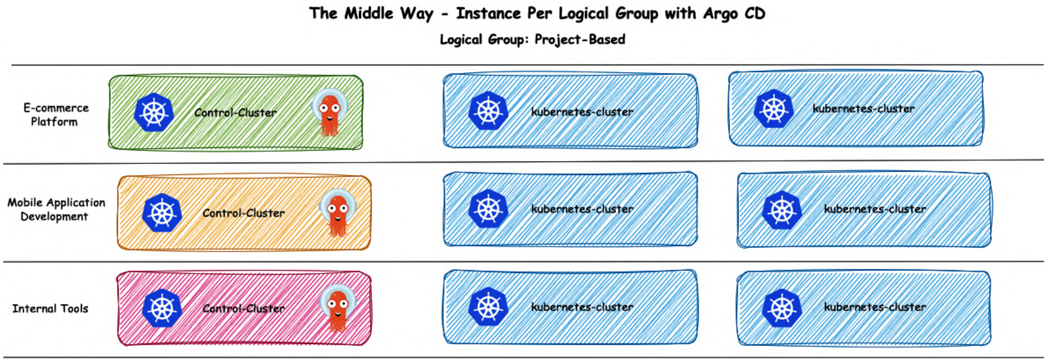


Figure 6.4 – Example of an instance per logical group in Argo CD-based projects

This architecture balances the demands of managing multiple clusters by effectively partitioning them into logical groups. It offers a solution that alleviates the challenges of maintaining too many individual instances while providing a more manageable and scalable approach. This grouping not only improves operational efficiency but also enhances the security and reliability of the system. The developer experience is also improved compared to an instance-per-cluster architecture as following a clear and understood convention for grouping reduces the cognitive load and simplifies integration processes.

Let's look at the possible and most sensible groupings. For example, groups can be logically sorted by departments:

- **Department-based**: Different departments such as development, operations, or QA each have their own Argo CD instance for managing their specific clusters.
    - **Development**: An instance for developers working on new product features
    - **Operations**: A separate instance for the operations team to manage deployment and infrastructure
    - **QA**: An instance for the QA team to test and validate products

At this point, developers have a development environment with different workload clusters that they can use autonomously. The operations department also has cluster groups and can operate infrastructure components that are important for the organization, such as LDAP servers, DNS servers, ACME servers, databases, and more. The QA department can test delivered software features on different clusters, in different versions, and conduct load tests with their tools.

Now, let's look at projects and geographical location:

- **Project-based**: For companies handling multiple projects, each project can be assigned a separate instance, facilitating focused management and autonomy:
    - **E-commerce platform**: An instance dedicated to the e-commerce project team
    - **Mobile application development**: A separate instance for teams working on mobile apps
    - **Internal tools**: For teams developing and maintaining internal company tools

  In this scenario, each project is assigned its own Argo CD instance. For instance, the team working on the e-commerce platform can fully control their deployment pipelines and feature rollouts, tailoring their workflows to the specific needs of the project. Similarly, teams dedicated to mobile application development can manage their deployments with a focus on mobile-specific requirements and testing environments. For internal tools, a dedicated instance allows the team to rapidly iterate and deploy updates, ensuring that internal operations run smoothly and efficiently.

- **Geographical location**: Companies with global operations can group clusters based on geographical regions for better localization and compliance management:
    - **North American operations**: An instance for clusters in North American regions
    - **Europe**: A dedicated instance for managing clusters in European countries
    - **Asia-Pacific** (**APAC**): An instance focused on the APAC region's specific needs and compliance

> For example, a dedicated Argo CD instance for North American operations allows teams to manage clusters as per local compliance and operational standards. In Europe, teams can address specific regional requirements, such as **GDPR compliance**, through a Europe-focused instance. Similarly, for the APAC region, an instance can cater to the unique operational and regulatory landscape, ensuring that deployments are optimized for local preferences and legal requirements. This geographical grouping not only enhances efficiency but also ensures adherence to regional regulations and cultural nuances, making it an essential strategy for global operations.

These groupings are valid, and I have personally seen this setup in various companies. However, it's more related to the organizational structure and how projects are managed. I have not yet seen the Argo CD setup in productive use in practice; I only know it from concepts or proof of concepts.

Although the centralized controller reduces certain disadvantages, such as the potential for a single point of failure in deployments, the centralization of admin credentials for all clusters, and the need for tuning individual components for scaling, these issues are shifted to the group level.

Through this setup, the following advantages and disadvantages arise:

Advantages	Disadvantages
Load distribution: By grouping clusters, this approach distributes the workload more evenly across groups, easing the burden on application controllers, repo servers, and API servers.	Multiple instance maintenance: The approach requires maintaining multiple Argo CD instances, one for each group, which can add to the administrative workload.
Group-specific credentials: Credentials are scoped per group, which simplifies access management while maintaining security.	Reducing single points of failure: While having a control cluster as the central management point per group reduces single points of failure, it also introduces a new vulnerability. If the control cluster encounters issues, it could potentially disrupt the management of all grouped deployments.
Unified view for deployment activities: Each group has a single view for all deployment activities, streamlining the monitoring and management process.	Centralized administration credentials: Admin credentials for all clusters in a group are stored in the control cluster, streamlining access management. This centralized approach can enhance security measures as it reduces the number of access points that need to be secured.

Advantages	Disadvantages
Reduced configuration duplication: As clusters in a group are likely to have similar RBAC, AppProject, and other configurations, this model significantly reduces the need for duplicate configurations.	Management cluster requirements: The requirement for a separate management cluster to host Argo CD instances adds a layer of infrastructure that needs to be set up and maintained, potentially complicating the overall system architecture.

Table 6.3 – Advantages and disadvantages of the middle-way approach

When to use the middle-way approach

The middle-way approach is particularly suited to organizations that manage a diverse range of clusters per logical group. It excels in scenarios where different departments, such as development, operations, or QA, require independent control over their respective clusters. This method is ideal for businesses handling multiple projects, each with unique requirements, allowing for focused and autonomous management. Additionally, for multinational companies, this approach facilitates effective management of clusters based on geographical locations, ensuring compliance with regional standards and operational efficiency.

When not to use the middle-way approach

Despite its benefits, this approach may not suit every scenario. For smaller organizations with limited resources, the task of maintaining multiple Argo CD instances can be daunting and resource-intensive. The need to tune each instance at scale, coupled with the necessity of a separate management cluster, adds layers of complexity that smaller or less complex environments might not warrant. In cases where centralized control is more practical and efficient, particularly in smaller setups, this approach might introduce unnecessary complications.

This approach presents a compromise between individual and centralized management models, distributing workloads across groups and reducing configuration duplication. It enhances the security and reliability of the system by limiting the impact of potential failures on specific groups. However, it requires careful planning and consideration of the organizational structure, resource availability, and the scale of operations to ensure it aligns with the specific needs of the organization. This approach, while not universally applicable, offers a flexible and efficient solution for medium to large-scale Kubernetes environments.

The next section deals with how to use a central Argo CD instance to provide the clusters with the necessary tools and how to ensure developer autonomy with dedicated Argo CD instances on the clusters.

The cockpit and fleet approach with Argo CD

In the dynamic world of Kubernetes and GitOps, The **cockpit and fleet** approach offers an innovative solution that combines centralized management with individual autonomy. This approach involves a platform team utilizing a central Argo CD instance for overarching control while also providing individual Argo CD instances for each developer's cluster (*Figure 6.5*). This approach is specifically designed for organizations that aim to streamline their Kubernetes operations and concurrently empower individual teams or departments with autonomy in their cluster management. Since July 2023, Flux has had an implementation that allows a similar approach and is called Hub and Spoke [*6*]:

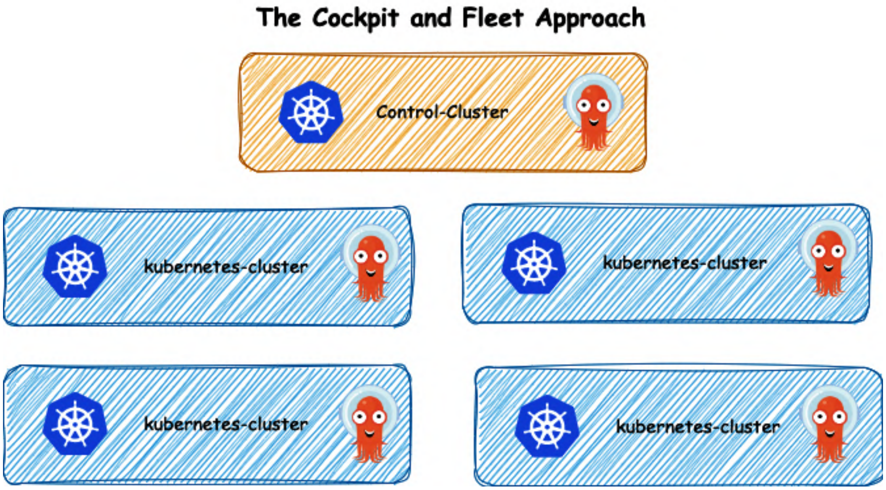


Figure 6.5 – Example of the cockpit and fleet approach

The central principle of this approach is anchored in two pivotal components – the cockpit and the fleet:

- **The cockpit**: Managed by the platform team, the cockpit involves operating a centralized Argo CD instance. This central instance functions as a command-and-control center that's responsible for deploying and managing essential infrastructure components across all the Kubernetes clusters within the organization. The primary role of the cockpit is to ensure that there's a uniform application of critical infrastructure elements across all clusters. This includes enforcing compliance with organizational standards and policies, thereby establishing a consistent and secure infrastructure framework.
- **The fleet**: In contrast to the centralized nature of the cockpit, the fleet provides individual developers or specific teams with dedicated Argo CD instances for each of their clusters. This decentralization empowers teams to manage their applications' life cycle independently, from configuration to deployment and updates. Such autonomy is vital in fostering innovation and agility, particularly in fast-paced development environments where rapid deployment and iterative updates are the norm.

Delving deeper into the approach

The dual-layered nature of the cockpit and fleet approach is crafted to address the diverse needs of large organizations with multiple clusters:

- The centralized cockpit offers a streamlined, holistic view of the organization's Kubernetes infrastructure. This centralization is crucial for large-scale operations where consistency in infrastructure management and policy enforcement is necessary. By having a unified control point, the platform team can efficiently manage shared resources, apply global security policies, and ensure compliance across all clusters.
- The decentralized fleet, on the other hand, caters to the specific needs of individual development teams or departments. Each team has the flexibility to tailor its cluster according to its project requirements. This setup is particularly beneficial in environments where different teams work on varied projects, each with its unique set of requirements and deployment strategies.

Operational dynamics

Implementing the cockpit and fleet approach necessitates a well-orchestrated operational model that balances centralized governance with decentralized autonomy:

- On one side, the platform team must ensure that the centralized cockpit is effectively managing the shared components and maintaining the required standards across all clusters. This involves regular updates, security patching, and monitoring of the centralized infrastructure.
- On the other side, individual teams managing their clusters via the fleet model need to align their development and deployment strategies with the broader organizational goals. They must also ensure their practices comply with the security and policy guidelines set by the platform team.

The following table compares some of the advantages and disadvantages of this approach:

Advantages	**Disadvantages**
Centralized view and control: Offers a unified view for platform context deployment activities across all clusters.	Security implications: Centralizing admin credentials for all fleet clusters could pose security risks.
Simplified management: Managing a single Argo CD instance for multiple Kubernetes clusters simplifies administrative tasks, easing the process of provisioning and maintaining the platform context for the fleet clusters.	Resource consumption: Additional resources are consumed, both for the management cluster and the fleet clusters, including hardware and engineering resources.

Advantages	Disadvantages
Reliability improvement: Each fleet cluster operates independently, which enhances overall reliability and allows for strict separation between teams.	Single point of failure: There's a potential risk of a single point of failure for fleet cluster deployments to provide the platform context.
Maintenance and expansion of fleet clusters: For platform teams, it's easier to roll out new tools or policies simultaneously across all clusters. This also impacts the process of maintaining or upgrading tools within the platform context.	Scaling and performance: Scaling requires tuning individual components within the platform context.

Table 6.4 – Advantages and disadvantages of the cockpit and fleet approach

When to use the cockpit and fleet approach

The cockpit and fleet approach is particularly well-suited for large-scale organizations managing a diverse range of projects across numerous clusters. This strategy is ideal for environments that require a combination of centralized control for shared resources and decentralized autonomy for individual teams or departments. It's especially beneficial in complex multi-cluster environments where a streamlined operation is needed to manage common infrastructure elements and policies efficiently. Moreover, organizations with different teams or departments, each having unique operational requirements, can leverage this approach to provide each unit with the necessary tools and autonomy for their specific projects. Global companies with operations across multiple regions also find this approach advantageous as it allows for centralized management of global standards while enabling local teams to manage clusters as per regional requirements.

When not to use the cockpit and fleet approach

Conversely, the cockpit and fleet approach may not be the most suitable for small to medium-sized businesses with a limited number of clusters and less complexity. In such cases, the overhead of creating a setup of maintaining both centralized and decentralized systems might outweigh the benefits. Organizations with uniform cluster needs across the board might find a simpler, more centralized approach more efficient and practical. Additionally, companies with limited resources in terms of personnel or infrastructure might face challenges in maintaining the dual management system effectively. Environments with simplified workflows, where development and deployment processes are straightforward and uniform across the organization, may not derive significant value from the added complexity of a hybrid approach.

The cockpit and fleet approach stands as a testament to the evolving landscape of Kubernetes management, offering a solution that is both comprehensive and flexible. It adeptly addresses the challenges of managing a vast Kubernetes infrastructure in large organizations, balancing the need for centralized control with the agility of decentralized management. The approach fosters a collaborative and efficient environment where the platform team and individual development teams work in harmony, each with their distinct yet interconnected roles. As organizations continue to grow and evolve in their Kubernetes journey, approaches such as cockpit and fleet become increasingly vital in navigating the complexities of cluster management at scale.

Choosing the right approach for your GitOps needs

Deciding on the right approach for GitOps can be challenging as there is no one-size-fits-all solution. In my experience across various industries and companies, two primary approaches have emerged: the cockpit and fleet approach in larger organizations, which scales with project needs, and the dedicated instance per cluster approach for smaller, independent teams. However, each approach has its drawbacks, particularly concerning security, especially in public cloud operations. To address these challenges and mitigate disadvantages, take a look at *Chapter 13*, *Security with GitOps*. Companies such as **Akuity** have begun offering SaaS and self-hosted solutions, which build upon GitOps with Argo CD but invert the principle.

This is illustrated in the following figure:

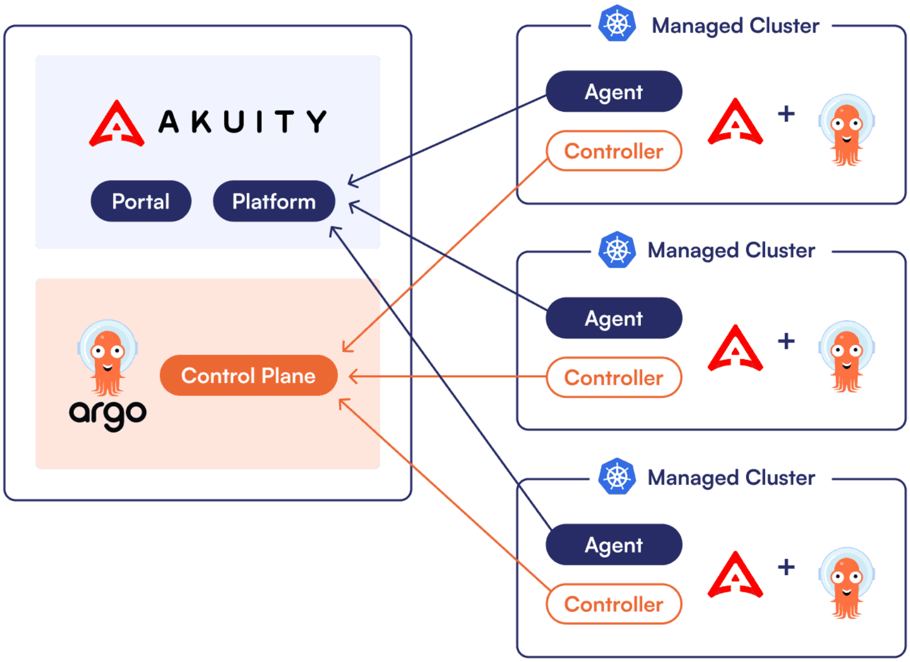


Figure 6.6 – Example of Akuity Platform's Argo CD SaaS and self-hosted offerings *[6]*

The Akuity Platform ingeniously integrates the cockpit and fleet approach of Argo CD management, where the Argo CD instances on fleet clusters register themselves with the central Argo CD "cockpit" in the Akuity Platform. This model blends the benefits of both instance-per-cluster and single-instance architectures, effectively addressing most of their limitations.

In this hybrid agent architecture, an agent runs inside each fleet cluster and establishes outbound communication back to the control plane in the cockpit. This setup significantly reduces the network traffic between the control plane and the cluster, addressing common security concerns as it does not require direct cluster access or admin credentials. This architecture is particularly advantageous for connecting external Argo CD instances to clusters in restricted environments, such as a local development cluster on a laptop.

The Akuity Platform simplifies the operational aspects of Argo CD. Unlike traditional models, which require a dedicated management cluster to host Argo CD, the Akuity Platform hosts the Argo CD instance and the custom resources. This innovation not only streamlines the management process but also introduces automatic snapshotting and disaster recovery features, effectively eliminating concerns around single points of failure.

From a visibility standpoint, the Akuity Platform offers a centralized view of all organizational Argo CD instances, akin to the single-instance architecture. The platform enhances open source capabilities by providing a dashboard for each instance, showcasing application health metrics and synchronization histories. It facilitates the management of settings, allowing configurations, typically complex YAML files, to be crafted easily using user-friendly wizards. Additionally, the inclusion of an audit log feature for all activity across the Argo CD instances greatly simplifies compliance reporting and monitoring.

If you look at the disadvantages of the cockpit and fleet approach, you can see why Akuity provides an innovative solution.

The following table shows how the Akuity Platform has minimized or eliminated most of the disadvantages mentioned in *Table 6.4*:

Disadvantages	Explanation
Security implications: Centralizing admin credentials for all fleet clusters could pose security risks.	The agent within the fleet cluster eliminates central credentials in the cockpit. It operates with outbound access back to the cockpit, removing the need for direct cluster access or admin credentials, thereby mitigating security concerns.
Resource consumption: Additional resources are consumed, both for the management cluster and the fleet clusters, including hardware and engineering resources.	Network traffic between the control plane and fleet clusters is reduced. Argo CD no longer needs to establish connections to fleet clusters at specific intervals. The syncing of applications is reduced as they now reside on the fleet clusters, lessening the load.
Single point of failure: There's a potential risk of a single point of failure for fleet cluster deployments to provide the platform context.	Fleet clusters autonomously retrieve and store the platform context through custom resources within the fleet cluster itself. This ensures continuity even in case of connection loss.
Scaling and performance: Scaling requires tuning individual components within the platform context.	No elimination. Tuning requirements shift to different aspects, such as setting specific domain filters for External-DNS per fleet cluster, which is essential for scaling. This necessitates tuning values so that they can be adapted to each cluster's needs.

Table 6.5 – How the Akuity Platform minimizes or eliminates most of the disadvantages mentioned in Table 6.4

In essence, the Akuity Platform adopts and enhances the cockpit and fleet approach, allowing fleet clusters' Argo CD instances to connect back to a central "cockpit," thereby providing a seamless, secure, and efficient method of managing large-scale Kubernetes environments. Akuity's approach stands out as an innovative solution, particularly for organizations grappling with the complexities and security concerns inherent in managing Kubernetes clusters.

So far, we have looked at approaches where Kubernetes clusters already exist. Next, we'll take a step back and create the Kubernetes clusters ourselves using the GitOps approach.

Centralized Kubernetes cluster creation – leveraging Cluster API and Argo CD for streamlined cluster deployment

In the world of modern software deployment, the synergy between Cluster API and Argo CD stands as a testament to the power and efficiency of GitOps practices. This approach not only facilitates the operation of Argo CD within clusters but also harnesses GitOps methodologies for deploying clusters themselves. These clusters form the foundational infrastructure for Argo CD and the workloads it manages, including various applications deployed through it.

This section delves into how Argo CD can be utilized, or how a self-service portal for teams can be provided, to streamline the deployment of Kubernetes clusters. The effectiveness of this approach is evidenced by companies such as **Kubermatic**, **CLASTIX**, and **Giant Swarm**, which have leveraged it to offer diverse managed Kubernetes solutions. These solutions range from standalone products to comprehensive managed services, aiming to simplify Kubernetes deployment for their clients.

This strategy allows organizations to centralize and automate the creation and management of Kubernetes clusters, ensuring a consistent and reliable infrastructure for deploying and managing applications using Argo CD. The use of GitOps in this context not only enhances the efficiency of these processes but also offers the scalability and flexibility needed to manage complex, multi-cluster environments effectively. By integrating Cluster API with Argo CD, organizations can create a powerful pipeline for deploying and managing Kubernetes clusters, which, in turn, can be used to deploy a wide range of workloads, including the Argo CD toolset itself and the application stacks for developers.

This approach represents a significant shift in how Kubernetes clusters are provisioned and managed, moving toward a more automated, scalable, and developer-friendly environment. It exemplifies the potential of GitOps to streamline not just application deployment but also the underlying infrastructure management, thereby enabling organizations to focus on innovation and development.

Introduction to Cluster API

The **Cluster API** [*7*] project represents a key initiative within the Kubernetes ecosystem that focuses on making the setup, update, and oversight of Kubernetes clusters more streamlined. Launched by the **Kubernetes Special Interest Group** (**SIG**) Cluster Lifecycle, this project utilizes Kubernetes-conformant APIs and design principles to automate the process of managing cluster life cycles for those responsible for platform operations. It facilitates defining and managing underlying infrastructure components – such as virtual machines, network resources, load balancers, and **virtual private clouds** (**VPCs**) – in a manner akin to how application developers handle application deployments. This approach ensures uniform and reliable deployment of clusters across diverse infrastructure settings.

A key aspect of Cluster API is its ability to provision Kubernetes-native, declarative infrastructure that applies to AWS. This incorporates principles and experiences from previous cluster managers, such as *kops* and *kubicorn*. Its features include being able to manage VPCs, gateways, security groups, and instances, support for **Elastic Kubernetes Service** (**EKS**), and the ability to deploy Kubernetes control planes in private subnets with a separate bastion server. SSH is not used for bootstrapping nodes, and only the minimal components are installed to bootstrap a control plane and worker nodes.

Cluster API is licensed under the Apache-2.0 license and offers an active community for developers and enthusiasts who wish to contribute to further development. There are regular office hours with maintainers where developers can participate in discussions and get support.

Cluster API offers a range of use cases in the realm of Kubernetes cluster management that cater to different needs within cloud-native ecosystems. Here are some common scenarios where Cluster API proves to be particularly useful:

- **Multi-cluster management**: Cluster API simplifies the management of multiple Kubernetes clusters across various environments. It allows for consistent and automated provisioning, upgrading, and operational tasks for many clusters, making it ideal for organizations managing a vast fleet of Kubernetes clusters.
- **Automated cluster life cycle management**: It automates the entire life cycle of Kubernetes clusters, including creation, scaling, upgrading, and deletion. This automation is particularly beneficial in scenarios where clusters need to be frequently scaled up or down based on demand or updated with the latest Kubernetes versions.
- **Hybrid cloud and multi-cloud deployments**: For organizations that operate in a hybrid or multi-cloud environment, Cluster API enables consistent deployment and management of Kubernetes clusters across different cloud providers. This uniformity is crucial for businesses looking to avoid vendor lock-in and maintain flexibility in their cloud strategy.
- **IaC**: Cluster API aligns with the IaC paradigm, allowing teams to define and manage clusters declaratively. This approach is beneficial for DevOps teams aiming to maintain infrastructure and configuration consistency through code.
- **Self-service clusters**: In larger organizations, different teams may require their own Kubernetes clusters. Cluster API enables a self-service model where teams can provision and manage their clusters autonomously while adhering to centralized policies and standards.
- **CI/CD pipelines**: Integrating Cluster API with CI/CD pipelines can streamline the process of testing and rolling out new applications or updates. It allows for dynamic creation and disposal of clusters as part of the CI/CD process, enabling more efficient testing and deployment workflows.
- **Disaster recovery**: Cluster API can be instrumental in disaster recovery strategies. Automating the creation of backup clusters and enabling quick replication of cluster states helps reduce downtime and ensures high availability.

- **Edge computing**: For Edge computing scenarios where Kubernetes clusters need to be deployed at multiple Edge locations, Cluster API provides a unified way to manage these clusters from a central point.
- **Learning and experimentation**: For educational purposes or experimentation, Cluster API allows users to quickly spin up and tear down Kubernetes clusters. This is useful for learning Kubernetes, testing new features, or experimenting with different configurations.

Each of these use cases demonstrates the versatility and utility of Cluster API in managing Kubernetes clusters efficiently and at scale, catering to the diverse needs of modern cloud-native applications and infrastructures.

How Cluster API is leveraged by different companies

In exploring the diverse landscape of Kubernetes management, it becomes evident that different organizations have unique requirements and strategies. These vary based on their specific operational needs, infrastructure preferences, and long-term technological goals. As a result, various implementations of Cluster API have emerged, each tailored to meet these differing demands. Some organizations opt for fully managed Kubernetes services, while others lean toward self-managed solutions to avoid vendor lock-in and maintain greater control over their infrastructure.

The following are only a fraction of the companies that use Cluster API in their substructure:

- **VMware Kubernetes solution (vSphere with Tanzu)**: VMware's integration of Kubernetes directly into the vSphere platform demonstrates a deep use of Cluster API, particularly with the vSphere provider. This allows developers to deploy and manage Kubernetes clusters directly from vSphere.
- **Cluster API Provider Azure (CAPZ)**: This is Microsoft's implementation of Cluster API for Azure and replaces AKS Engine for self-managed Kubernetes clusters. CAPZ leverages Azure's robust, scalable infrastructure to provide a seamless and efficient way to operate Kubernetes, simplifying cluster management tasks and enhancing the automation capabilities inherent in Azure's cloud services. This implementation ensures that users can maintain full control over their Kubernetes environments while benefiting from the native integrations and services offered by Azure.
- **Giant Swarm (Kubernetes platform)**: Giant Swarm uses Cluster API to create a unified application point for multiple self-managed Kubernetes clusters across different cloud provider endpoints. It offers a managed Kubernetes solution with the flexibility to deploy to various target cloud providers, emphasizing low vendor lock-in and subscription-based pricing.
- **CLASTIX (Kamaji)**: An entirely open source implementation of Cluster API, Kamaji is noted for its efficiency in scaling control planes on a management cluster, thereby reducing costs. The approach involves creating worker nodes and enabling them to join the respective tenants.

- **Kubermatic Kubernetes Platform (KKP)**: This platform, which started early with the first version of Cluster API, focuses on creating and managing instances for worker nodes and joining them to a cluster. The architecture includes a Master Cluster and Seed Clusters with a special machine controller for precise management.

Each of these implementations reflects different strategies and priorities, such as vendor lock-in considerations, customization capabilities, resource conservation, and integration with existing infrastructure. The choice of a specific implementation depends on the organization's requirements, including governance, compliance, and operational needs.

In this context, examining how different companies utilize Cluster API provides valuable insights into the practical applications and benefits of this tool. For instance, the Azure Provider for Cluster API (**CAPZ**) bridges the gap between Microsoft Azure's managed Kubernetes service, AKS, and Kubernetes-native management, aligning with Cluster API's standardized, declarative approach. Meanwhile, open source solutions such as **CLASTIX's Kamaji** emphasize flexibility and control, catering to organizations keen on avoiding vendor lock-in. In contrast, **VMware's vSphere with Tanzu** integrates Kubernetes into its platform, streamlining workflows and offering a seamless Kubernetes-native environment. Similarly, **Giant Swarm** and **Kubermatic Kubernetes Platform** target specific operational needs such as compatibility with Cluster API versions and resource conservation. Each of these implementations showcases the adaptability of Cluster API, underlining its importance in providing flexible, cloud-agnostic Kubernetes solutions in the modern technological landscape.

Cluster API, like any technology, comes with its own set of advantages and disadvantages. Understanding these can help in determining whether it's the right tool for a specific Kubernetes management scenario.

Here are a few advantages of Cluster API:

- **Consistency and standardization**: Cluster API provides a standardized way to manage Kubernetes clusters. This consistency is crucial for large-scale and multi-cloud environments.
- **Automation and scalability**: It automates the process of creating, configuring, and managing Kubernetes clusters, which is beneficial for organizations that need to scale their operations efficiently.
- **Declarative API**: Aligning with the Kubernetes principle of declarative configuration, Cluster API allows users to define their desired state for clusters, which the system then works to achieve.
- **Integration with the cloud-native ecosystem**: It integrates well with other tools in the Kubernetes ecosystem, offering a seamless experience for managing clusters as part of the broader cloud-native infrastructure.
- **Multi-cloud and hybrid cloud support**: Cluster API supports multiple cloud providers, making it easier to manage clusters in a hybrid or multi-cloud environment.
- **Community support**: Being a part of the Kubernetes project, it benefits from strong community support and ongoing development efforts.

Here are a few disadvantages of Cluster API:

- **Complexity**: Cluster API can be complex to understand and implement, especially for users new to Kubernetes or cloud-native technologies
- **Limited customization in some areas**: While it offers a standardized approach, this can sometimes limit customization options for specific use cases or environments
- **Dependency on Kubernetes expertise**: Effective use of Cluster API requires a good understanding of Kubernetes concepts and architecture
- **Resource overhead**: Running additional controllers and resources for managing clusters could lead to increased resource consumption in your Kubernetes environment
- **Learning curve**: For teams not familiar with Kubernetes' declarative model and API-centric management, there can be a significant learning curve

In summary, Cluster API is a powerful tool for organizations looking to automate and standardize their Kubernetes cluster management, especially across large-scale and multi-cloud environments. However, its complexity and the need for Kubernetes expertise might pose challenges for some teams. As with any technological decision, it's important to evaluate these factors in the context of specific organizational needs and capabilities.

In the next section, we will go hands-on and use Cluster API and GitOps to deploy Kubernetes clusters on Azure.

A deep dive into Cluster API and GitOps – hands-on

In this section, we'll use Cluster API to provision a Kubernetes cluster in Azure by using the declarative approach with Argo CD on VMs.

First, let's see what our environment looks like:

- Azure:
    - Azure tenant ID
    - Azure subscription
    - Azure app registration with Contributor access to the subscription
    - **Azure Kubernetes Service** (**AKS**)
    - The necessary Azure infrastructure, including **Virtual Machines Scale Sets** (**VMSS**), virtual networks, and more

- Managed cluster – AKS:
    - Argo CD running on a managed cluster
- Workload cluster – VMSS:
    - Control plane
    - Nodes
- Tools:
    - `kubectl`
    - `clusterctl`
    - `az cli`
    - `helm`

Initializing the management cluster

The `clusterctl` command takes a list of providers to install as input. When executed for the first time, `clusterctl init` automatically includes the `cluster-api` core provider in the list. If not specified, it also adds the `kubeadm` bootstrap and `kubeadm` control plane providers:

```
#export AZURE_SUBSCRIPTION_ID="<SubscriptionId>"
# Initialize Azure Service Principal credentials and Azure related
config below
export AZURE_TENANT_ID="<Tenant>"
export AZURE_CLIENT_ID="<AppId>"
export AZURE_CLIENT_SECRET="<Password>"
export AZURE_SUBSCRIPTION_ID="<SubscriptionId>"

# Define the names and namespace for AzureClusterIdentity resources
export AZURE_CLUSTER_IDENTITY_SECRET_NAME="cluster-identity-secret"
export CLUSTER_IDENTITY_NAME="cluster-identity"
export AZURE_CLUSTER_IDENTITY_SECRET_NAMESPACE="default"

# Convert credentials to Base64 for secure storage
export AZURE_SUBSCRIPTION_ID_B64="$(echo -n "$AZURE_SUBSCRIPTION_ID" |
base64 | tr -d '\n')"
export AZURE_TENANT_ID_B64="$(echo -n "$AZURE_TENANT_ID" | base64 | tr
-d '\n')"
export AZURE_CLIENT_ID_B64="$(echo -n "$AZURE_CLIENT_ID" | base64 | tr
-d '\n')"
export AZURE_CLIENT_SECRET_B64="$(echo -n "$AZURE_CLIENT_SECRET" |
```

```
base64 | tr -d '\n')"


# Construct a Kubernetes secret for Azure Service Principal, to be
used by AzureCluster
# This step secures the Service Principal's password within the
Kubernetes environment
kubectl create secret generic "${AZURE_CLUSTER_IDENTITY_SECRET_
NAME}" -from-literal=clientSecret="${AZURE_CLIENT_SECRET}" -namespace
"${AZURE_CLUSTER_IDENTITY_SECRET_NAMESPACE}"

# Begin the setup of the management cluster with Cluster API for Azure
clusterctl init --infrastructure azure
```

You should get an output similar to the following:

```
Fetching providers
Skipping installing cert-manager as it is already installed.
Installing Provider="cluster-api" Version="v1.6.0"
TargetNamespace="capi-system"
Installing Provider="bootstrap-kubeadm" Version="v1.6.0"
TargetNamespace="capi-kubeadm-bootstrap-system"
Installing Provider="control-plane-kubeadm" Version="v1.6.0"
TargetNamespace="capi-kubeadm-control-plane-system"
Installing Provider="infrastructure-azure" Version="v1.12.1"
TargetNamespace="capz-system"

Your management cluster has been initialized successfully!

You can now create your first workload cluster by running the
following:

  clusterctl generate cluster [name] --kubernetes-version [version] |
kubectl apply -f -
```

The output also mentions that the installation of `cert-manager` is skipped because it is already installed. This step is important because `cert-manager` plays a critical role in managing certificates within Kubernetes environments, ensuring secure communication between cluster components by automating the issuance and renewal of TLS certificates.

Moreover, the output reflects the successful setup of a management cluster with essential providers for Kubernetes cluster management, including Cluster API, Kubeadm, and Azure infrastructure, each installed in specific namespaces. This step is crucial for streamlining Kubernetes operations as workload clusters can be created with a simple command, thereby facilitating efficient cluster deployment and management.

Creating your first workload cluster

Once the management cluster is ready, you can create your first workload cluster.

> **Important note**
>
> Make sure you select a VM size that is available in your desired location for your subscription. To see the available SKUs, use the `az vm list-skus -l <your_location> -r virtualMachines -o table` command.

Please replace the following variables with your specific values:

```
# Specify the Virtual Machine sizes for the control plane and nodes.
export AZURE_CONTROL_PLANE_MACHINE_TYPE="Standard_D2s_v3"
export AZURE_NODE_MACHINE_TYPE="Standard_D2s_v3"
# Define the Azure region for resource deployment. Modify to match
your preferred region
export AZURE_LOCATION="centralus"
```

The following command prepares a YAML manifest for deploying a Kubernetes cluster named `capi-quickstart` and specifies Kubernetes version `1.29.0`, one control plane machine, and three worker machines. By saving this configuration to `capi-quickstart.yaml`, it enables automated and consistent cluster deployment, encapsulating the desired state and structure of the cluster in a single file.

Now, you are ready to generate the cluster YAML manifest:

```
clusterctl generate cluster capi-quickstart \
    --kubernetes-version v1.29.0 \
    --control-plane-machine-count=1 \
    --worker-machine-count=3 \
    > capi-quickstart.yaml
```

At this point, you should have a file that contains the following custom resources in the `capi-quickstart.yaml` file:

- `KubeadmConfigTemplate`: This is the schema for the `kubeadmconfigtemplates` API.
- `AzureClusterIdentity`: This is the schema for the `azureclustersidentities` API.
- `AzureMachineTemplate`: These templates define the specifications for creating Azure VMs within the cluster. This is the schema for the `azuremachinetemplates` API.
- `MachineDeployment`: This custom resource specifies the desired number of worker nodes and their properties. It helps in scaling the cluster by automatically managing the creation and scaling of worker nodes.

- `KubeadmControlPlane`: This defines the control plane for the Kubernetes cluster, including settings such as the number of control plane nodes and their configurations.
- `AzureCluster`: This custom resource represents the Azure-specific details of the cluster, such as the network configuration and virtual network details.
- `Cluster`: This defines the high-level cluster configuration, including control plane settings, worker node references, and provider-specific details. This is the top-level resource that represents the entire Kubernetes cluster.

Now, you can apply the file using `kubectl`. However, we'll leverage GitOps with Argo CD to maximize the benefits of the declarative approach. So, create an *application*, like this:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: capi-capz-quickstart
spec:
  destination:
    name: ''
    namespace: capi-capz-quickstart
    server: 'https://kubernetes.default.svc'
  source:
    path: >-
      ./chapter-6-gitops-architectural-designs-and-operational-
control/chapter-6-centralized-kubernetes-cluster-creation/
    repoURL: git@github.com:PacktPublishing/Implementing-GitOps-with-
Kubernetes.git
    targetRevision: HEAD
  sources: []
  project: default
  syncPolicy:
    automated:
      prune: false
      selfHeal: true
    syncOptions:
      - CreateNamespace=true
      - Validate=false
```

At this point, you can view the provisioning of the cluster:

```
kubectl get cluster
#Output like:
NAME                 CLUSTERCLASS    PHASE          AGE    VERSION
capi-quickstart                      Provisioning   2m
```

Now, view the provisioning state of the cluster:

```
clusterctl describe cluster capi-quickstart
#Output like:
NAME
READY
Cluster/capi-quickstart
True
├─ClusterInfrastructure - AzureCluster/capi-quickstart
              True
├─ControlPlane - KubeadmControlPlane/capi-quickstart-control-plane
True
│ └─Machine/capi-quickstart-control-plane-dct9z
True
└─Workers
  └─MachineDeployment/capi-quickstart-md-0
    └─3 Machines...
```

From the preceding output, it's evident that the cluster is partially operational. Having `READY True` across various components in the output indicates the operational status of the cluster. Specifically, the readiness of the cluster (`Cluster/capi-quickstart`), its Azure infrastructure (`AzureCluster/capi-quickstart`), and `KubeadmControlPlane/capi-quickstart-control-plane` being marked as `True` shows that these critical parts of the cluster are fully operational. The control plane nodes are running, but the worker nodes haven't started yet.

Important note

The control plane won't be ready until we install a **container network interface** (**CNI**).

The worker nodes are not yet operational because we need to deploy the CNI components. It's important to note that Azure does not currently support Calico networking. CAPZ clusters that use the default Calico configuration will experience issues with DNS functionality. To address this, we will deploy a Calico spec that utilizes VXLAN encapsulation for Pod traffic. You can deploy the Azure **Calico CNI** using the template provided here.

To obtain `kubeconfig` so that you can interact with the cluster, follow these steps:

1. The following command retrieves the `kubeconfig` details for the `capi-quickstart` Kubernetes cluster and saves it to a file:

   ```
   clusterctl get kubeconfig capi-quickstart > capi-quickstart.
   kubeconfig
   ```

2. Now, we need to install the CNI plugin on the workload cluster using the following command:

   ```
   helm repo add projectcalico https://docs.tigera.io/calico/charts
   --kubeconfig=./capi-quickstart.kubeconfig && \
   helm install calico projectcalico/tigera-operator
   --kubeconfig=./capi-quickstart.kubeconfig -f https://raw.
   githubusercontent.com/kubernetes-sigs/cluster-api-provider-
   azure/main/templates/addons/calico/values.yaml --namespace
   tigera-operator --create-namespace
   ```

   After performing the previous steps, you should observe that, after 5-10 minutes, the cluster's status changes to `provisioned`, and you gain access to the nodes:

   ```
   kubectl --kubeconfig=./capi-quickstart.kubeconfig get nodes

   #Output like:
   NAME                                  STATUS   ROLES
   AGE
   capi-quickstart-control-plane-kcqmm   Ready    control-plane
   8m26s
   capi-quickstart-md-0-2kj9c            Ready    <none>
   6m58s
   capi-quickstart-md-0-7krx6            Ready    <none>
   6m56s
   capi-quickstart-md-0-b8m7r            Ready    <none>
   7m2s
   ```

3. Now you can add the cluster to your Argo CD cockpit as a fleet ship and continue working.

In Azure, due to the peculiarities of CNI, doing this requires some additional work, such as setting up a webhook that deploys the CNI plugin as soon as `status control-plane=true` is achieved. This slightly restricts the self-service aspect and requires extension in the form of **CI/CD and webhooks**, for example. However, Cluster API offers various other providers where this may not be necessary:

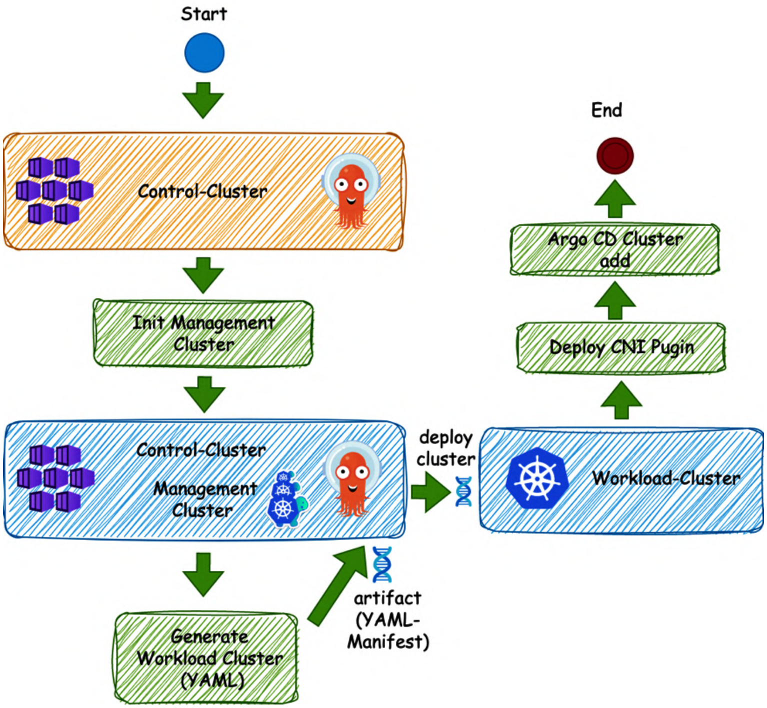


Figure 6.7 – Workflow visualized

In this example, we demonstrated how to utilize Cluster API to create a declarative setup that can be deployed by Argo CD to provision the infrastructure or workload clusters. *Figure 6.7* illustrates this process. Subsequently, Argo CD can be layered on top of it using `argocd cluster add`. This allows you to utilize tools such as **Kubernetes Service Catalog** (**KSC**) to deploy services based on labels.

Summary

In this chapter, we embarked on a comprehensive exploration of GitOps within Kubernetes environments, uncovering pivotal insights and strategies that are crucial for modern cloud-native deployments. We began by examining the criticality of tailoring architectural designs for scalability, resilience, and efficiency, all of which are foundational principles in today's dynamic cloud landscapes. This journey through architectural frameworks underscored the indispensability of IaC, not only for its collaborative and version control benefits but also for establishing immutable infrastructure that resists manual alterations in live environments. Emphasizing modular design, we highlighted how efficient microservices management can transform operational workflows.

Then, our exploration delved into the depths of architectural choices and their significant impact on the effectiveness of GitOps. We learned the importance of adopting a declarative model, an approach that seamlessly integrates version control, change management, and automated synchronization. This section illuminated the considerations necessary when selecting repository structures, weighing the merits of monolithic against multi-repository strategies. It also discussed the vital role of deployment strategies, environment isolation, scalability, and security, each a cornerstone in realizing a robust GitOps implementation.

We then transitioned to understanding the role of GitOps in enforcing systematic improvements in cloud-native architectures. Key aspects such as horizontal scaling, microservices, stateless applications, high availability, fault isolation, and disaster recovery were dissected. We explored how GitOps can manage deployments, ensuring resilience and efficiency through tactics such as auto-scaling, load balancing, and setting resource limits.

The second part of this chapter shifted our focus to various architectural approaches tailored for GitOps in Kubernetes environments. As organizations evolve, the need for scalable, secure, and efficient deployment strategies becomes paramount. We delved into the world of Argo CD, examining its capabilities in scaling performance, security, usability, and failover processes. We compared and contrasted the nuances between managing clusters with a centralized Argo CD instance and dedicated instances per cluster. The differences between Argo CD and Flux CD were also highlighted, providing a balanced view of their respective strengths and weaknesses.

Moreover, we explored the innovative cockpit and fleet approach with Argo CD, a strategy that goes beyond mere Kubernetes cluster management and includes provisioning clusters declaratively using Cluster API.

As we pave the way for the next chapter, we will delve into the necessary cultural shifts for successfully implementing and operating GitOps. We'll explore the transformation of treating infrastructure as an application and the principles of immutable infrastructure before delving into various **DevOps Research and Assessment** (**DORA**) metrics. We'll also discuss the critical need for continual improvement in GitOps and overcoming cultural barriers that may hinder its adoption. This sets the stage for a profound understanding that successful GitOps is not just about the right tools and technologies but also about cultural adaptation and evolution within the IT landscape.

References

- [*1*] `https://akuity.io/blog/argo-cd-architectures-explained`
- [*2*] `https://codefresh.io/blog/a-comprehensive-overview-of-argo-cd-architectures-2023/`
- [*3*] `https://www.youtube.com/watch?v=p8BluR5WT5w`
- [*4*] `https://github.com/openclarity/kubeclarity`
- [*5*] `https://github.com/aquasecurity/trivy-operator`
- [*6*] `https://github.com/fluxcd/flux2/releases/tag/v2.0.0`
- [*7*] `https://cluster-api.sigs.k8s.io`

7

Cultural Transformation in IT for Embracing GitOps

In the rapidly evolving landscape of information technology, the adoption of **GitOps** marks a significant paradigm shift, heralding not just a technological transformation but also a profound **cultural** metamorphosis within IT departments. This chapter delves into the multifaceted layers of this change, highlighting how GitOps fundamentally redefines operational processes and methodologies.

At its core, GitOps represents a fusion of software development and IT operations, driven by the principles of version control and collaboration, which are intrinsic to Git. It's a methodology where infrastructure and application deployment are treated as code, enabling a high degree of **automation** and precision in IT operations. This approach ensures that the same rigor that's applied to application development is now extended to the management of infrastructure, particularly within Kubernetes environments.

One of the most transformative aspects of GitOps is the establishment of an **immutable infrastructure**. Using reconciling and synchronization loops, GitOps automates the process of aligning the actual state of the infrastructure with the desired state defined in a Git repository. This alignment minimizes the need for manual interventions, reducing the potential for human error and improving the overall reliability and security of the IT systems.

A significant part of this chapter is dedicated to understanding the impact of GitOps on IT performance. We'll explore this through the lens of the **DevOps Research and Assessment** (**DORA**) metrics, a widely recognized set of indicators used to measure the effectiveness of DevOps practices. By applying these metrics to GitOps, organizations can quantitatively assess improvements in areas such as deployment frequency, change lead time, change failure rate, and mean time to recovery.

However, the journey of adopting GitOps extends beyond the implementation of tools such as Argo CD. It necessitates a continuous engagement with evolving GitOps trends and practices. As with any emerging technology, GitOps comes with its own set of challenges. These include the need for upskilling teams, adapting existing workflows, and **continuously optimizing** processes to align with best practices in GitOps.

The final section of this chapter addresses the broader organizational impact of implementing GitOps. Adopting GitOps is not just a technical upgrade; it's a catalyst for a cultural shift that permeates various departments within an organization. We'll share insights and experiences from multiple projects across diverse companies, illustrating how GitOps can drive a more collaborative, transparent, and efficient IT culture. These real-world examples provide valuable lessons on the challenges, strategies, and successes of integrating GitOps into the fabric of an organization.

This comprehensive examination of GitOps, from its technical foundations to its cultural implications, aims to provide you with a clear understanding of the transformative power of this methodology. It's a journey that transcends mere technology adoption, charting a path toward a more agile, responsive, and innovative IT culture.

We will cover the following main topics in the chapter:

- Treating infrastructure as an application
- The principles of immutable infrastructure
- An introduction to DORA metrics
- Understanding the need for continual improvement in GitOps
- Overcoming cultural barriers to adopt GitOps

Treating infrastructure as an application

In the evolving landscape of **DevOps**, the concept of treating infrastructure as applications – commonly known as **infra-as-apps** – is gaining momentum, especially in the context of GitOps. To understand this, we need to get a quick overview of **Infrastructure as Code** (**IaC**). IaC will be covered in detail in *Chapter 10*. This is a crucial component that's laid out, for example, by Terraform with its declarative approach, allowing infrastructure to be treated like an application.

Understanding IaC

IaC is a modern approach for managing and provisioning IT infrastructure using code instead of traditional manual processes. This method allows you to automate the setup, management, and configuration of computing resources such as servers, storage, networks, and applications. Treating your infrastructure as if it were software, IaC applies software development practices such as version control and testing to infrastructure management.

The main aim of IaC is to enhance efficiency, reliability, and consistency in infrastructure deployment and management. By using code to define infrastructure, manual intervention is reduced, decreasing human error, and speeding up deployment. IaC enables a more consistent and repeatable process for provisioning and configuring resources, simplifying the scaling and management of infrastructure over time.

Important note

Imperative: Directly use kubectl commands to create, update, or delete Kubernetes resources, offering quick, on-the-fly adjustments suitable for development and experimentation.

Declarative: Define the desired state in YAML manifests and apply them, letting Kubernetes ensure the actual state aligns with the declared one. This is ideal for production with benefits such as version control and reproducibility.

IaC typically employs a **declarative** or **imperative** approach to define the infrastructure's desired state. This specification allows the IaC tool to make necessary changes automatically, leading to a more agile, efficient, and reliable IT environment that better supports modern business demands.

The importance of IaC

The rise of cloud computing has been a significant driver behind IaC's growing adoption. Organizations face the challenge of managing and deploying resources across multiple environments, such as public, private, and hybrid clouds. IaC automates the deployment and management of these resources, easing the management of complex environments.

Additionally, the need for businesses to deliver applications and services swiftly and efficiently has made IaC increasingly important. In today's fast-paced digital world, organizations must rapidly adapt to market and customer demands. IaC streamlines IT operations, enabling quicker and more adaptable scaling.

Furthermore, IaC supports **DevOps principles**, bridging the gap between software development and IT operations. Treating infrastructure as code allows organizations to apply software development methodologies to infrastructure, enhancing collaboration and communication between development and operations teams.

How IaC works

The fundamental concept of IaC is treating infrastructure like software. This includes employing rigorous software development practices such as version control, infrastructure code testing, and continuous integration and deployment. This approach ensures infrastructure consistency, reducing downtime and maintenance time.

The IaC process starts with writing scripts that describe the infrastructure's desired state using declarative language. These scripts, stored in a version control system like any software, allow for versioning, auditing, and rollbacks. An IaC tool then applies these scripts to the actual infrastructure, ensuring it matches the desired state. This repeatability ensures consistency and reduces manual errors.

Understanding infrastructure as applications in Argo CD's GitOps framework

In the realm of Argo CD, the concept of an application extends beyond its traditional definition [*1*]. Here, an application encompasses not just the software but also its desired state configuration, the targeted deployment environment, and the policies governing how these resources are synced and managed. This approach marks a significant shift from the conventional understanding of applications.

Typically, engineering teams have viewed applications as software layers that run on a separate infrastructure. The standard process involves setting up the infrastructure first, often manually or using tools such as **Terraform** or **Cloud Formation** templates. Once this groundwork is laid, another team deploys their applications onto this prepared infrastructure. However, Argo CD introduces a more integrated perspective. In this GitOps-driven environment, an application's deployment is continuously monitored. Argo CD vigilantly compares the real-time state of the application against its predefined desired state. This constant comparison is crucial for maintaining consistency and reliability. The divergence between the actual state and the desired state can occur due to two primary reasons. First, there could be a drift in the actual state, which refers to unanticipated changes or modifications in the environment. Secondly, the desired state itself might be updated, necessitating a change in the deployment.

Whenever such divergences occur, Argo CD steps in to reconcile these differences. This act of reconciliation is at the heart of the GitOps style of deployment, which Argo CD upholds. This method aligns with the principles set by the **OpenGitOps standards**, ensuring a systematic and standardized approach to deployment and infrastructure management.

This paradigm shift, where infrastructure is treated akin to applications, brings about a more seamless and integrated workflow. It emphasizes the importance of treating infrastructure management with the same level of detail and care as application development, underscoring a holistic approach to system deployment and management.

Embracing infra-as-apps – bridging GitOps and infrastructure management

Traditionally, creating and managing infrastructure through declarative configurations has been a challenge, despite the popularity of tools such as **Terraform**, **Cloud Formation** templates, **Pulumi**, **Cloud Development Kit for Terraform** (**CDKTF**), and others. These tools offer a one-time or change-triggered application of the desired state, typically via **CI/CD pipelines**. However, this method falls short in one critical aspect of GitOps: ongoing state monitoring.

The gap lies in the fact that CI/CD pipelines, while efficient in deploying new plans or updates, do not continuously monitor for state drift. For instance, if a Terraform plan is executed and the actual state is altered externally later, the CI/CD system remains oblivious to these changes. This is where a GitOps operator such as Argo CD becomes indispensable. Argo CD's role is to relentlessly monitor the actual state of the infrastructure and ensure it aligns with the desired state, thereby preventing drift.

Adopting this *no-drift-allowed approach* has proven beneficial for teams, not only in managing their applications more efficiently but also in enhancing their ability to recover from failures rapidly. In this model, rolling back to a previous state is as simple as performing `git revert`.

The benefits that teams have experienced in managing applications can now be extended to infrastructure management, thanks to this paradigm shift. Our focus, however, will be on Crossplane. As an open source CNCF project, Crossplane enables users to provision and manage diverse cloud resources through the Kubernetes API. When integrated with a GitOps operator such as Argo CD, Crossplane allows for the deployment of applications representing a variety of cloud resources such as **Azure Key Vault**, **databases**, **Kubernetes clusters**, and load balancers across **Azure**, **AWS**, and other cloud platforms. This integration heralds a new era in infrastructure management, blending the robustness of GitOps with the versatility of modern cloud resources.

How IaC can be used to deploy infrastructure

Now that we have all the necessary background information and hopefully a sufficient understanding of the difference between the GitOps approach and using Terraform in terms of treating your infrastructure as an application, let's clarify this in practice by examining three use cases.

Before we begin, let's briefly explain the selected tools for this setup and why they were chosen. Starting with Azure:

- **Azure Kubernetes Service (AKS)**: This service offers a managed Kubernetes environment that simplifies how containerized applications are deployed, managed, and scaled with the robustness of Azure's infrastructure.
- **Container Registries**: This service provides a secure, scalable, and private registry for Docker container images, enhancing the management and deployment of containerized applications.
- **PostgreSQL Server on Azure**: This service offers a reliable and scalable cloud database service, ensuring efficient management and storage of application data.
- **Azure Key Vault**: This is a tool for securely storing and accessing secrets, keys, and certificates. It's crucial for managing sensitive information and enhancing overall security.

For Kubernetes, we have the following:

- **Argo CD**: A declarative, GitOps continuous delivery tool for Kubernetes, enabling automated deployment and management of applications
- **External-DNS**: This tool automates the management of DNS records, streamlining the process of connecting Kubernetes services with external DNS names

- **Cert-Manager**: This tool manages SSL/TLS certificates for Kubernetes, automating certificate issuance and renewal processes
- **External-Secrets Operator**: This tool integrates with systems such as Azure Key Vault to securely inject secrets into Kubernetes, enhancing secure access to sensitive data

For deployments for the IaC part, we have the following:

- **Terraform modules**: These are components within Terraform, an IaC tool, that are designed to enable modular and reusable infrastructure definitions. They facilitate a declarative approach by allowing users to define the desired state of their infrastructure in code, which Terraform then executes to create and manage the actual infrastructure, ensuring it matches the specified state.
- **Crossplane**: This is an IaC tool that integrates with the Kubernetes ecosystem, allowing for external resources, such as cloud services, to be managed through Kubernetes **custom resource definitions** (**CRDs**). It adopts a declarative model where users define their infrastructure requirements in a Kubernetes-native way, enabling consistent and unified management of both Kubernetes internal resources and external cloud infrastructure.

In combination, these tools form a comprehensive ecosystem. Azure's services provide a secure and scalable cloud platform, while Kubernetes tools such as Argo CD and Cert-Manager ensure efficient and secure application deployment and management. The integration between Azure Key Vault and the External-Secrets Operator in Kubernetes exemplifies how cloud infrastructure and Kubernetes can work seamlessly together for enhanced security and operational efficiency. Now, let's look at the use cases.

Use case 1 – deploy everything over Terraform

In the first use case, as illustrated in *Figure 7.1*, everything is deployed via Terraform. This means that initially, the required cloud infrastructure is deployed using Terraform:

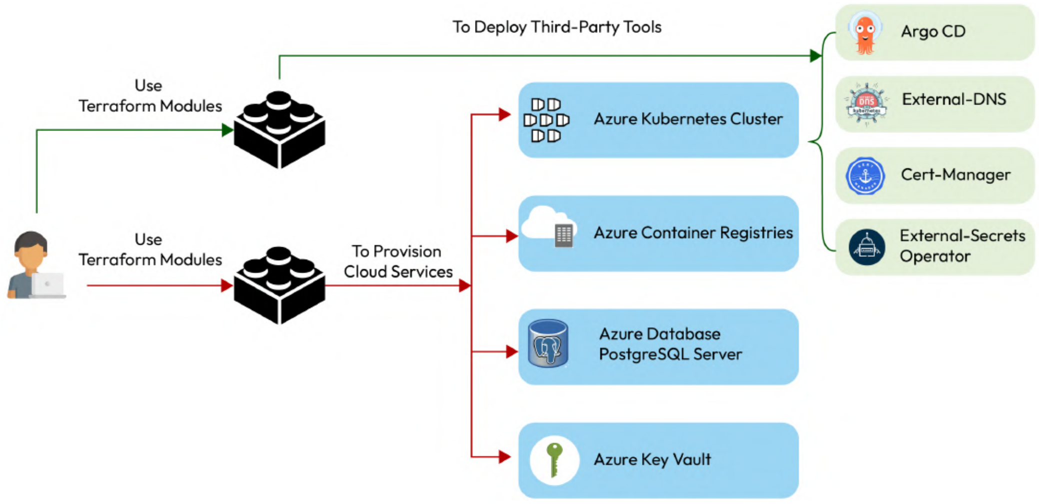


Figure 7.1 – Use case 1 – deploy everything over Terraform

To simplify this example, we've removed any unnecessary complexity, such as CI/CD pipelines. In a production setup, you would hopefully not execute this manually as a user, unless it's for initial bootstrapping or solving the chicken-and-egg problem. Once the infrastructure is deployed, additional Terraform modules are used to deploy further infrastructure in a Kubernetes context.

Important note

It's worth noting that there are native approaches available for integrating Terraform with GitOps, such as the **Terraform Controller**. This tool enables a variety of GitOps models tailored for Terraform resources, including full automation, hybrid automation for partial infrastructure management, state enforcement, and drift detection, all within a native framework. However, this chapter focuses more on the native use of tools.

This approach is sound and was even standard a few years ago. Here, the infrastructure is treated and deployed as an application through the declarative module approach.

However, there are some drawbacks to this method:

- **Lack of continuous monitoring**: This approach does not continuously monitor the infrastructure state for any drift, which is a key component in GitOps
- **Complexity in scalability**: As the infrastructure grows, managing it through Terraform alone can become increasingly complex, especially in terms of state management, multiple stages, and module dependencies

- **Limited dynamic response**: The approach lacks the dynamic response to changes that a GitOps workflow offers, where real-time adjustments to the infrastructure can be managed more seamlessly
- **Overhead in managing modules**: Relying heavily on Terraform modules for every aspect of deployment can lead to overhead in module management and version control
- **Potential for configuration drift**: Without continuous reconciliation, there's a higher risk of configuration drift over time as manual updates may not be tracked or recorded consistently

Use case 2 – integrating Terraform and Argo CD for deployment processes

The next approach, illustrated in *Figure 7.2,* combines the first method and then delegates the task to Argo CD. This approach is common in practice and often encountered in various projects with different clients. In this method, the Azure infrastructure is initially rolled out using Terraform modules. Subsequently, Argo CD is deployed as the initial instance using Terraform modules. Afterward, Argo CD takes over the GitOps part and deploys the Kubernetes context-related infrastructure. In this approach, a declarative method is best. Terraform modules are laid out declaratively, and the infrastructure, such as **Cert-Manager**, is also presented declaratively as *Applications* or *ApplicationSets*, as described in previous chapters:

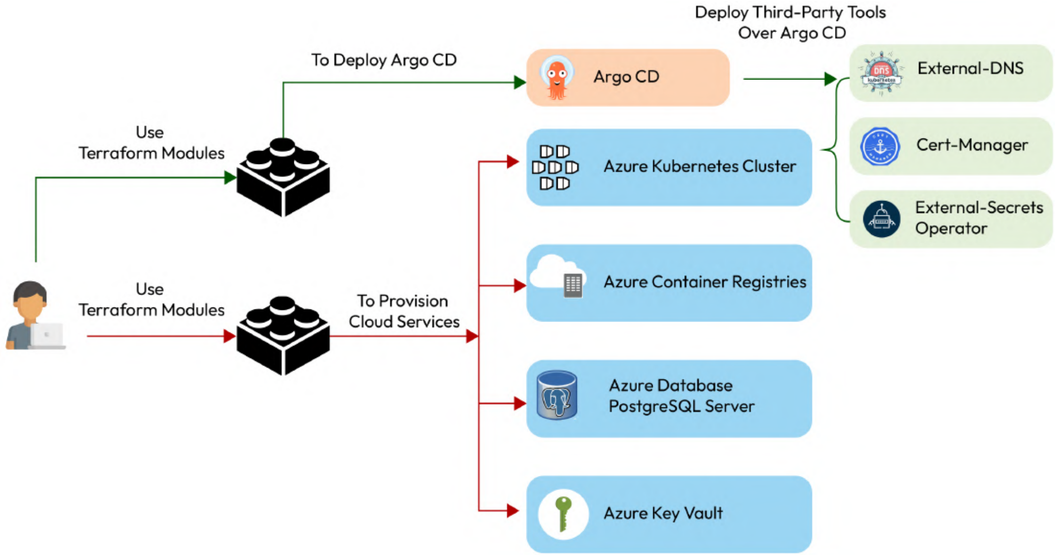


Figure 7.2 – Use case 2 – integrating Terraform and Argo CD for deployment processes

While this approach has its advantages, there are potential drawbacks to consider:

- **Complexity of integration**: Combining Terraform with Argo CD can add complexity, requiring a solid understanding of both tools
- **Initial learning curve**: For teams new to either Terraform or Argo CD, the learning curve might be steep due to the integration of these two different paradigms
- **Overhead in maintenance**: The combined approach might require more maintenance effort as it involves managing two systems rather than one
- **Risk of misconfiguration**: With two powerful tools at play, there's an increased risk of misconfiguration, which can lead to deployment issues or security vulnerabilities
- **Update coordination**: Coordinating updates between the infrastructure managed by Terraform and applications managed by Argo CD requires careful planning to avoid conflicts and ensure smooth operations

Use case 3 – unified management on Kubernetes – full-scale orchestration with CRs and Crossplane

The next approach, depicted in *Figure 7.3*, fully utilizes the Kubernetes API while focusing on **CRDs** and **custom resources** (**CRs**), which allows users to define their resource types and instantiate these types for custom configurations and functionality. This approach uses Crossplane for provisioning cloud infrastructure and deploying third-party tools necessary for Kubernetes infrastructure, such as Cert-Manager. It treats infrastructure as applications through the declarative nature of CRs, providing a fully trackable solution that leverages the Kubernetes ecosystem as an orchestration platform to keep resources in sync. This concept can be expanded so that virtually all cloud infrastructure is provisioned through Argo CD in combination with Crossplane. Everything is maintained within the cluster, requiring only an understanding of Kubernetes manifests – in this case, CRDs and CRs:

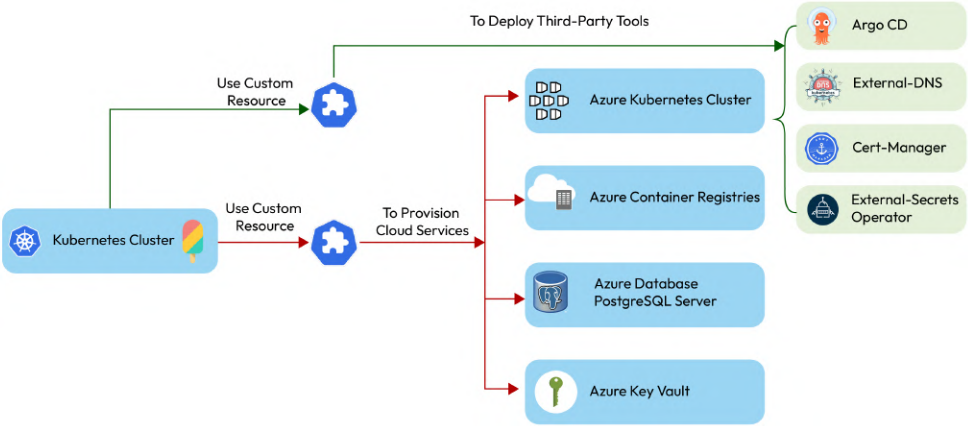


Figure 7.3 – Use case 3 – unified management on Kubernetes – full-scale orchestration with CRs and Crossplane

For a better understanding of how to create a resource in Azure with Crosslplane, CR, and Argo CD, here is a small example using the CR `VirtualNetwork` type. The requirement for this is that Kubernetes is installed and that the Azure provider is installed and configured [*3*].

Once the preparation is complete and the network provider is installed, we can create a managed resource of the `VirtualNetwork` type that can be used for AKS, for example.

First, create a CR, like so:

```
apiVersion: network.azure.upbound.io/v1beta1
kind: VirtualNetwork
metadata:
  name: crossplane-quickstart-network
spec:
  forProvider:
    addressSpace:
      - 10.0.0.0/16
    location: "West Europe"
    resourceGroupName: docs
```

Then, push the CR to your Git repository that will be managed by Argo CD. The CD part should now happen over GitOps with Argo CD and you should see the created managed Azure network resource in the portal.

However, this setup assumes the existence of a managed cluster where the necessary tools are deployed and configured. It comes with the following potential weaknesses:

- **The complexity of a Kubernetes ecosystem**: It requires a thorough understanding of Kubernetes, including CRDs and CRs, which might be challenging for teams less familiar with these concepts
- **Dependency on a managed cluster**: The approach is contingent on a managed Kubernetes cluster, limiting its applicability in environments where such a setup is not available
- **Resource intensiveness**: The approach might be resource-intensive, requiring more compute power and memory within the cluster
- **Configuration and maintenance**: Managing and maintaining the configurations of CRDs and CRs can be cumbersome and error-prone, especially in large-scale deployments and version upgrades

As you can see, there is no one-size-fits-all approach that meets all requirements. The right approach for a team depends greatly on the company's specific needs, the size of the teams, the team's skill set in various areas, and more. However, we should still take a moment to revisit why treating infra-as-apps is a game-changer.

Why infra-as-apps is a game-changer?

To summarize, here's a list of the benefits of infra-as-apps that makes it a game-changer:

- **Security**: By centralizing key management and shifting from direct cloud access to change management via Git, infra-as-apps enhances security. This approach minimizes risks associated with direct cloud access and improves audit trails by tracking changes through version control.
- **Efficient resource management**: This model streamlines the provisioning of new resources and the upgrading of existing ones. It allows for more agile and responsive infrastructure management, reducing the time and effort required for resource allocation and updates.
- **Improved CI/CD for pull requests**: Infra-as-apps focuses on enhancing pull requests within CI/CD pipelines, offering automated rollbacks and more thorough testing. This leads to more robust and reliable deployment processes, ensuring higher quality and stability in production environments.
- **Easier multi-cloud provisioning**: With tools such as Crossplane, infra-as-apps facilitates provisioning across various cloud environments. This capability simplifies the implementation of multi-cloud strategies, making it easier to manage and deploy resources across different cloud platforms efficiently.
- **Simplified infrastructure management**: This approach significantly streamlines the process of managing infrastructure, making it more efficient and less prone to human error.
- **Ease of onboarding and resource provisioning for developers**: It simplifies the process for developers to get started and manage resources, reducing the time and complexity involved in setting up and deploying applications.
- **Faster mean time to recover (MTTR)**: It improves the speed of recovery from failures.
- **Solid auditability**: It provides comprehensive audit trails and clear accountability, making it easier to track changes and maintain compliance with regulatory standards.

There are numerous companies, such as CERN, Splash, Grafana Labs, IBM, and SAP, that have already adopted Crossplane for a variety of use cases. A prime example of infra-as-apps in action can be seen at CERN, as presented by Ricardo Rocha at GitOpsCon US 2021, *A Multi-Cluster, Multi-Cloud Infrastructure with GitOps at CERN* [*2*]. CERN, renowned for operating the Large Hadron Collider, leverages infra-as-apps to manage an extensive infrastructure, including over 600 clusters, 3,000 nodes, 13,000 cores, 30 TBs of RAM, and 160 TBs of raw storage, to process vast amounts of data from experiments. They manage a massive array of clusters and computing resources using Kubernetes. CERN's implementation involves Helm charts with Crossplane configurations for cluster and resource management, allowing rapid scaling and multi-cloud resource allocation.

As a second example, IBM's use of Crossplane and GitOps to manage the infrastructure life cycle and application deployments is noteworthy. In *GitOpsify Everything: When Crossplane Meets Argo CD* [*4*], presented by Ken Murray, a CI/CD engineer at IBM, and Ying Mo, a software engineer, they explore the integration of Crossplane with Argo CD in various IT environments. IBM's approach involves using Crossplane as an abstraction layer for infrastructure provisioning and cluster management, enabling them to maintain a pool of Kubernetes clusters efficiently. This method leverages Helm templates for defining resource configurations and simplifies cluster management by allowing users to interact with simple resource types while Crossplane handles communication with cloud providers.

Infra-as-apps distinguishes itself from traditional **infra-as-code** by fully embracing GitOps at all levels. This approach treats manual changes to infrastructure as deviations from the Git-defined desired state, enhancing security and predictability in infrastructure management. It represents a significant shift in how infrastructure is managed, aligning it more closely with the dynamic and automated nature of application development and deployment.

GitOps has changed a lot, creating an immutable infrastructure that also has an impact on a company's culture. In the next section, we will look at immutable infrastructure.

Understanding the principles of immutable infrastructure

The **principles of immutable infrastructure**, combined with the GitOps approach, offer a transformative way of managing infrastructure and deployments in modern software development environments. Immutable infrastructure refers to a model where servers, once deployed, are never directly modified. Instead, changes require the server to be replaced with a new instance. This model drastically reduces issues caused by configuration drift and enhances the reliability of systems by treating servers as disposable units that can be replaced quickly, offering greater scalability and efficiency in deployment processes.

Incorporating GitOps into this model amplifies these benefits. GitOps leverages Git as the central source of truth for managing both software applications and infrastructure. It applies cloud-native patterns to deployments, often associated with Kubernetes but also applicable to various platforms. The core principles of GitOps include treating infrastructure as code, ensuring versioned and immutable deployments, automatic pulling of the desired state, and continuous reconciliation between the desired and actual states.

The essence of immutable infrastructure

Immutable infrastructure is a concept where servers, once deployed, are never modified; if a change is needed, a new instance is created and replaced. This approach is fundamentally different from traditional mutable infrastructure, where servers are continually updated and modified. The immutable model brings several key advantages.

The advantages of immutable infrastructure are as follows:

- **Consistency and reliability**: Immutable servers remain in a known, stable state, significantly reducing the risk of unexpected issues due to environmental drift or inconsistencies
- **Enhanced security**: With servers not being modified post-deployment, the attack surface remains constant, simplifying security management and anomaly detection
- **Simplified management and debugging**: Troubleshooting becomes more straightforward as the infrastructure stays in a predictable state
- **Scalability and performance**: New instances can be spun up and down efficiently to meet demand, maintaining a consistent performance level

However, this approach isn't without its challenges:

- **Increased storage and resource needs**: Immutable infrastructure can require more storage since each change involves creating a new instance
- **Potential deployment overhead**: Setting up an entirely new instance for every change can be more resource-intensive than updating an existing one
- **Complexity in configuration management**: Managing numerous server instances can become complex, particularly in large-scale environments

Integrating immutable infrastructure with GitOps

Incorporating GitOps into immutable infrastructure leverages Git's core principles – version control, collaboration, and automation – to operational management, enabling a system where infrastructure is treated as code, with all configurations and states meticulously maintained in Git repositories for enhanced consistency and traceability.

The advantages of GitOps are as follows:

- **Single source of truth**: Having a single repository for both application and infrastructure code ensures consistency and traceability
- **Automated, reliable deployments**: Continuous deployment via GitOps operators automates and simplifies the deployment process
- **Enhanced collaboration and transparency**: Pull request workflows enhance collaboration, providing clear audit trails and ensuring accountability in changes
- **Improved security and compliance**: Git's inherent features facilitate robust access control, encryption, and compliance adherence

Despite these benefits, GitOps is not without its drawbacks:

- **Complexity in scaling**: Managing multiple repositories and dealing with the complexity of large-scale deployments can be challenging.
- **Git limitations for operational tasks**: Git, primarily designed for code version control, may not be the best fit for operational updates, leading to potential conflicts.
- **Learning curve and adoption resistance**: Teams may face a steep learning curve or resistance to adopting new workflows, tools, and mindsets. I will cover this part later.

Synergy and challenges

When immutable infrastructure and GitOps are combined, they create a powerful synergy:

- **Infrastructure as a dynamic, version-controlled entity**: Infrastructure is treated similarly to application code, with changes tracked, reviewed, and deployed using GitOps methodologies
- **Rapid recovery and rollback capabilities**: The combination allows for quick recovery from failures and easy rollbacks to previous stable states
- **Streamlined operations**: This approach reduces the need for manual intervention, minimizing human error and streamlining operations

However, this integration also presents unique challenges:

- **Complex workflow management**: Balancing the immutable aspects of infrastructure with the dynamic nature of GitOps workflows requires careful planning and execution
- **Dependency on tooling and processes**: The effectiveness of this approach is heavily reliant on the right tooling and well-defined processes
- **Balancing security and agility**: Ensuring security in a rapidly changing environment necessitates a fine balance between rigid controls and operational flexibility

Achieving an effective production environment using immutable infrastructure

The optimal immutable infrastructure for production environments can be conceptualized by integrating several key practices, refined through experience. Here's how an effective production environment might look and why:

- **Read-only access to a Kubernetes prod cluster**: In a productive environment, a Kubernetes cluster should be treated like a managed service or a *black box* from a developer's perspective. This means that access to the cluster should be primarily read-only. Developers should not be permitted to make manual changes to the cluster, ensuring a controlled and stable environment.

- **Only create and delete resources through GitOps**: This principle dictates that all resources entering the cluster should be managed exclusively through GitOps. This includes the entire creation process, covering elements such as namespaces, deployments, services, ingress, service accounts, and more. This approach ensures that changes are **traceable**, **reversible**, and **consistent** with the source of truth in the Git repository.
- **Do not create namespaces over applications**: In the context of Argo CD and its concept of application, an application mustn't be able to create namespaces. This is because Argo CD does not support the deletion of namespaces created by an application *[5]*. This restriction is in place to prevent uncontrolled namespace creation, which could lead to conflicts and management issues in shared environments. In the following example, I will clarify the situation for you by illustrating the practical implications and potential risks associated with managing namespaces in a shared environment using Argo CD.

  Consider a scenario where five teams share a single Kubernetes namespace. Each team creates its own Argo CD application to deploy its service components, contributing to the overall microservices architecture. Suppose Team A decides to decommission its service. They proceed to delete their application and the associated deployed service, and they also attempt to delete the shared namespace through their application. However, Argo CD applications do not inherently have knowledge of or connections to other applications or unlinked resources within the same namespace. Consequently, if such a deletion were permitted, it would inadvertently result in the removal of the namespace and all other services deployed by the different teams within it. Fortunately, this scenario is not feasible in practice, as the Argo project team does not implement such a feature to avoid these kinds of disruptive and unintended consequences.
- **Use the right tools, such as PR-Generator**: Integrating tools such as PR-Generator in a GitOps workflow with Argo CD can greatly enhance the testing and deployment processes. PR-Generator can create resources with a specific prefix as soon as a PR is initiated, ensuring that resources are correctly created and cleaned up, respecting the *do not create namespaces over application* principle.

By adhering to these practices, several benefits can be realized:

- Every change is made through Git, ensuring a centralized and version-controlled change management process
- Changes are recorded in the Git history, providing a transparent and traceable record of all modifications
- Resources are cleanly managed, with no residual "orphaned" resources left in the cluster
- Clean and efficient workflow management using PRs and Git, facilitated by tools such as PR-Generator
- It establishes a standard and commitment within the organization, allowing these practices to be replicated across different projects

Implementing these principles in a production environment creates a robust, secure, and manageable infrastructure that leverages the benefits of both immutable infrastructure and GitOps methodologies.

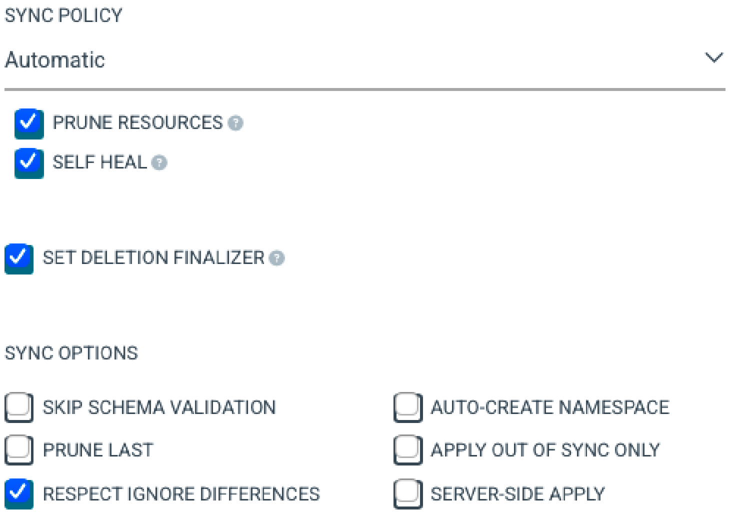
A potential folder structure for an application that utilizes both **Kustomize** and Argo CD might be structured as follows:

```
.
├── base
│   ├── kustomization.yaml
│   ├── namespace.yaml
│   ├── role.yaml
│   ├── rolebinding.yaml
│   ├── service.yaml
│   ├── serviceaccount.yaml
│   ├── statefulset.yaml
│   ├── templates
│   │   └── statefulset_template.yaml
│   └── values.yaml
└── overlays
    ├── production
    │   ├── kustomization.yaml
    │   ├── namespace.yaml
    │   └── statefulset.yaml
    └── staging
        ├── kustomization.yaml
        ├── namespace.yaml
        └── statefulset.yaml
```

This structure ensures that all resources can be deployed and cleaned up through Argo CD and the GitOps approach, eliminating the need to set the `AUTO-CREATE NAMESPACE` option in the Application Sync Options for namespace creation.

Application configuration for immutable infrastructure

Next, we will examine a potential application configuration that contributes to an immutable infrastructure setup:



SYNC POLICY

Automatic

☑ PRUNE RESOURCES
☑ SELF HEAL

☑ SET DELETION FINALIZER

SYNC OPTIONS

☐ SKIP SCHEMA VALIDATION
☐ PRUNE LAST
☑ RESPECT IGNORE DIFFERENCES
☐ AUTO-CREATE NAMESPACE
☐ APPLY OUT OF SYNC ONLY
☐ SERVER-SIDE APPLY

Figure 7.4 – Example of an application sync policy configuration

The following flags should be set if you want to get the most immutable infrastructure setup:

- **PRUNE RESOURCES**: Enable this option to allow Argo CD to automatically delete resources that are present in the cluster but no longer defined in the Git repository. This ensures that all resources in the cluster are managed through the GitOps approach.
- **SELF HEAL**: Enable this option. It ensures that any changes made directly in the cluster that deviate from the Git repository are automatically corrected by Argo CD, maintaining the desired state as defined in Git.
- **SET DELETION FINALIZER**: Enable this option to ensure that resources are only deleted from the cluster when they are removed from the Git repository. This prevents premature deletion and ensures resources are cleaned up only after their removal from Git.
- **RESPECT IGNORE DIFFERENCES**: Enable this to instruct Argo CD to ignore specific differences it doesn't recognize. You can define in the Argo CD configuration what specific differences to ignore.
- **AUTO-CREATE NAMESPACE**: Ensure this option is disabled to prevent Argo CD from automatically creating namespaces. Namespaces should be explicitly defined as resources in your Git repository.

By setting these policies, you ensure that resource creation, deletion, and synchronization are all managed through changes in your Git repository while adhering to the principles of GitOps. This approach ensures a tight coupling between the declared state in Git and the actual state in the cluster.

The combination of immutable infrastructure and GitOps represents a modern approach to IT operations and development. It brings together the best of stability, security, and dynamic, automated workflows. While it offers significant advantages in terms of reliability, security, and efficiency, it also poses challenges in terms of complexity, resource demands, and dependency on specific tools and practices. As with any IT methodology, success in implementing these concepts depends on a clear understanding of their benefits and limitations, careful planning, and an adaptive approach to evolving IT landscapes.

So far, we've looked at various aspects of transformation that come with GitOps. But what's the point of all this if you can't get or measure any added value from it? That's why, in the next section, we'll look at specific metrics that make product quality measurable.

Introducing DORA metrics

In this section, we'll introduce you to DORA metrics and delve into a set of key performance indicators developed by the DORA team. These metrics are pivotal in evaluating and improving software delivery and operational performance within IT teams. The four principal DORA metrics are **Deployment Frequency**, **Lead Time for Changes**, **Change Failure Rate**, and **MTTR**:

- **Deployment Frequency**: This assesses how often a team successfully deploys code to production. Frequent deployments are indicative of an efficient and responsive development process, allowing for quicker feedback and **continuous improvement**. To increase the frequency of deployments, adopt CI/CD practices. Automate your build and deployment pipelines using tools such as **Jenkins**, **GitHub Actions**, or **GitLab CI**. Regular, smaller deployments reduce risks and facilitate quicker feedback. Additionally, feature flags can be used to manage deployments and gradually roll out new features.
- **Lead Time for Changes**: This measures the duration from code commit to code deployment in production. This metric indicates the speed and agility of the development process. Shorter lead times often result in faster feedback from users and the ability to adapt and improve the software swiftly. Improve lead time by optimizing your development processes. This includes adopting Agile methodologies, increasing collaboration between teams, and integrating code review and testing into the development process. Utilize automated testing and **continuous integration** tools to ensure that code changes are tested and integrated quickly. Tracking tools can be used to monitor the time from code commit to deployment, helping identify bottlenecks.
- **Change Failure Rate**: This focuses on the percentage of deployments that result in a production failure. This metric is essential for understanding the reliability and stability of the software development process. A lower change failure rate signifies higher quality and reliability in software deployments. To reduce the change failure rate, focus on enhancing the **quality of code** and the reliability of deployments. Implement automated testing, including unit tests, integration tests, and end-to-end tests, to catch issues early. Employ static code analysis tools and conduct thorough code reviews. Practice continuous deployment with automated rollbacks, and use **canary releases** or **blue-green deployments** to minimize the impact of failed deployments.

- **MTTR**: This is the average time taken to recover from a failure in the production environment. This metric is crucial as it reflects the team's ability to quickly rectify issues, thereby minimizing downtime and maintaining service quality. Reducing MTTR involves improving incident response and recovery processes. Implement monitoring and alerting tools such as Prometheus, Grafana, or New Relic to quickly detect issues. Establish clear incident management protocols and on-call rotations. Use IaC tools such as Terraform to enable fast and consistent environment recovery. Regularly practice incident response drills to ensure the team is prepared for quick recovery.

These metrics align well with the GitOps approach, which emphasizes automation, monitoring, and quick feedback. GitOps can enhance Deployment Frequency and Lead Time for Changes through automated pipelines. It also helps in reducing the Change Failure Rate and MTTR by enabling quicker rollbacks and promoting more reliable deployments through practices like continuous integration and delivery.

However, while DORA metrics provide valuable insights, they come with challenges. Implementing these metrics requires a mature DevOps team and established CI/CD processes. Data collection and tagging need to be precise and actionable. The metrics must be adapted to the unique cadences and processes of different teams and products.

In terms of measuring these metrics independently from specific Git **Source Version Controls** (**SVCs**) such as DevOps, GitHub, or GitLab, organizations can look toward integrating various tools that provide necessary data visualizations and dashboards. Teams can leverage APIs from different tools to gather relevant data points for calculating these metrics. For example, Deployment Frequency can be monitored using tools that track code deployments, while Lead Time for Changes can be calculated by extracting timestamps for code commits and deployments from version control systems.

Organizations can choose to implement DORA metrics either by developing logic tailored to their environments or by utilizing established tools such as **GitLab's Value Stream Analytics Dashboard**. GitLab's dashboard [6] effectively visualizes the software development life cycle and computes DORA metrics. It provides an interactive interface for tracing essential phases of the software delivery process and enables teams to customize the dashboard to align with their specific workflow, enhancing the understanding of lead time, cycle time, and other vital metrics. This feature aids in identifying bottlenecks and areas needing improvement, thus boosting deployment frequency and overall DevOps performance.

Similarly, **SquaredUp** specializes in Azure DevOps integration [7] and offers a DORA metrics dashboard designed specifically for DevOps teams. This dashboard presents a comprehensive view of crucial performance indicators, including Deployment Frequency, Lead Time for Changes, Change Failure Rate, and MTTR. It supports teams in monitoring and analyzing their software delivery process, facilitating the identification of areas for enhancement. SquaredUp's solution emphasizes data-driven decision-making in software development, showcasing the importance of metrics in optimizing DevOps workflows.

The key to successfully utilizing DORA metrics lies in understanding the context of these measurements, interpreting them collectively rather than in isolation, and adapting them to suit the unique requirements of individual teams and projects.

In conclusion, DORA metrics provide a structured and quantifiable approach to assessing and improving DevOps performance. They offer a comprehensive view of the software delivery process, highlighting areas for improvement and fostering a **culture of continuous development** and operational efficiency.

Understanding the need for continual improvement in GitOps

In the ever-evolving landscape of software development, the adoption of GitOps marks a significant transition in collaborative approaches and operational efficiencies. GitOps, a methodology that combines Git with Kubernetes' operational workflows, is more than just a set of practices – it represents a continuous journey of improvement and adaptation. This section explores how GitOps necessitates and facilitates **continual improvement** across various domains such as security, development, platform engineering, and financial operations:

- **Evolving security practices with GitOps**: Security in the GitOps realm is dynamic and **continuously evolving**. Teams deploy policies using tools such as **Kyverno**, adjusting to the rapidly changing landscape of security threats. The emergence of new vulnerabilities and attack vectors demands a proactive and adaptive approach. GitOps supports this by allowing for quick iteration and deployment of **security policies**. However, the approach to managing secrets has seen shifts – from using tools such as Sealed Secrets to adopting External-Secrets Operators. This transition underscores the need for teams to constantly evolve their tools and practices, staying ahead in a domain where stagnation equates to vulnerability.
- **Developers' continuous innovation through GitOps**: For developers, GitOps is not just a method of deployment but a catalyst for **continuous innovation**. Techniques such as **blue-green** and **canary deployments** become more streamlined, while new tools such as PR-Generators enhance the deployment processes. These innovations are vital in a landscape where deployment strategies and tools are in a constant state of change. GitOps empowers developers to rapidly deploy, experiment, and iterate, thereby fostering a culture of continual improvement and agile responsiveness to market and technological changes.
- **Platform engineering and scalable infrastructure**: Platform engineering teams find GitOps an ally for managing infrastructure with efficiency and scalability. Patterns such as *App of Apps* and *ApplicationSets with Cluster Generator* in Argo CD demonstrate the evolving nature of infrastructure management. Tools such as Crossplane and Cluster API extend the capabilities of GitOps from mere application deployment to managing and provisioning diverse cloud resources. This evolution highlights the necessity for teams to continually adapt and enhance their skills and tools to manage increasingly complex and distributed systems effectively.

- **FinOps – financial optimization in the age of GitOps**: In the FinOps domain, GitOps introduces tools such as **OpenCost**, enabling detailed cost tracking and allocation down to the namespace level. Such granularity in cost allocation and the ability to adjust resource availability based on usage patterns represent a significant shift in managing cloud costs. It also illustrates the ongoing need for financial operations teams to integrate new tools and strategies to optimize costs continually. The use of tools such as **Kubegreen**, which adjusts resources based on demand, is a testament to the evolving nature of financial operations in cloud environments.

The journey with GitOps is ongoing and multifaceted. Teams across different domains – from security to development and platform engineering to FinOps – must not only adopt GitOps but also embrace its ethos of continual improvement. This involves regular evaluations, a willingness to adopt new tools and practices, and a commitment to learning and adaptation.

In the final section of this chapter, we will explore overcoming **cultural barriers**, adopting GitOps, and how different teams are currently working with varying approaches.

Overcoming cultural barriers to adopt GitOps

Adopting GitOps goes beyond just tool implementation; it involves a cultural shift within organizations. This transition involves breaking down traditional departmental barriers, nurturing collaboration, and uniting teams toward shared goals. GitOps acts as a unifying force, streamlining operations and development toward a more efficient IT environment. Continual improvement in GitOps is essential in the dynamic world of software development. This journey focuses on learning, adapting, and evolving practices.

When examining different definitions or contrasting GitOps with DevOps, the technical aspects can be summarized via the Weaveworks blog on GitOps culture, which elaborates on how GitOps extends beyond a set of tools or practices to encompass a broader cultural shift within organizations. This approach integrates Git at the heart of both operational and developmental processes, emphasizing transparency, accountability, and collaboration. GitOps culture is characterized by the centralization of Git in managing infrastructure, applications, and CI/CD processes, enhancing automation, stability, and consistency. It fosters an environment where team members are engaged in a continuous cycle of improvement, enabled by the visibility and control offered by Git-based workflows.

It might be right about the GitOps approach and the associated technical challenges, as well as the responsibility, collaboration, and more. I'm not saying it's wrong, because it isn't. However, I want to shift the focus to a perspective that also considers the daily work in various industries. Not every company is a software company like Weaveworks.

A project's story – exchange, experiences, and learnings

From my experience and discussions with **DevOps professionals**, **CTOs**, and **platform engineers**, it's clear that tool and approach changes are more straightforward than altering a team or company mindset, which requires time for transformation.

Some companies handle this like so:

- In our use of GitOps with Argo CD for infrastructure and application services, we've noticed significant improvements. For instance, we can easily identify drifts in our staging and production environments, which are monitored closely. For development and ephemeral environments, we use Helm and CD pipelines without tracking changes.
- Weaveworks on GitOps culture highlights how GitOps, particularly through Weave GitOps, fosters a strong team culture characterized by **excellent communication**, **coordination**, and **collaboration**. The clarity in **roles** and **responsibilities** provided by GitOps allows teams to **maintain autonomy**, supported by Kubernetes namespaces for organizing and isolating workloads. Additionally, Weave's acquisition of Magalix emphasizes security policies in GitOps, enabling admins to set and enforce policies throughout the GitOps process. This approach underscores the operational nature of GitOps culture, distinct from the aspirational culture of DevOps, focusing on precision and accountability.

And I believe Weaveworks hits the nail on the head when you look beyond the technical aspects and read closely. The dialog suggests that the GitOps approach fosters a new culture among different teams, who should master the same tool but for different use cases. The teams can work autonomously, but this requires excellent communication, coordination, and collaboration. Additionally, it involves defining roles and responsibilities. Every change or initiative should be clearly and transparently communicated to the other teams. This all sounds great, but how does it look in real projects with different teams? I will try to illustrate this with an example from one of my projects.

In the project, there's a platform team, several developer teams, and a security team. The security team's role is to ensure not only that security guidelines are adhered to within the organization, but also IT governance and compliance. Therefore, it feels responsible for using all possible means to protect the company and enforce policies throughout the GitOps process.

The platform team provides the necessary Kubernetes clusters and context for the developers, so they only need to focus on their applications and, at most, third-party tools. They see their role as enablers for the developers, who purchase and use the platform as customers, with maximum emphasis on **self-service**, **availability**, and **performance**. The developer teams consume Kubernetes as a product and develop their applications. Let's see how one team's reasonable decision can unintentionally harm another team:

- The security team enforces a policy that no application can run with elevated privileges, and some applications, both in the platform context and those self-developed, stop working. This results in application downtime and software failure.

The *rationale* for this action is to ensure that the company is protected.

- The platform team carried out a Kubernetes upgrade from 1.24 to 1.25 as support for the previous version ended. As a result, the third-party tools used by the security team no longer work. Some of the developers'applications also stop functioning. The change did not consider that the upgrade replaced *PodSecurityPolicies* with *PodSecurityStandards*.

  The *rationale* for this action is to ensure that a supported version is available and that the platform remains up to date as required by the security team. This responsibility includes ensuring that performance is optimized and reliable while maintaining adherence to the latest security protocols and compliance standards associated with the Kubernetes ecosystem.

- The developers use their autonomy and open a *NodePort* on a node with an external IP address for testing purposes. The application becomes externally accessible. The debugging works, but testing takes longer. Fortunately, the application uses **Log4j** version 2.10 for logging, simplifying the debugging process.

  The *rationale* for this action is to increase the performance of testing and debugging. The developers were under pressure to meet deadlines and chose a quick solution to test their applications, prioritizing immediate functionality over long-term security and compliance considerations.

Possible causes that arise from the different actions are that customers are frustrated, the company loses money per minute, and the reputation of the company declines. These are modified use cases and fortunately, my projects were not affected by the Log4J incident. But the point here is not what happened but that these cases occur and are justifiable from the team's point of view.

Reflecting on the causes of recent challenges, it seems that the varying explanations provided by the different teams are all valid. For example, when examining the changes made by the security and platform teams, they took accountability from their perspective, ensuring that the company was adequately protected and represented. However, they did not consider themselves responsible for application outages or customer loss, believing their actions were justified. This situation could have been potentially avoided with simple communication and transparency about the impending changes between teams.

From the developers' viewpoint, they were under pressure to meet a deadline to prevent further delays for a feature, which is already behind schedule, causing frustration among customers. Debugging was activated only in the development environment, and they assumed it was unnecessary to inform the security or platform teams, believing that these teams wouldn't understand the change and would impede it without reason. The developers felt they were acting in the company's best interest and aiming to satisfy their customers.

From the developers' viewpoint, they felt a strong sense of accountability to meet a deadline. However, they overlooked the importance of simple communication and transparency with the security or platform teams, assuming these teams wouldn't comprehend the change and might unnecessarily impede it. In their pursuit to act in the company's best interest and satisfy customers, they missed an opportunity for collaboration that could have harmonized their efforts with the broader organizational goals.

The learnings

The learnings from these incidents led to the creation of a guild, with representatives from each team attending meetings. This allowed for changes and their impacts to be discussed and made visible, fostering clear communication and collaboration between teams. Additionally, it helped different teams understand each other's perspectives, needs, and reasons, enhancing transparency. While not all decisions were discussed, and changes were sometimes made rapidly due to time constraints or urgency, such as in the case of critical security vulnerabilities, the overall situation improved. The increased transparency helped in better understanding the impact of changes, reducing the blast radius of these changes. Empathy among team members increased, and the culture of blaming significantly decreased.

Did things overall improve? *Yes*!

Were all decisions discussed and changes made only after consultation? *No*, as time constraints and urgency often lead to decisions being made quickly.

Essential Q&A from another recent project

In this section, we'll focus on specific questions and answers from a company, and various projects, that are still in the process of transformation, having started with the DevOps approach and now shifted to platform engineering, including their honest reasoning for this change:

- Q1: How did GitOps restructure your team organization?

  A1: Initially, we had separate DevOps teams per project. GitOps led to the creation of a central platform team to manage Kubernetes clusters efficiently.

- Q2: What were the challenges in transitioning to GitOps?

  A2: Our main challenge was adapting to the **immutable infrastructure** GitOps offers, especially understanding that all changes need to flow through Git commits.

- Q3: What benefits have you realized with GitOps?

  A3: GitOps has brought us a clear, traceable workflow. Every change, big or small, is now meticulously tracked via Git.

- Q4: How has GitOps affected team communication?

  A4: Communication has significantly improved. Our platform team has developed a **deeper understanding** of the developers' needs, leading to more **empathetic collaboration**.

- Q5: Can you share any successful cultural transformations?

  A5: Definitely. We've developed a **proactive error culture**, focusing on automation and **continuous learning**, which has fundamentally changed our problem-solving approach.

- Q6: How has daily work evolved for your teams with GitOps?

  A6: There's a noticeable increase in **motivation** and **responsibility** among team members, with a greater sense of **autonomy** in their roles.

- Q7: How did your team adapt to GitOps?

  A7: Surprisingly, there was a little resistance. The team understood the necessity of this transformation and was quite receptive to it.

- Q8: How do you measure the success of this transition?

  A8: While we don't have precise metrics yet, the reduction in **manual interventions** and the increase in **Git activities** are strong indicators of success.

- Q9: What is your team size?

  A9: Our team consists of around 20 developers and three platform engineers.

This conversation format allows for a clear understanding of how GitOps has transformed team dynamics, challenges, and workflow efficiency.

Adopting GitOps is more than implementing a set of tools; it involves cultural shifts within organizations. Overcoming traditional silos between departments, fostering a culture of collaboration and continuous learning, and aligning diverse teams toward common operational goals are essential steps in this journey. Thus, the adoption of GitOps becomes a unifying force, aligning varied operational and developmental efforts toward a more efficient, transparent, and responsive IT environment.

In conclusion, the necessity for continual improvement in GitOps is not just a recommendation; it is a requirement in the fast-paced and ever-changing world of software development and IT operations. As teams navigate this journey, the focus should remain on learning, adapting, and evolving practices to stay ahead in an environment where change is the only constant. So, GitOps is less of a destination and more of a path toward operational excellence and continual improvement.

Summary

This chapter provided a compelling view of the cultural revolution in IT, brought about by the adoption of GitOps. It started by establishing GitOps as not just a technological shift but a profound cultural metamorphosis within IT departments. This approach, merging software development with IT operations through Git, transforms operational processes and methodologies, treating infrastructure as an application. It enables automation and precision and extends rigorous development practices to infrastructure management.

A key transformation brought about by GitOps is the establishment of immutable infrastructure, automating the alignment of the infrastructure's actual state with its desired state, and reducing manual interventions and errors. This chapter also explored the impact of GitOps on IT performance through DORA metrics, offering a quantitative assessment of improvements in deployment frequency, change lead time, failure rate, and recovery time. This chapter also addressed the broader organizational implications of GitOps adoption. This goes beyond mere tool implementation; it's a catalyst for a far-reaching cultural shift, fostering a collaborative, transparent, and efficient IT culture. Real-world examples from various organizations illustrated how GitOps drives these changes.

All in all, this chapter revealed the transformative power of GitOps, transcending technology to chart a path toward a more agile and innovative IT culture.

In the next chapter, we'll explore the application of GitOps in conjunction with OpenShift, a Kubernetes downstream distribution from Red Hat, and examine how its management differs.

References

- [*1*] `https://codefresh.io/blog/infrastructure-as-apps-the-gitops-future-of-infra-as-code/`
- [*2*] `https://www.youtube.com/watch?v=h6xDWc6fXao`
- [*3*] `https://docs.crossplane.io/latest/getting-started/provider-azure/`
- [*4*] `https://www.youtube.com/watch?v=9odjdVqJkws`
- [*5*] `https://github.com/argoproj/argo-cd/issues/7875`
- [*6*] `https://docs.gitlab.com/ee/user/analytics/value_streams_dashboard.html`
- [*7*] `https://squaredup.com/dashboard-gallery/dora-metrics-dashboard-devops-team/`

Part 3:

Hands-on Automating Infrastructure and CI/CD with GitOps

In this part, you will get hands-on experience with automating infrastructure and CI/CD processes using GitOps. You will explore specific implementations on platforms such as OpenShift, Azure, and AWS, and integrate tools such as Terraform and Flux CD for enhanced automation. This practical section is designed to equip you with the necessary skills to apply GitOps in diverse environments, ensuring you can manage infrastructure and CI/CD pipelines effectively and efficiently.

This part includes the following chapters:

- *Chapter 8*, GitOps for OpenShift
- *Chapter 9*, GitOps for Azure and AWS Deployments
- *Chapter 10*, GitOps for Infrastructure Automation – Terraform and Flux CD
- *Chapter 11*, Deploying Real-World Projects with GitOps on Kubernetes

8

GitOps with OpenShift

In this chapter, we delve into the world of **GitOps** within the context of **Red Hat OpenShift**, an approach that revolutionizes the way we deploy, manage, and operate containerized applications.

As we journey through this chapter, we will explore the foundational principles of GitOps, highlighting how they seamlessly integrate with the robust features of Red Hat OpenShift. This combination not only enhances the efficiency and reliability of deployments but also aligns with modern DevOps practices, fostering collaboration between development and operations teams.

We will start by setting up a local OpenShift environment using **CodeReady Containers** (**CRC**), providing a hands-on approach to understanding how OpenShift operates and how it can be managed using GitOps principles. From there, we will dive into practical examples and best practices, including configuring automated deployments, managing application configurations, and ensuring high availability and security within your OpenShift cluster.

Whether you are new to OpenShift or looking to enhance your existing workflows, this exploration into GitOps will provide valuable insights and techniques for optimizing your cloud-native development processes.

In this chapter, we'll focus on these key areas:

- Introduction to Red Hat OpenShift
- Setting up GitOps in Red Hat OpenShift
- Leveraging Red Hat OpenShift's CI/CD for GitOps
- Automation and configuration best practices
- A Kubernetes-Red Hat OpenShift comparison

Technical requirements

For this chapter, ensure you have access to a Red Hat OpenShift cluster or a local setup via Red Hat OpenShift Local. Familiarity with GitOps practices and tools such as Argo CD, as well as basic development tools such as Git and a code editor, is necessary. A foundational understanding of container technology, Kubernetes architecture, and YAML syntax will aid in following the practical exercises. Ensure a stable internet connection for tool downloads and cluster access. If running OpenShift locally, your machine should ideally have at least 16 GB of RAM and sufficient storage.

The relevant code and resource files for this chapter can be found in the `Chapter08` folder of our dedicated GitHub repository: `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`.

Introduction to Red Hat OpenShift

OpenShift (`https://www.redhat.com/en/technologies/cloud-computing/openshift`), developed by Red Hat, is a leading enterprise Kubernetes platform that offers a comprehensive suite of features designed to streamline the deployment, scaling, and operations of containerized applications. It extends Kubernetes by simplifying many of the complex tasks associated with deploying and managing a containerized infrastructure, making it an ideal choice for enterprises looking to implement modern DevOps and GitOps practices.

One of the core strengths of OpenShift is its focus on **developer productivity** and **operational efficiency**. It provides a rich set of developer tools and a user-friendly console that makes it easier for developers to build, deploy, and manage their applications. OpenShift's **integrated development environment** (**IDE**) and built-in CI/CD capabilities enable developers to automate much of the software delivery process, from code build through to testing, deployment, and monitoring.

Security is another area where OpenShift shines. It incorporates built-in security features at every level of the application stack, from the **operating system** (**OS**) to the network, and up to the application services. This integrated approach ensures that security is not an afterthought but is woven into the very fabric of the application life cycle. Features such as **Security-Enhanced Linux** (**SELinux**) enforced isolation, automated vulnerability scanning, and default secure configurations help to protect applications and data from external threats and internal misconfigurations.

SELinux

SELinux is a kernel-integrated security feature that enforces **mandatory access control** (**MAC**) policies to enhance system security. It operates on the principle of least privilege, restricting system processes and users to the minimal permissions needed for their functions. SELinux helps prevent unauthorized access and can operate in enforcing mode, where it blocks and logs unauthorized actions, or in permissive mode, where it only logs violations without blocking them. This adds an important layer of security to Linux-based systems, including OpenShift, by controlling access with greater granularity than traditional access controls.

Furthermore, OpenShift is designed to support a wide range of workloads, from traditional monolithic applications to microservices and serverless functions. Its flexible architecture supports a variety of programming languages, frameworks, and databases, allowing teams to use the tools and technologies that best fit their needs. Additionally, OpenShift's container-native storage and networking solutions provide the scalability and performance required for high-demand applications.

In the context of GitOps, OpenShift provides a powerful platform for implementing declarative, version-controlled workflows for infrastructure and application deployment. By leveraging OpenShift's capabilities, teams can achieve higher levels of **automation**, **repeatability**, and **consistency**, which are central to the GitOps philosophy. With its robust ecosystem, OpenShift enables organizations to streamline their deployment pipelines, enhance security, and improve operational efficiency, making it an excellent choice for adopting GitOps methodologies.

Red Hat OpenShift environment setup

When setting up an OpenShift cluster for learning, testing, or development purposes, you have several options to consider, each with its own set of advantages. Understanding these alternatives will help you choose the best fit based on your objectives, available resources, and technical comfort level:

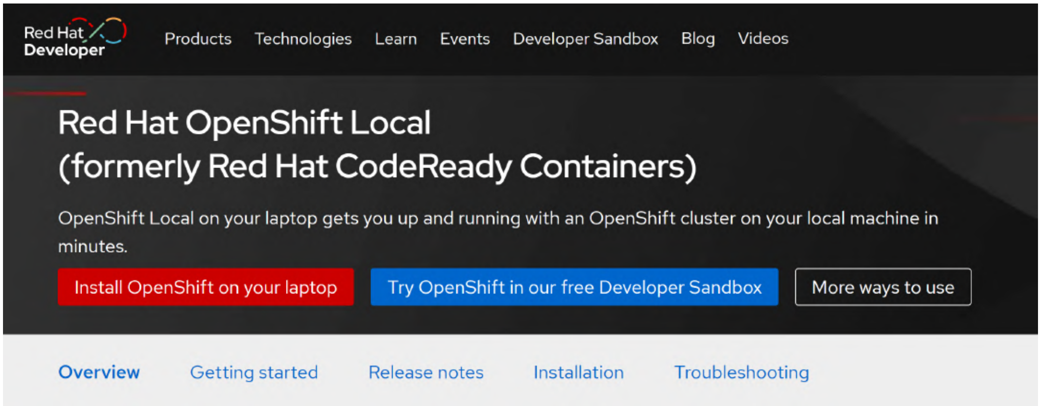
- **Developer Sandbox for OpenShift**: This is an excellent starting point for beginners or for those looking to test out OpenShift features without any setup. The Developer Sandbox (`https://developers.redhat.com/developer-sandbox`) provides a free, pre-configured OpenShift environment. All you need to do is register to gain 30 days of free access to a shared OpenShift and Kubernetes cluster. This option is ideal for users who prefer to avoid the complexities of manual setup and are looking for a quick way to start experimenting with OpenShift capabilities.
- **Cloud-based OpenShift cluster**: Setting up an OpenShift cluster in the cloud (Azure, AWS, or GCP) is a robust solution that offers scalability and real-world applicability. This approach is suitable for users looking to simulate a production environment or to work on larger, more complex projects. However, it's important to note that running a cluster in the cloud can incur costs, depending on the cloud provider and the resources consumed.
- **Bare metal setup**: Setting up OpenShift Local on bare metal is a challenging and resource-intensive task. Unlike in cloud environments, you must manually manage networking, storage, and hardware compatibility, requiring a high level of technical expertise. The system demands significant memory, multiple CPU cores, and substantial storage, which can be daunting for environments not initially designed for such loads. While a bare metal installation offers greater control and potential performance benefits, it comes with complexities and potential hurdles that need careful planning and understanding.
- **Red Hat OpenShift Local**: For those who prefer a local solution, setting up an OpenShift cluster on your machine using Red Hat CRC, can be an effective approach. This method allows for a more hands-on experience and works well for individual learning and development scenarios.

Red Hat OpenShift Local provides a fast and straightforward method for initiating OpenShift cluster construction. Tailored for local machine operation, it streamlines the setup and testing process, offering a cloud-like development environment right on your desktop. This tool supports various programming languages and simplifies the development of container-based applications by providing a minimal, pre-configured OpenShift cluster that requires no server-based infrastructure.

OpenShift Local enables the creation, building, and deployment of microservices directly into Kubernetes-hosted containers, supporting a range of OSs including Linux, macOS, and Windows 10. This makes it an ideal platform for local development and testing, mirroring a real OpenShift cluster's environment.

For the remainder of this chapter, we have chosen to utilize OpenShift Local due to its convenience and relevance to local development scenarios. It's crucial to note that running the examples effectively requires a minimum hardware allocation of 6 virtual CPUs and 12 GB of memory for the local cluster to operate correctly. This requirement ensures that users can experience the full capabilities of OpenShift Local in a GitOps context without significant performance issues.

To begin, navigate to `https://developers.redhat.com/products/openshift-local/overview` and click on **Install OpenShift on your laptop**, as shown in *Figure 8.1*.

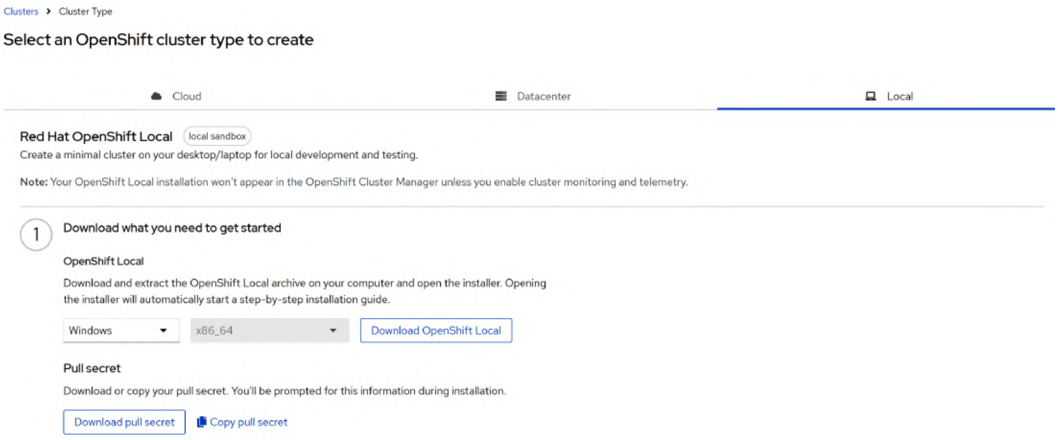


Red Hat OpenShift Local (formerly Red Hat CodeReady Containers)

OpenShift Local on your laptop gets you up and running with an OpenShift cluster on your local machine in minutes.

Figure 8.1 – The Red Hat OpenShift Local (CRC) homepage

To proceed with the setup of the cluster, it is necessary to download the installer by creating a Red Hat account if you do not already have one. From the displayed page (*Figure 8.2*), choose the installer corresponding to your OS (Linux, Windows, or macOS) where you plan to install the cluster. Additionally, make sure to select the **Download pull secret** or **Copy pull secret** option, as the pull secret will be required during the installation process. Remember, you can always return to this page if needed.



Clusters › Cluster Type

Select an OpenShift cluster type to create

Red Hat OpenShift Local

Create a minimal cluster on your desktop/laptop for local development and testing.

Note: Your OpenShift Local installation won't appear in the OpenShift Cluster Manager unless you enable cluster monitoring and telemetry.

1 Download what you need to get started

OpenShift Local

Download and extract the OpenShift Local archive on your computer and open the installer. Opening the installer will automatically start a step-by-step installation guide.

Pull secret

Download or copy your pull secret. You'll be prompted for this information during installation.

Figure 8.2 – Starting web page for downloading the CRC installer based on the target OS

At this point, it's important to note that a detailed step-by-step guide for installation is beyond the scope of this text. However, you can find comprehensive instructions in the official documentation provided by Red Hat.

For a complete walkthrough, please visit the following link: `https://access.redhat.com/documentation/en-us/red_hat_codeready_containers/1.0/html/getting_started_guide/getting-started-with-codeready-containers_gsg#doc-wrapper`.

This resource outlines all necessary steps to establish your local OpenShift cluster. As a standard procedure to verify the correctness of the installation, execute the following command:

```
$ crc version
```

This should produce output like the following:

```
CodeReady Containers version: 1.25.0+0e5748c8
OpenShift version: 4.7.13
```

Troubleshooting OpenShift CRC setup issues

Setting up your environment can be an elaborate process with potential difficulties. It requires careful attention to detail and an understanding of system configurations and dependencies. While following the provided guidelines and documentation can streamline the process, you may still encounter unexpected challenges. It's important to approach this setup with patience and be prepared to engage in some troubleshooting to address any issues that arise. Remember to properly start the cluster by using the following command:

```
$ crc start --cpus 6 --memory 12288
```

This command ensures that your CRC instance starts with the specified amount of computational resources: 6 CPUs and 12288 MB (approximately 12 GB) of memory. These settings are important for the optimal performance of your OpenShift cluster.

If you experience any issues during the setup process, try executing the following sequence of commands to troubleshoot and reset your environment:

- `crc delete -f`: This forcefully *deletes* the existing CRC instance, clearing any current configurations.
- `crc cleanup`: This command helps *clean up* any residual files or configurations left from previous instances, ensuring a clean slate.
- `crc setup`: Re-run the setup command to configure your system again for CRC.
- `crc config unset proxy-ca-file`: This removes any proxy configuration settings that might be causing issues.
- `crc start --log-level debug`: Start the CRC instance again, this time with debug logging enabled. This will provide more detailed output, which can be helpful in identifying where the setup process is encountering problems.

To finalize the setup of our cluster, an additional crucial step is required: the installation of the OpenShift **command-line interface** (**CLI**), commonly referred to as `oc`. This tool is essential for interacting with OpenShift clusters, allowing users to perform various tasks via the command line, such as deploying applications, inspecting cluster resources, and managing administrative activities. The `oc` CLI is specifically designed for developers and system administrators, offering advanced functionalities that surpass the basic features found in Kubernetes. This makes the `oc` CLI a vital component for those looking to automate processes via scripts or manage their clusters more directly, ensuring efficient operation within the OpenShift ecosystem.

Detailed steps for installing the `oc` CLI can be found in the official documentation. Please visit **Getting Started with the OpenShift CLI** at `https://docs.openshift.com/container-platform/4.8/cli_reference/openshift_cli/getting-started-cli.html` for comprehensive instructions on how to download, install, and begin using the `oc` CLI for managing your OpenShift clusters.

Check that `oc` is correctly installed by typing the following command:

```
$ oc version
```

The expected output of the preceding command should be like:

```
Client Version: 4.15.2
Kustomize Version: v5.0.4-0.20230601165947-6ce0bf390ce3
Server Version: 4.14.12
Kubernetes Version: v1.27.10+28ed2d7
```

Now that our cluster setup is complete, we are ready to progress to the next section, where we will focus on setting up GitOps in our Red Hat OpenShift cluster.

By applying GitOps, we can leverage the full potential of our newly configured OpenShift environment, ensuring a more efficient and automated operational framework. Let's move forward and explore the exciting capabilities of GitOps within OpenShift.

Setting Up GitOps in Red Hat OpenShift

Begin by installing the necessary GitOps tools. Argo CD (see the *Argo CD* section in *Chapter 4*) is a popular choice for OpenShift, as it integrates seamlessly with the Kubernetes environment. You can install Argo CD directly into your OpenShift cluster using **OperatorHub**, which offers an easy installation process and automatic management of the application life cycle:

1. To obtain the address of your local OpenShift cluster running on CRC, use the following command in your terminal:

   ```
   $ crc start
   ```

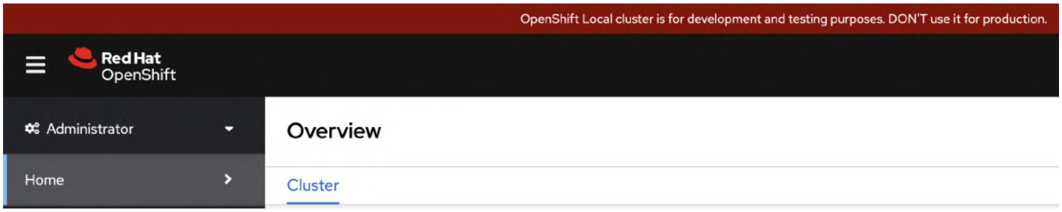
   The expected output of the `crc start` command will typically include several important pieces of information relevant to your local OpenShift cluster setup. Here's an example of what you might see:

   ```
   INFO Loading bundle: crc_vfkit_4.14.12_arm64...
   INFO A CRC VM for OpenShift 4.14.12 is already running
   Started the OpenShift cluster.
   The server is accessible via web console at:
     https://console-openshift-console.apps-crc.testing
   Log in as administrator:
     Username: kubeadmin
     Password: C57Yw-465gL-NXcpv-nri5w
   ```

```
Log in as user:
  Username: developer
  Password: developer
Use the 'oc' command line interface:
  $ eval $(crc oc-env)
  $ oc login -u developer https://api.crc.testing:6443
```

This output provides you with the URL for the OpenShift web console, along with login credentials for both the administrator and regular (developer) user accounts. It also includes instructions for logging in to your cluster using the `oc` CLI. Ensure you save these details, as you will need them to access your local OpenShift cluster.

2. Navigate to `https://console-openshift-console.apps-crc.testing` and log in as **Administrator** using the credentials provided by the `crc start` command, as illustrated in *Figure 8.3*. This will give you full access to the OpenShift cluster's administrative functionalities.



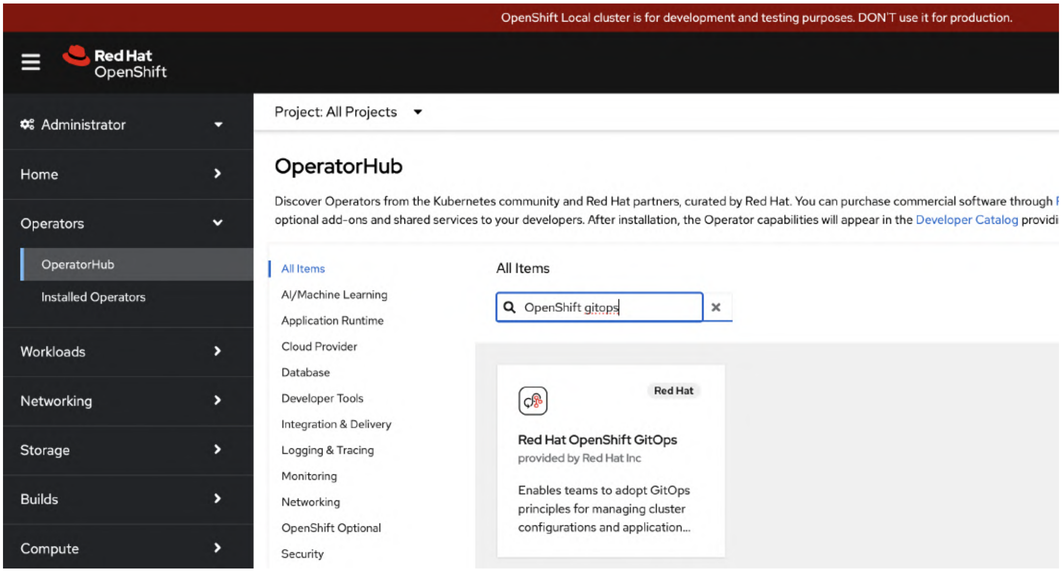
OpenShift Local cluster is for development and testing purposes. DON'T use it for production.

Figure 8.3 – The Cluster page in the Administrator mode

3. In the OpenShift web console, navigate to the left-hand panel and expand the **Operators** section.
4. Click on **OperatorHub**. Then, in the search box on the right-hand side, type `OpenShift GitOps` to locate the relevant operator for installation. This action will bring up the OpenShift GitOps operator, which you can then proceed to install on your cluster (refer to *Figure 8.4* for guidance).

OpenShift Operators

An **OpenShift Operator** is a method of packaging, deploying, and managing a Kubernetes application. OpenShift Operators extend the Kubernetes API to create, configure, and manage instances of complex stateful applications on behalf of a Kubernetes user. They automate the entire life cycle of the software they manage, from initial deployment to upgrade to everyday operations, following best practices. This approach simplifies the management of complex applications and services, allowing developers and operators to focus on the specifics of their applications while leveraging the scalability and reliability of Kubernetes systems. Essentially, Operators encapsulate the human operational knowledge required to run services and provide a more efficient and reliable way to manage applications in a cloud-native environment.


OpenShift Local cluster is for development and testing purposes. DON'T use it for production.

Discover Operators from the Kubernetes community and Red Hat partners, curated by Red Hat. You can purchase commercial software through
optional add-ons and shared services to your developers. After installation, the Operator capabilities will appear in the Developer Catalog providi

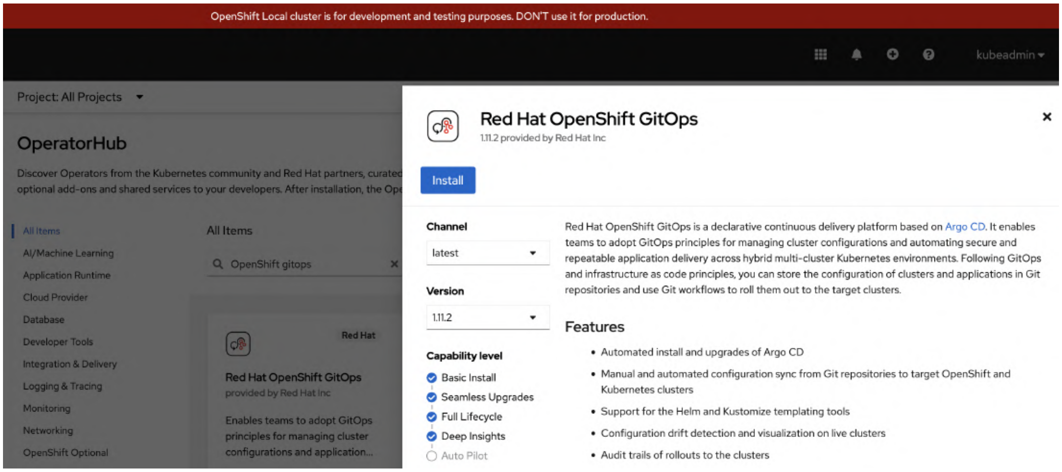
Search: OpenShift gitops

Red Hat OpenShift GitOps
provided by Red Hat Inc

Enables teams to adopt GitOps principles for managing cluster configurations and application...

Figure 8.4 – Navigating to OperatorHub and searching for OpenShift GitOps Operator

5. Once you have located the Red Hat OpenShift GitOps operator, click on it to view more details. Then, proceed to click on the **Install** button. When prompted, leave the default values as they are and continue with the installation process (see *Figure 8.5*). This will initiate the setup of the OpenShift GitOps operator on your cluster, equipping your environment with the necessary tools for GitOps workflows.


OpenShift Local cluster is for development and testing purposes. DON'T use it for production.

Red Hat OpenShift GitOps
1.11.2 provided by Red Hat Inc

Channel: latest

Version: 1.11.2

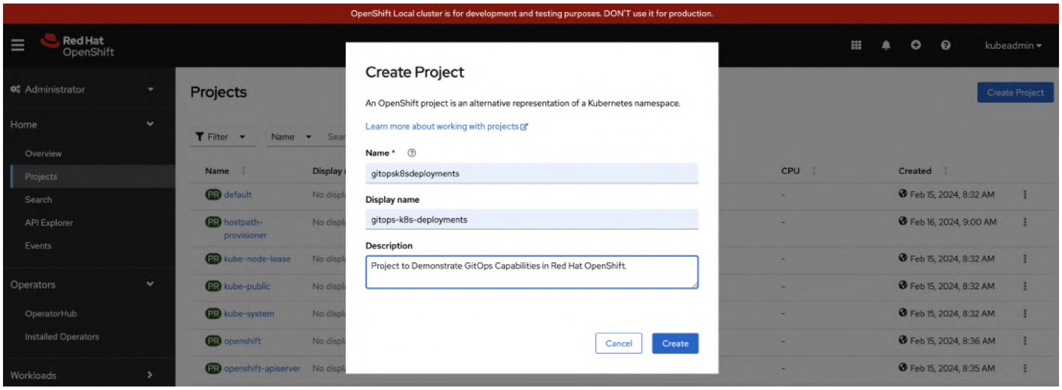
Capability level
- ● Basic Install
- ● Seamless Upgrades
- ● Full Lifecycle
- ● Deep Insights
- ○ Auto Pilot

Red Hat OpenShift GitOps is a declarative continuous delivery platform based on Argo CD. It enables teams to adopt GitOps principles for managing cluster configurations and automating secure and repeatable application delivery across hybrid multi-cluster Kubernetes environments. Following GitOps and infrastructure as code principles, you can store the configuration of clusters and applications in Git repositories and use Git workflows to roll them out to the target clusters.

Features
- Automated install and upgrades of Argo CD
- Manual and automated configuration sync from Git repositories to target OpenShift and Kubernetes clusters
- Support for the Helm and Kustomize templating tools
- Configuration drift detection and visualization on live clusters
- Audit trails of rollouts to the clusters

Figure 8.5 – A panel displaying details prior to operator installation

6. For **Installation mode**, select **All namespaces on the cluster (default)** to allow the operator to operate across all namespaces.
7. For **Installed Namespace**, choose **Operator recommended Namespace openshift-gitops-operator**, which is specifically suggested for the GitOps operator.
8. Click on **Install** and wait for the installation operations to be completed. This might take a few moments. After the installation is finished, click on **View Operator** to verify the installation's success, and ensure everything is set up correctly.
9. While remaining in the **Administrator** mode within the web console, navigate to **Create Project** to start a new project. Fill in the project creation form with the necessary information, following the guidelines or specifics you have for your project (*Figure 8.6*). This will establish a dedicated namespace for your GitOps deployments and related resources.



Name	Display	CPU	Created
PR default	No displ	-	Feb 15, 2024, 8:32 AM
PR hostpath-provisioner	No displ	-	Feb 16, 2024, 9:00 AM
PR kube-node-lease	No displ	-	Feb 15, 2024, 8:32 AM
PR kube-public	No displ	-	Feb 15, 2024, 8:32 AM
PR kube-system	No displ	-	Feb 15, 2024, 8:32 AM
PR openshift	No displ	-	Feb 15, 2024, 8:36 AM
PR openshift-apiserver	No displ	-	Feb 15, 2024, 8:35 AM

Create Project

An OpenShift project is an alternative representation of a Kubernetes namespace.

Learn more about working with projects

Name *: gitopsk8sdeployments

Display name: gitops-k8s-deployments

Description: Project to Demonstrate GitOps Capabilities in Red Hat OpenShift.

Figure 8.6 – Red Hat OpenShift – Create Project

> **Be aware!**
>
> The display name of your project can be adjusted according to your preferences, but it's important to remember this name as you will need to use it consistently in subsequent steps. Ensure that the name you choose is memorable and relevant to your project to avoid confusion later.

10. Once the project is created, the dashboard for your newly established project should appear in the OpenShift web console, as illustrated in *Figure 8.7*. It will provide an overview of your project's resources, activities, and status. Make sure the details align with your expectations and configurations, as this dashboard will be central to managing and monitoring your project's components in the upcoming steps.

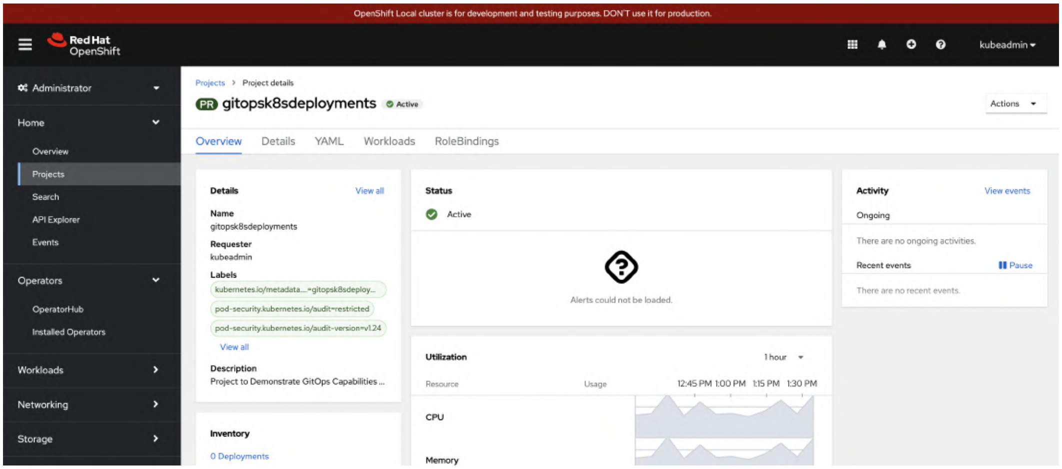
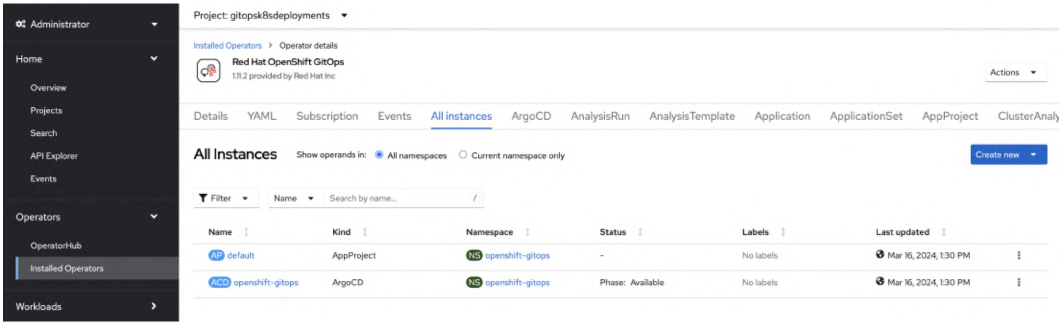


Figure 8.7 – Dashboard for the newly created project

11. In the OpenShift web console, go to **Operators | Installed Operators**. Find and click on **Red Hat OpenShift GitOps**, then select **All Instances** to view and manage your GitOps operator instances (refer to *Figure 8.8* for guidance).



Name	Kind	Namespace	Status	Labels	Last updated
AP default	AppProject	NS openshift-gitops	-	No labels	Mar 16, 2024, 1:30 PM
ACD openshift-gitops	ArgoCD	NS openshift-gitops	Phase: Available	No labels	Mar 16, 2024, 1:30 PM

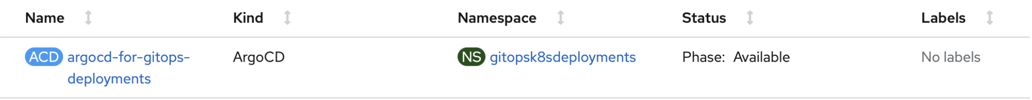
Figure 8.8 – List of instances where the operator has been installed and its status

12. Ensure that the project selected on the top-left is `gitopsk8sdeployments` before proceeding. Then, navigate to **Create new** and select **Argo CD** to set up a new instance.

13. In the form that appears, enter `argocd-for-gitops-deployments` into the name input box.

14. To create an external route for accessing the Argo CD server, click on **Server | Route** and ensure that **Enabled** is checked. This will allow you to access the Argo CD interface externally from your OpenShift cluster.

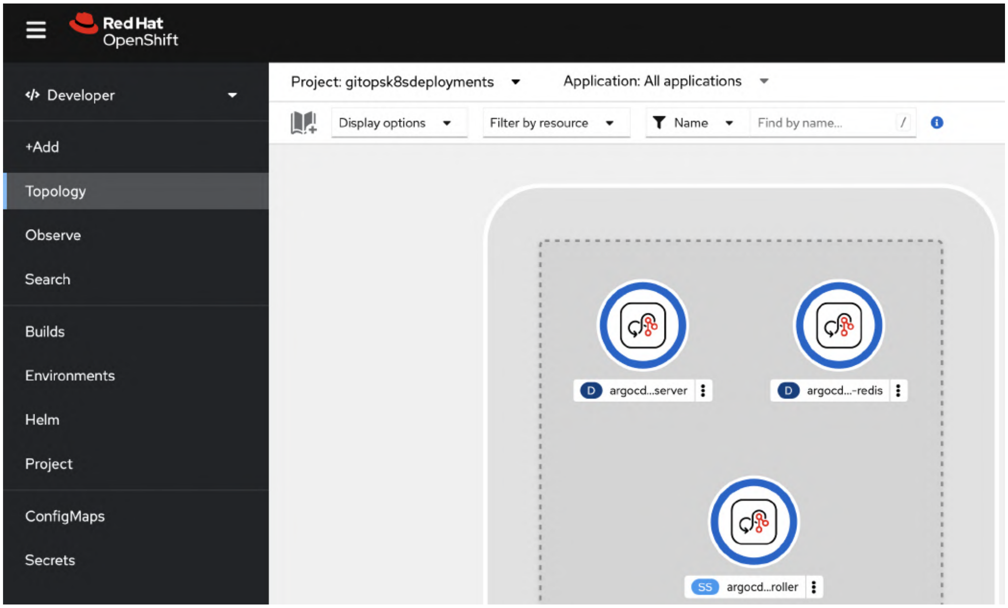
15. Wait for the **Status** value to change to **Available** as in *Figure 8.9*. This indicates that your Argo CD instance is fully deployed and ready for use within your OpenShift cluster. The status change might take a few minutes as the system finalizes the setup and deployment processes.



Name	Kind	Namespace	Status	Labels
ACD argocd-for-gitops-deployments	ArgoCD	NS gitopsk8sdeployments	Phase: Available	No labels

Figure 8.9 – New item displaying the created Argo CD instance for the specific namespace

Switch to the **Developer** mode in the OpenShift web console and click on **Topology**. This view will give you a graphical representation of the resources deployed so far, allowing you to visually understand the structure and relationships of your applications and services within the selected project.Top of Form



Project: gitopsk8sdeployments

Application: All applications

Figure 8.10 – A graphical representation of the resources deployed

16. To further test that Argo CD has been correctly set up, you can use `oc` and run the following command:

```
$ oc get pods -n gitopsk8sdeployments
```

This command will list all the pods within the `gitopsk8sdeployments` namespace, allowing you to verify that the Argo CD pods have been successfully deployed and are running as expected. If the Argo CD setup was successful, you should see the related pods listed without issues (all of them in the **Running** status):

```
NAME                                                            READY
STATUS     RESTARTS   AGE
argocd-for-gitops-deployments-application-controller-0           1/1
     Running    0           6m1s
argocd-for-gitops-deployments-redis-6c9476c7dc-76c6n             1/1
     Running    0           6m1s
argocd-for-gitops-deployments-repo-server-65cdc966d4-qzpvt       1/1
     Running    0           6m1s
```

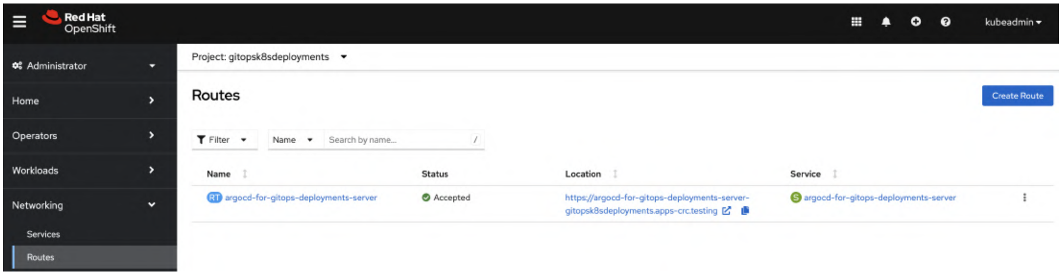
Now that we have successfully set up Argo CD and verified its correct installation, we are ready to proceed with the deployment. Let's move forward to the next steps where we will utilize our GitOps workflow within the OpenShift environment to manage and deploy our applications efficiently.

Leveraging Red Hat OpenShift's CI/CD for GitOps

In this section, we delve into leveraging Red Hat OpenShift's CI/CD capabilities for GitOps. Specifically, we will apply these principles to deploy the weather application, which was introduced previously in the *My city weather app* section in *Chapter 4*. Utilizing a deployment YAML file, we will orchestrate the application's rollout on the OpenShift cluster through our Argo CD instance, targeting the `gitopsk8sdeployments` namespace. This practical example will demonstrate how OpenShift's CI/CD tools, combined with GitOps methodologies, streamline and automate the deployment processes, ensuring a consistent, traceable, and efficient application delivery pipeline.

Let's explore how to integrate these powerful technologies to deploy our weather application seamlessly:

1. In the OpenShift web console, ensure you are in the **Administrator** mode.
2. Then, navigate to **Networking** and select **Routes**. Look for the entry named `argocd-for-gitops-deployments-server` in the list.



Project: gitopsk8sdeployments

Routes

Name	Status	Location	Service
RT argocd-for-gitops-deployments-server	Accepted	https://argocd-for-gitops-deployments-server-gitopsk8sdeployments.apps-crc.testing	S argocd-for-gitops-deployments-server

Figure 8.11 – Routes section of the Administrator mode

3. Click on the URL provided in the **Location** field of this entry. This action will take you directly to the Argo CD interface, where you can manage and monitor your GitOps deployments within the `gitopsk8sdeployments` namespace.

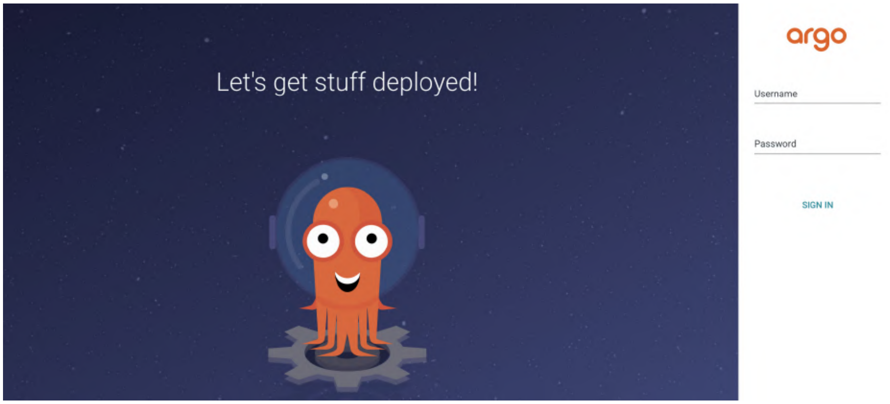
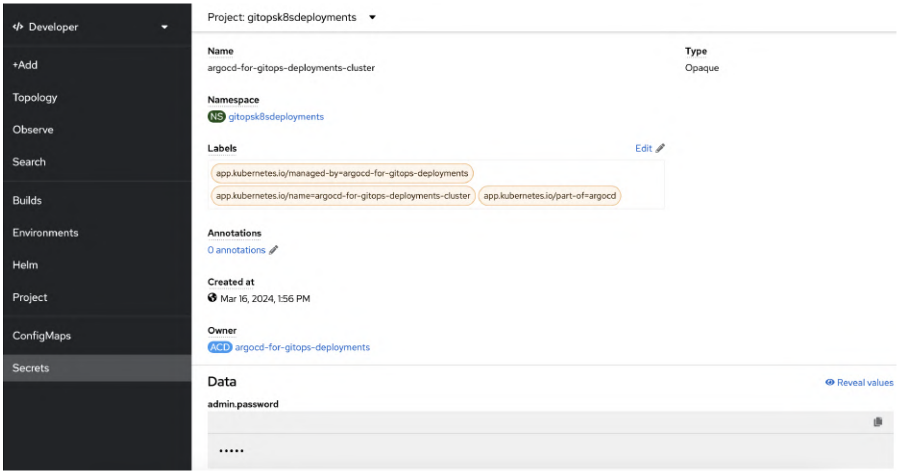


Figure 8.12 – The login page for the created Argo CD instance for our demo project

4. To retrieve the admin credentials, switch to the **Developer** mode in the OpenShift web console.
5. Then, click on **Secrets** in the side menu. Look for and select the secret named `argocd-for-gitops-deployments-cluster`. Within this secret's details, find and copy the value of the `admin.password` entry.



Project: gitopsk8sdeployments

Name
argocd-for-gitops-deployments-cluster

Type
Opaque

Namespace
NS gitopsk8sdeployments

Labels
app.kubernetes.io/managed-by=argocd-for-gitops-deployments
app.kubernetes.io/name=argocd-for-gitops-deployments-cluster
app.kubernetes.io/part-of=argocd

Annotations
0 annotations

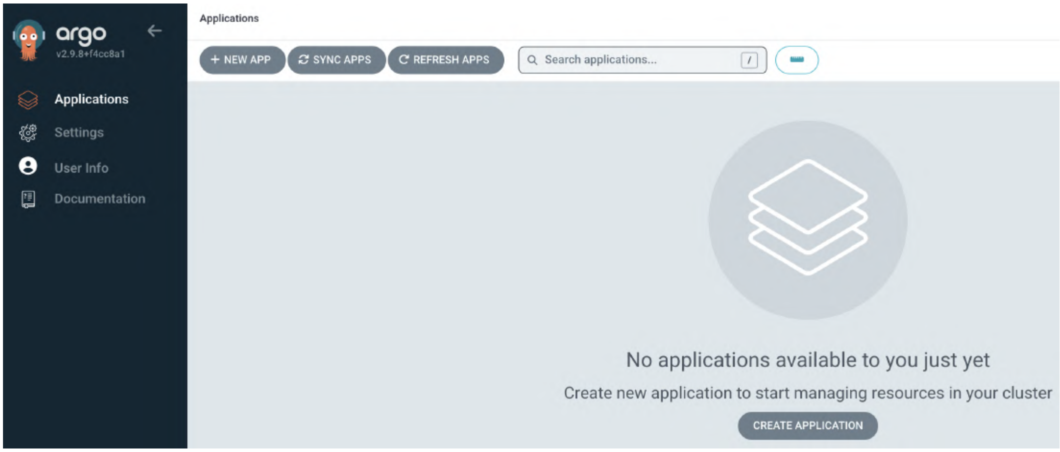
Created at
Mar 16, 2024, 1:56 PM

Owner
ACD argocd-for-gitops-deployments

Data
admin.password
•••••

Figure 8.13 – Secrets section of the gitopsk8sdeployments project

6. Once you access the Argo CD interface, the application list should initially be empty, as illustrated in *Figure 8.14*, since no applications have been deployed yet using Argo CD. At this point, you're ready to create a new application. This step will involve defining the source repository, the path within the repository where your Kubernetes manifests are located, and the destination where your application should be deployed, in this case, the OpenShift local cluster.



No applications available to you just yet

Create new application to start managing resources in your cluster

Figure 8.14 – Initial empty application list of the Argo CD instance

To create a new Argo CD application, you have the option to either use the Argo CD web UI or the `oc` CLI. If opting for the CLI, as demonstrated in an earlier section, you can apply a deployment file (e.g., `argocd-deployment.yaml`) that specifies the application's configuration.

7. Utilizing the code provided in the repository accompanying this chapter, you can create the new application by executing the following command:

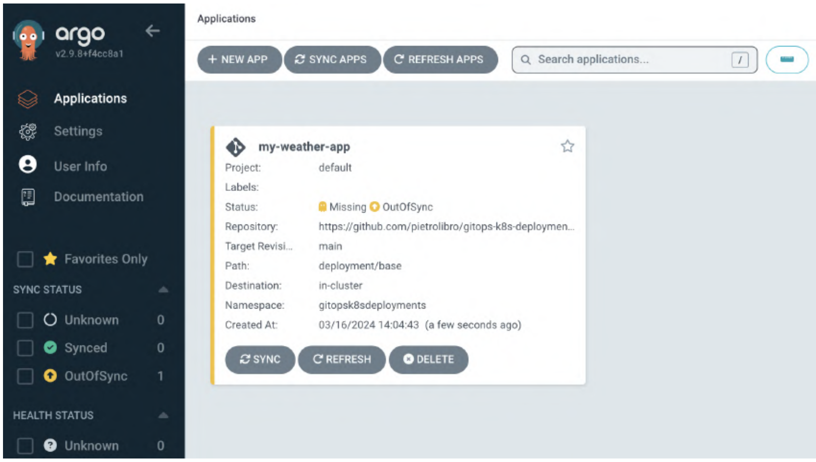
```
$ oc apply -f argocd-deployment.yaml
```

The content of the `argocd-deployment.yaml` file is as follows:

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: my-weather-app
spec:
  destination:
    name: ''
    namespace: gitopsk8sdeployments
    server: 'https://kubernetes.default.svc'
  source:
    path: deployment/base
```

```
      repoURL: 'https://github.com/pietrolibro/gitops-k8s-
  deployments-book-weather-app.git'
      targetRevision: main
    sources: []
    project: default
    syncPolicy:
      automated: null
      CreateNamespace: true
```

At this point, a new application titled `my-weather-app` will be visible in the Argo CD dashboard that was previously opened.



my-weather-app

Project: default

Labels:

Status: Missing OutOfSync

Repository: https://github.com/pietrolibro/gitops-k8s-deploymen...

Target Revisi... main

Path: deployment/base

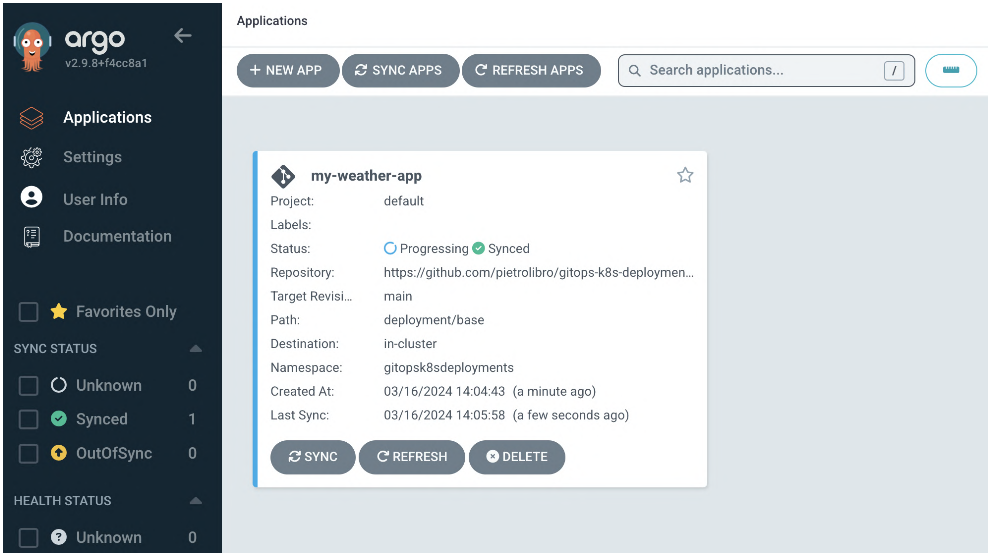
Destination: in-cluster

Namespace: gitopsk8sdeployments

Created At: 03/16/2024 14:04:43 (a few seconds ago)

Figure 8.15 – Argo CD instance populated with the newly created application

8. The app will initially appear in an **OutofSync** status, indicating that the actual state of the cluster does not match the desired state defined in the Git repository. To resolve this, click on **Sync** to synchronize the app with the desired state outlined in your repository. This action will start the deployment process according to the configurations defined in your Git repository.
9. Once the application is synchronized, *Figures 8.16-17*, it will align with the configurations defined in the manifest. This includes the deployment of two replicas, as specified in the manifest settings. The application's status should reflect that it is now in sync with the desired state, with the two replicas running as intended within your OpenShift cluster.

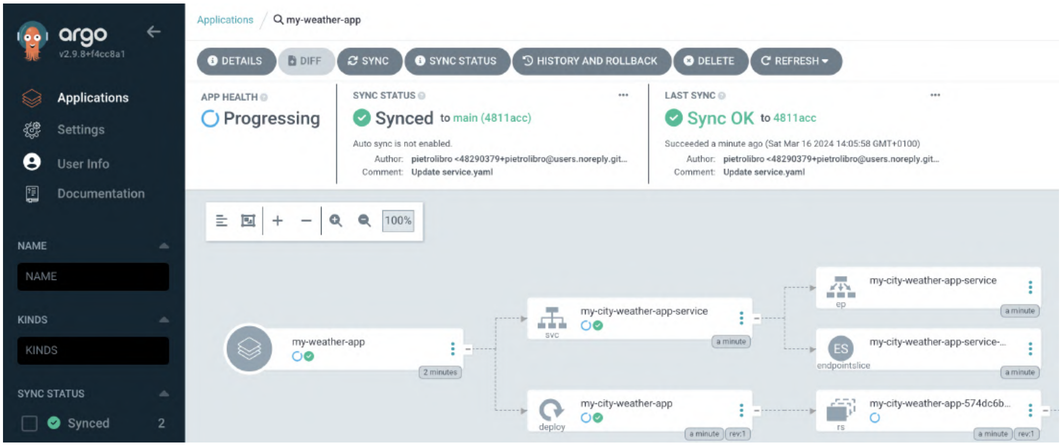


Applications

my-weather-app

Project: default
Labels:
Status: Progressing Synced
Repository: https://github.com/pietrolibro/gitops-k8s-deploymen...
Target Revisi... main
Path: deployment/base
Destination: in-cluster
Namespace: gitopsk8sdeployments
Created At: 03/16/2024 14:04:43 (a minute ago)
Last Sync: 03/16/2024 14:05:58 (a few seconds ago)

Figure 8.16 – Application panel after successful synchronization



Applications / my-weather-app

APP HEALTH
Progressing

SYNC STATUS
Synced to main (4811acc)
Auto sync is not enabled.
Author: pietrolibro <48290379+pietrolibro@users.noreply.git...
Comment: Update service.yaml

LAST SYNC
Sync OK to 4811acc
Succeeded a minute ago (Sat Mar 16 2024 14:05:58 GMT+0100)
Author: pietrolibro <48290379+pietrolibro@users.noreply.git...
Comment: Update service.yaml

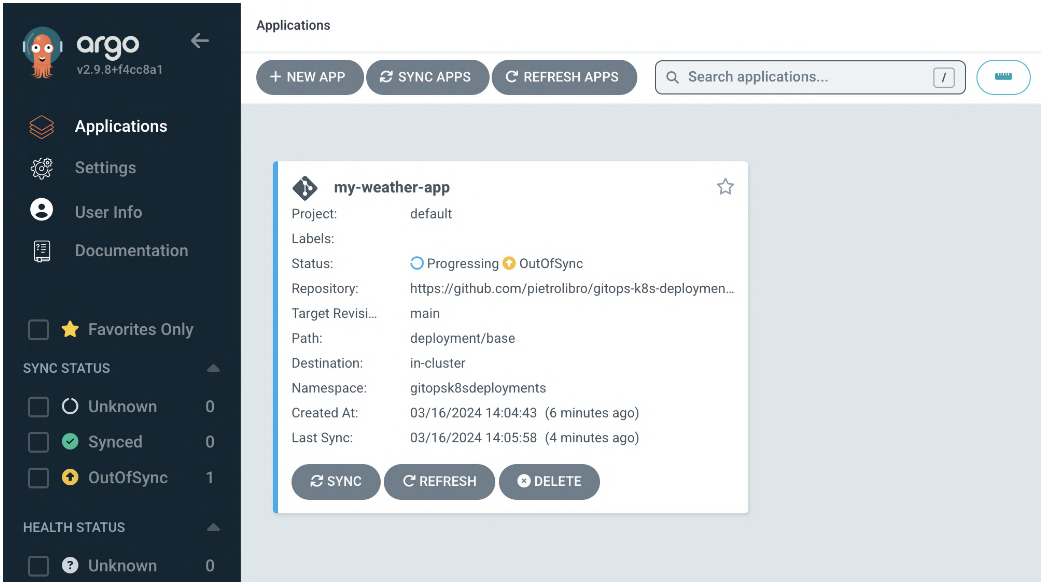
Figure 8.17 – Detailed status of deployed resources after synchronization

To test the GitOps CI/CD method of deployment, you can pull the repository locally using Git and then modify the number of replicas specified in the deployment file.

Changes to the application code alone will not cause the sync status to be marked as out of sync. The sync status is only affected by changes related to the deployment (e.g., YAML configuration files). GitOps tools such as Argo CD monitor the state of the configuration files in the repository. If there is a discrepancy between the desired state defined in the configuration files and the actual state in the Kubernetes cluster, it will be marked as out of sync. Therefore, only modifications to deployment files (such as YAML files) will impact the sync status, not changes to the application code itself.

This action embodies the iterative and controlled changes typical in GitOps workflows. If using Git directly is not your preference, you have the alternative to edit the file directly on GitHub, which simplifies the process while achieving the same result.

10. Once you have made and committed your changes, the application status in Argo CD will shift to **Out of Sync**. This indicates that the actual deployment no longer matches the desired state as defined in your Git repository. In our demonstration, we are operating under a manual sync approach, meaning that after any changes, you will need to manually trigger a sync in Argo CD. This step is essential to apply the new configurations, highlighting the GitOps principle where updates are made through version-controlled files, with manual intervention kept to a minimum to ensure deliberate and recorded changes.



my-weather-app

Project: default
Labels:
Status: Progressing OutOfSync
Repository: https://github.com/pietrolibro/gitops-k8s-deploymen...
Target Revisi... main
Path: deployment/base
Destination: in-cluster
Namespace: gitopsk8sdeployments
Created At: 03/16/2024 14:04:43 (6 minutes ago)
Last Sync: 03/16/2024 14:05:58 (4 minutes ago)

Figure 8.18 – Application in the OutOfSync status after changes to the manifest files

11. To resynchronize the application and apply the updated configurations, simply click the **Sync** button in the Argo CD dashboard. After initiating the sync, monitor the process to ensure it completes successfully. Once the synchronization is successful, the application's status will update to reflect that it is now in sync with the desired state as defined in your Git repository, incorporating the changes you made, such as the updated number of replicas.

As we transition from understanding the foundational aspects of Red Hat OpenShift and Kubernetes, we now shift our focus toward applying automation and configuration best Practices within these environments.

Automation and configuration best practices

OpenShift is widely adopted for hosting critical applications globally. As such, applications are expected to meet high availability standards, often aiming for five nines (99.999%) to ensure continuous service delivery. OpenShift assists in deploying applications in a highly available manner to mitigate outages, even when instances or infrastructure are compromised.

This summary outlines some key practices for deploying highly available applications on OpenShift:

- **Multiple replicas**: Deploying multiple pod instances ensures that the deletion or failure of a single pod doesn't lead to downtime. A recommended practice is to have at least two replicas for each application to ensure continuity.
- **Update strategy**: Employ **rolling** and **recreate** strategies to update applications. The rolling strategy, which is the default, minimizes downtime by gradually replacing old pods with new ones. For critical applications, `RollingUpdate` is recommended.
- **Graceful shutdown**: Ensure your applications handle the **SIGTERM** signal gracefully, allowing them to shut down properly. Adjust `terminationGracePeriodSeconds` as needed to give your applications sufficient time to close out tasks.
- **Probes**: Use **liveness**, **readiness**, and **startup** probes to monitor application health. Liveness probes help restart applications in a failed state, readiness probes manage traffic flow to the application, and startup probes allow time for initial app startup before other probes start monitoring.
- **External dependencies**: Manage your application's external dependencies effectively. Use `initContainers` or `startupProbes` to ensure dependencies are ready before starting your main application and employ readiness probes to maintain health checks.
- **PodDisruptionBudgets (PDBs)**: Implement PDBs to limit the number of pods that can be simultaneously unavailable during maintenance, thus preserving application availability.
- **Autoscaling**: Utilize **HorizontalPodAutoscaler** (**HPA**) and **VerticalPodAutoscaler** (**VPA**) to dynamically adjust the number of pod replicas or resources per pod based on the current load, ensuring that your application can handle varying levels of traffic.

- **Pod topology spread constraints**: Spread your pod replicas across different failure domains (e.g., nodes, racks, or zones) to avoid simultaneous downtime and ensure better availability and resilience.
- **Deployment strategies**: Use blue/green or canary deployment strategies to minimize disruptions during application updates. These strategies enable safer rollouts and easier rollback in case of issues.

For critical applications hosted on OpenShift, adopting these best practices will significantly boost availability, resilience, and continuous service delivery, meeting and even exceeding enterprise-level expectations and **service-level agreements** (**SLAs**).

Moreover, factors such as node sizing, resource allocation, and scheduling strategies play a vital role in enhancing the performance and dependability of your OpenShift framework. Regarding application security, we strongly recommend adhering to the following best practices to fortify the security posture of your applications:

- **Utilize trusted base container images**: Prioritize the use of official or vendor-provided container images, as they are rigorously tested, secured, and maintained. If you opt for community-supported images, ensure they are from reputable sources. Avoid images from unknown origins found in public registries such as Docker Hub, as they may pose security risks.
- **Regularly update base container images**: Staying updated with the latest versions of container images is crucial, as they include the most recent security patches and fixes. Implement automated processes in your CI pipeline to pull the latest base images during the build process and to trigger rebuilds of your application images when new base image updates are released.
- **Differentiate between build and runtime images**: Separate your build and runtime environments by using distinct images for each phase. The build image should contain only what is necessary for compiling and building the application, while the runtime image should be slimmed down to include only the essentials needed for running the application. This approach minimizes the attack surface by reducing unnecessary dependencies and vulnerabilities.
- **Adhere to restricted security context constraints (SCCs)**: Whenever feasible, configure your container images to comply with OpenShift's restricted SCC. This constraint enforces stringent security policies, limiting the potential for security breaches. Adapting your applications to operate under the restricted SCC maximizes the security level, safeguarding the cluster in the event of an application compromise. Refer to OpenShift's documentation on supporting arbitrary user IDs for additional guidance.
- **Secure inter-component communications**: Ensure that communication between application components is secured, particularly if sensitive data is being transmitted. While OpenShift's network is designed to be secure, using **Transport Layer Security** (**TLS**) adds an extra layer of protection. For a more streamlined approach, consider utilizing **OpenShift Service Mesh**, which can manage TLS, taking the burden off individual applications and providing centralized control over security policies.

Having explored key practices to enhance application availability, resilience, and security within OpenShift, we conclude this section. These guidelines provide a foundation for maintaining robust, secure applications that meet enterprise standards. As we transition, the forthcoming section will delve into a comparative analysis between Kubernetes and Red Hat OpenShift. This comparison will illuminate the distinct features and benefits of each platform, helping you understand their unique advantages in various operational environments. Let's proceed to uncover the differences and similarities that define Kubernetes and Red Hat OpenShift, setting the stage for informed decision making in your container orchestration strategies.

A comparison of Kubernetes Red Hat OpenShift

Table 8.1 provides a high-level comparison between Kubernetes and OpenShift. While Kubernetes offers a flexible and widely adopted container orchestration platform, OpenShift provides a more integrated and security-focused enterprise solution with extensive support and built-in features for developer productivity and operational efficiency.

Feature/Aspect	**Kubernetes**	**OpenShift**
Origin	Open-source project by Google	Red Hat's enterprise Kubernetes distribution
Installation	Manual, requires setup of networking, storage, etc.	Streamlined with automated setups and configurations
Web UI	Basic dashboard available	Comprehensive and user-friendly console with advanced features
Security	Basic security features: additional tools are often needed	Enhanced security with built-in authentication and authorization, SELinux integration
CI/CD integration	Plugins and third-party tools required	Built-in CI/CD tools such as OpenShift Pipelines and **Source-to-Image** (**S2I**)
Networking	Pluggable **Container Network Interface** (**CNI**)	Default SDN with additional networking features
Storage	Supports PersistentVolumes with manual configuration	Simplified storage integration with automatic provisioning
Developer tools	Limited in-built tools	Extensive tools including OpenShift Do (`odo`), S2I, and DevSpaces
Monitoring and logging	Basic with third-party integration (e.g., Prometheus)	Integrated monitoring and logging stack
Registry	Requires external or third-party registry	Integrated container image registry

Feature/Aspect	Kubernetes	OpenShift
Enterprise support	Available through third-party providers	Direct support from Red Hat
Community and support	Large open-source community	Strong open-source community with enterprise support
Pricing	Free, costs associated with cloud providers or infrastructure	Subscription-based with various plans for enterprise use

Table 8.1 – A Kubernetes and Red Hat OpenShift feature/aspect comparison

In concluding our comparison between Kubernetes and Red Hat OpenShift, we've uncovered the distinct features, advantages, and considerations of each platform. This analysis equips you with the knowledge to make informed decisions based on your specific operational needs and goals. As we move forward, let's apply this understanding to further enhance our deployment strategies and operational efficiencies within the context of container orchestration and cloud-native development.

Summary

In this chapter, we embarked on an in-depth exploration of leveraging GitOps within the Red Hat OpenShift environment, highlighting the integration of automation and configuration best practices to enhance operational efficiency and security. We began by setting up a local OpenShift cluster using CRC, providing a sandbox for readers to apply real-world applications of GitOps principles. This practical approach demystified the process of deploying and managing containerized applications using declarative configurations and automated workflows.

We delved into various best practices, from using trusted base images and setting appropriate resource limits to implementing robust security measures such as TLS for inter-component communication. Each practice was designed to fortify the deployment process, reduce vulnerabilities, and ensure applications remain resilient and available. By comparing Kubernetes with Red Hat OpenShift, we provided insights into how OpenShift extends Kubernetes capabilities, offering enhanced features that cater to enterprise needs.

Throughout the chapter, the emphasis was placed on hands-on examples, ensuring that readers not only understand theoretical concepts but also how to apply them effectively in their own OpenShift environments. As we conclude, the knowledge and strategies discussed here pave the way for more secure, efficient, and reliable application deployments, reflecting the evolving landscape of cloud-native development and the increasing importance of GitOps methodologies.

In the next chapter, we will explore the integration of GitOps practices within Azure and AWS, unveiling advanced strategies for cloud-native deployments.

9

GitOps for Azure and AWS Deployments

In the evolving landscape of cloud computing, the adoption of GitOps practices stands out as a transformative approach to streamlining the deployment and management of applications and infrastructure. This chapter embarks on a detailed exploration of applying GitOps principles within the robust ecosystems of Azure and **Amazon Web Services** (**AWS**), two of the world's leading cloud platforms. This chapter aims to unravel the complexities of leveraging cloud-native capabilities, providing readers with a comprehensive understanding of how to harness the full potential of **Azure Kubernetes Service** (**AKS**), **Azure DevOps**, **Elastic Kubernetes Service** (**EKS**), and **AWS CodePipeline** in a GitOps workflow.

Through real-world case studies, expert insights, and practical guidance, we delve into the nuances of setting up **continuous integration/continuous deployment** (**CI/CD**) pipelines, managing configurations, and ensuring consistent, automated deployments across these platforms. By the end of this chapter, readers will be equipped with the knowledge to implement efficient, secure, and scalable GitOps workflows, marking a significant step forward in their cloud-native journey.

In this chapter, we'll focus on these key areas:

- Cloud GitOps essentials – Azure and AWS
- Deployment on Azure and AWS with GitOps
- Integrating Azure and AWS in GitOps workflows
- GitOps applications in cloud environments
- GitOps strategies for Azure and AWS deployments for Kubernetes

Technical requirements

Before delving into the intricacies of implementing GitOps for Azure and AWS deployments, it's important to build upon the foundational knowledge established in the preceding chapters of this book. The principles of GitOps, containerization technologies such as Docker, Kubernetes concepts, and CI/CD principles discussed earlier provide a solid starting point for understanding the advanced applications highlighted in this chapter. Additionally, access to Azure and AWS accounts and a basic understanding of their services will be crucial for following along with practical exercises and case studies. Familiarity with version control systems, especially Git, will not only enhance comprehension but also facilitate the effective application of the GitOps practices detailed in our exploration of cloud-native deployments across Azure and AWS.

The relevant code and resource files for this chapter can be found in the `Chapter09` folder of our dedicated GitHub repository: `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`.

Azure and AWS accounts

While this book delves into the intricacies of implementing GitOps within Azure and AWS ecosystems, providing a comprehensive exploration of their respective tools and practices, a detailed, step-by-step guide for creating Azure and AWS accounts falls beyond our scope. To embark on the practical journey of applying the concepts and examples outlined in the upcoming sections, it is essential for readers to have active Azure and AWS accounts. We encourage you to consult the official documentation provided by Azure at `https://azure.microsoft.com/en-us/`, and AWS at `https://aws.amazon.com/`, for the most current and detailed instructions on setting up your accounts. These accounts are indispensable for deploying the examples and applying the GitOps practices discussed, serving as the foundation upon which you can build and experiment with the cloud-native capabilities of Azure and AWS.

In the forthcoming section, we will assume that readers possess an active and properly configured Azure or AWS account, along with basic knowledge of the **Azure CLI**, **AWS CLI**, or their respective web portals.

Cloud GitOps essentials – Azure and AWS

As the cloud computing landscape continues to evolve, the adoption of GitOps principles has become a cornerstone for achieving operational excellence and automation in **cloud-native deployments**. This chapter introduces the GitOps essentials for both Azure and AWS, showcasing how these leading cloud platforms support the seamless integration of GitOps workflows to enhance deployment speed, reliability, and scalability.

Cloud-native developments and deployments

Cloud-native development represents a transformative approach to building and deploying applications that fully exploit the advantages of the cloud computing model. At its core, it involves leveraging managed services, microservices architectures, containers, and declarative APIs to create applications that are scalable, resilient, and easily updated. This paradigm shift encourages organizations to move away from monolithic architectures, enabling faster development cycles, enhanced scalability, and greater flexibility in responding to market demands. Cloud-native technologies, including Kubernetes, Docker, and serverless functions, play pivotal roles in this ecosystem, providing the tools necessary for developers to build applications that are not only highly available and fault tolerant but also capable of thriving in the dynamic environment of the cloud. By adopting cloud-native practices, businesses can accelerate their digital transformation journeys, enhancing their ability to innovate and compete in an increasingly digital world.

Azure and AWS each offer unique tools and services – such as AKS, Azure DevOps, EKS, and AWS CodePipeline – that empower teams to implement GitOps practices effectively. Through a comprehensive exploration of these essentials, readers will gain insight into leveraging the cloud-native capabilities of Azure and AWS to streamline their deployment processes, ensuring that infrastructure and application management is as efficient and error free as possible. This unified approach provides a solid foundation for understanding how GitOps can be optimally applied within the distinct but complementary ecosystems of Azure and AWS.

Azure GitOps essentials

Azure, with its rich ecosystem and integration capabilities, offers fertile ground for implementing GitOps principles, enhancing automation, consistency, and scalability in deployments. At the heart of Azure's GitOps capabilities lies the AKS, which simplifies the deployment, management, and operations of Kubernetes, enabling a seamless GitOps workflow. Coupled with **Azure DevOps** – a suite that provides a range of tools including **Azure Repos** for Git hosting and **Azure Pipelines** for CI/CD – developers can establish a robust GitOps pipeline that ensures continuous integration and deployment with minimal manual intervention. Leveraging these services, users can maintain a high degree of control and visibility over their deployments, benefiting from the declarative nature of GitOps to manage infrastructure and applications efficiently.

Azure DevOps

Azure DevOps represents a suite of development tools provided by Microsoft, designed to support the complete software development life cycle. At its core, Azure DevOps facilitates CI/CD practices, enabling teams to automate the build, test, and deployment phases of their applications. Specifically, in the context of GitOps, Azure DevOps becomes a powerful ally, allowing teams to manage infrastructure and application code in a Git repository, automatically apply changes to Kubernetes environments, and maintain a consistent state across development, testing, and production environments.

For implementing Kubernetes GitOps deployment with Azure DevOps, the following base steps can guide you through the process:

1. **Set up a Git repository**: Begin by setting up a Git repository within Azure Repos or any other Git hosting service. This repository will hold your Kubernetes manifest files, representing the desired state of your application and infrastructure in a declarative manner.
2. **Create Azure Pipelines**: Utilize Azure Pipelines to define your CI/CD workflows. For GitOps, the CD pipeline plays a crucial role. It should be configured to trigger automatically upon changes to the main branch of your repository, where your Kubernetes manifest files are stored.
3. **Define environments**: In Azure DevOps, define environments to represent your deployment targets, such as development, staging, and production. These environments can be linked to your Kubernetes clusters managed by AKS or any Kubernetes clusters.
4. **Automate deployment with Helm or Kustomize**: Use Helm charts or Kustomize for managing complex Kubernetes applications. Azure Pipelines can be configured to use Helm or Kustomize to package and deploy applications, adhering to the GitOps principle of declarative configuration.
5. **Implement CD**: The CD pipeline should be designed to automatically apply changes to your Kubernetes environment using `kubectl`, Helm, or a GitOps tool such as Flux or Argo CD. This step involves fetching the latest configuration from your Git repository and applying it to the designated environment, ensuring that the actual state matches the desired state declared in Git.
6. **Monitor and rollback**: Finally, leverage Azure Monitor and other observability tools to keep an eye on your deployments. In case of any issues, your GitOps workflow should support easy rollbacks by simply reverting changes in your Git repository and re-running the pipeline to restore the previous state.

Kubernetes deployment with Azure DevOps

In this section, we embark on a hands-on journey to deploy a Kubernetes cluster on Azure, leveraging the powerful combination of Terraform, **Azure Container Registry** (**ACR**), and Azure DevOps pipelines. We'll start by creating a straightforward AKS cluster and an ACR, and then establish a system-managed identity between them to facilitate secure interactions. The climax of our journey involves orchestrating the deployment of this image to the AKS cluster using a meticulously crafted Azure DevOps pipeline. This practical walkthrough is designed to illuminate the seamless integration of these components, illustrating how they converge to streamline the deployment process within a GitOps framework. Through this example, readers will gain a comprehensive understanding of deploying applications to Kubernetes using Azure's robust ecosystem. The following are the necessary steps to successfully complete this example.

In a real-world scenario, executing the Terraform script would involve creating an **Azure service principal** with the appropriate permissions to provision resources. However, for this example, we are simplifying the process by using an Azure administrator account with full authorizations. It's important to note that this approach is not recommended for production environments due to security concerns. Log in to Azure by typing the following command in a new terminal window:

```
$ az login
```

To initialize Terraform, we use the `terraform init` command. For this step, we will utilize the `main.tf` and `versions.tf` Terraform files located in the `iac/azure` folder of the GitHub repository accompanying this chapter.

Why Terraform?

Terraform is a preferred tool for **infrastructure as code** (**IaC**) because it uses a declarative configuration language, which simplifies defining and managing complex environments by specifying the desired end state of the infrastructure. It supports multiple cloud providers, including AWS, Azure, and Google Cloud, enabling consistent IaC practices across different platforms. Additionally, Terraform's state management feature keeps track of the current infrastructure status, allowing it to efficiently plan and apply changes while minimizing errors. Its extensive module ecosystem and robust community support further enhance its capability to automate and manage infrastructure at scale.

The `main.tf` file orchestrates the setup of essential Azure services for a Kubernetes-based deployment. Initially, it establishes a resource group called `aks-k8s-deployments-rg` in the `switzerlandnorth` region, serving as a container for all related Azure resources. Following this, an ACR named `aksgitops3003204acr` is provisioned within the same resource group and region, with the **Basic SKU** option and administrative access disabled for enhanced security. The core of the deployment, an `AKS` cluster named `aksgitopscluster`, is then created, featuring a single default node pool configured with minimal resources to ensure cost efficiency. The cluster is set up with a system-assigned identity, streamlining the authentication process across Azure services.

Finally, the Terraform configuration secures the integration between the AKS cluster and the ACR by assigning the necessary role permissions, enabling seamless container image pulls from the registry to the Kubernetes environment. Feel free to modify the names of the resource group or ACR and the region where the resources are deployed. In a terminal window, execute the following commands. The entire process will take a few minutes to complete:

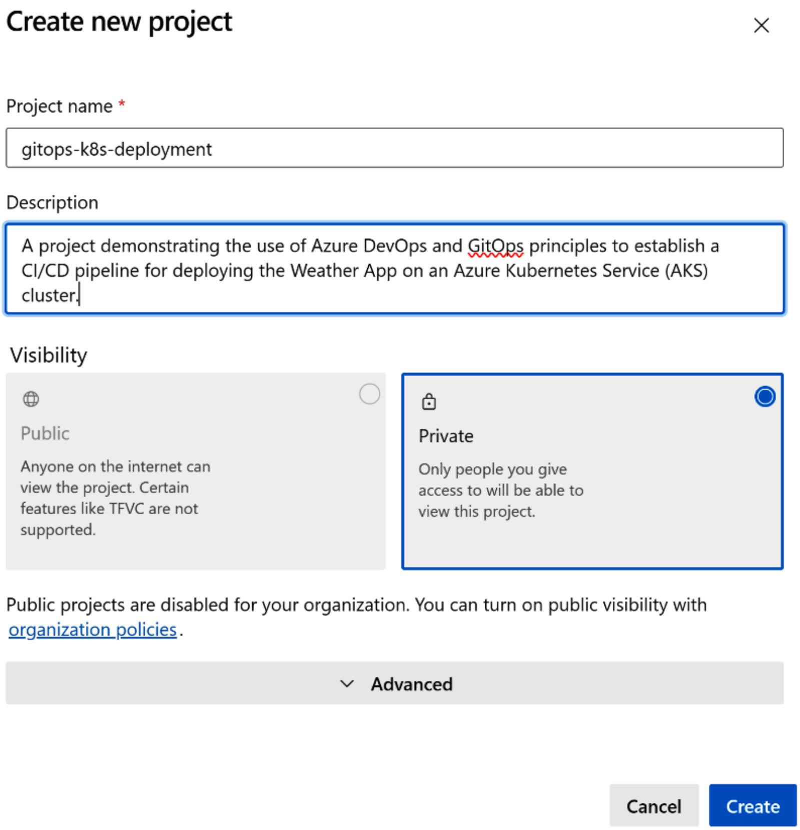
```
$ terraform init
$ terraform plan -out aksplan
$ terraform apply -auto-approve aksplan
```

Await the successful provisioning of resources, then integrate the `kubectl` configuration for cluster management:

```
$ az aks get-credentials --resource-group aks-k8s-deployments-rg
--name aksgitopscluster
```

The following are the necessary steps to create a new Azure DevOps project and properly set up an Azure DevOps pipeline:

1. Establish a new Azure DevOps project, as illustrated in *Figure 9.1*, within your existing Azure account to manage the CI/CD pipeline and project artifacts, opting for a **Private Azure DevOps** project setting. For more information on setting up Azure DevOps, please visit `https://learn.microsoft.com/en-us/azure/devops/user-guide/sign-up-invite-teammates?view=azure-devops&tabs=microsoft-account`. Assign a name to the project, such as `gitops-k8s-deployment`, and optionally add a project description. Select **Private** for **Visibility**, and then click the **Create** button.



Create new project

Project name *

gitops-k8s-deployment

Description

A project demonstrating the use of Azure DevOps and GitOps principles to establish a CI/CD pipeline for deploying the Weather App on an Azure Kubernetes Service (AKS) cluster.

Visibility

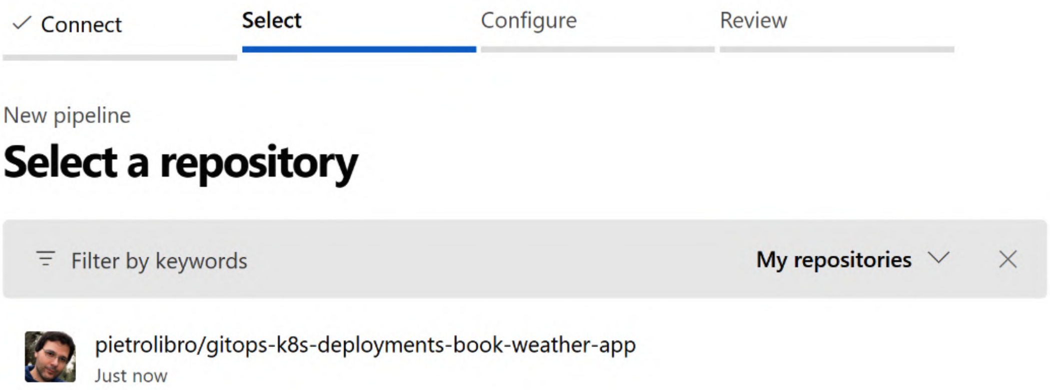
○ Public
Anyone on the internet can view the project. Certain features like TFVC are not supported.

● Private
Only people you give access to will be able to view this project.

Public projects are disabled for your organization. You can turn on public visibility with organization policies.

Figure 9.1 – The Azure DevOps window for creating a new project

2. In the **Where is your code?** window, select **Pick** next to the GitHub option to choose the correct GitHub repository you wish to associate with the Azure DevOps project you previously created. You can directly use the GitHub repository associated with this chapter or select one that you have created, as seen in *Figure 9.2*.



✓ Connect | Select | Configure | Review

New pipeline

Select a repository

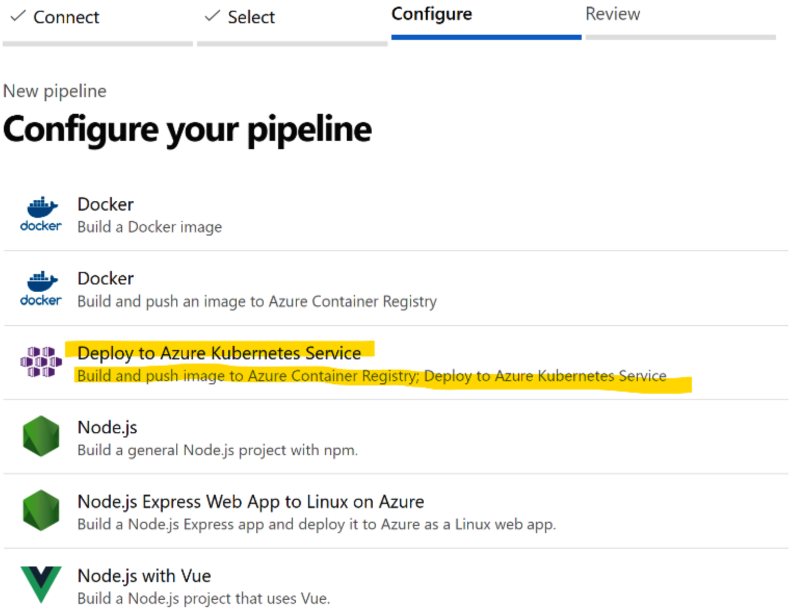
Filter by keywords | My repositories

pietrolibro/gitops-k8s-deployments-book-weather-app

Just now

Figure 9.2 – Selecting a repository panel

3. At this stage, we're prepared to configure the pipeline. In the configuration panel, select **Deploy to Azure Kubernetes Service** as illustrated in *Figure 9.3*.



✓ Connect | ✓ Select | Configure | Review

New pipeline

Configure your pipeline

Docker
Build a Docker image

Docker
Build and push an image to Azure Container Registry

Deploy to Azure Kubernetes Service
Build and push image to Azure Container Registry; Deploy to Azure Kubernetes Service

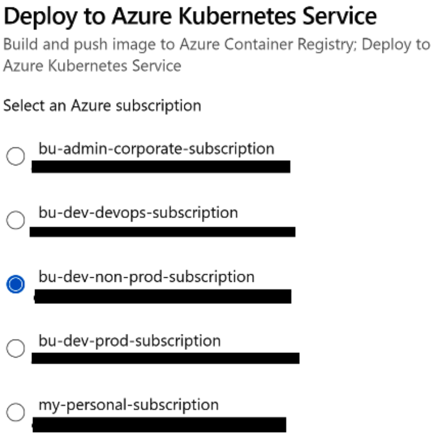
Node.js
Build a general Node.js project with npm.

Node.js Express Web App to Linux on Azure
Build a Node.js Express app and deploy it to Azure as a Linux web app.

Node.js with Vue
Build a Node.js project that uses Vue.

Figure 9.3 – Configure your pipeline section of the wizard

4. When prompted, as visible in *Figure 9.4*, select the Azure subscription where the AKS cluster was previously configured and deployed using the Terraform script.



Deploy to Azure Kubernetes Service

Build and push image to Azure Container Registry; Deploy to Azure Kubernetes Service

Select an Azure subscription

○ bu-admin-corporate-subscription
[redacted]

○ bu-dev-devops-subscription
[redacted]

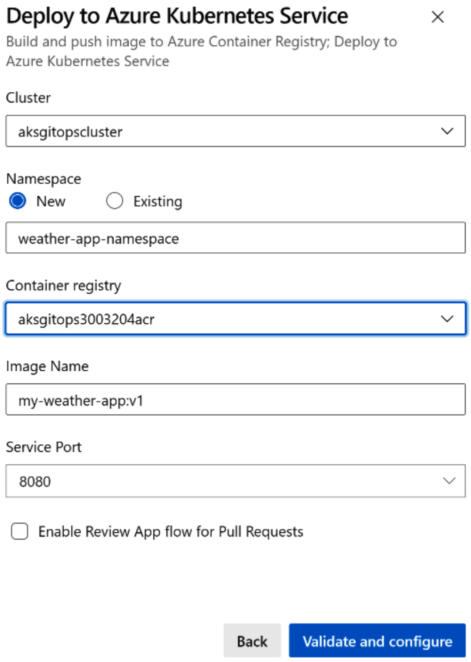
● bu-dev-non-prod-subscription
[redacted]

○ bu-dev-prod-subscription
[redacted]

○ my-personal-subscription
[redacted]

Figure 9.4 – Popup window for selecting the Azure subscription

5. In the **Deploy to Azure Kubernetes Service** window, enter the required information as shown in *Figure 9.5*. Deploy the application into a new namespace, such as `weather-app-namespace`.



Deploy to Azure Kubernetes Service

Build and push image to Azure Container Registry; Deploy to Azure Kubernetes Service

Cluster: aksgitopscluster

Namespace: ● New ○ Existing

weather-app-namespace

Container registry: aksgitops3003204acr

Image Name: my-weather-app:v1

Service Port: 8080

☐ Enable Review App flow for Pull Requests

Figure 9.5 – Deploy to AKS settings window

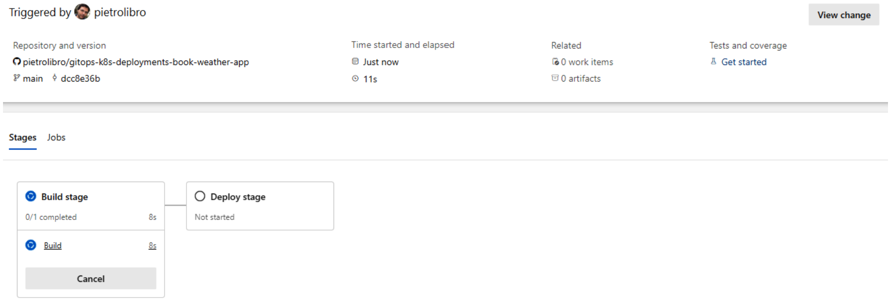
6. Click the **Validate and configure** button to review the YAML pipeline. Before continuing, since the Dockerfile is not present in the root of the directory, but in the `src` subdirectory, we need to edit the last line of the following code:

```
variables:
# Container registry service connection …
dockerfilePath: '**/Dockerfile'
```

We edit the preceding code this way:

```
dockerfilePath: '**/src/dockerfile'
```

7. Then, click the **Save and run** button, leave the default values as they are, and click the **Save and run** button again. The pipeline will be triggered as illustrated in *Figure 9.6*.



Triggered by pietrolibro

Repository and version
pietrolibro/gitops-k8s-deployments-book-weather-app
main dcc8e36b

Time started and elapsed
Just now
11s

Related
0 work items
0 artifacts

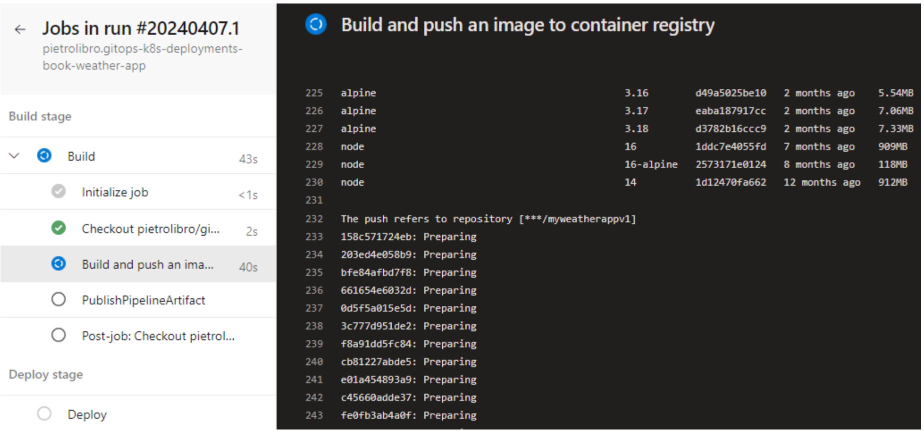
Tests and coverage
Get started

Build stage
0/1 completed 8s
Build 8s

Deploy stage
Not started

Figure 9.6 – Example of a triggered Azure DevOps pipeline

8. Click on the **Build** stage to view more details, as illustrated in *Figure 9.7*. This time, there's no need to build the image locally and then push it to the registry, as everything is handled within the Azure DevOps pipeline.



Jobs in run #20240407.1
pietrolibro.gitops-k8s-deployments-book-weather-app

Build stage
Build 43s
Initialize job <1s
Checkout pietrolibro/gi… 2s
Build and push an ima… 40s
PublishPipelineArtifact
Post-job: Checkout pietrol…

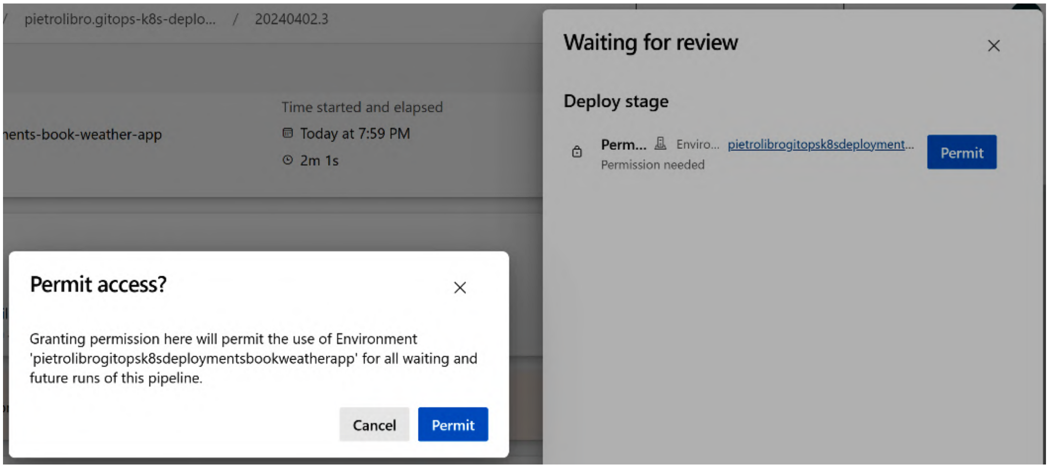
Deploy stage
Deploy

Build and push an image to container registry

```
alpine                                    3.16        d49a5025be10   2 months ago    5.54MB
alpine                                    3.17        eaba187917cc   2 months ago    7.06MB
alpine                                    3.18        d3782b16ccc9   2 months ago    7.33MB
node                                      16          1ddc7e4055fd   7 months ago    909MB
node                                      16-alpine   2573171e0124   8 months ago    118MB
node                                      14          1d12470fa662   12 months ago   912MB

The push refers to repository [***/myweatherappv1]
158c571724eb: Preparing
203ed4e058b9: Preparing
bfe84afbd7f8: Preparing
661654e6032d: Preparing
0d5f5a015e5d: Preparing
3c777d951de2: Preparing
f8a91dd5fc84: Preparing
cb81227abde5: Preparing
e01a454893a9: Preparing
c45660adde37: Preparing
fe0fb3ab4a0f: Preparing
```

Figure 9.7 – Details of building and pushing an image to the container registry

If required, authorize permissions for the pipeline, as illustrated in *Figure 9.8*.



Permit access?

Granting permission here will permit the use of Environment 'pietrolibrogitopsk8sdeploymentsbookweatherapp' for all waiting and future runs of this pipeline.

Figure 9.8 – Popup window to grant permissions for the current and subsequent runs of the pipeline

9. At this point, after a few seconds, the pipeline should complete successfully, as illustrated in *Figure 9.9*. An email should be sent to your account notifying you of the pipeline's successful completion.

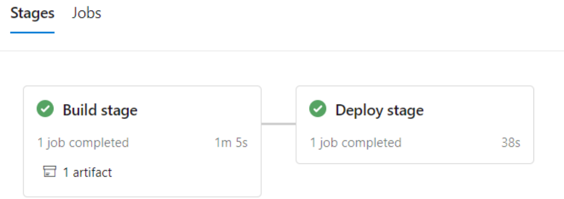


Figure 9.9 – Successful completion of the pipeline

10. To verify the deployment, you can utilize tools such as Visual Studio Code or execute the following command in the terminal:

```
$ kubectl get pods --namespace weather-app-namespace
```

The output should be as follows:

```
NAME                    READY    STATUS     RESTARTS    AGE
myweathera…           1/1      Running    0           7m50s
```

At this juncture, we can execute a `port-forward` command to display the app in the browser, as illustrated in *Figure 9.10*.

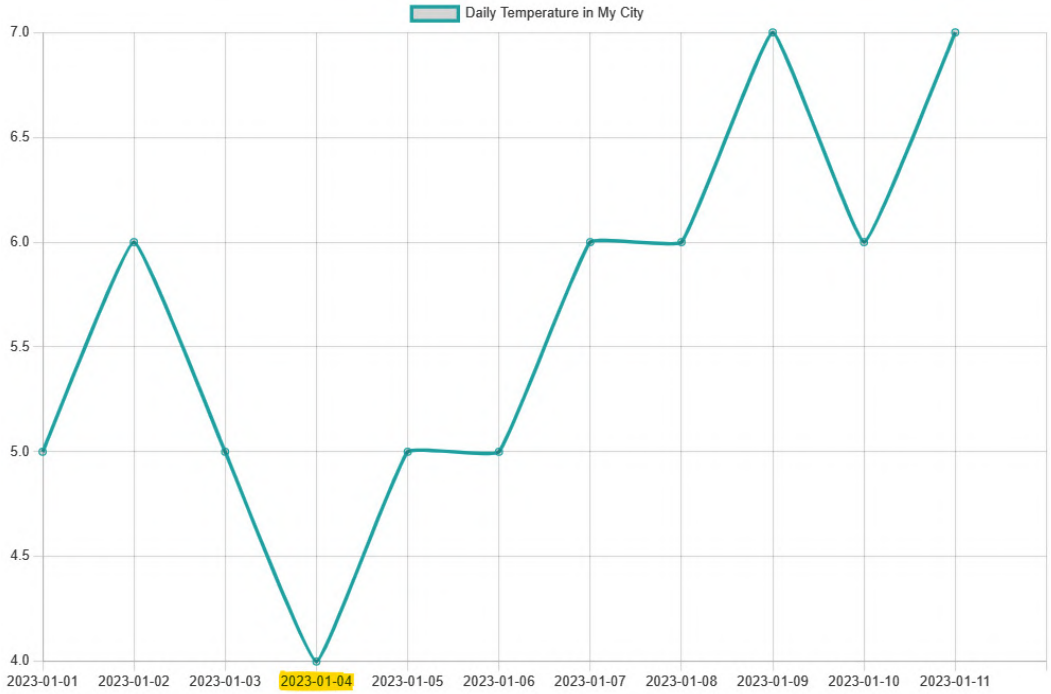
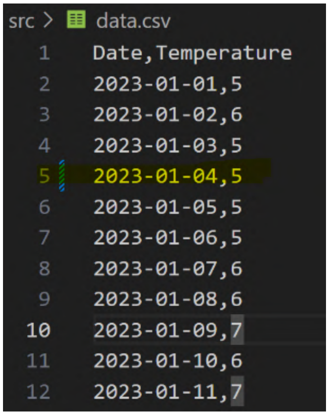


Figure 9.10 – A screenshot of weather-app running as a container on AKS

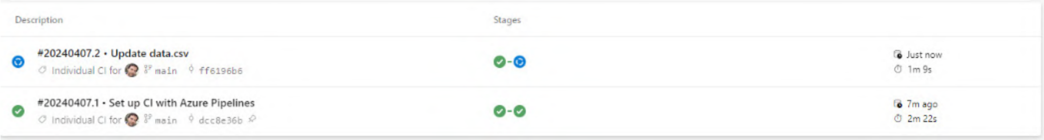
11. Now, to explore the CI/CD capabilities of Azure DevOps, you could, for example, edit the `data.csv` file, changing the value for the `2023-01-04` day from `4.0` to, say, `5.0` (see *Figure 9.11*). This file can be edited directly in your GitHub repository or by cloning the repository locally and using Git.



```
Date,Temperature
2023-01-01,5
2023-01-02,6
2023-01-03,5
2023-01-04,5
2023-01-05,5
2023-01-06,5
2023-01-07,6
2023-01-08,6
2023-01-09,7
2023-01-10,6
2023-01-11,7
```

Figure 9.11 – The updated data.csv file used as a data source from our weather-app application

At this stage, the pipeline will be automatically triggered once more, as illustrated in *Figure 9.12*.



Description	Stages	
#20240407.2 • Update data.csv Individual CI for main ff6196b6		Just now 1m 9s
#20240407.1 • Set up CI with Azure Pipelines Individual CI for main dcc8e36b		7m ago 2m 22s

Figure 9.12 – New execution of the pipeline after pushing the updated data.csv file

12. By executing a new `port-forward` command targeting the newly deployed pod, we will have the opportunity to visualize the updated chart (see *Figure 9.13*).

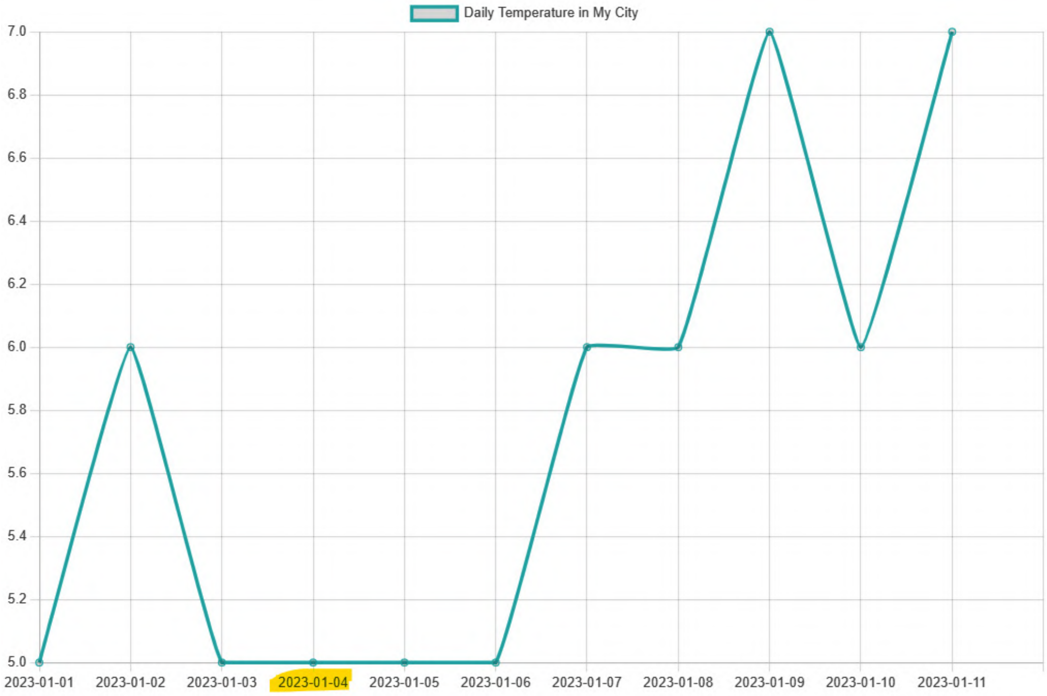


Figure 9.13 – The weather-app application after a new deployment, triggered by pushing the updated data.csv file

13. To avoid incurring unnecessary costs, remember to destroy all Azure resources created for this example by typing the following command (or using the Azure portal):

```
$ terraform destroy --auto-approve
```

Congratulations! You have successfully completed your Azure DevOps CI/CD pipeline with deployment to AKS using an ACR instance. Now, it's time to transition to the AWS ecosystem to explore how things operate on that side.

AWS GitOps essentials

AWS embraces the GitOps model by offering a suite of services designed to facilitate the management of cloud-native applications and infrastructure with high efficiency and reliability. The EKS stands out as AWS's managed Kubernetes service, compared to other major cloud providers' offerings such as **Google Kubernetes Engine** (**GKE**) from Google Cloud and AKS from Microsoft Azure. Each provides similar functionalities but with unique features tailored to their respective ecosystems, optimizing the deployment process and ensuring the automatic scaling and management of containerized applications. Integrating EKS with AWS CodePipeline, a service that automates the build, test, and deploy phases of your release process, enables a GitOps approach where the entire infrastructure is treated as code. This integration empowers teams to implement CD practices, allowing for rapid and safe application updates. AWS's commitment to GitOps is evident in its tooling and services, which support immutable infrastructure, automated deployments, and detailed monitoring, aligning with the GitOps principles of declarative configuration and version control.

AWS CodePipepline

AWS CodePipeline (`https://aws.amazon.com/codepipeline/`) is a fully managed CI/CD service that automates the build, test, and deployment phases of your release process. It allows you to create pipelines that automate the steps required to release your software changes continuously. With CodePipeline, you can define your release process as a series of stages, where each stage performs a specific action, such as fetching source code from a version control system, building and testing your application, and deploying it to your infrastructure.

One of the key features of AWS CodePipeline is its flexibility and integration with other AWS services. You can easily integrate CodePipeline with services such as **AWS CodeBuild** for building your application, **AWS CodeDeploy** for deploying it to EC2 instances or **AWS Lambda functions**, and **AWS Elastic Beanstalk** for deploying and managing web applications.

CodePipeline also supports integration with third-party tools and services through custom actions, allowing you to extend its capabilities to fit your specific requirements. Additionally, it provides visibility into your release process through its web-based console, allowing you to monitor the progress of your pipelines and troubleshoot any issues that arise. The following are the steps for implementing Kubernetes GitOps deployment with AWS CodePipeline:

1. **Set up AWS CodePipeline**: Begin by setting up AWS CodePipeline, which orchestrates the CI/CD workflow for your Kubernetes deployment.
2. **Connect to the GitHub repository**: In the CodePipeline configuration, connect to your GitHub repository where your Kubernetes manifests and deployment scripts are stored.
3. **Configure source stage**: Define the source stage in the CodePipeline configuration, specifying the GitHub repository and branch to pull the Kubernetes manifests from.
4. **Add build stage**: Create a build stage in the CodePipeline configuration to execute any necessary build steps for your Kubernetes deployment, such as compiling code or packaging artifacts.

5. **Integrate with EKS**: Incorporate EKS into your CodePipeline workflow. This may involve setting up connections or permissions between AWS services and CodePipeline.
6. **Implement GitOps principles**: Ensure that your CI/CD pipeline adheres to GitOps principles, such as storing all configuration and deployment manifests in version control, automating deployment processes, and using pull-based synchronization for cluster updates.
7. **Define deployment strategy**: Define the deployment strategy for your Kubernetes application, specifying parameters such as rollout strategy, scaling options, and health checks.
8. **Trigger deployments**: Configure AWS CodePipeline to trigger deployments automatically whenever changes are pushed to the GitHub repository, maintaining the CD of your Kubernetes application.
9. **Monitor and debug**: Implement monitoring and logging mechanisms to track the performance and health of your Kubernetes deployments. Ensure that you have tools in place to debug and troubleshoot any issues that may arise during deployment.
10. **Iterate and improve**: Continuously iterate on your CI/CD pipeline, incorporating feedback and making improvements to enhance the efficiency, reliability, and security of your Kubernetes deployments over time.

Wrapping up, AWS CodePipeline streamlines the release process, empowering teams to deliver software changes more rapidly and reliably. By automating the deployment pipeline, CodePipeline helps accelerate time to market and enhances overall productivity, enabling organizations to respond swiftly to customer needs and market demands.

Kubernetes deployment with AWS CodePipeline

In this section, we embark on a hands-on journey to deploy a Kubernetes cluster on AWS, leveraging Terraform, EKS, and AWS CodePipeline. We'll start by provisioning an EKS cluster and an Amazon **Elastic Container Registry** (**ECR**). The highlight of our journey involves orchestrating the deployment of this image to the EKS cluster using AWS CodePipeline. As done for Azure in the *Kubernetes deployment with Azure DevOps* section, this practical walkthrough demonstrates the seamless integration of these components, showcasing how they streamline the deployment process within a GitOps framework. Through this example, readers will gain a comprehensive understanding of deploying applications to Kubernetes using AWS's powerful services.

For the following walkthrough example, we assume that the reader has a valid AWS account and has already installed and configured AWS CLI version 2. Please review the following AWS links for reference:

- **AWS CLI user guide**: `https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-welcome.html`
- **AWS CLI examples**: `https://docs.aws.amazon.com/cli/latest/userguide/welcome-examples.html`

The following are the necessary steps to successfully complete this example:

1. Ensure that AWS CLI is correctly configured by typing the following command:

   ```
   $ aws –version
   ```

   The output should resemble the following:

   ```
   aws-cli/2.15.35 Python/3.11.8 Windows/10 exe/AMD64 prompt/off
   ```

2. To initialize Terraform, we use the `terraform init` command. For this step, we will utilize the `main.tf` Terraform files located in the `iac/aws` folder of the GitHub repository accompanying this chapter.

   This `main.tf` Terraform configuration file orchestrates the setup of essential AWS infrastructure components for deploying a Kubernetes cluster and managing container images. It begins by defining the AWS provider and version required for the deployment. Following this, it creates an ECR repository named `eksgitops3003204ecr` to store Docker images with mutable tag mutability. Next, the configuration provisions a **virtual private cloud** (**VPC**) and associated subnets using the `terraform-aws-modules/vpc/aws` module. This VPC, named `eks-cluster-vpc`, spans across two availability zones in the `eu-central-1` AWS region. Subsequently, the configuration sets up an EKS cluster utilizing the `terraform-aws-modules/eks/aws` module. The cluster, named `eksgitopscluster`, operates on version 1.29 and allows public access to its endpoint. It's integrated with the previously created VPC and utilizes private subnets for enhanced security.

   Additionally, the configuration establishes a managed node group within the EKS cluster, configured with one instance of the `t3.small` type. This node group provides the computing resources necessary for running containerized applications within the Kubernetes environment. To facilitate seamless interaction between the EKS cluster and the ECR repository, an **identity and access management** (**IAM**) policy named `ecr-pull-policy` is created. This policy grants EKS nodes the permissions required to pull container images from the specified ECR repository.

   Finally, the IAM policy is attached to the IAM role associated with the EKS cluster, ensuring that the cluster nodes have the necessary permissions to retrieve container images for deployment.

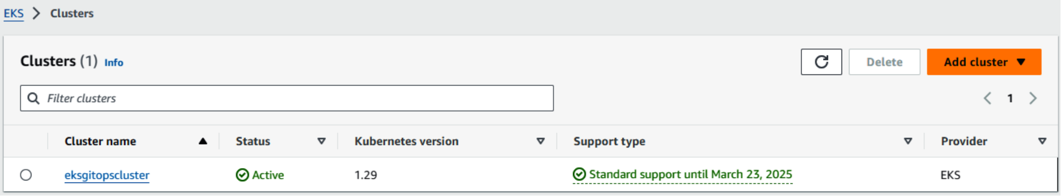
IAM policy

An IAM policy is a document that defines permissions and is used to manage access to AWS resources. IAM policies grant specific rights to users, groups, or roles, determining what actions they can perform on which resources. These policies are made up of statements that include components such as **Effect** (allow or deny), **Action** (the specific actions permitted or denied), and **Resource** (the specific resources to which the actions apply). IAM policies help ensure secure and granular access control within an AWS environment, enabling administrators to enforce the principle of least privilege by giving entities only the permissions they need to perform their tasks.

3. Type the following commands to start the cluster creation:

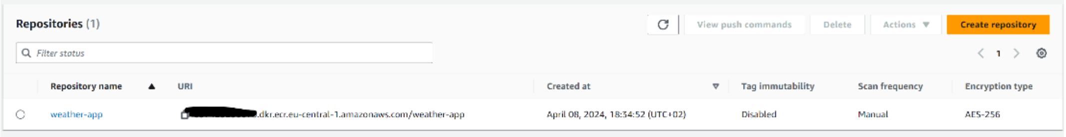
```
$ terraform init
$ terraform plan -out aksplan
$ terraform apply -auto-approve aksplan
```

The entire process will take nearly 10 minutes to complete. *Figure 9.14* illustrates the EKS cluster in the AWS console, while *Figure 9.15* shows the created ECR registry.



	Cluster name	Status	Kubernetes version	Support type	Provider
○	eksgitopscluster	Active	1.29	Standard support until March 23, 2025	EKS

Figure 9.14 – The EKS cluster after the execution of the Terraform script



	Repository name	URI	Created at	Tag immutability	Scan frequency	Encryption type
○	weather-app	[redacted].dkr.ecr.eu-central-1.amazonaws.com/weather-app	April 08, 2024, 18:34:52 (UTC+02)	Disabled	Manual	AES-256

Figure 9.15 – The ECR registry after the execution of the Terraform script

4. Integrate the `kubectl` configuration for cluster management:

```
$ aws eks --region eu-central-1 update-kubeconfig --name
eksgitopscluster
```

5. Type the following command to test the access:

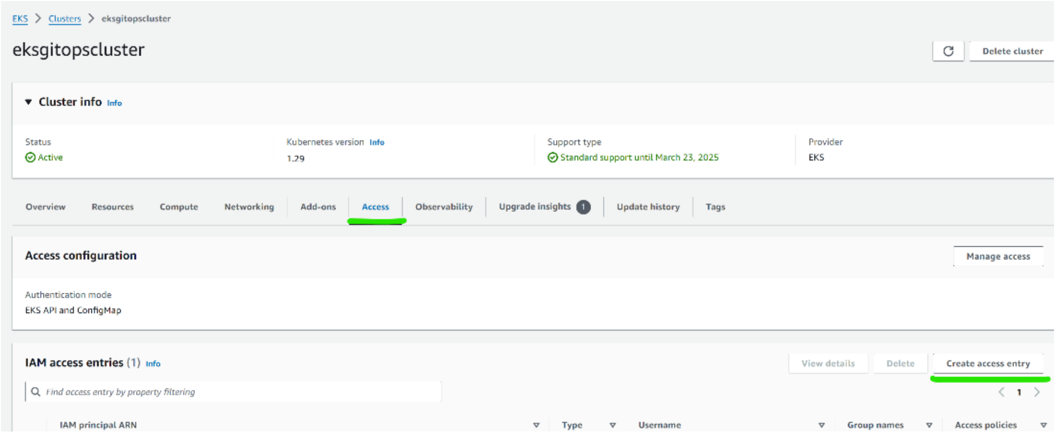
```
$ kubectl cluster-info
```

The output should be as follows:

```
Kubernetes control plane is running at
https://54CE9D2FAC3008E8E5B3D4873E92E7B2.yl4.eu-central-1.eks.
amazonaws.com
CoreDNS is running at https://54CE9D2FAC3008E8E5B3D4873E92E7B2.
yl4.eu-central-1.eks.amazonaws.com/api/v1/namespaces/kube-
system/services/kube-dns:dns/proxy
```

If you encounter authentication issues or generic issues after merging the cluster management, you need to add an access entry for the EKS cluster. This involves adding the following permissions to the user configured for the AWS CLI, as specified at the beginning of this section:

1. In the AWS console, navigate to the **Access** tab of the EKS cluster (e.g., `eksgitopscluster`) and click on the **Create access entry** button, as illustrated in *Figure 9.16*:



Status: Active

Kubernetes version: 1.29

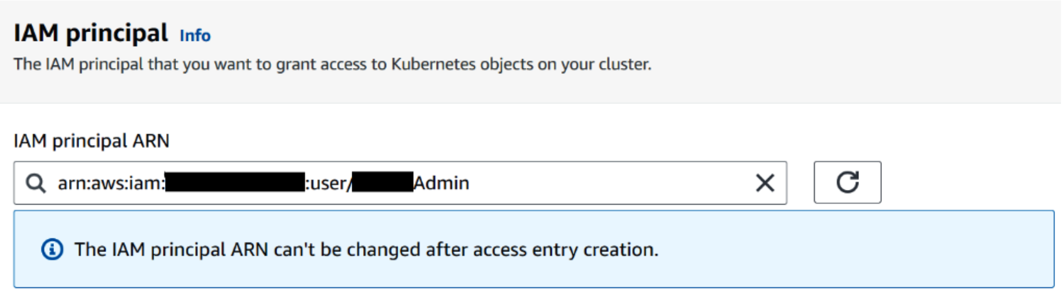
Support type: Standard support until March 23, 2025

Provider: EKS

Authentication mode: EKS API and ConfigMap

Figure 9.16 – The EKS cluster page and the Access tab for creating a new access entry

2. In the **IAM Principal ARN** field, you have to select the IAM ARN that has been used to configure the AWS CLI and execute AWS CLI commands, as illustrated in *Figure 9.17*:



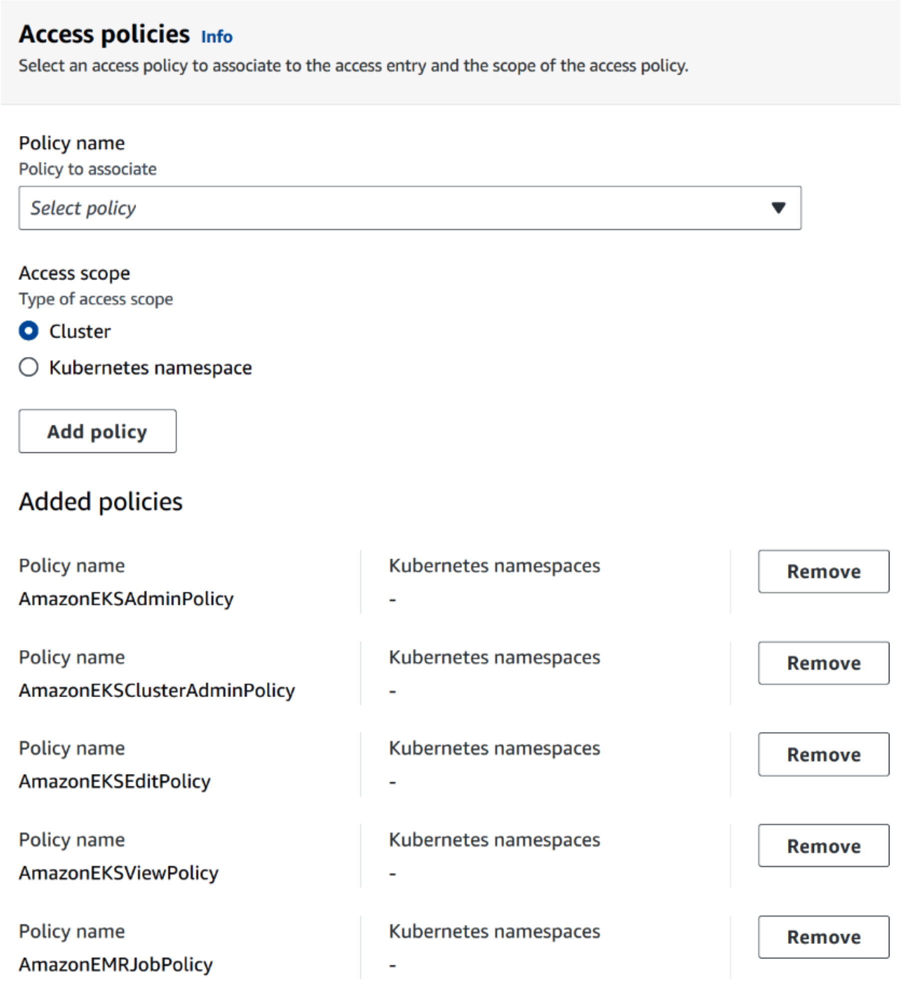
The IAM principal that you want to grant access to Kubernetes objects on your cluster.

arn:aws:iam:[redacted]:user/[redacted]Admin

The IAM principal ARN can't be changed after access entry creation.

Figure 9.17 – The IAM principal ARN selection for AWS CLI configuration

3. Then, click the **Next** button at the bottom of the page, and for this example, add all the following policy names (see *Figure 9.18*) with **Cluster** as the selected **Access scope**:



Access policies Info

Select an access policy to associate to the access entry and the scope of the access policy.

Policy name

Policy to associate

Select policy ▼

Access scope

Type of access scope

● Cluster

○ Kubernetes namespace

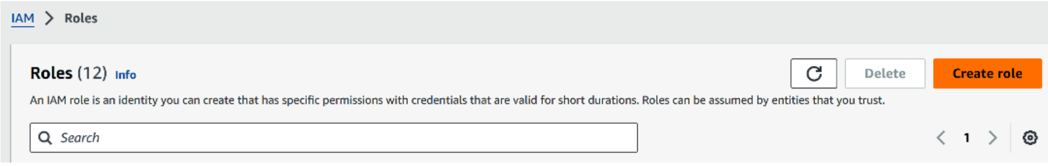
Add policy

Added policies

Policy name	Kubernetes namespaces	
AmazonEKSAdminPolicy	-	Remove
AmazonEKSClusterAdminPolicy	-	Remove
AmazonEKSEditPolicy	-	Remove
AmazonEKSViewPolicy	-	Remove
AmazonEMRJobPolicy	-	Remove

Figure 9.18 – The Access policies section for attaching permissions to the new IAM principal

4. Click the **Next** button and then **Create**. Now, you should be able to administer the cluster using AWS CLI without any issues. For more information and details, please refer to the official documentation: `https://docs.aws.amazon.com/eks/latest/userguide/access-entries.html#updating-access-entries`.
5. Before proceeding with the creation of an instance of AWS CodePipeline, we first need to create an **IAM role** named `GitOpsCodeBuildRole`. This role will allow the pipeline to build a new image for our `weather-app` application whenever new code is committed to the repository, push the image to the created ECR, and deploy it to EKS. In the AWS console, navigate to **IAM** |**Roles**, and click on the **Create role** button, as illustrated in *Figure 9.19*.

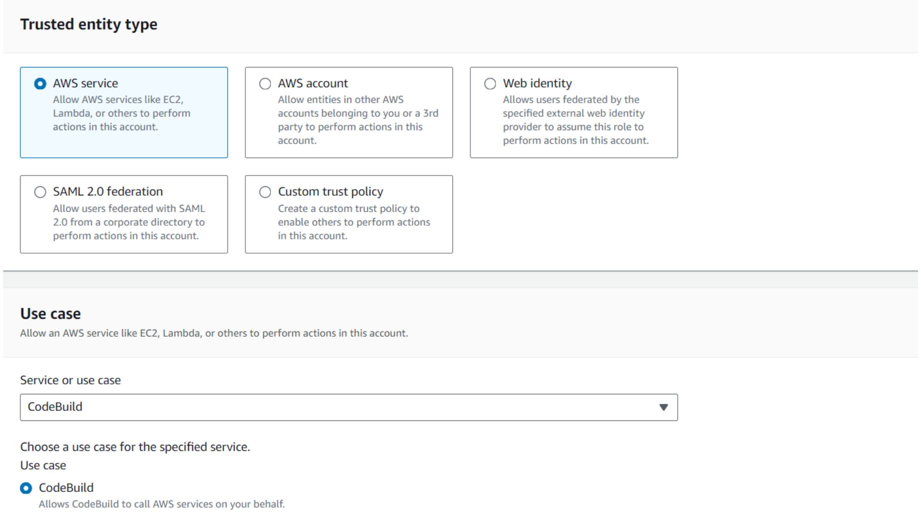


An IAM role is an identity you can create that has specific permissions with credentials that are valid for short durations. Roles can be assumed by entities that you trust.

Figure 9.19 – The Roles page in the AWS Management Console, where you can begin creating a new IAM role

> **IAM role**
>
> An **IAM role** in AWS is a set of permissions that define what actions can be performed on AWS resources. Roles are used to delegate access to users, applications, or services within AWS, allowing them to interact securely with various AWS services. Roles are defined with policies that specify the actions allowed or denied, and they are assumed by entities such as AWS services, IAM users, or AWS resources. This approach ensures secure access control and helps enforce the principle of least privilege, where users and services have only the permissions necessary to perform their tasks.

6. In the **Select trusted entity** section, choose **AWS service**, and in the **Use case** panel, select **CodeBuild**, as shown in *Figure 9.20*.



Trusted entity type

● AWS service
Allow AWS services like EC2, Lambda, or others to perform actions in this account.

○ AWS account
Allow entities in other AWS accounts belonging to you or a 3rd party to perform actions in this account.

○ Web identity
Allows users federated by the specified external web identity provider to assume this role to perform actions in this account.

○ SAML 2.0 federation
Allow users federated with SAML 2.0 from a corporate directory to perform actions in this account.

○ Custom trust policy
Create a custom trust policy to enable others to perform actions in this account.

Use case
Allow an AWS service like EC2, Lambda, or others to perform actions in this account.

Service or use case
CodeBuild

Choose a use case for the specified service.
Use case
● CodeBuild
Allows CodeBuild to call AWS services on your behalf.

Figure 9.20 – The Trusted entity type and Use case section on the Create role page

7. On the **Add permissions** page, add the following permissions:

    - `AmazonEC2ContainerRegistryFullAccess`
    - `AmazonS3FullAccess`
    - `AWSCodeBuildAdminAccess`
    - `AWSCodeCommitFullAccess`
    - `CloudWatchLogsFullAccess`

    Then, add the following **inline policy**:

```
{
      "Version": "2012-10-17",
      "Statement": [
            {
                  "Effect": "Allow",
                  "Action": "eks:Describe*",
                  "Resource": "*"
            }
      ]
}
```

Inline policy

An **inline policy** is a set of permissions that can be directly attached to an IAM user, group, or role. Unlike managed policies, which are standalone entities, inline policies are embedded directly within the resources they are intended to control. This allows for more granular control and management of permissions at a more specific level. Inline policies are often used when certain permissions need to be applied only to a specific user, group, or role, rather than being shared across multiple entities.

Before reviewing and creating the new IAM role, it's important to ensure the addition of the specified trust relationship to control who can assume the role, enhancing security and compliance. This setup prevents unauthorized access and ensures that only designated entities can access certain AWS resources:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "codebuild.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

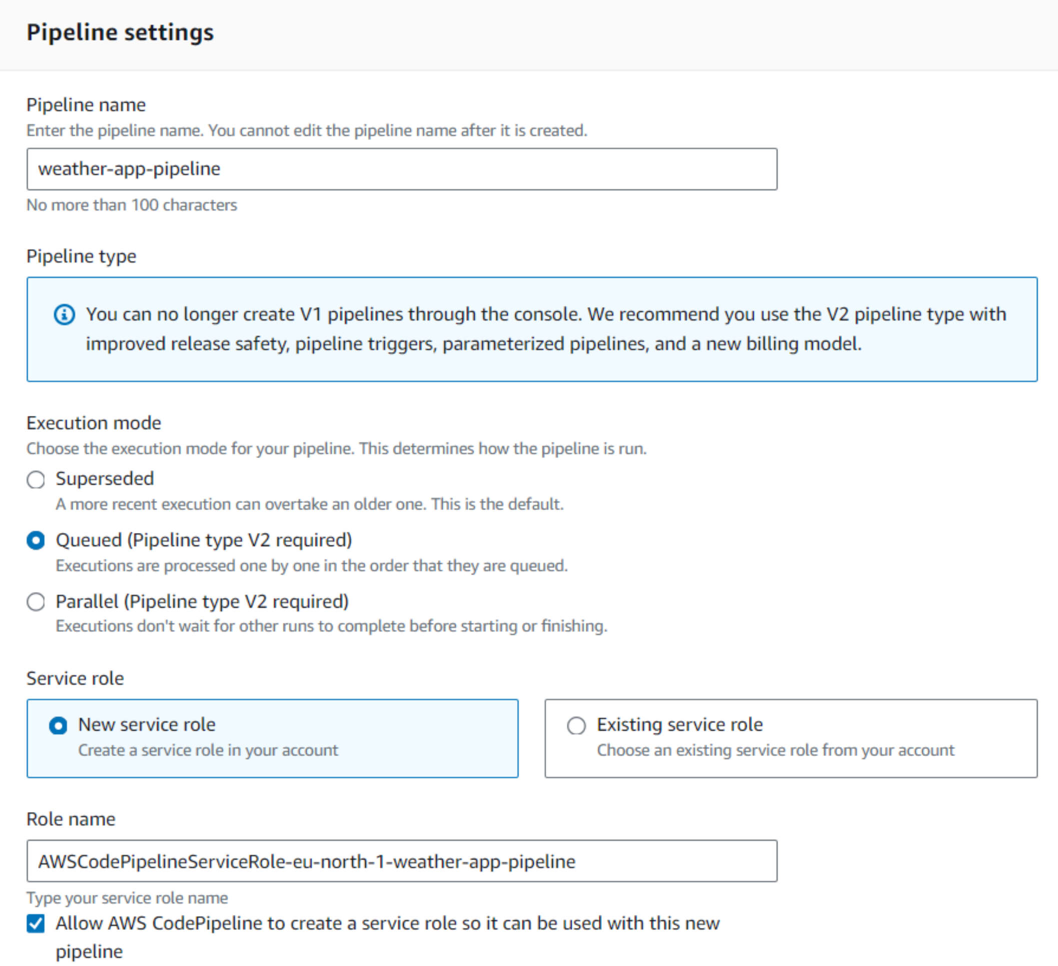
To complete the process, simply click the **Create** button. Now, we're all set to create the AWS CodePipeline instance.

8. Following a similar approach as done for Azure DevOps, refer to the *Kubernetes deployment with Azure DevOps* section in this chapter; it's time to automate the application deployment to AWS EKS using AWS CodePipeline.

   To set up the CodePipeline correctly, navigate to the AWS console, go to **Services | CodePipeline | Create pipeline**, and configure the pipeline settings with the following values, as illustrated in *Figure 9.21*:

   - **Pipeline name**: `weather-app-pipeline`
   - **Pipeline type**: **V2**
   - **Execution mode**: **Queued**
   - **Service role**: **New service role**

Then, click the **Next** button.



Pipeline settings

Pipeline name

Enter the pipeline name. You cannot edit the pipeline name after it is created.

weather-app-pipeline

No more than 100 characters

Pipeline type

You can no longer create V1 pipelines through the console. We recommend you use the V2 pipeline type with improved release safety, pipeline triggers, parameterized pipelines, and a new billing model.

Execution mode

Choose the execution mode for your pipeline. This determines how the pipeline is run.

○ Superseded
A more recent execution can overtake an older one. This is the default.

● Queued (Pipeline type V2 required)
Executions are processed one by one in the order that they are queued.

○ Parallel (Pipeline type V2 required)
Executions don't wait for other runs to complete before starting or finishing.

Service role

● New service role
Create a service role in your account

○ Existing service role
Choose an existing service role from your account

Role name

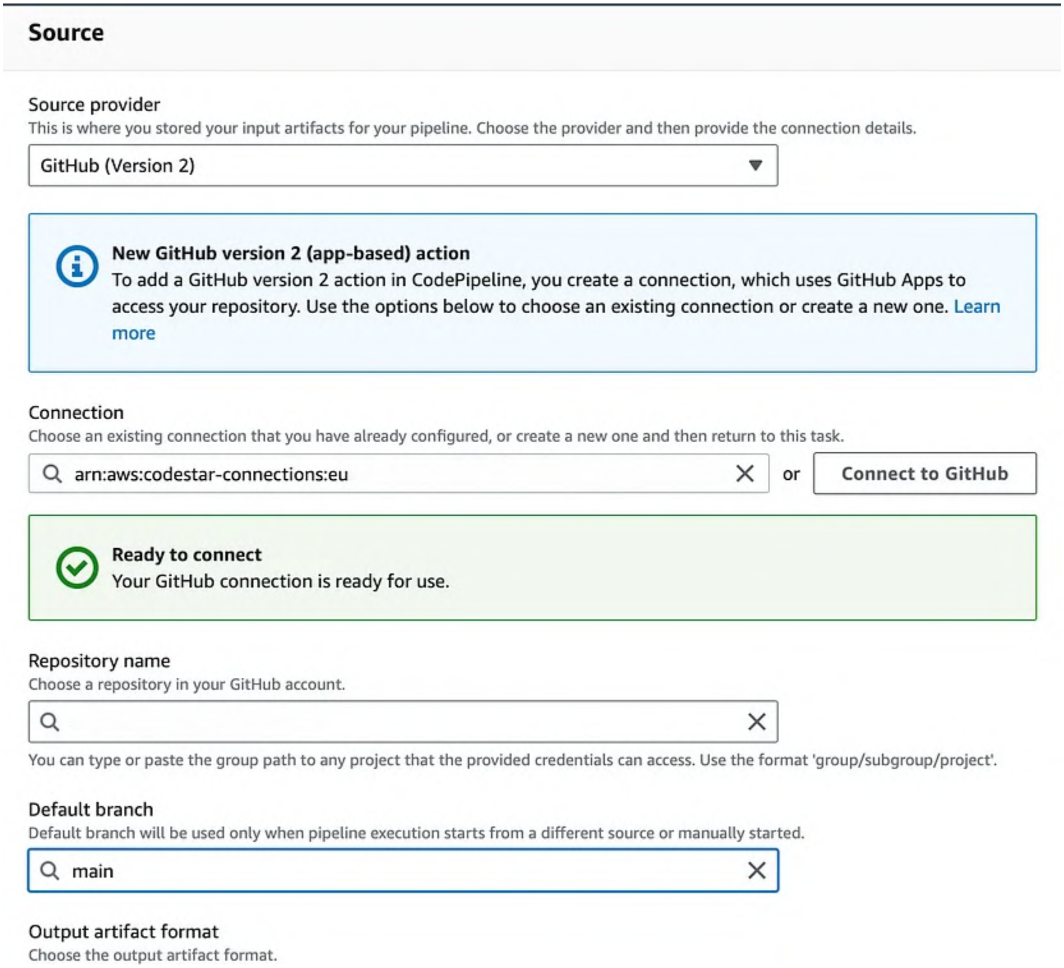
AWSCodePipelineServiceRole-eu-north-1-weather-app-pipeline

Type your service role name

☑ Allow AWS CodePipeline to create a service role so it can be used with this new pipeline

Figure 9.21 – Initial configurations for the new AWS CodePipeline

9. In **Add source stage**, select `GitHub (Version 2)` as **Source provider**, then click on **Connect to GitHub** and follow the instructions to authenticate your GitHub connection. Once the connection is established, choose the GitHub repository in the **Repository name** field. Set the **Default** branch to `main`. For **Output artifact format**, select the `CodePipeline` default. Finally, click the **Next** button.



Source

Source provider

This is where you stored your input artifacts for your pipeline. Choose the provider and then provide the connection details.

GitHub (Version 2) ▼

New GitHub version 2 (app-based) action

To add a GitHub version 2 action in CodePipeline, you create a connection, which uses GitHub Apps to access your repository. Use the options below to choose an existing connection or create a new one. Learn more

Connection

Choose an existing connection that you have already configured, or create a new one and then return to this task.

arn:aws:codestar-connections:eu

Ready to connect

Your GitHub connection is ready for use.

Repository name

Choose a repository in your GitHub account.

You can type or paste the group path to any project that the provided credentials can access. Use the format 'group/subgroup/project'.

Default branch

Default branch will be used only when pipeline execution starts from a different source or manually started.

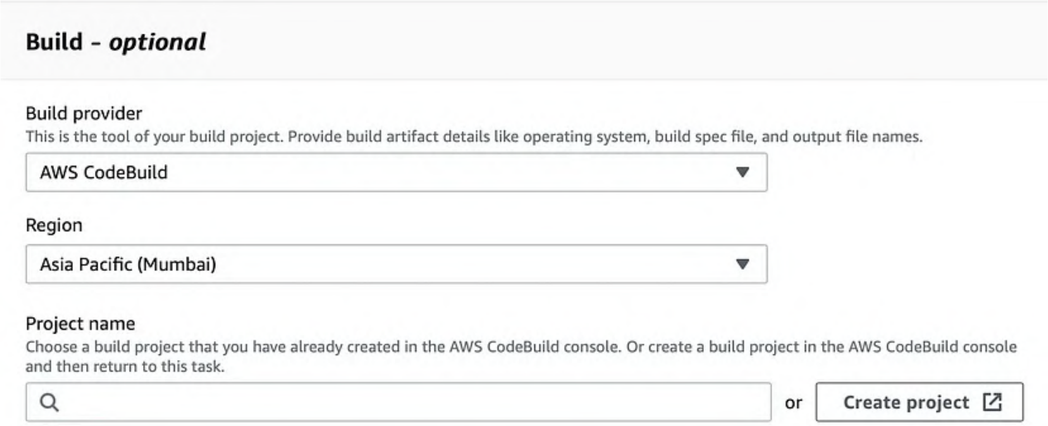
main

Output artifact format

Choose the output artifact format.

Figure 9.22 – Connection with the GitHub repository source

10. In **Add build stage**, select **AWS CodeBuild** as the build provider (refer to *Figure 9.23*). Choose the same region where EKS and ECR resources have been previously deployed, and then click on **Create project**.



Build - *optional*

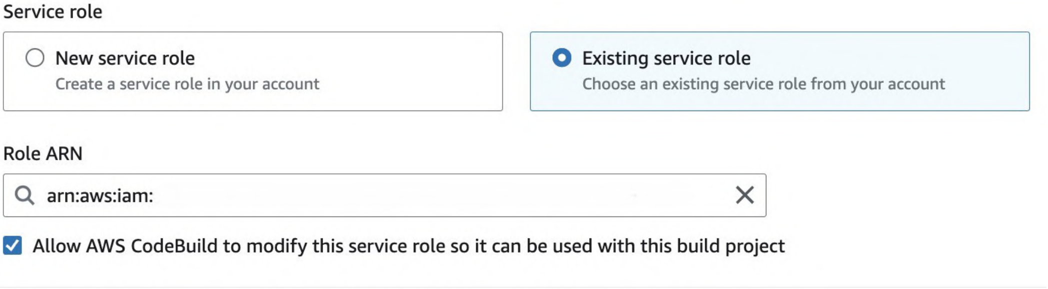
Build provider
This is the tool of your build project. Provide build artifact details like operating system, build spec file, and output file names.
AWS CodeBuild ▼

Region
Asia Pacific (Mumbai) ▼

Project name
Choose a build project that you have already created in the AWS CodeBuild console. Or create a build project in the AWS CodeBuild console and then return to this task.

Figure 9.23 – The section of the Build panel related to selecting the build provider and region

11. In the **Create build project** window, set `weather-app-build` as the project name and keep all other values as default, except for **Service role** where you need to select **Existing service role** and enter the previously created role (see *Figure 9.24* for more details): `arn:aws:iam::[AWS_ACCOUNT_ID]:role/GitOpsCodeBuildRole`.



Service role

○ New service role
Create a service role in your account

● Existing service role
Choose an existing service role from your account

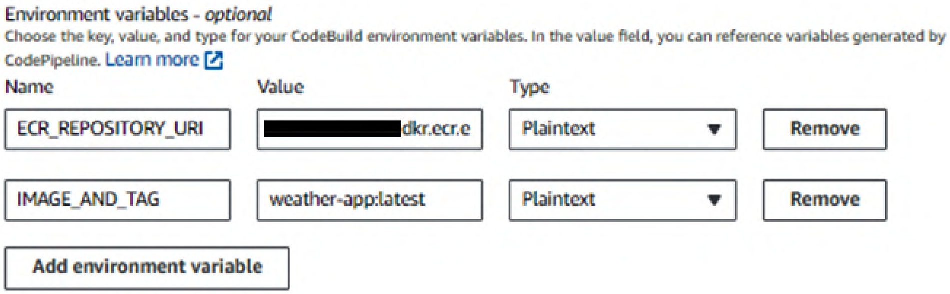
Role ARN
arn:aws:iam:

☑ Allow AWS CodeBuild to modify this service role so it can be used with this build project

Figure 9.24 – The section of the Build panel related to the selection of service roles and roles

12. In the **Build** panel, select **Insert build commands**, then click on **Switch to editor**, and add the content of the `buildspec.yaml` file. Then, click the **Continue to CodePipeline** button. Click on the **Add environment variable** button and add two environment variables (refer to *Figure 9.25*):

    - `ECR_REPOSITORY_URI`: `[AWS_ACCOUNT_ID].dkr.ecr.eu-central-1.amazonaws.com`
    - `IMAGE_AND_TAG`: `weather-app:latest`



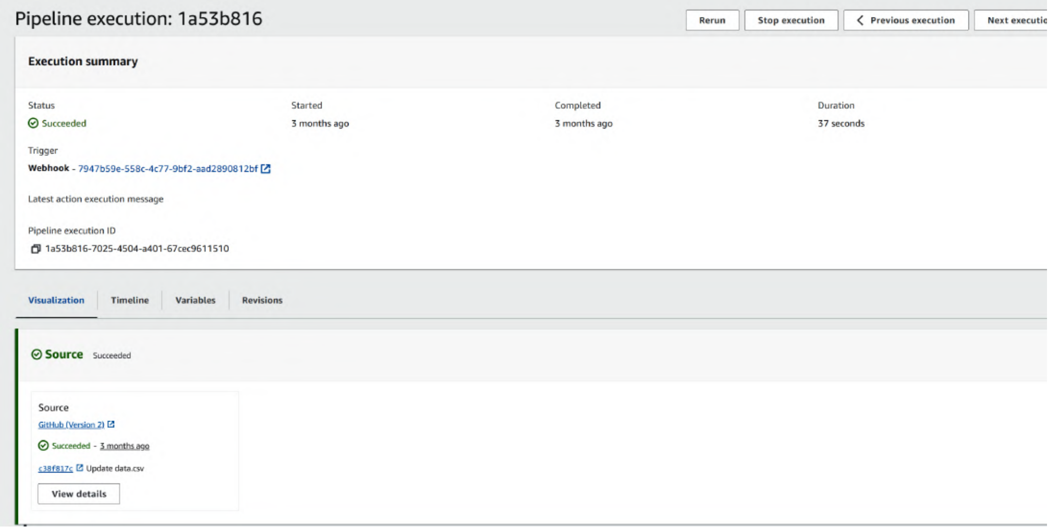
Environment variables - *optional*

Choose the key, value, and type for your CodeBuild environment variables. In the value field, you can reference variables generated by CodePipeline. Learn more

Name	Value	Type
ECR_REPOSITORY_URI	[redacted]dkr.ecr.e	Plaintext
IMAGE_AND_TAG	weather-app:latest	Plaintext

Figure 9.25 – The section of the Build panel related to adding environment variables

13. For **Build type**, select **Single Build**. Click the **Next** button. At the **Add deploy stage** step, click on the **Skip deploy stage** button, and then on **Next**. Review the pipeline at the final stage and click the **Create pipeline** button. At this point, the CodePipeline will be triggered and completed successfully in one minute (see *Figure 9.26*).



Pipeline execution: 1a53b816

Execution summary

Status	Started	Completed	Duration
Succeeded	3 months ago	3 months ago	37 seconds

Trigger

Webhook - 7947b59e-558c-4c77-9bf2-aad2890812bf

Latest action execution message

Pipeline execution ID

1a53b816-7025-4504-a401-67cec9611510

Source Succeeded

Source

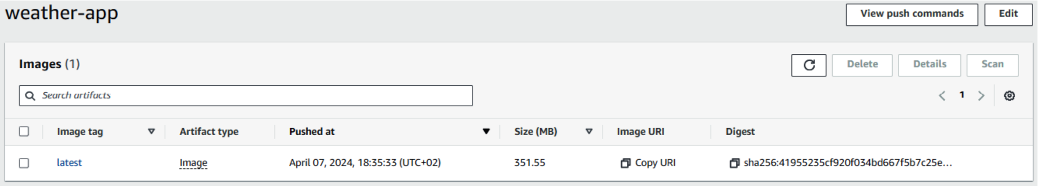
GitHub (Version 2)

Succeeded - 3 months ago

c38f817c Update data.csv

Figure 9.26 – weather-app-pipeline immediately after its definition

14. At the end of the pipeline execution, we can verify that the Docker image of `weather-app` has been correctly built and pushed to the ECR, as illustrated in *Figure 9.27*.



weather-app

Images (1)

☐	Image tag	Artifact type	Pushed at	Size (MB)	Image URI	Digest
☐	latest	Image	April 07, 2024, 18:35:33 (UTC+02)	351.55	Copy URI	sha256:41955235cf920f034bd667f5b7c25e...

Figure 9.27 – The weather-app:latest image successfully pushed to the ECR

15. To verify that the deployment has been successfully completed, execute the following command:

```
$ kubectl get deployments
```

Here is the output:

```
NAME                   READY   UP-TO-DATE   AVAILABLE   AGE
my-city-weather-app    1/1     1            1           64m
```

16. At this point, we can execute a `port-forward` command to display the app in the browser, as illustrated in *Figure 9.28*.

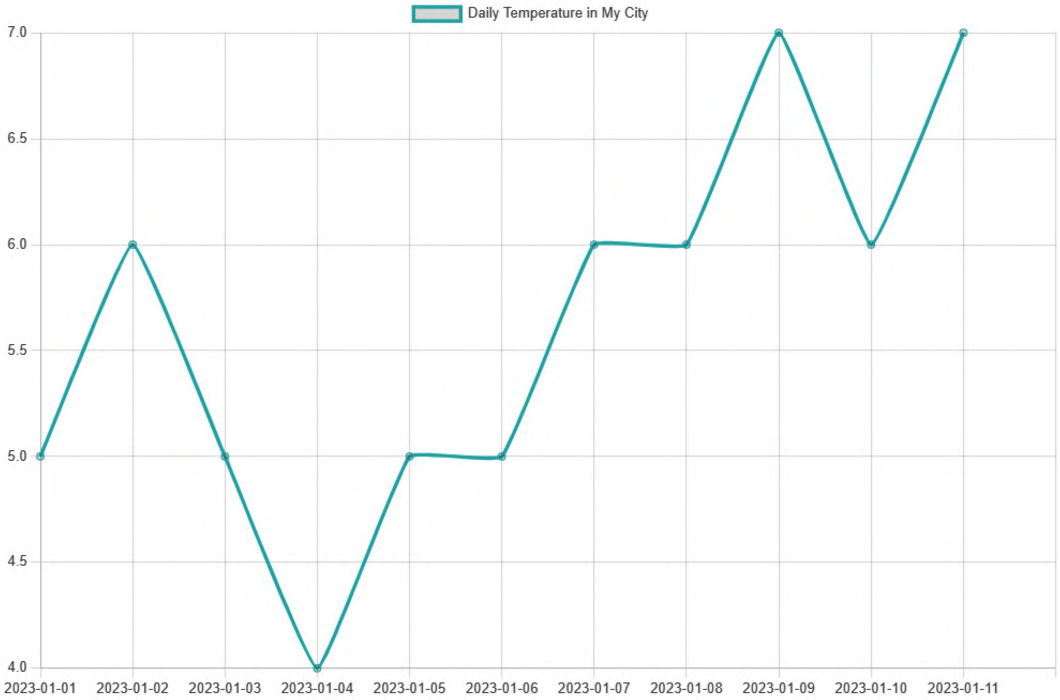


Figure 9.28 – weather-app rendered in the browser after the deployment

17. Attempt to edit the data source file as described in *Step 11* of the *Kubernetes deployment with Azure DevOps* section. After pushing the changes, `weather-pipeline` will be triggered immediately, as illustrated in *Figure 9.29*. Perform a new `port-forward` command to view the updated version of the chart.

	Name	Latest execution status	Latest source revisions	Latest execution started	Most recent executions
○	weather-app-pipeline (Type: V2 | Execution mode: QUEUED)	In progress	-	Just now	View details

Figure 9.29 – The pipeline is immediately triggered after pushing the updated data.csv file

To avoid incurring unnecessary costs, remember to delete all AWS resources created for this example after you've finished by executing the appropriate commands or using the AWS Management Console.

Congratulations! At this stage, you've successfully completed the deployment of the weather application on Azure using Azure DevOps and on AWS using CodePipeline. Now, let's delve into GitOps applications in cloud environments in the next section.

GitOps applications in cloud environments

The advent of GitOps has significantly revolutionized the way cloud environments are managed and deployed, by embedding the principles of version control and collaboration into the operational fabric of cloud-native applications. GitOps applications extend beyond mere deployment automation, encapsulating the entire life cycle of cloud resources and services. This includes provisioning, scaling, updating, and decommissioning, all governed through Git pull requests. The methodology fosters a transparent, auditable, and easily reversible process, enhancing both the security and compliance of cloud applications. Moreover, GitOps practices ensure that the desired state of the cloud environment is declaratively defined and maintained, promoting consistency and reliability across different stages of development and production. This systematic approach minimizes discrepancies between environments, significantly reducing *it works on my machine* issues and streamlining the path to production.

Cross-cloud strategies

In today's multi-cloud landscape, organizations often leverage the unique advantages of Azure, AWS, and other cloud providers to optimize their operations and costs. Managing deployments across such diverse environments can present challenges, particularly in maintaining consistency and efficiency. GitOps offers a unified strategy for managing these deployments, facilitating cross-cloud interoperability and configuration management. The following is a list of benefits of adopting GitOps in a cross-cloud setup:

- **Unified configuration management**: By storing infrastructure definitions and configurations as code in a Git repository, teams can use the same GitOps workflows to manage deployments across Azure, AWS, and other clouds. This centralizes control and ensures consistency across cloud environments.

- **Interoperability and portability**: Leveraging containerization and Kubernetes, applications can be designed to run seamlessly across different clouds. GitOps practices, coupled with these technologies, simplify the process of deploying and managing these applications, irrespective of the underlying cloud platform.
- **Automated synchronization**: Tools such as Argo CD or Flux can be used to automatically synchronize the desired state in the Git repository with the actual state in each cloud environment. This ensures that all environments adhere to the same configurations and policies, facilitating a smooth and consistent operational workflow across clouds.
- **Environment parity**: GitOps enables teams to replicate environments across different clouds with high fidelity. This is particularly useful for testing, where an application deployed on Azure can be tested under similar conditions on AWS, ensuring that the application behaves consistently across platforms.
- **Secrets management**: Managing secrets and sensitive information can be challenging in a cross-cloud setup. GitOps workflows can integrate with cloud-specific secrets management services such as Azure Key Vault or AWS Secrets Manager, allowing secure handling of secrets while maintaining the flexibility of cross-cloud deployments.

Adopting a GitOps approach for cross-cloud strategies not only simplifies the complexity inherent in multi-cloud environments but also enhances operational efficiency, security, and compliance. By treating infrastructure and application configurations as code, teams gain the agility to adapt to changes swiftly, ensuring their deployments remain aligned with organizational goals and industry best practices. In the next section, we will introduce which GitOps strategies should be adopted for Kubernetes deployments on Azure and AWS.

GitOps strategies for Azure and AWS deployments for Kubernetes

In the realm of Kubernetes deployments, GitOps strategies offer a paradigm shift towards more efficient, transparent, and reliable operations. By leveraging GitOps principles, organizations can automate deployment processes, ensure consistency across environments, and significantly enhance their operational workflows. The following are insights into GitOps strategies tailored for Kubernetes deployments on Azure and AWS.

Azure GitOps strategies

Adopting GitOps strategies for AKS entails a detailed approach to integrating source control, CI/CD pipelines, and configuration management to enhance deployment processes. A pivotal strategy involves the deployment of IaC using tools such as ARM templates or Terraform, which are stored in Git repositories. This approach enables the declarative management of AKS configurations. *Chapters 10* and *11* will provide comprehensive examples of automating IaC with Terraform on Azure. These automations facilitate automated and repeatable deployments, thereby improving the stability and scalability of applications.

The use of Azure Policy in conjunction with GitOps further enforces compliance and governance across Kubernetes clusters, ensuring that deployments adhere to organizational and regulatory standards. Integrating Azure Monitor with GitOps workflows enables teams to implement observability as a core component of their operations, allowing for proactive monitoring and troubleshooting of AKS deployments.

AWS GitOps strategies

AWS offers a robust ecosystem for implementing GitOps with EKS. The foundation of AWS GitOps strategies lies in leveraging Amazon ECR for the Docker container registry, AWS CodeCommit for source control, and AWS CodePipeline for continuous integration and deployment. Similar to Azure, AWS advocates for the use of IaC, with AWS CloudFormation or Terraform as preferred tools, to manage EKS cluster configurations and resources in a declarative manner.

An effective GitOps strategy on AWS encompasses the integration of AWS CodeBuild and AWS CodeDeploy within the CI/CD pipeline, automating the build, test, and deployment phases directly from Git repositories. Moreover, the AWS App Mesh service can be integrated into GitOps workflows to manage microservices more effectively, providing end-to-end visibility and network traffic control across applications.

For both Azure and AWS, implementing GitOps for Kubernetes deployments revolves around four key principles: version control, automated deployments, merge/pull requests for change management, and observability. By adhering to these principles, organizations can achieve automated, predictable, and secure application deployments at scale. Utilizing GitOps not only simplifies Kubernetes cluster management but also aligns operational practices with development workflows, fostering a culture of collaboration and continuous improvement.

Summary

By now, we have gained a thorough understanding of how GitOps can be effectively implemented within the Azure and AWS cloud environments. The chapter covered the necessary tools and processes, such as AKS, Azure DevOps, AWS EKS, and AWS CodePipeline, to establish robust CI/CD pipelines and manage deployments seamlessly. With practical examples and expert advice, this chapter ensured that readers can apply these concepts to achieve more automated, consistent, and secure cloud-native deployments. Emphasizing the importance of a solid foundation in Git, Docker, and Kubernetes, the chapter prepared readers to embrace the full potential of GitOps in their cloud computing ventures. In the next chapter, we will explore the integration of GitOps with Terraform and Flux, focusing on infrastructure. We'll cover the essential steps to align infrastructure as code with real-time operations, using Terraform for provisioning and Flux for CD. The discussion will also highlight best practices and common challenges in this process.

10

GitOps for Infrastructure Automation – Terraform and Flux CD

In this chapter, we explore the powerful synergy between Terraform and Flux CD, two pivotal tools that epitomize the principles of GitOps and **Infrastructure as Code** (**IaC**). As cloud architectures become increasingly complex and dynamic, the need for sophisticated, automated tooling to manage these environments has become critical. Terraform allows users to define and provision cloud infrastructure using a high-level configuration language, creating a blueprint of resources that can be versioned and reused. Complementing this, Flux CD automates the deployment of applications and infrastructure changes by continuously syncing them from Git repositories to Kubernetes clusters. This integration not only simplifies the management of cloud resources but also ensures consistency and reliability across various stages of the deployment pipeline. Through detailed discussions, practical examples, and expert insights, this chapter will equip you with the knowledge to harness the combined power of Terraform and Flux CD, enabling you to automate your infrastructure efficiently and predictably in a cloud-native world.

In this chapter, we will delve into the following key areas:

- Introducing infrastructure automation with Terraform and Flux CD
- Setting up Terraform in a GitOps workflow
- Exploring Flux CD – enabling **continuous deployment** (**CD**) in Kubernetes
- Combining Terraform and Flux CD for enhanced automation
- Version control and automation with Terraform and Flux CD
- Multi-environment management with Terraform and Flux CD

Technical requirements

Before diving into the integration of Terraform and Flux CD for GitOps workflows, it is essential to have a firm grasp of the foundational concepts discussed in earlier chapters of this book. Understanding the fundamentals of IaC, key Kubernetes concepts, and **continuous integration** (**CI**)/CD principles will significantly enhance your comprehension of the material in this chapter. Practical familiarity with Terraform and initial setup experience with Kubernetes clusters are indispensable for applying the techniques covered. Access to a **version control system** (**VCS**), particularly Git, is critical as it forms the backbone of the GitOps methodology. This chapter assumes you have these prerequisites in place, enabling you to fully engage with the advanced strategies of using Terraform and Flux CD for efficient and scalable infrastructure automation. Before diving into deployment examples, it's essential that you have at least a local Kubernetes cluster running. For more information about the different alternatives and OS bases, refer to the *Local cluster setup* section in *Chapter 2*. The relevant code and resource files for this chapter can be found in the `Chapter10` folder of our dedicated GitHub repository: `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`.

Introducing infrastructure automation with Terraform and Flux CD

In today's rapidly evolving cloud computing landscape, the demand for agility, scalability, and reliability in managing infrastructure is paramount. Traditional manual approaches are no longer sufficient for modern applications. Enter IaC and GitOps, reshaping infrastructure management in cloud-native environments. Manual work remains crucial for platform engineers, such as understanding cloud resources to write effective Terraform scripts, and is often the first step before updating the Terraform Git repository. Leverage the GitOps framework to address cloud-native delivery challenges such as preventing configuration drifts and ensuring secure, error-free deployments. Implementing GitOps on Kubernetes introduces challenges, including managing complex configurations and maintaining synchronization between the Git repository and clusters. IaC is a methodology that treats infrastructure provisioning, configuration, and management as code. In essence, it enables developers and operations teams to define infrastructure resources—such as **virtual machines** (**VMs**), networks, and storage—using declarative or imperative code. We have already explored the concept of IaC in the previous chapter, where we utilized Terraform scripts to provision necessary resources in Azure and AWS.

This code is version-controlled, allowing for reproducible, consistent, and automated deployments. By codifying infrastructure configurations, IaC streamlines the deployment process, reduces human error, and enhances collaboration between development and operations teams.

GitOps, on the other hand, extends the principles of IaC to the realm of continuous delivery and deployment. Changes to the infrastructure are made via Git commits, enabling versioning, auditability, and collaboration. GitOps promotes a declarative approach to infrastructure management, where desired state configurations stored in Git repositories are automatically reconciled with the actual state of the infrastructure. This reconciliation is typically facilitated by specialized tools such as Flux CD.

Building on the foundational principles of IaC and GitOps introduced earlier, let's consider a practical example, as shown in *Figure 10.1*, of how companies can leverage Terraform and Flux CD for efficient infrastructure management.

Imagine a technology company that needs to rapidly scale its operations across multiple cloud environments while maintaining strict compliance and security standards. By using Terraform, the company can codify its infrastructure requirements into version-controlled configurations, which allows for quick deployment and easy replication of environments across AWS, Azure, or Google Cloud. With Flux CD integrated into this setup, any changes to the Terraform configurations in the Git repository automatically trigger updates within the Kubernetes clusters. This seamless integration ensures that the infrastructure evolves consistently with the application code, reducing discrepancies and potential errors. The result is a robust, scalable, and compliant infrastructure that can adapt swiftly to the changing needs of the business, all while minimizing manual oversight and intervention:

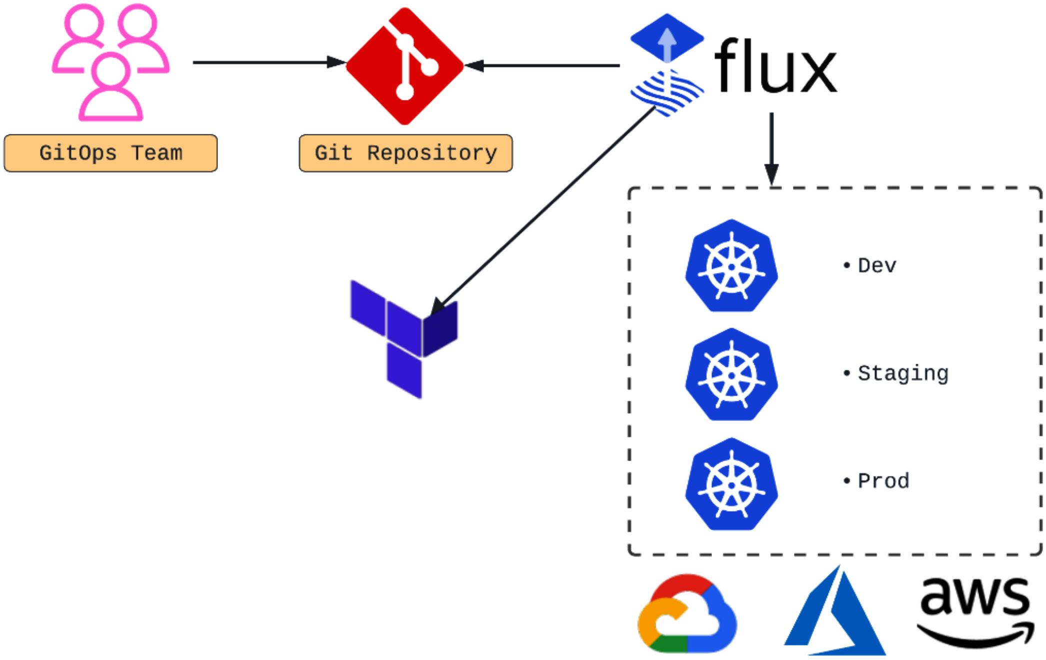


Figure 10.1 – Example of a company leveraging Terraform and Flux CD for efficient infrastructure management

In the context of modern cloud environments, where agility and scalability are imperative, the adoption of IaC and GitOps offers numerous benefits. Firstly, it enables organizations to achieve infrastructure automation, allowing for rapid provisioning, scaling, and modification of resources. Secondly, it enhances reproducibility and consistency across environments, mitigating the risk of configuration drift and ensuring reliability. Thirdly, it fosters collaboration and visibility, as infrastructure configurations are stored and version-controlled in accessible Git repositories. Finally, it promotes resilience and observability, with automated reconciliation mechanisms ensuring that the infrastructure remains in the desired state. We will delve deeper into the topics of IaC and GitOps in the upcoming sections of this chapter, using practical examples.

Now, it's time to begin configuring our environment.

Setting up Terraform in a GitOps workflow

Having established the foundational concepts of IaC and GitOps, we now understand how the integration of Terraform and Flux CD can *transform infrastructure management*. With this robust groundwork in place, we will guide you through configuring Terraform to work seamlessly with Flux CD, setting the stage for a cohesive and automated infrastructure deployment process that aligns with modern cloud-native practices. Terraform was introduced and briefly discussed in *Chapter 4* and further explored in the context of *Kubernetes deployments with Azure DevOps and AWS CodePipeline* in *Chapter 9*. For the remainder of this chapter, we will assume that the reader has a solid understanding of Terraform and that it is already properly set up locally. For more information on what Terraform is and how to install it, please visit the following link: `https://developer.hashicorp.com/terraform/tutorials/aws-get-started/install-cli`. To check that Terraform is correctly installed, in a new terminal, type the following command:

```
$ terraform version
```

The output of the preceding command should be similar to this:

```
Terraform v1.7.5
```

In our example of setting up Terraform within a GitOps workflow, we'll use Flux to monitor changes in a Git repository containing Terraform scripts. Flux will then apply these changes to manage Azure cloud resources, specifically targeting a designated resource group, a virtual network, and a subnet.

To successfully implement this setup, you must meet the following prerequisites:

- **A functional Kubernetes cluster**: For this tutorial, we'll use `minikube` installed locally. Alternatively, you could choose a managed cluster environment such as **Azure Kubernetes Service** (**AKS**) or **Amazon Elastic Kubernetes Service** (**EKS**).
- **Helm and kubectl**: Ensure both tools are installed and configured correctly.

- **GitHub account**: While this example uses GitHub, Flux bootstrap supports other repositories as well. For additional details, refer to the Flux documentation at `https://fluxcd.io/flux/installation/#bootstrap-providers`.
- **GitHub personal access token (PAT)**: This is necessary for private repositories or for operations that require authentication. More information about PATs is available at `https://docs.github.com/en/authentication/keeping-your-account-and-data-secure/managing-your-personal-access-tokens`.
- **An Azure account**: You may also use an Azure account, based on your preference (please note that the examples in this and the upcoming sections are specifically designed for Azure).

Before delving deeper into the setup of the workflow, the next section will discuss what Tofu Controller is and why it is the engine and core of the reconciliation process.

Tofu Controller (formerly Weave TF-Controller)

Tofu Controller (refer to `https://github.com/flux-iac/tofu-controller`), formerly known as **Weave TF-Controller**, serves as a pivotal element within the Flux ecosystem, enabling the reconciliation of **OpenTofu** and Terraform resources in a GitOps manner.

OpenTofu

OpenTofu, formerly known as OpenTF, is an open source, community-driven fork of Terraform managed by the Linux Foundation. Designed to enhance the management of IaC within cloud-native environments, OpenTofu seamlessly integrates with existing GitOps workflows, enabling users to declaratively manage their infrastructure using tools such as Terraform. Both OpenTofu and Terraform use declarative **HashiCorp Configuration Language** (**HCL**) syntax for provisioning infrastructure but differ in their licensing: Terraform now operates under a **Business Source License** (**BSL**), while OpenTofu is released under the open source **Mozilla Public License** (**MPL**). OpenTofu's primary goal is to bridge the gap between complex infrastructure setups and the scalable, automated management capabilities required by modern cloud environments. By leveraging OpenTofu, organizations can efficiently apply GitOps principles to their infrastructure, ensuring consistency, reliability, and swift deployment of cloud resources.

With Tofu Controller, organizations can adopt GitOps practices for their infrastructure at a pace that suits their operational needs and readiness—meaning that there is no requirement to convert all resources to GitOps immediately.

Flexibility and pace emphasizing the **at-your-own-pace** approach, Tofu Controller allows for incremental adoption of GitOps. This flexibility is crucial for organizations that may want to gradually shift their operations without the need for a full-scale immediate transformation. It enables teams to manage changes and adaptations more comfortably, ensuring that each step toward GitOps can be thoroughly planned and implemented.

Tofu Controller supports various GitOps models, each catering to different needs and stages of infrastructure management:

- **GitOps Automation Model**: This model provides a comprehensive approach to GitOps, encompassing everything from provisioning to enforcement. For instance, it could manage the entire lifecycle of an AKS/EKS cluster via GitOps, automating numerous underlying processes.
- **Hybrid GitOps Automation Model**: Ideal for environments with pre-existing resources, this model allows selective GitOps adoption. For example, if an organization already has an AKS/EKS cluster, it can choose GitOps if only certain components such as the node group or security group integrate new capabilities while maintaining existing configurations.
- **State Enforcement Model**: Ensures that the current infrastructure state matches the Terraform state file (`TFSTATE`), using GitOps principles to enforce the desired state without making additional changes and correcting any deviations to maintain consistency and reliability.
- **Drift Detection Model**: Monitors for configuration drifts between the actual state and the Terraform state file, detecting changes not reflected in the `TFSTATE` file and allowing organizations to proactively address and manage these changes.

For the current example and for the upcoming sections, we will utilize the **GitOps Automation** model previously mentioned.

Getting started with the setup

Now that we have introduced the main topics and outlined the necessary requirements, we have reached the hands-on part of the workflow setup. To complete the first part of this setup, we need to complete the following numbered steps:

1. **Create a dedicated minikube cluster for Flux**: To create a `minikube` cluster dedicated to Flux setup, in a new terminal, type the following command:

   ```
   $ minikube start --profile flux-cluster
   ```

   After a few minutes, the output should be like the following:

   ```
   😄  [flux-cluster] minikube v1.33.0 on Ubuntu 22.04 (amd64)
   ✨  Automatically selected the docker driver
   📌  Using Docker driver with root privileges
   👍  Starting "flux-cluster" primary control-plane node in "flux-cluster" cluster
   …
   🌟  Enabled addons: storage-provisioner, default-storageclass
   🏄  Done! kubectl is now configured to use "flux-cluster" cluster and "default" namespace by default
   ```

   This indicates that your cluster is ready to be used.

2. **Install the Flux CLI**:

    I. To set up the Flux CLI, type the following command:

```
$ curl -s https://fluxcd.io/install.sh | sudo bash
```

For more information or troubleshooting, please refer to the official Flux documentation at `https://fluxcd.io/flux/installation/`.

    II. To verify that the Flux CLI is correctly installed, type the following command:

```
$ flux –version
```

The output of the preceding command should look like this:

```
flux version 2.2.3
```

3. **Create a GitHub repository**: For this walkthrough, I've created a GitHub repository named `gitops-terraform-workflow`, but feel free to create a repository according to your preferences. We will use a private repository for this setup. To create a GitHub account, please refer to the *GitHub account* section in *Chapter 3* of this book. After creating the GitHub repository, clone it locally to your laptop. For the Git commands, please refer to the *Basics of Git* section in *Chapter 3* of this book.

4. **Flux bootstrap for GitHub**:

    I. The `flux bootstrap` GitHub command installs Flux controllers on a Kubernetes cluster, enabling synchronization with a GitHub repository. This setup involves committing Flux manifests to the repository and configuring Flux for self-updates from Git. Once executed, all operations on the cluster, including Flux updates, can be managed via `git push`, removing the need for direct cluster interaction. To interact with the GitHub API, the command requires a GitHub PAT with administrative permissions, which can be set as an environment variable:

```
$ export GITHUB_TOKEN=[GITHUB_PAT_TOKEN]
```

    II. Now, you can run the bootstrap for a repository on your personal GitHub account by entering the following command:

```
    flux bootstrap github \
      --token-auth \
      --owner=[GITHUB_ACCOUNT] \
      --repository=gitops-terraform-workflow \
      --branch=main \
      --path=clusters/local \
      --personal
```

The execution of the command should take a few minutes, and a successful output should look like the following:

```
► connecting to github.com
► cloning branch "main" from Git repository "https://github.
com/pietrolibro/gitops-terraform-workflow.git"
…
► confirming components are healthy
✓ helm-controller: deployment ready
✓ kustomize-controller: deployment ready
✓ notification-controller: deployment ready
✓ source-controller: deployment ready
✓ all components are healthy
```

III. By running the following `kubectl` command, you should notice that a `flux-system` namespace has been created:

```
$ kubectl get namespaces
NAME              STATUS   AGE
default           Active   26m
flux-system       Active   3m30s
kube-node-lease   Active   26m
kube-public       Active   26m
kube-system       Active   26m
```

5. **Install the TF-Controller**: Now, we are ready to install the TF-Controller previously mentioned:

I. Enter the following command:

```
$ kubectl apply -f https://raw.githubusercontent.com/flux-iac/
tofu-controller/main/docs/release.yaml
```

II. Now, we are ready to install the previously mentioned Tofu Controller. Enter the following command:

```
helmrepository.source.toolkit.fluxcd.io/tf-controller created
helmrelease.helm.toolkit.fluxcd.io/tf-controller created
```

6. **Create an Azure service principal (SP)**: To provision Azure resources into a specific Azure subscription, we need to create an SP with the `Contributor` role. To do this, type the following commands:

```
subscriptionId="$(az account list --query "[?name=='AZURE_
SUBSCRIPTION_NAME'].id" --output tsv)"
az account set --subscription=$subscriptionId
az ad sp create-for-rbac --role="Contributor" --scopes="/
subscriptions/$subscriptionId" --name gitops-terraform-sp
```

The output contains sensitive credentials that you must protect. Ensure that you do not include these credentials in your code or check them into your source control. For more information on managing Azure SPs, visit `https://aka.ms/azadsp-cli`:

```
{
  "appId": "application-id",
  "displayName": "gitops-terraform-sp",
  "password": "generated-password",
  "tenant": "tenant-uid"
}
```

The values correspond to the Terraform variables as follows: `appId` serves as the `client_id` variable, `password` is used as the `client_secret` variable, and `tenant` is the `tenant_id` variable as defined previously.

7. **Define Kubernetes Secrets**:

   I. We are running Terraform in a Kubernetes cluster using Flux to automate Terraform operations, so we need to use Kubernetes Secrets to store Azure credentials and the PAT. These secrets are then mounted into your Pod where Terraform executes. To define a secret for the GitHub repository, please refer to the `github-repository-secret.yaml` file located in the definitions folder of the chapter's repository. The content of the file is as follows:

```
apiVersion: v1
kind: Secret
metadata:
  name: github-repository-secret
  namespace: flux-system
type: Opaque
data:
  username: <github_accountname-base64>
  password: <github_pat-base64>
```

   II. The username and password need to be `base64` encoded. To encode these using the command line, you can use the following command: `echo -n 'your-string' | base64`. After encoding, apply the secret using `kubectl` in the usual manner:

```
$ kubectl apply -f github-repository-secret.yaml
```

   III. Now, we need to create a Kubernetes secret for the Azure credentials of the SP we previously created. Use the following command to do so:

```
$ kubectl create secret generic azure-creds \
    --from-literal=ARM_SUBSCRIPTION_ID='SP_SUB_ID' \
    --from-literal=ARM_TENANT_ID='SP_TENANT_ID' \
    --from-literal=ARM_CLIENT_ID='SP_APPID' \
```

```
    --from-literal=ARM_CLIENT_SECRET='SP_PASSWORD' \
    --namespace flux-system
```

IV. To verify the creation of the secret, use the following command:

```
$ kubectl get secrets -n flux-system
NAME        TYPE                        DATA   AGE
azure-creds                 Opaque      4      16s
flux-system                 Opaque      2      34m
github-repository-secret  Opaque      2      25s
…
```

Kubernetes Secret

A Kubernetes Secret is a resource used in Kubernetes to hold a small amount of sensitive data such as passwords, tokens, or keys. This data is stored in `base64`-encoded format and can be used by pods to handle operations that require confidentiality and security, helping to prevent sensitive information from being exposed in your application code or logs. The term *opaque* in the context of Kubernetes Secrets refers to the default type of Secret. It is used when the specific type of Secret (such as a service account token or Docker configuration) doesn't need to be indicated. Essentially, *opaque* indicates that the content of the Secret is arbitrary and not structured to represent any specific type of confidential data.

8. Now, we need to link the GitHub repository with the GitHub repository secret using the `github-repository-definition.yaml` file located in the definitions folder (the `flux-gitops-definitions` folder defined in the repository accompanying this chapter). Apply it using the usual command:

```
$ kubectl apply -f github-repository-definition.yaml
gitrepository.source.toolkit.fluxcd.io/gitops-terraform-
repository created
```

9. **Docker image for the TF-Controller**: Now, we need to create a custom Docker image to run the TF-Controller. The Dockerfile definition is contained in the Docker directory. For Docker commands, please refer to *Publishing the image to a container registry* section in *Chapter 2*. To build a custom Docker image, use the following command:

```
$ docker build -f ./Docker/dockerfile -t pietrolibro/gitops-
terraform:tf_az_cli_1_1 .
```

10. Then, you need to publish it in a repository reachable from your Kubernetes cluster. To push the command, after logging in to Docker, use the following command:

```
$ docker push pietrolibro/gitops-terraform:tf_az_cli_1_1
```

11. **Create a Terraform script**: Please refer to the Terraform script named `main.tf` in the `iac/azure/vnet` subfolder of the repository accompanying this chapter. The Terraform script provisions a set of network resources in Microsoft Azure. It configures the Azure provider and defines a resource group named `gitops-terraform-rg` in the Switzerland North region, serving as a container for related Azure resources. The script also establishes a virtual network called `gitops-terraform-vnet` within this resource group, with an address space of `10.0.0.0/16`. Additionally, it creates a default subnet within this virtual network, allocated with an address range of `10.0.0.0/24`. This setup structures the Azure cloud infrastructure effectively, facilitating the management and scalability of network components within the specified region.

At this point, we can commit and push the code to the GitHub repository, concluding the first part of our journey of setting up Terraform within our GitOps workflow. The next stop on our journey, which will be described in the upcoming sections, focuses on enabling CD in Kubernetes with Flux.

Exploring Flux CD – enabling CD in Kubernetes

Flux CD automates the application of configurations and updates to a Kubernetes cluster based on changes pushed to a Git repository. This approach not only ensures consistency across environments but also enhances security and auditability by maintaining a clear, version-controlled history of all deployments. Flux continuously monitors a designated Git repository for changes to Kubernetes manifests, Helm charts, or configuration files. When a change is detected, Flux automatically applies these updates to the corresponding Kubernetes cluster. This automation is comprehensive, extending not only to applications but also to the Kubernetes infrastructure itself, enabling entire environments to be managed as code.

Key features of Flux CD include the following:

- **Automated synchronization**: Flux regularly checks the Git repository for changes and synchronizes the state of the cluster accordingly
- **Immutable deployments**: By treating infrastructure as immutable, Flux discourages manual interventions and promotes reproducibility
- **Security and compliance**: Changes are pulled from Git, ensuring that only approved changes are deployed, thereby enhancing security and compliance

It's time to return to our journey, which we paused at the end of the previous section. Now, we need to configure the Flux instance to track its respective directory or branch in the Git repository. This is where the beauty of the Terraform Controller comes into play—it does all the hard work for you.

We need to create a YAML file (refer to the `azvnet-gitops-automation.yaml` file defined in the repository for this chapter) like so:

```
apiVersion: infra.contrib.fluxcd.io/v1alpha2
kind: Terraform
metadata:
  name: gitops-terraform-automation
  namespace: flux-system
spec:
  interval: 1m
  approvePlan: auto
  destroyResourcesOnDeletion: true
  path: ./iac/azure/vnet
  sourceRef:
…
  runnerPodTemplate:
    spec:
      env:
        - name: ARM_SUBSCRIPTION_ID
          valueFrom:
            secretKeyRef:
              name: azure-creds
              key: ARM_SUBSCRIPTION_ID
        …
```

This YAML file defines a `Terraform` resource for use within the Flux CD framework, specifically designed to manage IaC deployments automatically. The configuration details the API version and specifies that the resource type is `Terraform`, indicating its role in managing and applying Terraform configurations through Flux.

The resource is named `gitops-terraform-automation` and is located within the `flux-system` namespace, which is typically designated for Flux CD operations. The specifications include an interval setting of 1 minute, indicating how frequently Flux checks the Git repository for changes in the designated directory. It is configured to automatically approve and apply changes (`approvePlan: auto`) and to destroy all managed resources if this Terraform configuration is deleted from Kubernetes (`destroyResourcesOnDeletion: true`). The `./iac/azure/vnet` path specifies where in the Git repository the Terraform configurations are located, which Flux will monitor.

The `sourceRef` instance points to a Git repository named `flux-system`, identifying the specific repository Flux should monitor. This setup allows Flux to automatically manage updates and maintain the environment as specified in the repository.

The `runnerPodTemplate` section configures the environment variables for the pod that executes the Terraform commands, securing access to Azure services. Variables such as `ARM_SUBSCRIPTION_ID`, `ARM_CLIENT_ID`, `ARM_CLIENT_SECRET`, and `ARM_TENANT_ID` are essential for Azure interactions and are pulled securely from a Kubernetes secret named `azure-creds`. This ensures that sensitive credentials are managed securely and only made available to the necessary processes.

By leveraging this configuration, Flux CD automates the deployment and management of Azure virtual networks defined in the Terraform configurations at the specified path in the repository. The automation encompasses monitoring for changes, applying updates, and ensuring the environment remains consistent with the declared state in the repository. This setup epitomizes the principles of GitOps by using Git as a **single source of truth** (**SSOT**) and automating the reconciliation process between the desired state defined in Git and the actual state of the infrastructure in Azure.

To make the magic happen and bring the configuration to life, we need to apply it using `kubectl`:

```
$ kubectl apply -f ./definitions/azvnet-gitops-automation.yaml
terraform.infra.contrib.fluxcd.io/gitops-terraform-automation created
```

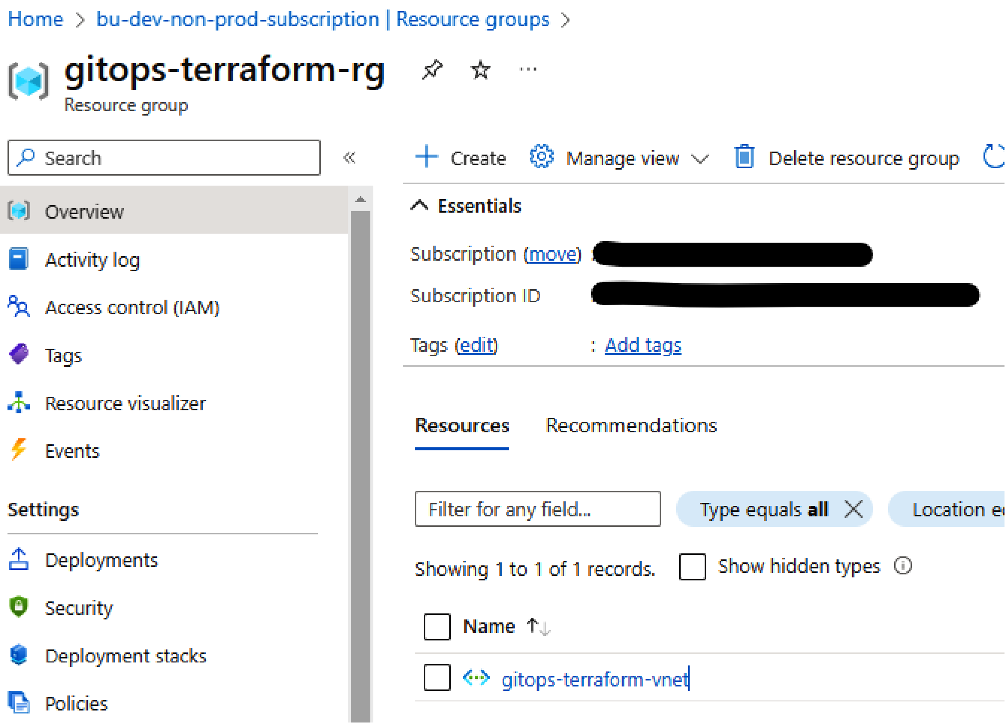
To monitor and verify the reconciliation process initiated by Flux CD, you can use the `kubectl` command to query the status of resources managed by Flux. Here's how to check the reconciliation process:

```
$ kubectl get terraforms.infra.contrib.fluxcd.io -n flux-system  -w
```

This command displays the current status of the Terraform configuration:

```
NAME         READY     STATUS                         AGE
gitops-ter… Unknown   Reconciliation in progress   9s
gitops-ter… Unknown   Initializing                 15s
gitops-ter… Unknown   Terraform Planning           22s
gitops-ter… Unknown   Plan generated               30s
gitops-ter… Unknown   Applying                     30s
gitops-ter… Unknown   Applying                     30s
gitops-ter… Unknown   Applied successfully: main@
sha1:2eeaefba687fbd3d4caea404fc332ee7c5d8b144   43s
gitops-ter… True      Applied successfully: main@
sha1:2eeaefba687fbd3d4caea404fc332ee7c5d8b144   44s
```

As indicated by the output of the previous command, the Terraform script has been successfully applied at a certain point. For further confirmation, we can open the Azure portal and visually inspect the provisioned infrastructure, as illustrated in *Figure 10.2*:



Subscription (move) [redacted]

Subscription ID [redacted]

Tags (edit) : Add tags

Showing 1 to 1 of 1 records.

☐ Name

☐ gitops-terraform-vnet

Figure 10.2 – Azure resources provisioned by Terraform and GitOps

Fantastic work! You have successfully integrated IaC using Terraform along with GitOps principles through Flux CD. By adopting this approach, you have laid a robust foundation for managing your cloud resources in a way that promotes consistency, reduces errors, and streamlines operations.

Combining Terraform and Flux CD for enhanced automation

Combining Terraform and Flux CD for enhanced automation offers significant advantages in managing cloud infrastructure efficiently and resiliently. Here's how this integration can be advantageous in various practical scenarios.

Providing new infrastructure by updating Terraform files

One of the standout benefits of integrating Terraform with Flux CD is the simplicity with which new infrastructure components can be added. For example, suppose you need to add a new subnet to an existing Azure virtual network. By merely updating the Terraform file to include the new subnet definition and committing this change to your repository, Flux CD automatically detects the update and applies it. This seamless process eliminates the need for manual intervention in the cloud environment, thereby reducing the potential for human error and accelerating deployment times. Automation ensures that infrastructure adjustments, such as scaling out to accommodate growth, are handled promptly and accurately.

We can immediately test this aspect by updating the `main.tf` file presented in *step 8* of the *Setting up Terraform in a GitOps workflow* section. Append the following code block to the end of the file:

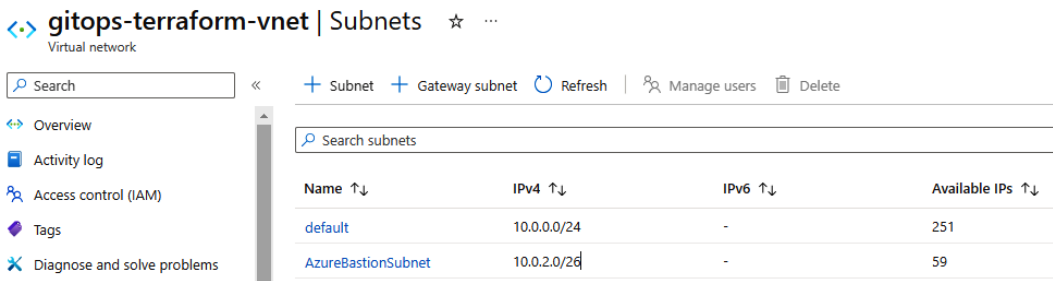
```
resource "azurerm_subnet" "azure_bastion_subnet" {
  name                 = «AzureBastionSubnet"
  resource_group_name  = azurerm_resource_group.gitops_terraform_
rg.name
  virtual_network_name = azurerm_virtual_network.gitops_terraform_
vnet.name
  address_prefixes     = [«10.0.1.0/26"]
}
```

This Terraform code block creates a subnet named `AzureBastionSubnet` within a specified Azure virtual network and resource group. It assigns the subnet an IP address range of `10.0.1.0/26`. The subnet resource is identified within Terraform as `azure_bastion_subnet`, and it references the names of both the virtual network and resource group from other resources defined in the Terraform configuration.

At this point, simply commit and push the code to trigger a reconciliation loop. You will then see the process in action, as illustrated next:

```
gitops-ter…   Unknown   Reconciliation in progress    3h12m
gitops-ter…   Unknown   Initializing                3h12m
gitops-ter…   Unknown   Terraform Planning            3h12m
gitops-ter…   Unknown   Plan generated                3h12m
gitops-ter…   Unknown   Applying                    3h12m
gitops-ter…   Unknown   Applied successfully: main@
sha1:a3b32ed48dda027b0c5e40a65c7be56e1cc7dd50       3h13m
gitops-ter…   True      Applied successfully: main@
sha1:a3b32ed48dda027b0c5e40a65c7be56e1cc7dd50       3h13m
```

Open the Azure portal to visually verify the results, as illustrated in *Figure 10.3*:



Name	IPv4	IPv6	Available IPs
default	10.0.0.0/24	-	251
AzureBastionSubnet	10.0.2.0/26	-	59

Figure 10.3 – After the reconciliation process, the gitops-terraform-vnet virtual network in the Azure portal contains two subnets

Enhanced disaster recovery capabilities

In the event of a disaster, the combination of Terraform and Flux CD significantly enhances recovery capabilities. Terraform's ability to describe the complete IaC means that entire environments can be replicated swiftly and accurately. Flux CD plays a crucial role by continuously ensuring that the state of the cloud environment matches the configurations defined in your Git repository. If a disaster strikes, you can quickly redeploy your infrastructure to a known good state as defined in the repository, drastically reducing downtime and the complexity often associated with recovery processes.

Creating and managing multi-stage environments

When managing complex deployments that involve multiple stages, such as development, staging, and production, the division of the repository becomes essential. With Terraform and Flux CD, you can split your repository appropriately to manage these environments separately yet consistently. Each environment can have its configuration files and Terraform plans, which Flux CD can manage independently. This setup not only keeps your deployments organized but also enforces consistency and isolation between environments. Changes can be tested in development or staging environments through automated pipelines and only promoted to production once verified, ensuring stability and reliability in your production deployments.

As seen so far, this strategic integration of Terraform and Flux CD not only streamlines the operational aspects of cloud infrastructure management but also enhances strategic capabilities such as quick scaling, robust **disaster recovery** (**DR**), and effective multi-environment handling. With Flux CD, updates and changes are applied systematically and predictably, reducing the likelihood of human error and the overhead associated with manual deployments.

In the next section, we will explore how Terraform and Flux CD utilize version control and automation to enhance infrastructure management, ensuring transparency, collaboration, and security throughout the deployment process.

Version control and automation with Terraform and Flux CD

Terraform uses version control to manage the state of your infrastructure, which allows for tracking changes, auditing, and collaborating on infrastructure development. By storing Terraform configurations in version-controlled repositories, teams can review changes through pull requests, maintain a history of configurations, and revert to previous states when necessary. This process enhances transparency and collaboration among team members, fostering a more controlled and secure environment for deploying infrastructure.

Security and best practices with Terraform and Flux CD

When combining Terraform with Flux CD, security is significantly enhanced by the inherent design of GitOps and the nature of declarative infrastructure management. All changes to the infrastructure are committed to version control, which means every change is auditable and traceable.

To enhance security further, we can deploy the following:

- **Access control**: Implement strict access controls on your Git repositories. Use branch protection rules to ensure that changes are reviewed and approved before they are merged, preventing unauthorized modifications to your infrastructure.
- **Secrets management**: Never store secrets directly in your Git repositories. Instead, leverage Kubernetes Secrets or integrate with a secrets management tool such as HashiCorp Vault or Azure Key Vault. Flux CD can securely pull secrets and inject them into the deployment pipeline.
- **Automated compliance scans**: Integrate compliance scanning tools into your CI/CD pipeline to automatically check Terraform plans for compliance with security policies and best practices before they are applied.

Best practices for configuration and maintenance

Here are some best practices for configuration and maintenance:

- **Immutability**: Treat infrastructure as immutable. Rebuild infrastructure from scratch frequently to ensure that the actual state aligns with the state defined in Terraform configurations.
- **Code review and pair programming**: Enforce code reviews for all infrastructure changes. Pair programming can also be used for critical changes, involving more than one set of eyes before changes go live.
- **CI**: Integrate CI processes to test infrastructure changes automatically. This includes testing Terraform plans and applying them to a non-production environment before they affect your production infrastructure.

Best practices for managing multi-environment configurations

Managing multiple environments (dev, staging, production) effectively requires a well-organized repository structure.

Here are some best practices:

- **Environment branching**: Maintains a single repository, but uses different branches for each environment, merging changes from development branches to production branches through a controlled process.
- **Parameterization**: Use variables and outputs to customize Terraform configurations for different environments. Avoid hardcoding values directly into Terraform files.
- **Environment parity**: Keep your development, staging, and production environments as similar as possible. This reduces the chances of encountering unexpected behaviors in production that weren't evident in dev or staging. The optimal approach to managing multi-environment configurations hinges on several factors, such as organization size, project complexity, regulatory demands, and the balancing act between security and operational efficiency. For organizations in highly regulated industries, such as finance or healthcare, where security is paramount, using separate repositories for each environment might be advisable to enhance security by isolating production settings from development and testing environments. This method limits access to sensitive configurations and minimizes the risk of inadvertent changes. Alternatively, if rapid development and deployment are critical, a single repository with separate directories for each environment can simplify the management of changes across environments, though it necessitates strict access controls and vigilant monitoring to protect production configurations. For those seeking a balance between security and simplicity, a hybrid approach could be suitable, involving a single repository for all non-sensitive environments and a separate one for production. Regardless of the chosen strategy, implementing robust version control practices, clear access controls, and automated deployment pipelines will be essential to maintain efficiency and safeguard the integrity of environments.

Git workflow strategies

To effectively manage your infrastructure and streamline your development workflow, it's essential to adopt best practices that align with your team's dynamics and project requirements. Consider the following strategies:

- **Trunk-based development versus Git Flow**: Choose a Git workflow that suits your team's needs. Trunk-based development encourages shorter-lived branches and might be more suitable for fast-paced environments, while Git Flow can provide more control through designated branches for features, releases, and hotfixes.

- **Automated testing and promotion**: Use automated testing to validate changes in lower environments. Only promote changes to the next environment once they pass all tests. Automate this promotion using Flux CD, which can monitor different branches or folders and apply changes to the respective environments.

Now that we have a comprehensive understanding of Terraform, GitOps, and Flux, along with best practices, we can combine all these elements in a practical example. This example will focus on managing a multi-environment setup, which we will describe in the next section.

Multi-environment management with Terraform and Flux CD

In the final section of this book, you'll explore a detailed application of the concepts we've discussed, using a Kubernetes cluster (`minikube`) to manage a multi-environment setup with Flux CD for IaC. This real-world example will demonstrate how to effectively deploy separate environments for **development** (`dev`), **staging** (`staging`), and **production** (`prod`). *Figure 10.4* illustrates the described example:

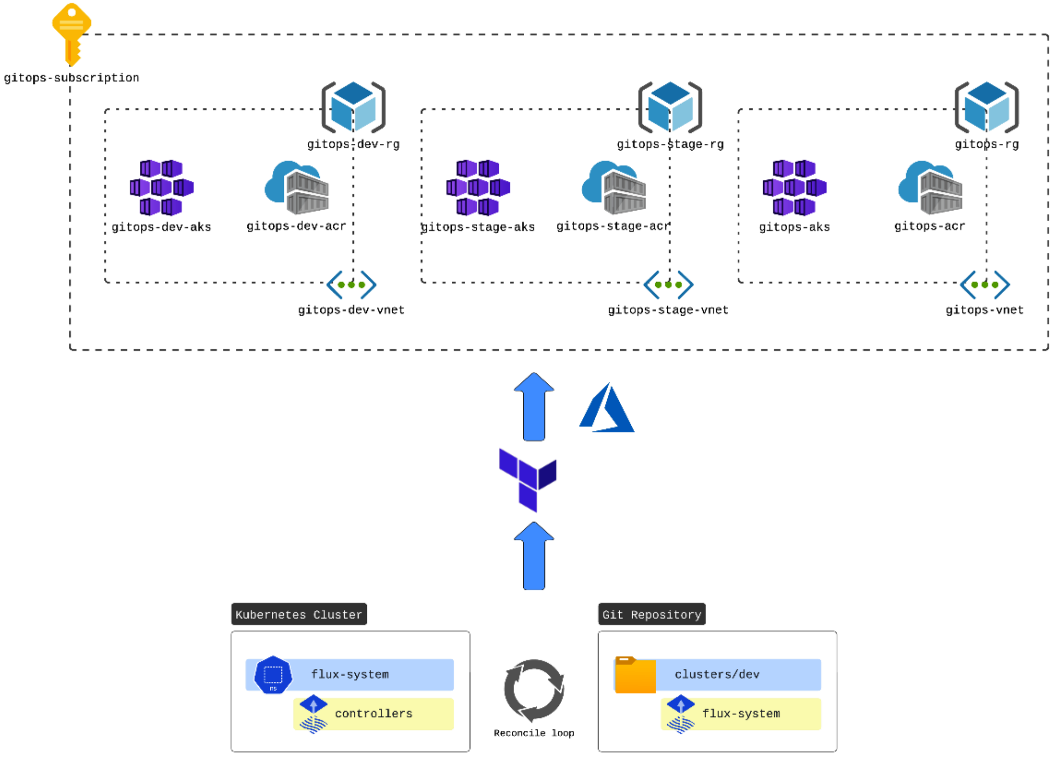


Figure 10.4 – Example of multi-environment management with Terraform and Flux CD

GitHub will serve as the VCS, and the directory hierarchy will be organized as follows:

```
|---clusters/mgmt
|---flux-gitops-definitions
|---multi-env
     |---iac
          |---azure
               |---base
               |---dev
               |---staging
               |---prod
```

In this structure, we have the following:

- `clusters/mgmt`: Contains the Flux configuration files installed by the bootstrap for the management cluster (`minikube`, running locally).
- `multi-env/iac/azure`: Contains subdirectories for the different environments (`dev`, `staging`, and `prod`), each intended for managing infrastructure specific to that environment using Terraform scripts.
- `Base`: Contains the main Terraform files (`main.tf` and `variables.tf`), which define the Azure resources (such as AKS, virtual network, **Azure Container Registry** (**ACR**), and so on) from an infrastructure perspective. These files are copied into the corresponding environment subfolder as needed to tailor the infrastructure setup for each specific environment.

The `flux-gitops-definitions` directory contains various manifest YAML files essential for configuring the GitHub repository and secret. This setup involves the following:

- **GitHub repository configuration**: YAML files in this directory help link the Flux installation with the GitHub repository, specifying where Flux should listen for updates and which branch to monitor.
- **Secrets management**: Includes the setup for Kubernetes secrets that store sensitive information, such as GitHub access tokens or cloud provider credentials. These secrets ensure that Flux can securely access the repository and interact with other services as needed.
- **Resource definitions**: Manifests that define how resources should be applied, rolled out, and managed across different environments.

For the following example, we will continue using the same GitHub repository as before (although you are welcome to create a new one if you prefer). Additionally, we will initiate a new `minikube` cluster:

1. To do this, start by opening a new terminal window and enter the following command:

   ```
   $ minikube start --profile mgmt-cluster
   ```

2. While waiting for the competition to start, we'll need to export `GITHUB_TOKEN`. This can be done by entering the following command in your terminal:

```
export GITHUB_TOKEN=ghp_XYZ
```

3. Now that `GITHUB_TOKEN` is set, you can proceed to install Flux on your cluster using the bootstrap command. Execute the following in your terminal:

```
flux bootstrap github \
      --token-auth \
      --context=mgmt-cluster \
      --owner=[GITHUB_ACCOUNT] \
      --repository=[GITHUB_REPOSITORY] \
      --branch=develop \
      --path=clusters/mgmt \
      --personal
```

Once the command has finished executing, you should see an output similar to the following:

```
► connecting to github.com
► cloning branch "develop" from Git repository "https://github.
com/...
✓ cloned repository
► generating component manifests
…
✓ kustomize-controller: deployment ready
✓ notification-controller: deployment ready
✓ source-controller: deployment ready
✓ all components are healthy
```

4. Now that Flux is successfully bootstrapped, the next step is to install the TF-Controller. Here's how to proceed:

```
$ kubectl apply -f https://raw.githubusercontent.com/flux-iac/
tofu-controller/main/docs/release.yaml
```

5. Once the command is executed, you should verify that the TF-Controller has been installed correctly and is running. You can check the status of the pods in the namespace where the TF-Controller is installed, typically in the `flux-system` namespace or a specific namespace designated for it:

```
$ kubectl get pods -n flux-system
NAME                        READY   STATUS    RESTARTS   AGE
helm-controller…            1/1     Running   0          4m47s
kustomize-contr…              1/1     Running   0          4m47s
notification-co…            1/1     Running   0          4m47s
source-controll…            1/1     Running   0          4m47s
```

Look for pods related to the TF-Controller and ensure they are in the `Running` state.

6. Since we are using a new `minikube` cluster (or alternative cluster) and our goal is to provision resources on Azure, we need to create the corresponding secret:

```
$ kubectl create secret generic azure-creds \
    --from-literal=ARM_SUBSCRIPTION_ID='SP_SUB_ID' \
    --from-literal=ARM_TENANT_ID='SP_TENANT_ID' \
    --from-literal=ARM_CLIENT_ID='SP_APPID' \
    --from-literal=ARM_CLIENT_SECRET='SP_PASSWORD' \
    --namespace flux-system
```

7. To set up a multi-environment infrastructure using Terraform, you'll need to create a new (empty) `main.tf` file in each environment-specific subdirectory (`dev`, `staging`, `prod`) within the `multi-env/iac/azure` main directory. Here's how to structure the `main.tf` file for each environment:

```
provider "azurerm" {
  features {}
}
```

8. Now, change your working directory to where `flux-gitops-definitions` is located. This directory should contain your YAML files for the GitHub repository configuration (`github-repository-definition.yaml`) and the secret (`github-repository-secret.yaml`). First, apply the *GitHub repository secret*, which contains the credentials that Flux CD will use to access your GitHub repository:

```
$ kubectl apply -f github-repository-secret.yaml
```

9. Next, apply the *GitHub repository definition*. This definition will link your Kubernetes setup with the GitHub repository, setting the groundwork for Flux CD to monitor changes and manage resources according to GitOps principles:

```
$ kubectl apply -f github-repository-definition.yaml
```

10. Before continuing, it's important to commit and push the changes we've made so far. Then, we need to apply the automation configurations for each environment by executing the following commands:

```
$ kubectl apply -f dev-iac-automation.yaml
$ kubectl apply -f staging-iac-automation.yaml
$ kubectl apply -f prod-iac-automation.yaml
```

11. Afterward, check the reconciling process to ensure that the configurations are being applied correctly:

```
$ kubectl get terraforms.infra.contrib.fluxcd.io -n flux-system
-w
NAME                              READY
STATUS
                              AGE
dev-cluster-tf-automation         True
No drift: develop@sha1:c93...    3m48s
prod-cluster-tf-automation        True
No drift: develop@sha1:c93...    3m31s
staging-cluster-tf-automation     True
No drift: develop@sha1:c93...    3m37s
dev-cluster-tf-automation         Unknown
Reconciliation in progress
                       3m48s
staging-cluster-tf-automation     Unknown
Reconciliation in progress
                       3m38s
prod-cluster-tf-automation        Unknown
Reconciliation in progress
                       3m33s
```

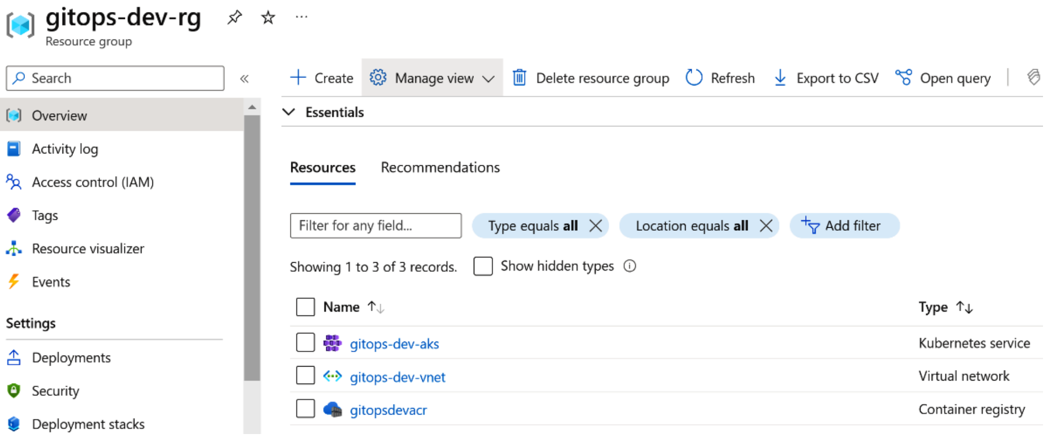
12. At this point, the reconciliation process is in progress, but no resources will be provisioned because the `main.tf` file in each directory does not define any resources. To address this, copy the `main.tf` and `variables.tf` files from the `base` directory to each subdirectory corresponding to the specific environments. Then, for each environment, open the `variables.tf` file and update it according to the needs of the `dev`, `staging`, and `prod` environments:

```
variable "environment" {
  …
  default      = "dev"
}
variable "location" {
  …
}
variable "rg" {
  …
  default      = «gitops-dev-rg"
}
```

13. Now, it's time to commit and push the updated code to your Git repository. After doing so, you can observe the reconciliation process to see how Flux CD responds to the changes:

```
$ kubectl get terraforms.infra.contrib.fluxcd.io -n flux-system
-w
```

This procedure provides real-time feedback on how Flux CD manages and applies updates from your Git repository. The reconciliation process may take some time to fully provision the resources defined in the Terraform scripts. The results are illustrated in *Figure 10.5*:



☐ Name	Type
☐ gitops-dev-aks	Kubernetes service
☐ gitops-dev-vnet	Virtual network
☐ gitopsdevacr	Container registry

Figure 10.5 – Resources provisioned by the GitOps automation reconciliation process

Well done! Now, you can try making changes to the infrastructure code in one of the environments and see how the GitOps automation updates and syncs the infrastructure with your repository. What we've explored is just a glimpse of using GitOps for managing multi-environment setups. We haven't covered using pull requests or installing applications yet—these topics will be addressed in the next chapter.

Summary

In this chapter, we explored the dynamic synergy between Terraform and Flux CD, showcasing how these tools bolster infrastructure automation using GitOps and IaC principles. The chapter introduced Terraform for defining and provisioning cloud infrastructure, alongside Flux CD, which automates the deployment process by continuously syncing changes from Git repositories to Kubernetes clusters. It thoroughly covered the fundamentals of integrating Terraform within a GitOps workflow, ensuring seamless management and scaling of cloud resources. Special attention was given to best practices for maintaining multiple environments such as development, QA, and staging, emphasizing the importance of environment-specific configurations to maintain consistency, reduce errors, and streamline operations across various cloud setups. Practical examples throughout the chapter demonstrated how to utilize these tools to effectively manage complex deployments, with a strong focus on the critical roles of version control and automated reconciliation.

Building on the foundational concepts and best practices discussed, the next chapter will delve deeper into practical applications and real-world scenarios, showcasing how these strategies can be effectively implemented to optimize cloud infrastructure management.

11

Deploying Real-World Projects with GitOps on Kubernetes

In this chapter, you will embark on a practical journey that bridges the gap between theoretical knowledge and real-world knowledge application. As you delve into the intricate process of setting up a GitOps and Kubernetes-based development environment, you will gain firsthand experience in designing, developing, and deploying an application within this innovative framework. Through detailed guidance on architectural design, **Continuous Integration and Continuous Delivery** (**CI/CD**) processes, application scaling, and security, this chapter aims to equip you with the essential skills and insights needed to implement these cutting-edge technologies effectively in your projects. Whether you're looking to enhance your organizational capabilities or to refine your personal technical expertise, the comprehensive real-life example provided here will serve as an invaluable resource for anyone aspiring to master GitOps and Kubernetes in practical settings.

In this chapter, our focus will be on the following key areas:

- Establishing a GitOps and Kubernetes development environment
- Implementing CI/CD with GitOps
- Designing for scalability and efficiency
- Resource management and scalability
- Monitoring and securing your application

Technical requirements

This chapter builds on your existing knowledge of Git, Kubernetes, and GitOps tools such as **Argo CD** and Flux CD, which you acquired in earlier chapters. We will use an **Azure AKS** cluster deployed by Terraform using a GitHub workflow. Ensure that you have access to a Kubernetes setup and are familiar with CI/CD principles to fully benefit from the exercises.

All necessary code and resources are provided in the *Chapter 11* folder of our dedicated GitHub repository:

`https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`

Establishing a GitOps and Kubernetes development environment

Establishing a proper development environment is crucial for the successful implementation of GitOps practices. This environment serves as the backbone for both development and operations teams, enabling seamless integration and continuous delivery of applications. A well-configured development environment ensures that all changes to applications and infrastructure are version-controlled, traceable, and aligned with the declarative configurations stored in Git. This consistency between the development environment and production setups reduces the likelihood of errors and deployment failures, fostering a more reliable and robust delivery pipeline. By emphasizing the importance of a correct setup from the outset, teams can leverage GitOps to its fullest potential, ensuring that automated processes govern deployments and infrastructure management efficiently and effectively.

Installing and configuring Kubernetes for GitOps involves setting up your Kubernetes cluster in a way that integrates seamlessly with GitOps tools such as Flux CD (see the *Flux integration with Kubernetes* section in *Chapter 4*) or Argo CD (see the *Argo CD integration with Kubernetes* section in *Chapter 4*). What follows is a step-by-step guide that covers the setup process, ensuring that your Kubernetes environment is ready for a GitOps workflow:

1. Install a Kubernetes cluster and choose your environment. For learning and development, consider using **K3s** (refer to the *Exploring K3s as a lightweight Kubernetes distribution* section in *Chapter 2*) or **minikube**. Both are suitable for running Kubernetes locally on your machine. For production or more scalable environments, consider cloud solutions such as **Amazon EKS**, Azure AKS, or **Google GKE**. To install minikube, follow the official minikube installation guide at `https://minikube.sigs.k8s.io`. For deploying Kubernetes on cloud platforms, refer to the specific setup guides provided by the respective cloud providers. For the real-world scenario described in this chapter, we will use an AKS cluster.
2. Verify installation. Ensure that `kubectl`, the Kubernetes command-line tool, is installed and configured to communicate with your cluster. You can verify this by running the following:

   ```
   $ kubectl cluster-info
   ```

   This command should return the cluster details confirming that Kubernetes is up and running:

   ```
   Kubernetes control plane is running at https://127.0.0.1:32769
   CoreDNS is running at https://127.0.0.1:32769/api/v1/namespaces/
   kube-system/services/kube-dns:dns/proxy
   ```

3. Set up your namespace. It's good practice to create a dedicated namespace for your GitOps tools:

```
$ kubectl create namespace gitops
```

4. Set up permissions. Set up **Role-Based Access Control** (**RBAC**) rules to ensure that your GitOps tools have the necessary permissions to manage resources. Most GitOps tools have specific RBAC configurations outlined in their setup guides. We will see a concrete example of how to set up RBAC in the *Configuring Kubernetes RBAC for user and role management* section of this chapter.
5. Install your GitOps tool. Depending on your preference, select from tools such as Flux CD, Argo CD, Helm, or Kustomize. Each tool has unique strengths and supportive community backing. For more details about the mentioned GitOps tools, refer to the *Overview of popular GitOps tools* section in *Chapter 4*. Additionally, you can explore the *A deep dive into Helm and Kustomize*, *Argo CD integration with Kubernetes*, and *Flux CD integration with Kubernetes* sections of the same chapter.
6. Set up a Git repository. Configure the GitOps tool to track your Git repository where your Kubernetes manifests are stored. For guidance, refer to the *Kubernetes deployment with Azure DevOps* or the *Kubernetes deployment with AWS CodePipeline* section, both in *Chapter 4*. This setup process involves pointing the tool to your repository and specifying which branch and path to monitor for changes.
7. Validate and test. Start by deploying a simple application using your GitOps tool to confirm that changes in your Git repository automatically trigger deployments in your Kubernetes cluster. Monitor the deployment using the GitOps tool's dashboard or CLI to ensure that the application is deployed and running as expected. Test updates and rollbacks by modifying the application's manifest in your Git repository and noting whether the changes are automatically implemented.

Most of the pipeline points have already been covered in more detail in previous chapters. They will be revisited in the next section, where will see how to implement a real-world scenario for CI/CD with GitOps.

Implementing CI/CD with GitOps

To implement a real-world CI/CD GitOps scenario, we need an application that no longer uses mocked data but instead utilizes concrete data.

In this section, we will expose a backend service for a weather application that fetches data from a real weather service, such as **OpenWeatherMap** (`https://openweathermap.org/`), to the public internet.

Given that the requirements for setting up our GitOps environment (installing a Kubernetes cluster and choosing your environment, verifying installation, and installing your GitOps tool) have already been met in the previous section of this chapter, the next step is to create a new GitHub repository. For example, you might create `gitops-for-real-world`, with a directory named `Step-01`. This directory will be used to add the code and files for subsequent steps.

Before proceeding, you need to create a free account with the `OpenWeatherMap` service or another similar service of your choice. Services like these typically require a token to query their API, which is used for authentication and billing purposes. It's crucial to keep this token *confidential* and not share it. Please refer to the `OpenWeatherMap` documentation to create a new token. Soon, we will add this token as a **secret** in the Kubernetes cluster.

Final objective and implementation

To achieve our final objective, this section and the ones that follow will demonstrate the use of a **Python Flask application**, packaged as a **Docker image**. This image is built with a new tag at each commit and deployed on an Azure AKS cluster, which is provisioned automatically by the pipeline using Terraform for the **Infrastructure as Code** (**IaC**) component. Initially, the entire deployment chain—both IaC and the application—will be managed entirely by our GitHub CI/CD pipeline. Later, we will transition to using Argo CD for the deployment while keeping the CI processes within the GitHub workflow.

Ultimately, to test our service after it has been exposed to the public internet, we will perform **weather** requests for a specified city via the query string in a URL, such as `http://public-ip/weather?city=zurich`. The response will be in JSON format, which can be rendered directly in the browser or with tools such as **curl**.

Our pipeline will be developed as a GitHub workflow and will be composed as illustrated in *Figure 11.1*.

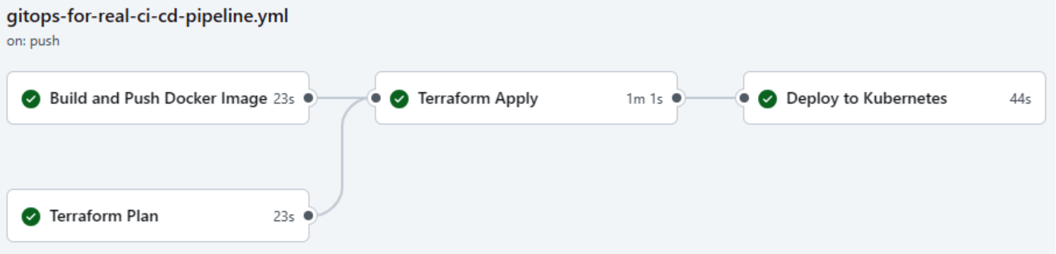


Figure 11.1 – A GitHub workflow pipeline

CI/CD pipeline using GitHub Actions and Terraform

The pipeline in *Figure 11.1* leverages Terraform for infrastructure management and deploys a Dockerized application to an AKS cluster, providing a practical example of modern DevOps practices. The code of the pipeline is too long to be explicitly added as content in this chapter. What follows are some important aspects that should be considered for a better understanding of that. The workflow description is contained in the `gitops-for-real-ci-cd-pipeline.yml` file in the `.github/workflows/` directory of the repository accompanying this chapter.

Workflow trigger conditions

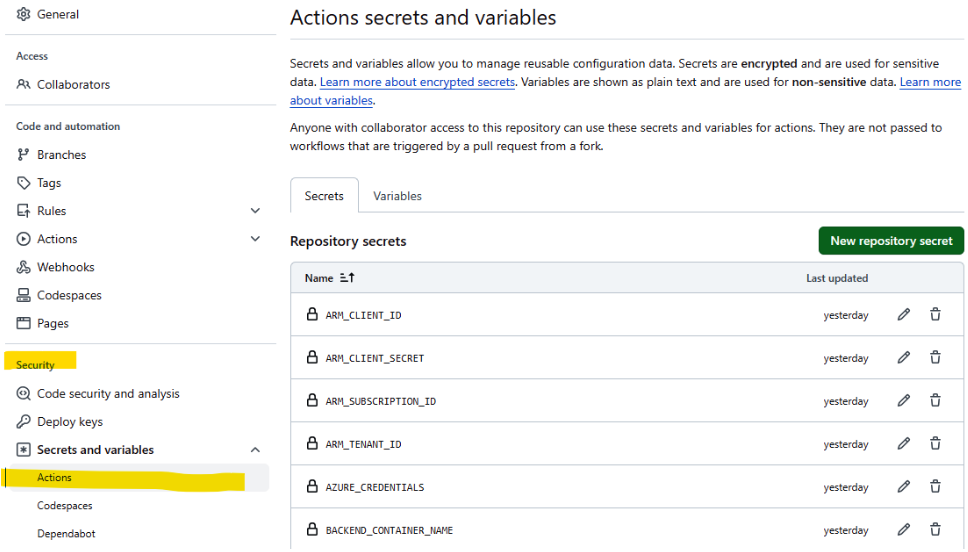
The pipeline is configured to trigger on any push or pull request to the main branch with one exception: changes exclusively in the `Step-01/deployment` directory do not initiate the workflow. This precaution prevents redundant runs when only Kubernetes manifest files are updated, ensuring efficient use of resources and avoiding potential conflicts in continuous deployment scenarios.

Terraform plan and apply

The workflow begins with the `terraform-plan` job. This job executes several critical steps:

1. **Environment setup**: The job initializes by checking out the repository and setting up the Azure CLI with credentials stored securely as GitHub secrets. This step ensures that the workflow has access to manage resources in Azure.

   All the passwords, tokens, and other sensitive information used in the pipeline need to be configured as GitHub Actions Secrets, as illustrated in *Figure 11.2*.



Actions secrets and variables

Secrets and variables allow you to manage reusable configuration data. Secrets are **encrypted** and are used for sensitive data. Learn more about encrypted secrets. Variables are shown as plain text and are used for **non-sensitive** data. Learn more about variables.

Anyone with collaborator access to this repository can use these secrets and variables for actions. They are not passed to workflows that are triggered by a pull request from a fork.

Repository secrets

Name	Last updated
ARM_CLIENT_ID	yesterday
ARM_CLIENT_SECRET	yesterday
ARM_SUBSCRIPTION_ID	yesterday
ARM_TENANT_ID	yesterday
AZURE_CREDENTIALS	yesterday
BACKEND_CONTAINER_NAME	yesterday

Figure 11.2 – GitHub secrets on the Actions secrets and variables page

2. **Terraform initialization**: In this step, the job runs `terraform init` to prepare the Terraform environment, configuring backend storage for Terraform state files in Azure Blob Storage. The following code is extracted from the main pipeline:

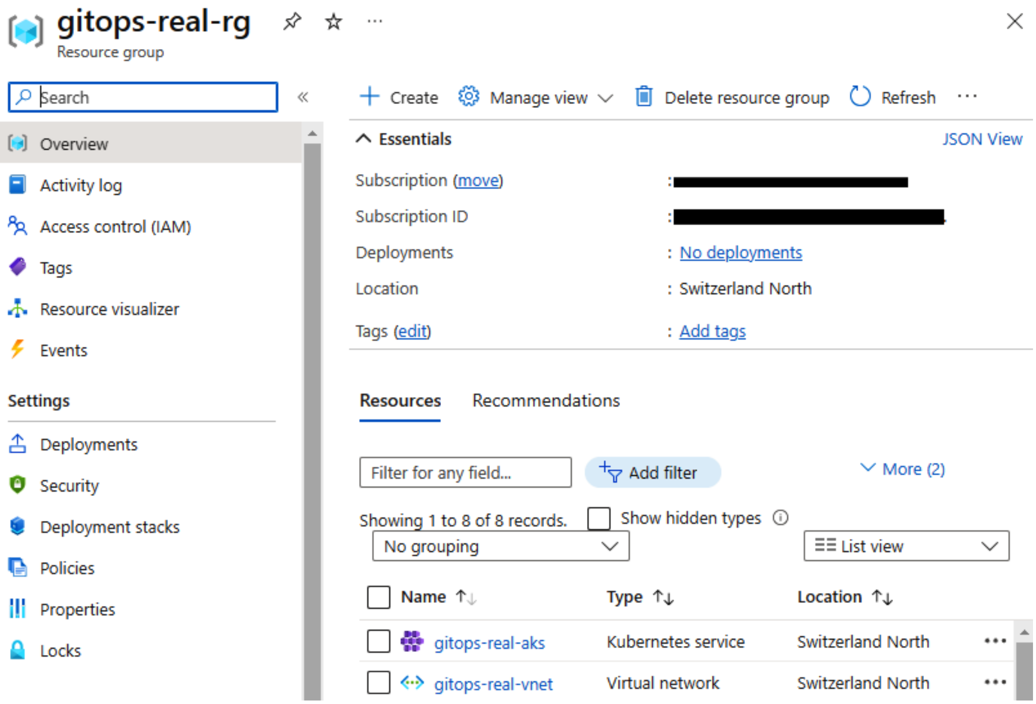
```
- name: Terraform Init
  run: |
    terraform init \
    --backend-config=»resource_group_name=${{ secrets.BACKEND_
RESOURCE_GROUP_NAME }}» \
    --backend-config=»storage_account_name=${{ secrets.BACKEND_
STORAGE_ACCOUNT_NAME }}» \
    --backend-config=»container_name=${{ secrets.BACKEND_
CONTAINER_NAME }}» \
    --backend-config=»key=${{ secrets.BACKEND_KEY }}» \
    --reconfigure
    working-directory: ./Step-01/terraform
    env:
    ARM_CLIENT_ID: ${{ secrets.ARM_CLIENT_ID }}
    ARM_CLIENT_SECRET: ${{ secrets.ARM_CLIENT_SECRET }}
    ARM_SUBSCRIPTION_ID: ${{ secrets.ARM_SUBSCRIPTION_ID }}
    ARM_TENANT_ID: ${{ secrets.ARM_TENANT_ID }}
```

3. **Execution plan**: Terraform then generates an execution plan (`terraform plan`), which is reviewed automatically to determine whether there are changes to apply. The plan is saved as an **artifact** for review and used in the subsequent `terraform-apply` job.

GitHub artifacts

An artifact in GitHub Actions is a file or a collection of files produced during a workflow run. Artifacts can include binary files, logs, test results, or any other type of data that needs to be stored after a job is completed. These artifacts are typically used for storing build and test outputs to be used for debugging, deployment, or further processing in subsequent steps or future runs. GitHub stores these artifacts for a specified period, allowing them to be downloaded or shared across different jobs within the same workflow. This feature facilitates effective CI/CD practices by ensuring that outputs from one part of a workflow can easily be accessed and utilized in other parts, enhancing automation and continuity throughout the software development life cycle.

Following planning, the `terraform-apply` job applies the approved changes to the infrastructure, ensuring that the actual state matches the expected state defined in the Terraform configurations. This part can take a few minutes due to the provisioning of the resources to Azure. Opening the Azure portal, the final provisioning should be similar to what is illustrated in *Figure 11.3*.



Subscription (move) : [redacted]

Subscription ID : [redacted]

Deployments : No deployments

Location : Switzerland North

Tags (edit) : Add tags

Showing 1 to 8 of 8 records.

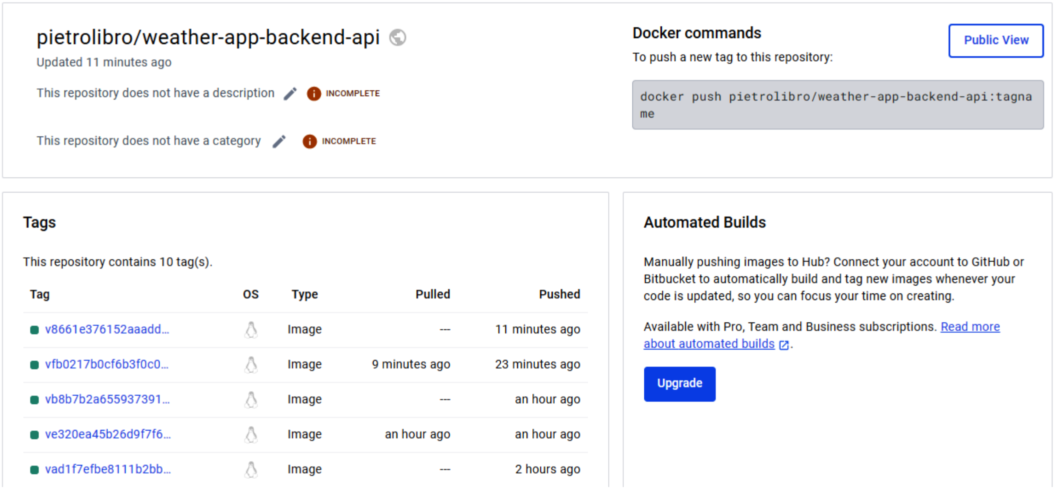
Name	Type	Location
gitops-real-aks	Kubernetes service	Switzerland North
gitops-real-vnet	Virtual network	Switzerland North

Figure 11.3 – Azure resources automatically provisioned by the GitHub workflow

Docker image build and push

Parallel to infrastructure management, the `docker-build-and-push` job handles the application side:

1. **Docker preparation**: The job sets up Docker environments using QEMU and Buildx, tools that enhance Docker's capabilities on CI environments.
2. **Building and Pushing Docker Images**: It then builds the Docker image from a **Dockerfile** located in the `Step-01` directory and pushes it to Docker Hub, tagging it with the **commit SHA** for immutability and traceability, as illustrated in *Figure 11.4*.



pietrolibro/weather-app-backend-api

Updated 11 minutes ago

This repository does not have a description INCOMPLETE

This repository does not have a category INCOMPLETE

Docker commands

To push a new tag to this repository:

```
docker push pietrolibro/weather-app-backend-api:tagna
me
```

Tags

This repository contains 10 tag(s).

Tag	OS	Type	Pulled	Pushed
v8661e376152aaadd...		Image	---	11 minutes ago
vfb0217b0cf6b3f0c0...		Image	9 minutes ago	23 minutes ago
vb8b7b2a655937391...		Image	---	an hour ago
ve320ea45b26d9f7f6...		Image	an hour ago	an hour ago
vad1f7efbe8111b2bb...		Image	---	2 hours ago

Automated Builds

Manually pushing images to Hub? Connect your account to GitHub or Bitbucket to automatically build and tag new images whenever your code is updated, so you can focus your time on creating.

Available with Pro, Team and Business subscriptions. Read more about automated builds.

Figure 11.4 – A Docker repository containing the built images with tags corresponding to the SHA number

Kubernetes deployment

After the Docker image is pushed and the infrastructure is ready, the `deploy-to-kubernetes` job proceeds:

1. **Cluster configuration**: The job configures `kubectl` with the credentials for the Kubernetes cluster managed in Azure, ensuring that commands are executed against the correct cluster, as reported in the following code:

   ```
   - name: Update Kubeconfig
     run: az aks get-credentials --resource-group gitops-dev-rg
   --name gitops-dev-aks --overwrite-existing
   ```

2. **Secrets and configurations**: It then deploys necessary Kubernetes secrets and configurations, such as API keys needed by the application, using best practices for secret management. The following code is extracted from the pipeline:

   ```
   - name: Deploy to Kubernetes
     run: |
       cd ./Step-01/deployment
       kubectl create namespace weather-app-for-real \
          --dry-run=client -o yaml | kubectl apply -f -
       kubectl create secret generic weather-api-key \
          --from-literal=WEATHER_API_KEY=${{ secrets.WEATHER_API_
   TOKEN }} \
   ```

```
        --namespace weather-app-for-real \
        --dry-run=client -o yaml | kubectl apply -f -
```

The use of the `--dry-run=client -o yaml | kubectl apply -f -` command sequence can play a significant role in ensuring that a Kubernetes deployment is **idempotent**.

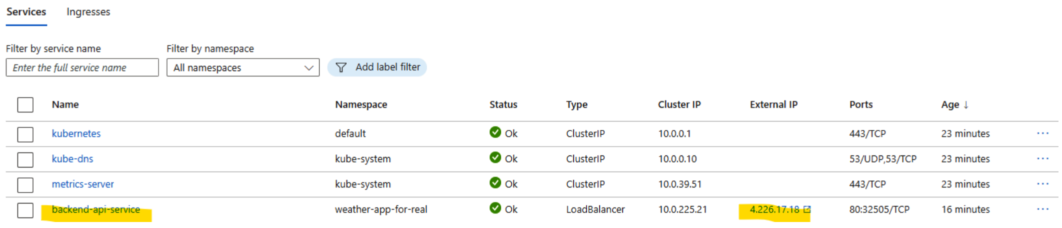
> **Idempotency**
>
> Idempotency, in the context of deploying resources, means that running the same deployment command multiple times will result in the same state without causing unintended changes or side effects after the initial application.

3. **Application deployment**: Applies the Kubernetes deployment manifest (`Step-01/deployment/backend-api-deployment.yaml`), which references the newly built Docker image, ensuring that the latest version of the application is deployed:

```
$ kubectl apply -f backend-api-deployment.yaml
```

> **Beware!**
>
> If you want to access the remote AKS cluster from your local development, you need to login Azure and execute the following command: `az aks get-credentials --resource-group gitops-real-rg --name gitops-real-aks`

4. **Testing**: Unlike local development, for this real-world example, we specified a **LoadBalancer** port in the deployment file, so AKS is automatically using a public IP address to expose our service to the public internet, as illustrated in *Figure 11.5*.



☐	Name	Namespace	Status	Type	Cluster IP	External IP	Ports	Age ↓	
☐	kubernetes	default	✓ Ok	ClusterIP	10.0.0.1		443/TCP	23 minutes	···
☐	kube-dns	kube-system	✓ Ok	ClusterIP	10.0.0.10		53/UDP,53/TCP	23 minutes	···
☐	metrics-server	kube-system	✓ Ok	ClusterIP	10.0.39.51		443/TCP	23 minutes	···
☐	backend-api-service	weather-app-for-real	✓ Ok	LoadBalancer	10.0.225.21	4.226.17.18	80:32505/TCP	16 minutes	···

Figure 11.5 – A public IP address used to expose the backand-api-service to the public internet

5. At this point, we can query our service using a URL like the one shown in *Figure 11.6* to obtain a response:

```
{
  "current": {
    "cloud": 25,
    "condition": {
      "code": 1003,
      "icon": "//cdn.weatherapi.com/weather/64x64/day/116.png",
      "text": "Partly cloudy"
    },
    "feelslike_c": 25.6,
    "feelslike_f": 78,
    "gust_kph": 11.6,
    "gust_mph": 7.2,
    "humidity": 36,
    "is_day": 1,
    "last_updated": "2024-05-12 18:00",
    "last_updated_epoch": 1715529600,
    "precip_in": 0.01,
    "precip_mm": 0.19,
    "pressure_in": 29.94,
    "pressure_mb": 1014,
    "temp_c": 24,
    "temp_f": 75.2,
    "uv": 5,
    "vis_km": 10,
    "vis_miles": 6,
    "wind_degree": 46,
    "wind_dir": "NE",
    "wind_kph": 3.6,
    "wind_mph": 2.2
  },
  "location": {
    "country": "Switzerland",
    "lat": 47.37,
    "localtime": "2024-05-12 18:12",
    "localtime_epoch": 1715530370,
    "lon": 8.55,
    "name": "Zurich",
    "region": "",
    "tz_id": "Europe/Zurich"
  }
}
```

Figure 11.6 – An example of querying the service for Zurich city using real weather data

What we have obtained so far is a fully working CI/CD pipeline that exposes a service to the real world. We want to take it a step further by separating the CI pipeline from the CD pipeline using ArgoCD, as described in the next section.

Using Argo CD for the continuous deployment

In the world of modern software delivery, it's crucial to ensure that our deployment practices are as reliable and scalable as possible. Argo CD, a declarative GitOps continuous delivery tool for Kubernetes, significantly enhances these aspects by automating deployment processes and syncing the desired application state defined in a Git repository with the production environment.

Transitioning to Argo CD

In this section, we will evolve our GitHub Actions workflow by transitioning the `deploy-app-to-kubernetes` stage to an `argo-cd-deployment` stage. The `argo-cd-deployment` stage in our GitHub Actions workflow encapsulates the following key operations:

1. **Argo CD setup**: First, the workflow initializes Argo CD in the Kubernetes cluster if it's not already installed. This includes setting up the necessary namespaces and applying the Argo CD installation manifests directly from the official sources.
2. **Repository configuration**: The workflow then adds the Git repository containing the Kubernetes manifests to Argo CD. This step involves configuring Argo CD to monitor changes in the repository, which hosts the deployment definitions for the application.
3. **Application deployment via Argo CD**: It then ensures that the specific namespace for the application is created and ready for deployment.
4. **Application sync**: Next, it applies the `argocd_deployment.yaml` file, which defines the Argo CD application. This manifest specifies the path to the Kubernetes deployment manifests within the Git repository, the revision target (e.g., `branch`), and synchronization policies.
5. **Sync trigger**: Optionally, this step triggers a manual sync if immediate deployment is required, though typically Argo CD would automatically detect changes based on its polling strategy.

Managing downtime and ensuring continuity with Argo CD

When Argo CD experiences temporary downtime, the primary impact is on the synchronization and automated reconciliation of deployments in Kubernetes environments. During this period, any changes committed to the Git repository will not be synchronized with Kubernetes clusters, which means that updates, fixes, and new feature deployments are postponed. The automated reconciliation process, which ensures that the actual state of the Kubernetes environment matches the desired state specified in the Git repository, is also interrupted. This means that any discrepancies or configuration drifts that occur during the downtime will not be addressed until Argo CD is back online. Upon restoring Argo CD, it will automatically begin to process and apply all changes made during its downtime. The system will fetch the latest configurations from Git and proceed with the necessary reconciliations to align the Kubernetes clusters with the desired states from the repository. It's important to note that the running applications themselves are not directly affected by Argo CD's downtime; they will continue to operate as configured prior to the outage. However, to manage critical updates during such downtimes, teams might need to perform manual interventions, which should be handled carefully to avoid further complications once Argo CD resumes normal operation. Robust monitoring and alert systems are recommended to quickly detect any issues with Argo CD and to minimize the impact of such downtimes.

6. To see the new workflow in action, we need to replace the contents of the `gitops-for-real-ci-cd-pipeline.yml` file in the `Step-02-ArgoCD-Deployment` folder with the contents of the file named in the same manner located in the `.github/workflows` subdirectory. We must then commit and push the updated code to trigger a workflow run, as illustrated in *Figure 11.7*:

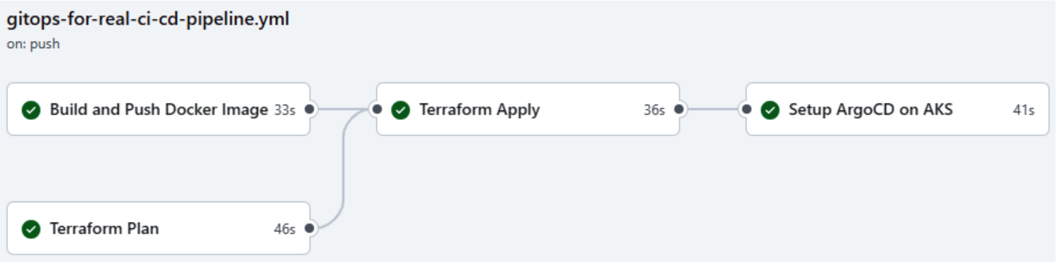
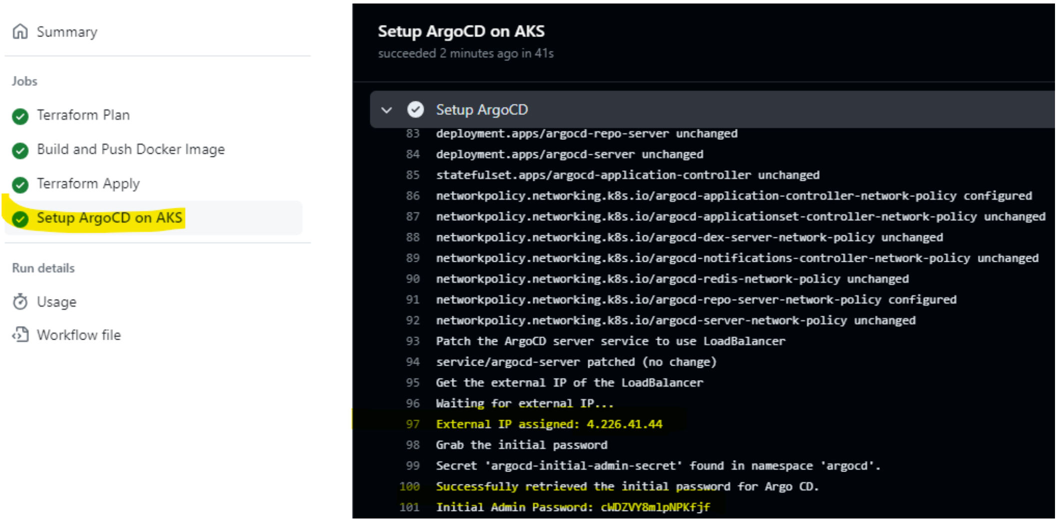


Figure 11.7 – A new workflow run is triggered after the commit and push of the new workflow definition

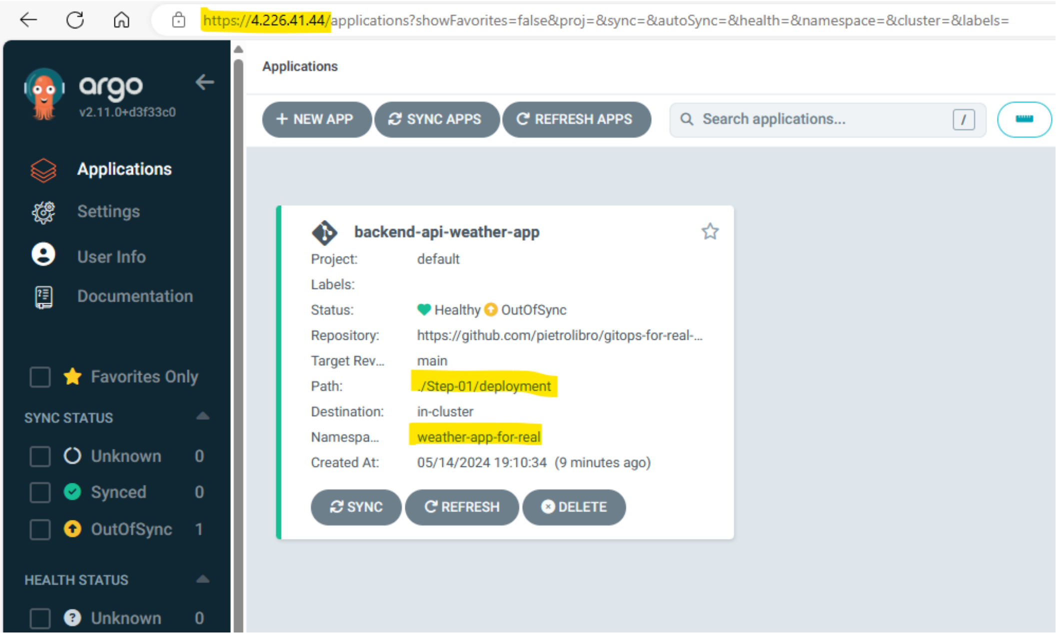
The external IP and the admin password that were automatically generated during the setup process can be found in the log of the `Setup ArgoCD on AKS` task, as illustrated in *Figure 11.8*. Be careful: this kind of information should not be exposed in real production environments. It is just a shortcut for the scope of this example.



```
deployment.apps/argocd-repo-server unchanged
deployment.apps/argocd-server unchanged
statefulset.apps/argocd-application-controller unchanged
networkpolicy.networking.k8s.io/argocd-application-controller-network-policy configured
networkpolicy.networking.k8s.io/argocd-applicationset-controller-network-policy unchanged
networkpolicy.networking.k8s.io/argocd-dex-server-network-policy unchanged
networkpolicy.networking.k8s.io/argocd-notifications-controller-network-policy unchanged
networkpolicy.networking.k8s.io/argocd-redis-network-policy unchanged
networkpolicy.networking.k8s.io/argocd-repo-server-network-policy configured
networkpolicy.networking.k8s.io/argocd-server-network-policy unchanged
Patch the ArgoCD server service to use LoadBalancer
service/argocd-server patched (no change)
Get the external IP of the LoadBalancer
Waiting for external IP...
External IP assigned: 4.226.41.44
Grab the initial password
Secret 'argocd-initial-admin-secret' found in namespace 'argocd'.
Successfully retrieved the initial password for Argo CD.
Initial Admin Password: cWDZVY8mlpNPKfjf
```

Figure 11.8 – The Setup ArgoCD on AKS task log containing sensitive information

At this point, we can log in to the admin UI of the deployed instance of Argo CD by typing `https://4.226.41.44/` into your preferred browser (see *Figure 11.9*):

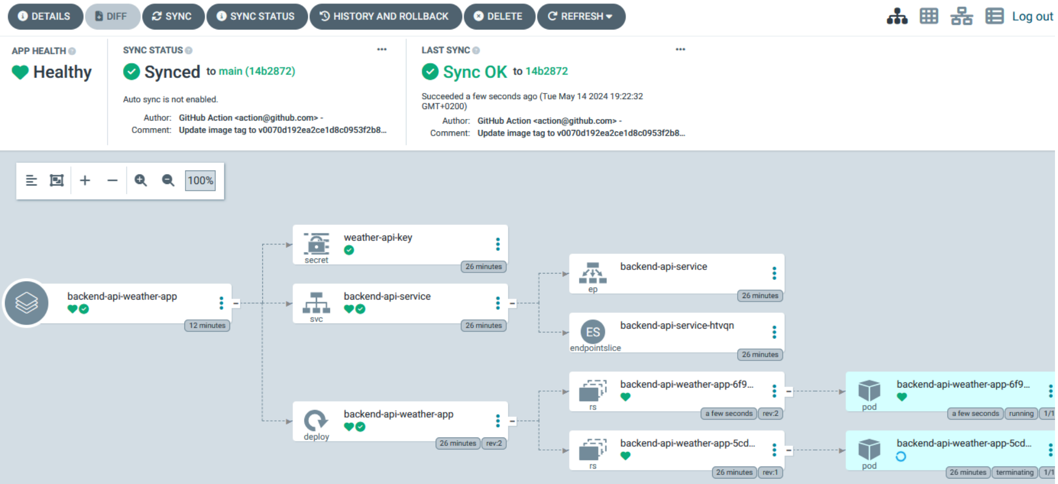


backend-api-weather-app

Project: default
Labels:
Status: Healthy OutOfSync
Repository: https://github.com/pietrolibro/gitops-for-real-...
Target Rev... main
Path: ./Step-01/deployment
Destination: in-cluster
Namespa... weather-app-for-real
Created At: 05/14/2024 19:10:34 (9 minutes ago)

Figure 11.9 – The Argo CD home page after logging in, showing the deployment of the backend-api-weather-app pod

Voilà! For this example, we didn't activate **auto-sync**, so we just need to click on the **Sync Apps** button to synchronize our weather app application, as illustrated in *figure 11.10*.



APP HEALTH
Healthy

SYNC STATUS
Synced to main (14b2872)
Auto sync is not enabled.
Author: GitHub Action <action@github.com> -
Comment: Update image tag to v0070d192ea2ce1d8c0953f2b8...

LAST SYNC
Sync OK to 14b2872
Succeeded a few seconds ago (Tue May 14 2024 19:22:32 GMT+0200)
Author: GitHub Action <action@github.com> -
Comment: Update image tag to v0070d192ea2ce1d8c0953f2b8...

Figure 11.10 – The Argo CD application is correctly synchronized with the GitHub repository

To test the Argo CD synchronization process, try changing the number of replicas from `1` to `5` (for instance) in the `backend-api-deployment.yaml` file and pushing the change to GitHub. The workflow will not be triggered because we specified the following value:

```
paths-ignore:
- 'Step-01/deployment/**'
```

However, Argo CD will notice the out-of-sync state and a new synchronization will be needed. Now that we have our CI/CD pipeline in place and working perfectly, it is time to introduce the topics of scalability and efficiency in the next section.

Designing for scalability and efficiency

In this section, we will delve into designing for scalability and efficiency. These traits are essential in the architecture of modern applications, exemplified by our weather app. Scalability ensures that the application can handle growth, whether it's an increasing number of users, data volume, or transaction frequency, without compromising performance. Efficiency involves optimizing resource use, which is crucial for minimizing costs and enhancing response times. We will explore architectural principles that support scalability, such as **microservices** and load balancing, and discuss how to manage compute, storage, and networking resources effectively. Additionally, we will look at tools and strategies to test scalability to ensure that the architecture can withstand real-world demands. By mastering these elements, you'll learn how to design a scalable and efficient architecture that is well-suited for deployment on Kubernetes, enhancing the overall performance and reliability of applications such as our weather app.

Architectural principles

Architectural principles for designing scalable and efficient systems are critical in modern application development, especially as demands for performance and reliability increase. Key strategies include decoupling components to minimize dependencies, which facilitates easier maintenance and scaling. Emphasizing statelessness allows for the replication and distribution of components, enhancing the application's resilience and responsiveness.

Load balancing is essential to distribute incoming network loads evenly across multiple systems, preventing any single server from becoming overwhelmed and increasing the application's availability. Horizontal scaling, or scaling out by adding more instances rather than adding resources to a single instance, is more cost-effective and increases fault tolerance.

Database sharding partitions data into smaller, more manageable segments, which is particularly beneficial for large datasets or high throughput demands. Sharding is great for improving performance. Caching frequently accessed data reduces latency and backend load by serving common requests without redundant data processing.

Asynchronous processing of tasks enhances throughput and user experience by handling operations in a non-blocking manner. Adopting a microservices architecture allows for independent deployment, scaling, and management of each service. This modular approach not only boosts performance but also simplifies management as applications evolve, making it ideal for cloud-native environments managed by platforms such as Kubernetes. Although microservices architecture has been mentioned, it will not be part of the example discussed in this chapter.

Microservices architecture

Microservices architecture is an architectural style that structures an application as a collection of loosely coupled services, each of which implements a specific business capability. This approach enables developers to build and deploy services independently, which enhances flexibility and accelerates development cycles. By breaking down an application into small, manageable components, microservices allow for more granular scaling and efficient resource utilization. Each service can be developed, deployed, and scaled independently, often using different programming languages and technologies that best suit the task at hand. This modularity improves fault isolation, making it easier to identify and fix issues without affecting the entire system. Moreover, microservices facilitate CI/CD practices, promoting a more agile and resilient development process. Overall, microservice architecture fosters a more robust and scalable application environment that is capable of adapting to evolving business needs and technological advancements.

Resource management

Effective resource management is crucial in application design and operation, especially in environments that aim to maximize efficiency and performance while minimizing costs. Managing compute, storage, and networking resources involves careful planning and orchestration to ensure that each component of an application has the necessary resources to perform optimally without wastage:

- **Compute management**: This involves provisioning the right amount of CPU and memory resources to meet the application's requirements. Techniques such as **auto-scaling** and load balancing help distribute compute workloads evenly across the available infrastructure.
- **Storage management**: This ensures that data storage resources are allocated efficiently, keeping data accessibility and redundancy in mind. This includes choosing appropriate storage types and implementing data partitioning strategies to enhance performance and scalability.
- **Networking management**: This focuses on efficiently configuring network resources to ensure fast and secure data transfer between application components. Proper network configuration reduces latency and prevents bottlenecks, making it essential for real-time data processing and delivery.

Together, these resource management practices ensure that applications can scale effectively and remain robust under varying operational conditions. Implementing resource management strategies also involves monitoring and analyzing resource usage to make informed decisions about adjustments and improvements, ensuring that resources are utilized in the most efficient way possible.

Testing for scalability

Testing for scalability is crucial for ensuring that applications perform well under expected loads and can handle growth in users, transactions, and data efficiently. Scalability testing involves a variety of techniques to simulate different environments and stresses on the system to uncover potential issues before they impact users:

- **Load testing**: Simulates a specific expected number of concurrent users or transactions to assess how the application behaves under normal conditions
- **Stress testing**: Pushes the application beyond its normal operational limits to discover its maximum capacity and understand its behavior under extreme conditions
- **Soak testing**: Runs the application under a heavy load for a prolonged period to identify issues such as memory leaks or slow degradation of performance
- **Spike testing**: Checks the application's ability to handle sudden and large spikes in traffic
- **Scalability testing**: Tests whether the application can scale up or down based on demand by gradually increasing the load and observing how additional resources affect the application's capacity

These tests often utilize automated testing tools and are conducted in staged environments that closely mimic real-world traffic patterns. Tools such as **Apache JMeter**, **LoadRunner**, and **Gatling**, along with cloud services such as **AWS CloudWatch** and **Google Cloud Monitoring**, are commonly employed to facilitate these tests. Through regular scalability testing across development and deployment phases, teams can ensure that their applications are robust, scalable, and ready to handle real-world operational demands.

Resources management and scalability

In this section, we will continue to use the weather app to see how resource management, horizontal scaling, and scalability testing work in a real-world scenario. We can start with the optimization of resource usage.

Optimizing resource usage

To optimize resource usage in Kubernetes, setting up resource requests and limits is crucial. These settings ensure that pods receive the right amount of CPU and memory resources to function properly while also preventing any single application from consuming excessive cluster resources, which could affect other applications:

- **Requests**: These are the amount of resources Kubernetes guarantees for a container. If a container requires more resources than its request and they are available on the node, it can consume more.
- **Limits**: This is the maximum amount of resources a container can use. If a container tries to exceed this limit, the system will throttle its CPU usage. If the container exceeds its memory limit, Kubernetes might terminate it, depending on the situation.

To test the use of requests and limits, we can try to update the `Step-01/deployment/backend-api-deployment.yaml` file by adding the following code block immediately after `key: WEATHER_API_KEY row`:

```
…          key: WEATHER_API_KEY
resources:
          requests:
            cpu: «100m»
            memory: "100Mi"
          limits:
            cpu: «150m»
            memory: "150Mi"
```

The previously mentioned code block specifies the resource requests and limits for a container. Here's what each line means:

- `requests`:
  - `cpu: "100m"`: This requests 100 millicores (where 1,000 m equals 1 CPU core) for the container
  - `memory: "100Mi"`: This requests 100 mebibytes of memory
- `limits`:
  - `cpu: "150m"`: This sets a limit of 150 millicores for CPU usage by the container
  - `memory: "150Mi"`: This sets a memory limit of 150 mebibytes

Now that we have defined requests and limits, in the next section, we will see how to implement the **Horizontal Pod Autoscaler** (**HPA**) in the next section.

Implementing the HPA

Implementing an HPA in Kubernetes is an effective way to automatically scale the number of pod replicas in a deployment, replication controller, or replica set based on observed CPU utilization or other select metrics such as memory usage or custom metrics. Here's a step-by-step guide to setting up an HPA:

1. Ensure that the **Metrics Server**, which collects resource metrics from Kubelets and exposes them in Kubernetes through the Metrics API, is installed in the cluster. This is crucial for the HPA to make scaling decisions. We can install it with the following command:

   ```
   $ az aks update --enable-azure-monitor-metrics --name gitops-
   real-aks --resource-group gitops-for-real-rg
   ```

2. Setup will take a few minutes. At completion, verify the correct installation with the following command:

   ```
   $ kubectl get deployment metrics-server -n kube-system
   ```

   This command should return the deployment details confirming that the Metrics Server is up and running:

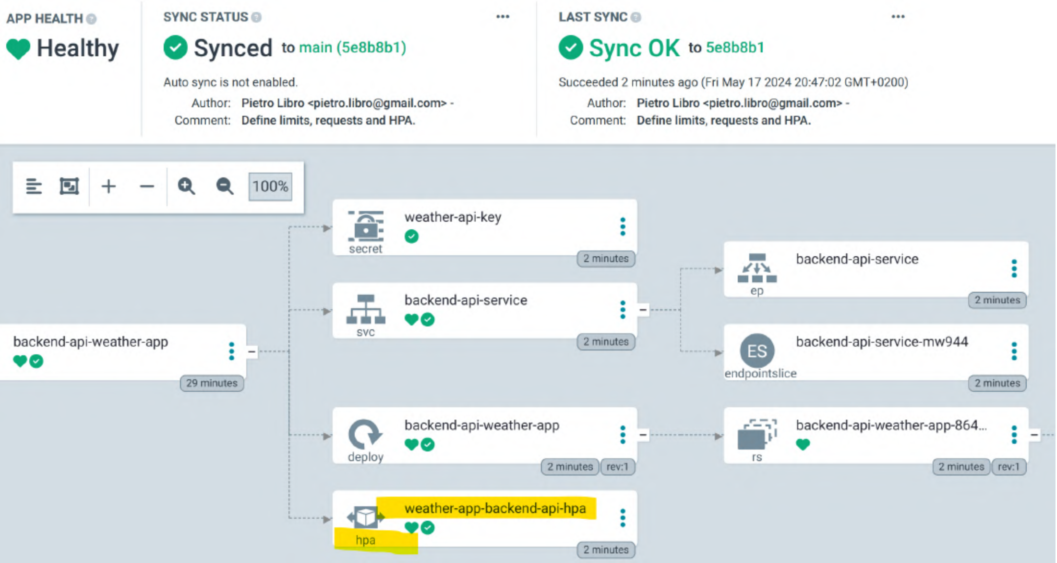
   ```
   NAME            READY   UP-TO-DATE   AVAILABLE   AGE
   metrics-server  2/2     2            2           3h37m
   ```

3. Create an HPA that scales based on CPU utilization:

   ```
   apiVersion: autoscaling/v1
   kind: HorizontalPodAutoscaler
   metadata:
     name: weather-app-backend-api-hpa
     namespace: weather-app-for-real
   spec:
     scaleTargetRef:
       apiVersion: apps/v1
       kind: Deployment
       name: backend-api-weather-app
     minReplicas: 1
     maxReplicas: 5
     targetCPUUtilizationPercentage: 5
   ```

   This HPA is configured to maintain between `1` and `5` replicas of the pod, scaling up or down when the CPU utilization reaches 50%.

4. Create a new `hpa.yaml` file in the `Step-01/deployment` folder with the content that we have described. Commit and push the code, then wait for the Argo CD application synchronization or force it if auto-sync is off.



APP HEALTH
Healthy

SYNC STATUS
Synced to main (5e8b8b1)
Auto sync is not enabled.
Author: Pietro Libro <pietro.libro@gmail.com> -
Comment: Define limits, requests and HPA.

LAST SYNC
Sync OK to 5e8b8b1
Succeeded 2 minutes ago (Fri May 17 2024 20:47:02 GMT+0200)
Author: Pietro Libro <pietro.libro@gmail.com> -
Comment: Define limits, requests and HPA.

Figure 11.11 – The Argo CD application synchronized with the HPA configuration in place

5. Use the following command to monitor the status and effectiveness of your HPA:

```
$ kubectl get hpa -w --namespace weather-app-for-real
```

This will show the current number of replicas and whether the HPA is in the process of scaling up or down based on the current CPU utilization against the target set.

Testing for scalability – an example

Now that we have implemented the HPA and set up monitoring for it, our next step is to observe how Kubernetes dynamically scales the number of pods in response to changes in CPU utilization. By simulating varying loads, we can watch the HPA adjust the pod count to maintain optimal performance. This process ensures that our application scales efficiently, handling increases or decreases in demand without manual intervention. Understanding this behavior is crucial for optimizing resource management and cost-effectiveness within our Kubernetes environment. What follows is a guided step-by-step testing scenario:

1. Create a Bash script for testing HPA. Use the `curl` command to make requests to the exposed weather service using various cities and include a random value in the query string to avoid caching. You can use the `hpa-testing.sh` script present in the repository accompanying this chapter as reference. Before executing the script, update `$baseUrl` to match your weather service's URL. This might look as follows:

```
# Base URL of your weather service
baseUrl="http://20.250.198.208/weather"
```

2. Run this Bash script:

    I. Open a new terminal and make the script executable with the following:

    ```
    $ chmod +x test_weather_app.sh
    ```

    II. Execute the script by running the following:

    ```
     ./hpa_testing.sh.
    ```

    After a while, the percentages described by the `TARGET` column should increase. To speed up the testing scenario and see the number of pods increase faster, execute another instance of the same script in another terminal.

3. Check the HPA status (see the sixth point in the *Implementing the HPA* section of this chapter). Use the following command to check the HPA status:

    ```
    $ kubectl get hpa -w --namespace weather-app-for-real
    ```

    You should see the current number of replicas and their scaling activities based on the CPU utilization:

    ```
    NAME                TARGETS     MINPODS      MAXPODS
    REPLICAS
    weather-app-…       cpu: <unk>%/5%   1     5             1
    weather-app-…    cpu: 5%/5%   1            5             1
    weather-app-…    cpu: 20%/5% 1             5             4
    …
    weather-app-…    cpu: 3%/5%   1         5             4
    weather-app-…    cpu: 1%/5%     1             5             4
    ```

The HPA monitors the CPU utilization of the deployment and adjusts the number of pods accordingly to ensure optimal performance.

Initially, the CPU utilization is marked as `<unknown>`, likely due to metrics not being available or still being initialized. When the utilization stabilizes at 5%, which matches the target set in the HPA, there's no change in the number of replicas and they remain at one. As CPU usage increases to 20%—well above the 5% target—the HPA reacts by scaling up the number of replicas from `1` to `4` to handle the increased load.

This elevated level of resource use persists briefly, keeping the replicas at `4`. However, when the utilization drops significantly to 3% and further down to 1%, the HPA initially doesn't scale down immediately, possibly due to stabilization settings that prevent oscillations in pod count. Ultimately, as the low utilization continues, the HPA scales the number of pods back down to `1`.

This sequence demonstrates the HPA's capability to dynamically scale application resources based on real-time data, thus ensuring that the deployment scales efficiently in response to workload changes. This dynamic adjustment helps manage resources effectively, maintaining application responsiveness and optimizing operational costs. The responsiveness of the HPA to changes in CPU utilization exemplifies how Kubernetes can automate scaling to maintain performance and resource efficiency without manual intervention.

As we ensure efficient resource management and scalability, it is equally important to turn our attention to monitoring and securing your application. In the next section, we will explore these crucial aspects of operational excellence.

Monitoring and securing your application

Operational excellence in software deployment and management is a crucial factor for the success of any technology-driven organization. The keys to achieving this excellence are monitoring, scaling, and security, each serving as foundational pillars that ensure the smooth and efficient operation of applications in production environments.

Monitoring is vital as it provides the visibility needed to understand the behavior of applications and systems in real time. Effective monitoring strategies help in identifying performance bottlenecks, predicting system failures, and gathering valuable data to aid in decision-making processes. This continuous oversight allows teams to respond proactively to issues before they affect the user experience or lead to more significant disruptions.

Security practices are critical to safeguard sensitive data and protect infrastructures from breaches and attacks. In an era where cyber threats are evolving rapidly, ensuring that robust security measures are in place is non-negotiable. Security protocols help in maintaining trust with customers, complying with regulatory requirements, and avoiding the financial and reputational damage associated with data breaches.

Together, monitoring, scaling, and security form the backbone of operational excellence, supporting a stable, efficient, and secure environment for deploying and managing applications. Organizations that master these aspects are better positioned to leverage technology for business success, ensuring that they can deliver continuous value to users while adapting to the ever-changing digital landscape.

Monitoring

Grafana and Prometheus are powerful tools that are widely used in the monitoring and observability landscape. They are particularly valuable for managing cloud-native applications deployed in dynamic environments such as Kubernetes:

- **Prometheus**: Prometheus is an open source monitoring system with a robust query language. It collects and stores its metrics as time-series data, meaning that each metric is stored with its exact time of recording. Prometheus is highly effective for recording real-time metrics in a high-availability environment. It supports a pull model for fetching data from monitored services, allowing it to actively scrape data from registered targets at specified intervals. This data can then be queried and analyzed to monitor the health and performance of applications.
- **Grafana**: Grafana is an open source analytics and visualization platform that integrates seamlessly with a multitude of data sources, including Prometheus. Grafana is used to create comprehensive dashboards that provide visualizations of metrics data. These dashboards allow developers and operations teams to visually interpret complex data to understand application behavior and resource usage, making it easier to spot trends, patterns, and potential problems.

Together, Prometheus and Grafana offer a powerful combination for data gathering, storage, and visualization, enhancing the ability to observe system behaviors, troubleshoot issues, and ensure that system performance aligns with user expectations and business objectives. This duo is particularly effective in a DevOps context, where continuous monitoring and feedback loops are critical to the software development and deployment life cycle.

Setting up Prometheus and Grafana

The following are the necessary steps to set up Prometheus and Grafana on the AKS cluster:

1. Add the Prometheus and Grafana Helm chart repository:

```
$ helm repo add prometheus-community https://prometheus-
community.github.io/helm-charts
```

2. Update the Helm repository. Ensure that we are using the most up-to-date version:

```
$ helm repo update
```

3. Install the Helm Chart in a namespace called `gitops-real-monitoring`:

```
$ helm install prometheus \
  prometheus-community/kube-prometheus-stack \
  --namespace gitops-real-monitoring \
  --create-namespace
```

4. Check the entire deployment by typing the following command:

```
$ kubectl get all -n gitops-real-monitoring
```

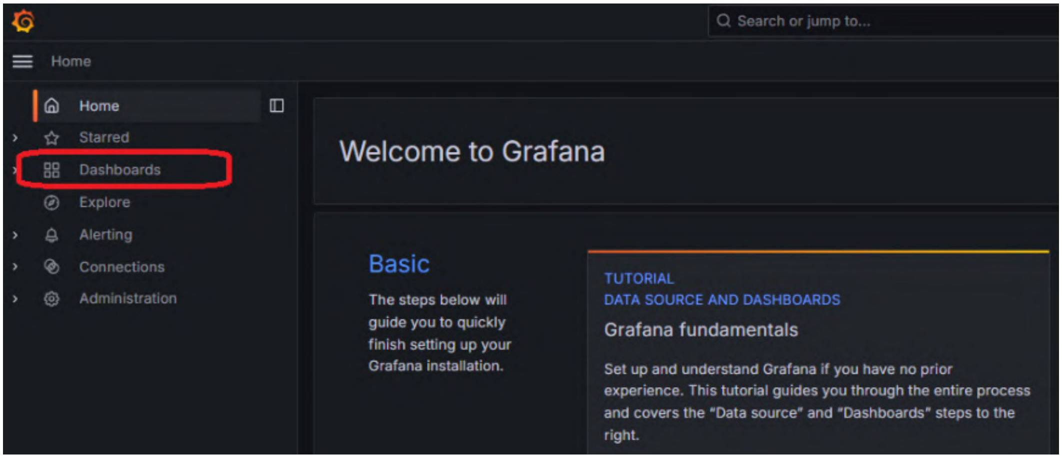
5. Since we are not using a `LoadBalancer` service's type, to log in to Grafana, we need to execute a port-forward:

```
$ kubectl port-forward svc/prometheus-grafana -n gitops-real-
monitoring 9000:80
```

6. Log in with the `admin` username the password defined for the prom-operator.
7. Expose Prometheus using the following command:

```
$ kubectl port-forward svc/prometheus-kube-prometheus-prometheus
-n gitops-real-monitoring 9001:9090
```

8. After logging in, you should be able to see the Grafana homepage. Click on the **Dashboards** menu item as illustrated in *Figure 11.12*:



Welcome to Grafana

Basic

The steps below will guide you to quickly finish setting up your Grafana installation.

TUTORIAL

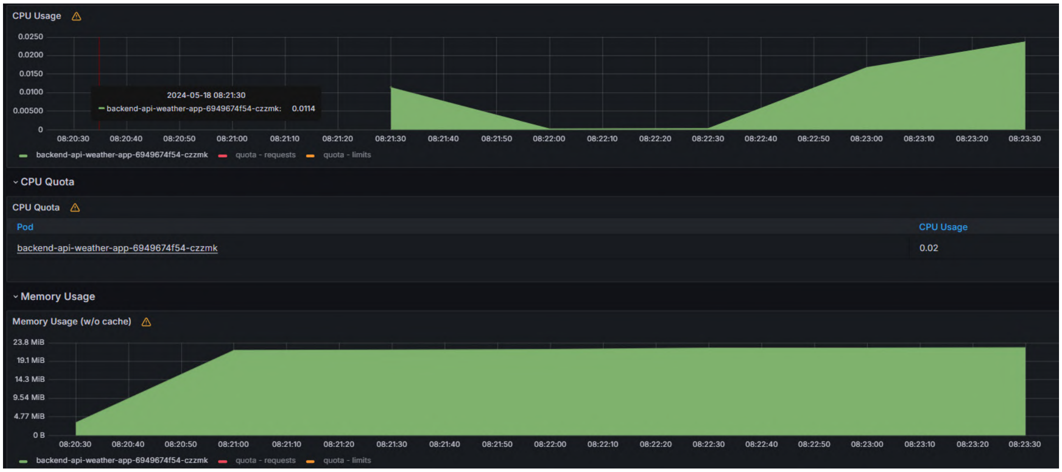
DATA SOURCE AND DASHBOARDS

Grafana fundamentals

Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.

Figure 11.12 – The Grafana home page, with the Dashboards menu item highlighted

9. Click on **Kubernetes | Compute Resources | Namespace (Pods)** and change the namespace to `weather-app-for-real`. You will see some interesting metrics on the Pods that are running there, as illustrated in *Figure 11.13*.



Pod	CPU Usage
backend-api-weather-app-6949674f54-czzmk	0.02

Figure 11.13 – CPU, memory, and other metrics for the backend-api-weather pod

At this point, you have correctly and successfully set up Grafana and Prometheus. Now, you can see interesting statistics about the usage of `backend-api-weather-app`, which can be used to fine-tune the resource limits and requests, as discussed in the *Optimize resource usage* section of this chapter. In the next section, we will introduce another important aspect of Kubernetes management in the real world: Kubernetes security.

Understanding Kubernetes security

Kubernetes, while robust and scalable, presents a unique set of security challenges that stem from its dynamic and distributed nature. Securing a Kubernetes cluster involves safeguarding the infrastructure, the applications running on it, and the data that it processes. Given the complexity of Kubernetes environments, security must be integrated into every layer of the cluster. Key aspects of Kubernetes security include the following:

- **Authentication and authorization**: This ensures that only verified users can access the cluster with methods such as certificates and tokens. It also controls user actions using mechanisms such as RBAC and **Attributed-Based Access Control** (**ABAC**).
- **API security**: Protecting the Kubernetes API server, which acts as the central control unit for the cluster, is crucial. Securing access to the API involves using SSL/TLS encryption, API request auditing, and limiting IP access through network policies.
- **Network security**: Enforcing policies that control the flow of traffic between pods and external networks helps prevent unauthorized access and limits the potential for lateral movement within the cluster.

- **Pod security admission**: This is a Kubernetes admission controller that enforces security settings on pods at creation time, using predefined security profiles (**privileged**, **baseline**, and **restricted**) to ensure compliance with best security practices and prevent privilege escalations.
- **Secrets management**: Kubernetes manages sensitive data (such as passwords and tokens) using secrets. Proper handling and security of secrets, including encryption at rest and in transit, is vital to protect sensitive information.

The importance of a layered security approach

Given the complexities of Kubernetes, a single security measure is often not enough. A layered security approach that includes network segmentation, threat detection, secure access controls, and ongoing vulnerability management is crucial for protecting Kubernetes environments from threats.

In the next section, we will see a practical example of how to manage access to the weather app's resources within a specific namespace using RBAC.

Configuring Kubernetes RBAC for user and role management

Here is a step-by-step guide to configuring RBAC for the weather app:

1. Define a role that specifies the permissions for managing specific resources related to the weather app, such as deployments, services, and pods within a designated namespace. The following are the definitions for the `weather-app-manager` role:

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: Role
   metadata:
     namespace: weather-app-for-real
     name: weather-app-manager
   rules:
   - apiGroups: ["", "apps"]
     resources: ["deployments", "replicasets", "pods", "services"]
     verbs: ["get", "list", "watch", "create", "update", "delete"]
   ```

   Here are the definitions for the `weather-app-user` role:

   ```
   apiVersion: rbac.authorization.k8s.io/v1
   kind: Role
   metadata:
     namespace: weather-app-for-real
     name: weather-app-user
   ```

```
rules:
- apiGroups: ["", "apps"]
  resources: ["pods", "services"]
  verbs: ["get", "list", "watch"]
```

In a production environment, user management might be handled outside Kubernetes through OIDC, LDAP, or other authentication services. For demonstration purposes, this example assumes that user credentials and certificates are managed by your Kubernetes administrator or through a cloud provider's IAM system.

2. Create a `RoleBinding` resource to grant the specified role to a user. This binding will apply the `weather-app-manager` role to a user named `weather-app-user`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: weather-app-manager-binding
  namespace: weather-app-for-real
subjects:
- kind: User
  name: weather-app-user
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: weather-app-manager
  apiGroup: rbac.authorization.k8s.io
```

3. Create another `RoleBinding` resource to grant the `weather-app-user` role to a user named `weather-app-operator`:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: weather-app-operator-binding
  namespace: weather-app-for-real
subjects:
- kind: User
  name: weather-app-operator
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: weather-app-operator
  apiGroup: rbac.authorization.k8s.io
```

4. Move the YAML files to the `Step-01/deployment` folder, push the changes to GitHub, and synchronize the Argo CD app.
5. Verify that the `weather-app-user` and `weather-app-operator` users have the necessary permissions using the `kubectl auth can-i` command:

   ```
   $ kubectl auth can-i delete pods --as weather-app-operator -n
   weather-app-for-real
   $ kubectl auth can-i delete pods --as weather-app-manager -n
   weather-app-for-real
   ```

   The expected output should be as follows:

   ```
   no
   yes
   ```

6. We can list all roles and role bindings in the namespace, or cluster roles affecting the user, with commands such as the following:

   ```
   $ kubectl get
   rolebindings,roles,clusterrolebindings,clusterroles
   --all-namespaces -o yaml
   ```

For reference, the `Step-03-Security` directory in the repository accompanying this chapter contains the YAML files with the role and role-binding definitions described so far.

> **Beware!**
>
> To avoid incurring unexpected expenses due to Azure resources, please remember to destroy any undesired Azure provisioned resources.

This section concludes our journey through a real-world GitOps pipeline and deployment. While an entire book might not be enough to delve deeply into every aspect of GitOps with Kubernetes, security, and deployments, we believe that the sections covered so far provide a comprehensive overview. They offer valuable insights into setting up an effective GitOps pipeline for your future Kubernetes projects.

Summary

This chapter provided a comprehensive guide to deploying real-world projects on Kubernetes with GitOps. By following the detailed instructions and examples, you learned how to set up a GitOps and Kubernetes development environment, implement CI/CD processes, design for scalability and efficiency, manage resources, and secure your application. This practical knowledge equipped you with the skills needed to effectively implement these cutting-edge technologies in your projects, enhancing your organizational capabilities and personal technical expertise.

As you now have a solid foundation in deploying and managing applications with GitOps on Kubernetes, the next chapter will delve into observability with GitOps, providing essential strategies to monitor and gain insights into your applications' performance and health.

Part 4:
Operational Excellence Through GitOps Best Practices

In this part, you will focus on achieving operational excellence through best practices in GitOps. You will learn about integrating observability, enhancing security, managing financial operations, and preparing for future trends in GitOps. This section aims to provide a comprehensive guide to maintaining high standards of operational efficiency and security, while also addressing sustainability and financial considerations, thus ensuring that your GitOps practices are both cutting edge and sustainable.

This part includes the following chapters:

- *Chapter 12*, Observability with GitOps
- *Chapter 13*, Security with GitOps
- *Chapter 13*, FinOps, Sustainability, AI, and Future Trends for GitOps

12

Observability with GitOps

Welcome to a focused exploration of integrating observability into Kubernetes environments through the lens of **GitOps** practices. As cloud-native applications grow in complexity and scale, the ability to observe, understand, and react to their behavior becomes increasingly critical. This chapter is designed to bridge the gap between traditional operational methods and the dynamic, automated world of GitOps, offering a pathway to more resilient, responsive, and efficient systems.

At the heart of this journey is the fusion of **Site Reliability Engineering** (**SRE**) principles with the GitOps framework. GitOps, a term that has rapidly gained traction in the DevOps community, leverages the power of Git as a single source of truth for declarative infrastructure and applications. By applying GitOps, we not only automate and streamline deployment processes but also enhance the observability and manageability of Kubernetes environments.

This chapter delves into the essential concepts of **observability** within the GitOps paradigm, distinguishing between internal and external observability to provide a comprehensive understanding of system states. Internal observability sheds light on the workings within the system—through **metrics**, **logs**, and **traces**—while external observability focuses on the experience outside the system, such as user interactions and external dependencies.

A significant emphasis is placed on **SLO-driven** performance management. **Service-Level Objectives** (**SLOs**) serve as a quantifiable measure of performance and reliability, guiding our efforts in system optimization and improvement. Coupled with the **DevOps Research and Assessment** (**DORA**) metrics—deployment frequency, lead time for changes, change failure rate, and time to restore service—this approach offers a robust framework for assessing and enhancing the effectiveness of GitOps practices.

Furthermore, the chapter introduces the concept of distributed tracing, a critical component in understanding the flow of requests through microservices architectures. Implementing distributed tracing, with tools such as **Linkerd** within a GitOps workflow, provides deep insights into the interactions and dependencies of system components, facilitating rapid diagnosis and resolution of issues.

Lastly, we address the setup of monitoring and alerting systems using cutting-edge tools such as **OpenTelemetry**. This setup is crucial for proactive system management, allowing teams to detect and respond to anomalies before they escalate into more significant issues.

This chapter mainly talks about theories and ideas. It's a good idea to read everything from start to finish. After you're done, you'll get to put some of these ideas together in a special way and try them out yourself with a real example.

Embarking on this intermediate guide to observability with GitOps, you are taking a step toward mastering the art and science of maintaining highly observable, performant, and reliable cloud-native applications. Let's dive in and unlock the full potential of your Kubernetes deployments.

As such, the following main topics are covered in the chapter:

- Exploring the fundamentals of SRE for GitOps and Kubernetes
- Understanding internal versus external observability
- Exploring SLO-driven multi-stage performance with DORA
- Implementing distributed tracing in GitOps with Linkerd
- Implementing monitoring in GitOps with tools such as Uptime Kuma and OpenTelemetry
- Looking at alerting strategies in a GitOps framework
- Scaling observability with GitOps

Exploring the fundamentals of SRE for GitOps and Kubernetes

In the evolving landscape of cloud-native applications, the integration of SRE principles with GitOps and Kubernetes represents a significant leap toward operational excellence. This section aims to provide a concise overview of these foundational concepts, equipping you with the knowledge to apply SRE practices effectively within your GitOps workflows and Kubernetes environments.

The intersection of SRE with GitOps

SRE is a discipline that incorporates aspects of software engineering into the realm of IT operations. The core philosophy of SRE is to treat operations as if they were a software problem, focusing on automating and optimizing system reliability and performance. Google introduced SRE to maintain large-scale services with high availability and performance goals. The key principles include defining clear SLOs, reducing organizational silos, embracing risk, and automating manual tasks.

GitOps is a paradigm that applies Git's version-control systems to manage infrastructure and application configurations. It emphasizes automation, immutability, and declarative specifications, making it an ideal framework for implementing SRE practices. GitOps enables teams to apply software development principles such as **code review**, **version control**, and **continuous integration/continuous deployment** (**CI/CD**) to infrastructure management, ensuring consistency, reliability, and speed.

SRE principles in a Kubernetes context

Kubernetes, an open source platform for automating deployment, scaling, and operations of application containers, complements the GitOps approach by providing a dynamic and scalable environment for managing containerized applications.

Integrating SRE principles into Kubernetes through GitOps involves several key practices:

- **Automating Deployment and Scaling**: Use GitOps to automate the deployment of Kubernetes resources and applications, ensuring they meet predefined SLOs. Automating scaling decisions based on traffic patterns or system load helps maintain performance and reliability.
- **Error Budgets and Risk Management**: Define error budgets as part of your SLOs to balance the rate of change with system stability. GitOps can help enforce these budgets by automating rollback or deployment procedures based on error budget consumption.
- **Monitoring and Observability**: Implement comprehensive monitoring and observability frameworks to track the health of your services. Kubernetes offers built-in tools such as Prometheus for monitoring and Grafana for visualization, which can be integrated into your GitOps pipeline for real-time insights and alerting.
- **Incident Management**: Automate incident response within your GitOps workflow. Use Kubernetes' self-healing features, such as auto-restarting failed containers and rolling updates, to minimize downtime and maintain service availability.

The integration of SRE principles with GitOps and Kubernetes offers a powerful approach to managing cloud-native applications. By focusing on automation, monitoring, and reliability, teams can achieve higher levels of efficiency and performance. This foundational knowledge serves as a stepping stone toward mastering the complexities of modern IT operations, enabling you to build and maintain resilient and scalable systems in an ever-changing technological landscape.

In the next section, we look at the difference between internal and external observability and how to achieve optimal system performance by balancing the two observabilities.

Understanding internal (white box) versus external (black box) observability

Understanding the nuances of internal versus external observability is crucial for effectively managing and optimizing cloud-native applications. This distinction guides how we monitor and interpret the behavior of systems deployed using GitOps practices in Kubernetes environments. Here, we delve into what constitutes internal and external observability, their respective roles, and how to leverage both to achieve a comprehensive view of your system's *health* and *performance*.

Internal or white box observability explained

Internal observability focuses on the metrics, logs, and traces that are generated from within the system itself. It's akin to looking under the hood of a car while it's running to gauge the health and performance of its engine and other components. In the context of Kubernetes and GitOps, internal observability involves the following:

- **Metrics**: Numerical data that represents the state of your system at any given moment. This could include CPU usage, memory consumption, network I/O, and more.
- **Logs**: Text records of events that have occurred within your system. Logs are invaluable for debugging issues and understanding the sequence of events leading up to an incident.
- **Traces**: Detailed information about requests as they flow through your system, highlighting how different components interact and where bottlenecks or failures occur.

To make it clear what is meant by *internal* in this context, *Figure 12.1* has been created. However, before explaining the diagram in detail, the framework should be explained. This chapter is not about explaining tools such as OpenTelemetry (see [*1*] in the *Further reading* section at the end of the chapter), **Grafana Loki** [*2*], **Prometheus** [*3*], or **Jaeger** [*4*]. Nor is it about the detailed workings of how OpenTelemetry functions and how best to configure it – that would require a chapter or even a book of its own. Later in the chapter, the basic functionality of OpenTelemetry will be outlined, along with the necessary context for GitOps. Therefore, we will view *Figure 12.1* as a black box, focusing on what happens in a Kubernetes cluster and how internal observability relates to it.

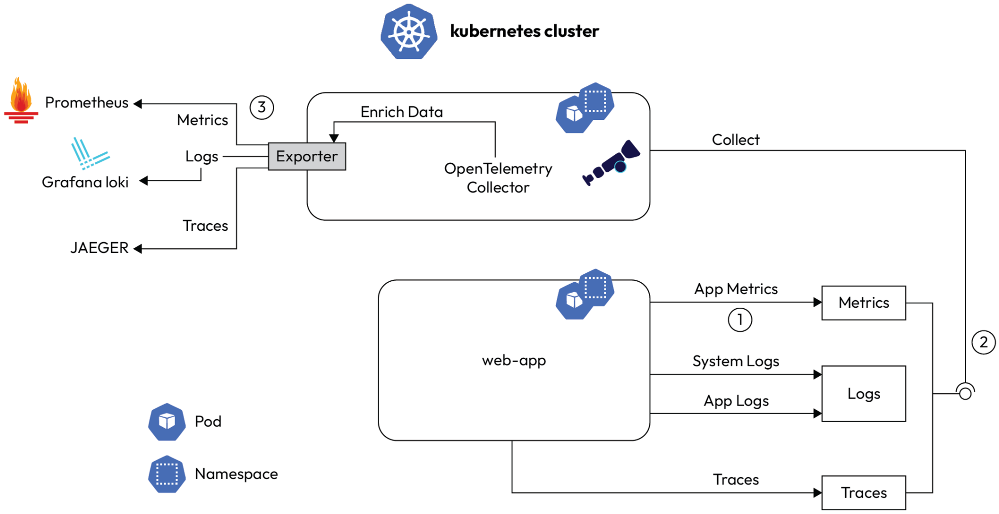


Figure 12.1: Internal observability with OpenTelemetry

> **Important note – logs format**
>
> To ensure that collected system and application logs (*Figure 12.1*) can be effectively utilized, they must be in a standardized and structured format. This format should enable the easy extraction and analysis of relevant information. The analyzed data can then be translated into concrete SLOs that help monitor and ensure the performance and reliability of services.

Here's a brief classification of the tools that will serve as endpoints in *Figure 12.1*:

- **OpenTelemetry** is a unified observability framework for collecting, processing, and exporting telemetry data (logs, metrics, and traces) to help understand software performance and behavior
- **Grafana Loki** is a log aggregation system optimized for storing and querying massive amounts of log data efficiently, integrating seamlessly with Grafana for visualization
- **Prometheus** is an open source monitoring system with a powerful query language designed to record real-time metrics in a time-series database
- **Jaeger** is a distributed tracing system that enables you to monitor and troubleshoot transactions in complex distributed systems

In our example, everything runs within a Kubernetes cluster. For instance, we have a web app, such as an online store, which generates application logs such as which user has logged in, system logs such as unexpected shutdowns, metrics such as the CPU and RAM usage of individual containers, and traces that map the journey of requests through the application's components (*1* in *Figure 12.1*).

Then, the **OpenTelemetry Collector** (*2* in *Figure 12.1*) gathers metrics, logs, and traces and enriches them with relevant data such as timestamps, service names, and environment details. Subsequently, the exporter, which is part of the Collector, makes logs, metrics, and traces available to the appropriate endpoints (*3* in *Figure 12.1*).

For example, the logs are pushed to Grafana Loki, which can then be used by Grafana as a database. The metrics are pushed to Prometheus, which can also serve as a database for Grafana. The traces are pushed to Jaeger, which can likewise act as a database for Grafana. This enables the construction of observability dashboards and alerts in Grafana, providing comprehensive insights into the system's performance and health.

Of course, one could argue that the nodes can be globally distributed, and the collection of logs can also occur across distributed clusters, and so on.

However, the key understanding here is that *internal* refers to the production of logs, metrics, and traces by the running pods on the nodes.

I hope it has become clear at this point what is meant by *internal* and that everything here pertains to the system level on the nodes, the application logs that are written on the nodes, or the network overlay level between the nodes through which packets are sent (service mesh).

External or black box observability defined

External observability, on the other hand, is concerned with understanding the system from an outsider's perspective, primarily focusing on the experience of the end users. It measures the output of your system and how changes within the system affect those outputs. Key aspects include the following:

- **User Experience Metrics**: These metrics gauge the responsiveness and reliability of your application from the user's viewpoint, such as page load times, transaction completion rates, and error rates.
- **Synthetic Monitoring**: Simulated user interactions with your application to test and measure its performance and availability from various locations around the world.
- **Dependency Checks**: Monitoring the health and performance of external services your application relies on. This helps in identifying whether an issue within your system is due to an external dependency.

This section focuses on examining external monitoring. To simplify it for better visualization, we use a service called Uptime Kuma [5] in *Figure 12.2*. For instance, it runs on a Kubernetes cluster and monitors a web app, such as an online store, through a URL accessible on the internet. For our example, to better illustrate the external aspect, we use the `packthub` website.

Getting external observability means using system-wide metrics that are not part of the core functionality of our application. This includes monitoring external services and third-party components such as networking and CPU usage. For example, within a Kubernetes cluster, an internal service in the same namespace can be directly monitored. Alternatively, in a different namespace, monitoring can be done via internal DNS names. This approach does not operate at the system level of the nodes but through permitted accesses in the overlay network using a service mesh with kube-proxy

Important note

Uptime Kuma is a self-hosted monitoring tool that can run on a Kubernetes cluster to keep tabs on services such as web applications. By monitoring accessible URLs over the internet, such as an online store, it provides insights into the uptime and performance of these services from an external perspective. This external monitoring extends beyond merely watching over system metrics at the node level, enabling the observation of services across namespaces through internal DNS names, facilitated by the Kubernetes networking model and service meshes.

In *Figure 12.2*, a simple *HTTP(s)* check is set up, expecting a `200–299` code. This allows for external monitoring of a site and setting up alerts for when the site goes down, the certificate expires, or the response time increases.

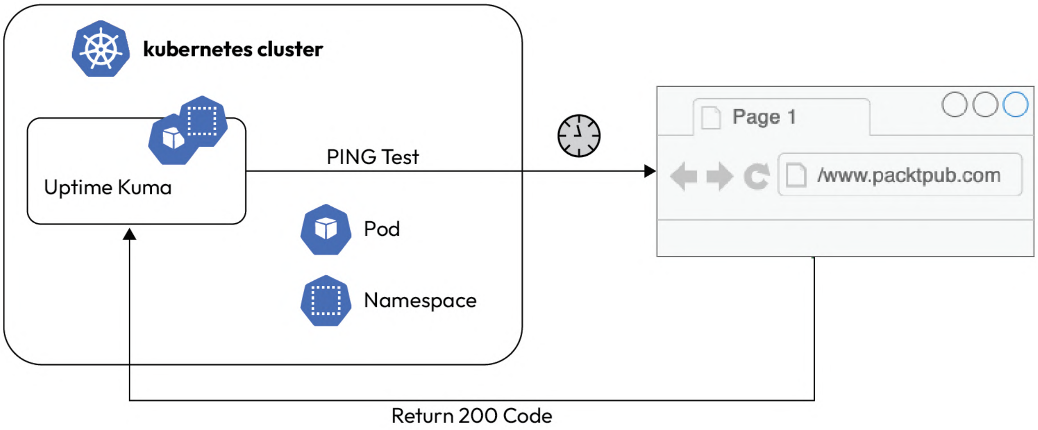
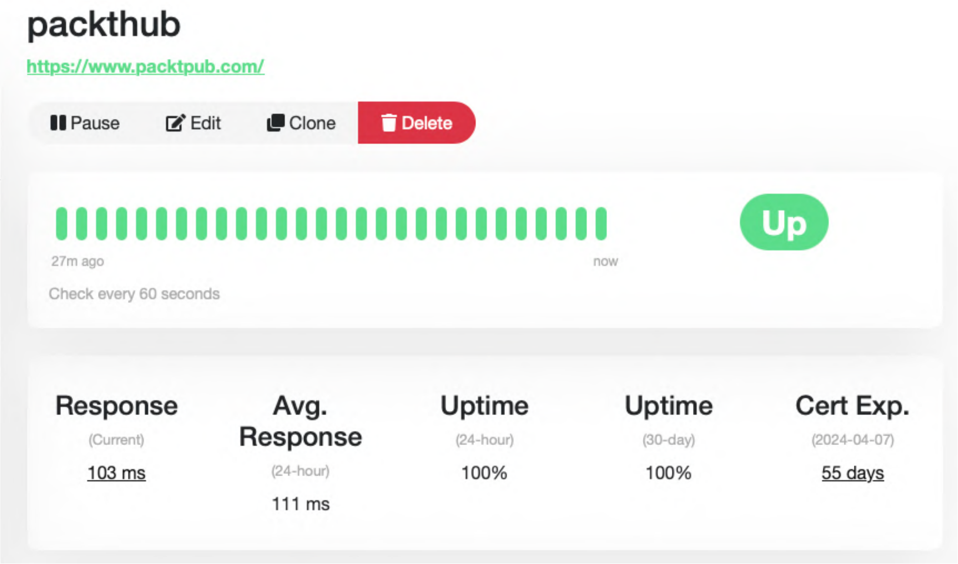


Figure 12.2: External observability with Uptime Kuma

In *Figure 12.3*, you can see the uptime, which is at 100%. Additionally, you can see when the certificate expires and what the response or average response time is.



packthub

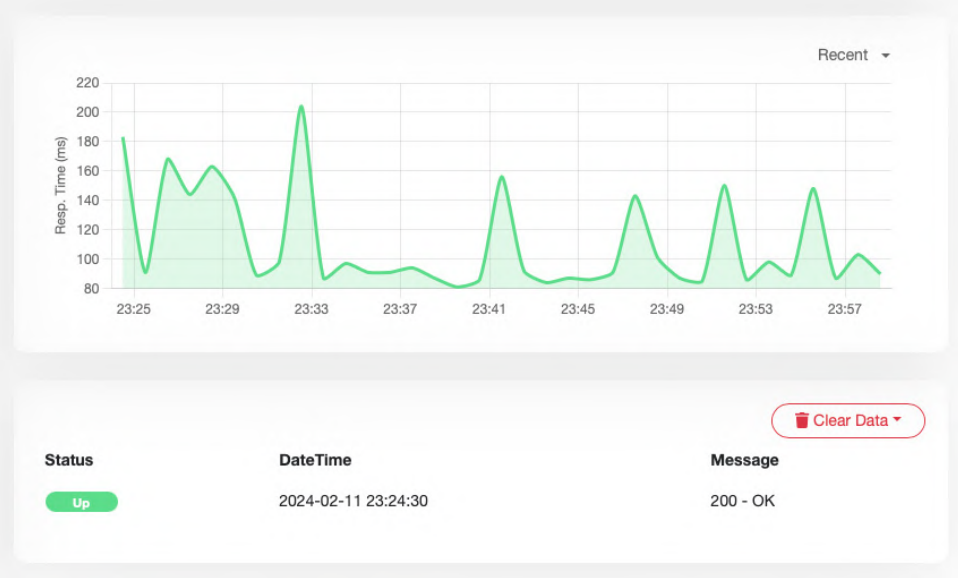
https://www.packtpub.com/

Check every 60 seconds

Response	Avg. Response	Uptime	Uptime	Cert Exp.
(Current)	(24-hour)	(24-hour)	(30-day)	(2024-04-07)
103 ms	111 ms	100%	100%	55 days

Figure 12.3: External observability with Uptime Kuma – dashboard part 1

The second part of the dashboard (*Figure 12.4*) displays the response time for a specific interval, as well as the current **Up** status, which is queried every 60 seconds and returns a `200` code.



Status	DateTime	Message
Up	2024-02-11 23:24:30	200 - OK

Figure 12.4: External observability with Uptime Kuma – dashboard part 2

At this point, it should hopefully have been clarified what is meant by *external* and how this can be implemented with the help of tools such as Uptime Kuma. This allows for the determination of **Service Level Agreements** (**SLAs**), which, depending on the criticality or contract, can be extremely important. Understanding this with alerting is also crucial.

Balancing internal and external observability

To achieve optimal system performance and reliability, it's essential to balance internal and external observability. Internal observability allows you to diagnose and resolve issues within your infrastructure and applications, while external observability ensures that those fixes translate into a better user experience. The integration of GitOps practices into Kubernetes enhances this balance by automating the deployment and management of observability tools and practices:

- **Implementing Observability in GitOps**: Use Git repositories to define your observability stack, ensuring that monitoring, logging, and tracing tools are automatically deployed and configured across all environments consistently.
- **Automated Feedback Loops**: Establish automated feedback loops that integrate observability data into your GitOps workflows. This can help in automatically rolling back changes that negatively impact system performance or user experience.

In conclusion, mastering the interplay between internal and external observability is key to maintaining and optimizing cloud-native applications. By leveraging both perspectives, teams can ensure that their systems are not only running smoothly internally but are also delivering the desired outcomes and experiences for their users. Integrating these observability practices into your GitOps and Kubernetes strategies enables a more proactive, data-driven approach to system management and improvement.

The next section is about useful metrics that can be collected to gain insights into the deployment across multiple stages or clusters.

Exploring SLO-driven multi-stage performance with DORA

In the realm of cloud-native applications, particularly those managed through GitOps in Kubernetes environments, the adoption of SLOs and the integration of DORA metrics offer a strategic framework for achieving and sustaining high performance. This approach combines the precision of SLOs with the insights provided by DORA metrics to guide continuous improvement across multiple stages or clusters of application development and deployment.

At this point (*Figure 12.5*), it is about observing the metrics, which are defined by the company as indicators such as latency, error rate, and so on, and how GitOps helps to measure performance and reliability throughout the CI/CD procedure.

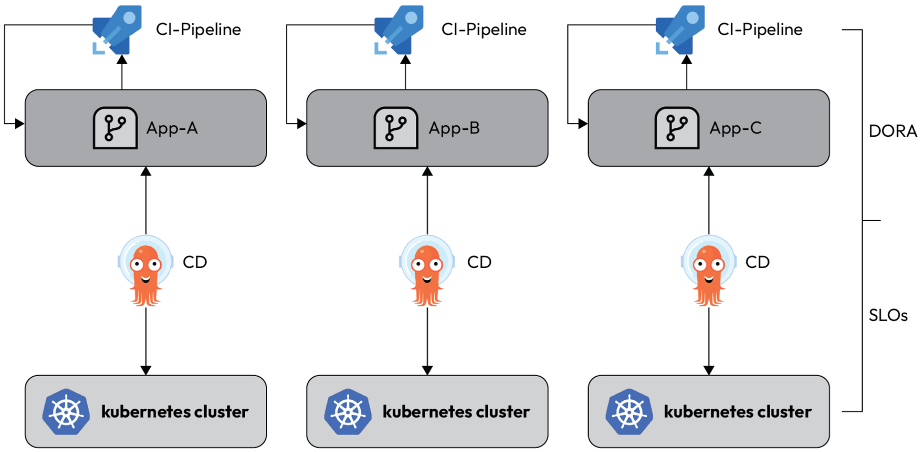


Figure 12.5: How GitOps with DORA and SLOs contribute to observability

The performance and efficiency of an application or its entire stack can be evaluated over several Kubernetes clusters. GitOps plays a crucial role not just in facilitating the distributed deployment of applications throughout these clusters but also in enabling a more profound comprehension of system behaviors, thereby fostering ongoing enhancements in the processes of software delivery.

Let's first understand what an SLO is and the role of DORA:

- **Understanding SLOs**: SLOs are specific, measurable goals that reflect the desired level of service performance and reliability. SLOs are derived from **Service-Level Indicators** (**SLIs**), which are the quantitative measures of service levels, such as latency, error rates, or uptime. Setting SLOs involves determining the acceptable thresholds for these indicators, and balancing the need for reliability with the desire for innovation and rapid development.
- **The Role of DORA Metrics**: The **DORA** metrics (**deployment frequency**, **lead time for changes**, **change failure rate**, and **time to restore service**) serve as key indicators of DevOps performance. These metrics provide insights into the efficiency and effectiveness of software delivery processes, helping teams to identify areas for improvement. In a GitOps context, these metrics can be closely monitored to ensure that the automation and orchestration provided by GitOps workflows are optimizing the software delivery pipeline.

Integrating SLOs with DORA metrics

The integration of SLOs with DORA metrics creates a powerful framework for managing performance in Kubernetes environments:

- **Deployment Frequency and SLOs**: By aligning deployment frequency with SLOs, teams can ensure that they are releasing new features and updates at a pace that does not compromise service reliability.

- **Lead Time for Changes and SLOs**: Monitoring the lead time for changes in relation to SLO performance can help teams streamline their development and deployment processes, ensuring that changes are made swiftly without affecting service quality.
- **Change Failure Rate and SLOs**: Keeping the change failure rate within the thresholds defined by SLOs ensures that most changes enhance rather than detract from service performance.
- **Time to Restore Service and SLOs**: In instances where service levels drop below SLO thresholds, the time to restore service metric becomes crucial. Quick restoration not only meets SLO requirements but also minimizes disruption to end users.

Applying a multi-stage approach

A multi-stage approach to SLO-driven performance leverages DORA metrics at each stage of the GitOps workflow:

- **Planning**: Use SLOs to define performance and reliability goals at the outset of a project or feature development
- **Development**: Integrate DORA metrics into the development process to track progress and ensure that coding practices align with SLOs
- **Deployment**: Automate deployment processes through GitOps to maintain a high deployment frequency while adhering to SLO-defined performance criteria
- **Observation**: Continuously monitor SLIs and DORA metrics post-deployment to assess whether SLOs are being met and identify areas for improvement

Incorporating SLO-driven performance strategies and DORA metrics into GitOps and Kubernetes practices offers a structured path to enhancing the *reliability*, *efficiency*, and *quality* of cloud-native applications. This approach not only optimizes operational processes but also fosters a culture of continuous improvement, ensuring that organizations can adapt and thrive in the fast-paced world of cloud computing. To incorporate this feedback loop, the SRE team should collaborate with application developers to obtain end-to-end improvement.

The following section provides an overview of integrating traces with GitOps, which improves the observability and reliability of cloud-native applications by automating the deployment and configuration of Linkerd via GitOps practices.

Implementing distributed tracing in GitOps with Linkerd

In the complex ecosystem of cloud-native applications, understanding the intricate web of service interactions is crucial for diagnosing issues, optimizing performance, and ensuring reliability. Distributed tracing emerges as a vital tool in this context, offering visibility into the flow of requests across microservices.

Important note – tracing OpenTelemetry versus Linkerd

While OpenTelemetry was mentioned previously for distributed tracing, it is important to explain the difference between OpenTelemetry and Linkerd and their preferred use cases. OpenTelemetry is a collection of tools, APIs, and SDKs used to instrument, generate, collect, and export telemetry data (metrics, logs, and traces) to help understand software performance and behavior.

Linkerd is preferred when you need a robust service mesh to manage and observe service-to-service communication within a Kubernetes environment, particularly when you want seamless integration without modifying your application code.

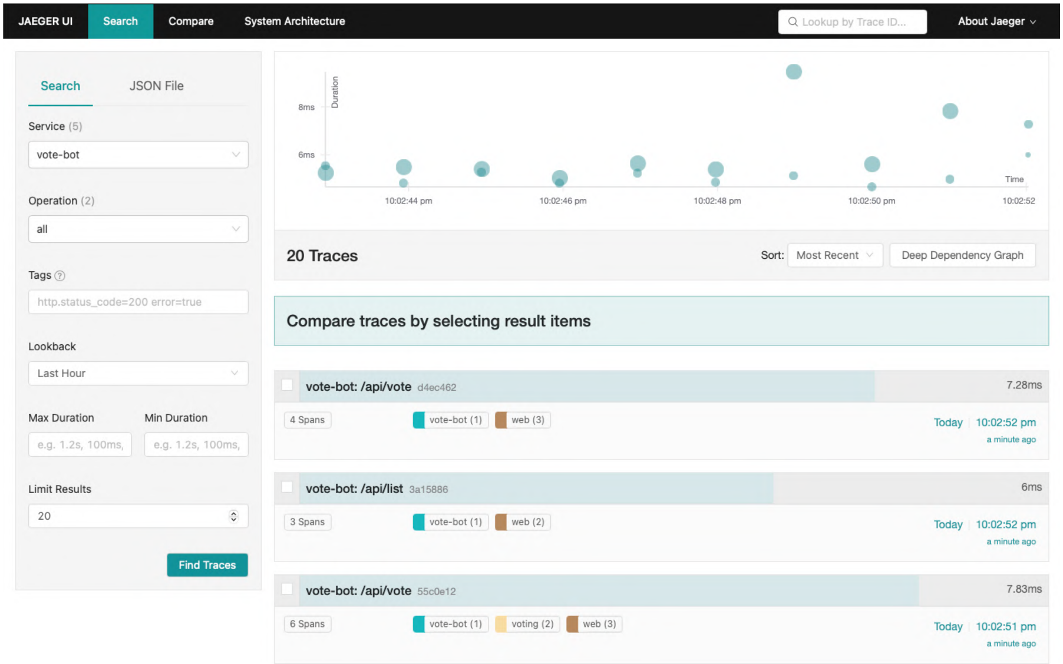
When integrated into a GitOps workflow with Kubernetes, tools such as Linkerd can streamline the deployment and management of distributed tracing, enhancing observability and operational efficiency:

- **Distributed Tracing**: Distributed tracing provides a detailed view of how requests traverse through the various services in a microservices architecture. Each request is tagged with a unique identifier, enabling the tracking of its journey and interactions across services. This visibility is invaluable for pinpointing failures, understanding latencies, and optimizing service interactions.
- **Why Linkerd for Distributed Tracing?**: Linkerd is a lightweight, open source service mesh designed for Kubernetes. It provides critical features such as secure service-to-service communication, observability, and reliability without requiring modifications to your code. Linkerd's support for distributed tracing allows developers and operators to gain insights into the request path, latency contributions by various services, and the overall health of the service mesh.

Integrating Linkerd into your GitOps workflows involves defining the service mesh configuration and the distributed tracing settings within your Git repository. This GitOps approach ensures that the deployment and configuration of Linkerd are fully automated, consistent, and traceable across all environments. Let's break down the integration process:

- **Installation and Configuration**:
- **Define Linkerd Installation**: Use Git to manage the declarative specifications for Linkerd's installation and configuration, ensuring that it aligns with your organization's security and observability requirements.
- **Automate Deployment**: Utilize GitOps with Argo CD to automate the deployment of Linkerd into your Kubernetes clusters. This automation includes the installation of the Linkerd control plane and the injection of Linkerd sidecars into your service pods.
- **Configure Distributed Tracing**:
- **Trace Collector Integration**: Specify configurations for integrating Linkerd with a distributed tracing system (such as Jaeger or Zipkin) within your Git repository. This includes setting up Linkerd to send trace data to the collector.

- **Service Annotation**: Annotate your Kubernetes service manifests to enable tracing with Linkerd. These annotations instruct Linkerd sidecars to participate in distributed tracing by forwarding trace data.
- **Visualization and Analysis**:
- **Leverage Tracing Dashboards**: Utilize the integrated tracing dashboards provided by Jaeger (*Figure 12.6*) or Zipkin to visualize and analyze trace data. These tools offer powerful capabilities to filter, search, and drill down into the details of individual traces.



Service (5): vote-bot

Operation (2): all

Tags: http.status_code=200 error=true

Lookback: Last Hour

Max Duration: e.g. 1.2s, 100ms,

Min Duration: e.g. 1.2s, 100ms,

Limit Results: 20

20 Traces

Sort: Most Recent

☐ vote-bot: /api/vote d4ec462 7.28ms
4 Spans vote-bot (1) web (3) Today 10:02:52 pm a minute ago

☐ vote-bot: /api/list 3a15886 6ms
3 Spans vote-bot (1) web (2) Today 10:02:52 pm a minute ago

☐ vote-bot: /api/vote 55c0e12 7.83ms
6 Spans vote-bot (1) voting (2) web (3) Today 10:02:51 pm a minute ago

Figure 12.6: Jaeger UI for distributed tracing of service calls in a Kubernetes cluster

Implementing distributed tracing with Linkerd in a GitOps framework significantly enhances the observability and reliability of cloud-native applications. By automating the deployment and configuration of Linkerd through GitOps, teams can ensure a consistent and scalable approach to monitoring microservices interactions. This capability is essential for maintaining high-performance, resilient applications in the dynamic landscape of Kubernetes environments.

In the next part of the chapter, we will look at how tools such as Uptime Kuma and OpenTelemetry can help to enable both external and internal observability with the help of GitOps.

Implementing monitoring in GitOps with tools such as Uptime Kuma and OpenTelemetry

In the dynamic and distributed world of cloud-native applications, effective monitoring and alerting are essential for ensuring system reliability, performance, and security. Integrating these practices within a GitOps framework not only streamlines the deployment and management of monitoring tools but also aligns operational practices with the principles of **Infrastructure as Code** (**IaC**). This approach, particularly when leveraging powerful tools such as OpenTelemetry, provides a cohesive and automated methodology for observing system behaviors and responding to incidents from the internal point of view. But you also have tools, such as Uptime Kuma, that enable the external observability of services.

Monitoring in a GitOps framework involves collecting, analyzing, and displaying metrics and logs from across your infrastructure and applications. This data-driven approach allows teams to understand system performance, identify trends, and detect anomalies. By defining monitoring configurations and dashboards as code within a Git repository, teams can apply version control, review processes, and automated deployments to monitoring infrastructure, ensuring consistency and reliability. The distribution of these dashboards, for example, can be deployed across an *N* number of clusters using GitOps.

Uptime Kuma – the external watchdog for your online services

Uptime Kuma is an open source monitoring tool designed to track the uptime, downtime, and performance of various services and websites. It's a self-hosted solution, meaning it runs on your own hardware or cloud infrastructure, providing full control over your monitoring environment. Uptime Kuma offers a user-friendly interface and is becoming a popular choice among developers and system administrators for its simplicity, flexibility, and cost-effectiveness. Uptime Kuma operates by sending requests to your services or websites at regular intervals and monitoring their responses to determine their availability and response time.

In comparison, **Datadog** and **Prometheus with Grafana** offer different approaches to monitoring. Datadog is a comprehensive, cloud-based monitoring and analytics platform that provides end-to-end visibility into the performance of your applications, infrastructure, and logs. It is particularly known for its integration capabilities with a wide range of third-party services and its advanced analytics features.

Prometheus, on the other hand, is an open source monitoring and alerting toolkit designed primarily for reliability and scalability. It excels at collecting and storing time-series data, which can then be visualized using Grafana, a powerful open source platform for monitoring and observability. Grafana allows users to create customizable dashboards to visualize metrics collected by Prometheus. While Prometheus supports monitoring various protocols such as HTTP, HTTPS, DNS, TCP, and ICMP ping through the use of exporters such as Blackbox Exporter, it requires additional setup and configuration to achieve this.

In the following subsections, we take a look at various key features and strengths of Uptime Kuma in order to gain a better understanding of the tool.

Key Features

The key features of Uptime Kuma are as follows:

- **Multi-Protocol Support**: Uptime Kuma supports monitoring via HTTP(S), TCP, DNS, and more
- **Customizable Alerts**: Users can configure alerts based on various criteria and choose their preferred notification methods
- **Performance Metrics**: Tracks response times, allowing users to monitor the performance of their services in addition to their availability
- **SSL Certificate Monitoring**: It can monitor the expiration of SSL certificates, alerting users before their certificates expire
- **Ping Monitoring**: Offers the ability to monitor the availability and latency of servers using ICMP ping

Core functionalities

Here's a breakdown of its core functionalities and how it works:

- **Monitoring Services**: Uptime Kuma can monitor various types of services including HTTP(S) websites, TCP ports, HTTP(s) endpoints with specific expected statuses, DNS records, and more. It allows users to configure the monitoring intervals, timeouts, and specific conditions that define the availability of each service.
- **Alerts and Notifications**: When a service goes down or meets specific conditions set by the user (e.g., high response time), Uptime Kuma can send alerts through various channels. It supports numerous notification methods including email, SMS (through third-party services), Telegram, Discord, Slack, and more, ensuring that users are promptly informed about status changes.
- **Status Page**: Uptime Kuma provides a public or private status page that displays the uptime status of all monitored services. This page can be used to communicate with team members or customers about the current status of various services, enhancing transparency and trust.
- **Detailed Reporting**: It offers detailed reports and analytics on the uptime, downtime, and response times of monitored services. These insights can help identify patterns, potential issues, and areas for improvement in your infrastructure or application performance.
- **Easy Setup and Configuration**: Setting up Uptime Kuma is straightforward. It can be deployed on various platforms including Docker, which makes it easy to install and run on most environments. The web-based interface provides a simple and intuitive way to add and configure the services you want to monitor.

Uptime Kuma is a versatile and user-friendly tool for monitoring the uptime and performance of websites and services. Its self-hosted nature gives users full control over their monitoring setup, making it a secure and customizable option for businesses and individual users alike. With its broad protocol support, flexible alerting system, and detailed analytics, Uptime Kuma provides a comprehensive solution for ensuring the reliability and performance of online services.

OpenTelemetry – a unified observability framework

OpenTelemetry is an open source observability framework designed to provide comprehensive insights into the behavior of software applications. It achieves this by collecting, processing, and exporting telemetry data – specifically logs, metrics, and traces. OpenTelemetry aims to make it easy for developers and operators to gain visibility into their systems, helping to debug, optimize, and ensure the reliability of applications across various environments.

Key features

At the core of OpenTelemetry is **instrumentation**, a process that involves integrating OpenTelemetry libraries or agents into your application code or runtime environment. This integration allows OpenTelemetry to capture detailed telemetry data from the application:

- **Manual Instrumentation**: Developers can manually instrument their code using the OpenTelemetry API. This involves adding specific code snippets that generate telemetry data such as custom metrics, logs, or traces for specific operations within the application.
- **Automatic Instrumentation**: OpenTelemetry provides auto-instrumentation agents that can be attached to an application. These agents automatically capture telemetry data without requiring modifications to the application code, ideal for legacy systems or for common libraries and frameworks.

OpenTelemetry collects three main types of telemetry data:

- **Logs**: Records of discrete events that have occurred within the application, providing detailed context about operations, errors, and other significant activities
- **Metrics**: Numerical data that represents the measurements of different aspects of the application and system performance over time, such as request rates, error counts, and resource utilization
- **Traces**: Detailed information about the execution paths of transactions or requests as they travel through the application and its services, showing how different parts of the system interact

Core functionalities

Here's a breakdown of its core functionalities and how it works:

1. **Processing and Enrichment**: Once telemetry data is collected, OpenTelemetry can process and enrich this data. Processing may include aggregating metrics, filtering logs, or adding additional context to traces to make the data more useful and meaningful. This step is crucial for reducing noise and enhancing the relevance of the data collected.
2. **Exporting Data**: OpenTelemetry supports exporting telemetry data to a wide range of backend observability platforms where the data can be analyzed, visualized, and monitored. It provides exporters for popular monitoring solutions, cloud-native observability tools, and custom backends. The OpenTelemetry Collector, a component that can be deployed as part of your infrastructure, plays a key role in this process. It can receive, process, and export telemetry data from multiple sources, acting as a central hub for observability data.
3. **Analysis and Action**: The final step in the OpenTelemetry workflow involves analyzing the exported telemetry data using observability platforms. These platforms allow teams to visualize data through dashboards, set up alerts based on specific conditions, and derive insights that can inform troubleshooting, performance optimization, and decision-making processes.

Implementing monitoring with OpenTelemetry

Here's how you can implement monitoring with OpenTelemetry in GitOps:

- **Define Monitoring Configuration as Code**: Store OpenTelemetry Collector configurations in your Git repository, specifying how data is collected, processed, and exported. This setup ensures that monitoring configurations are subject to the same review and deployment practices as application code.

- **Automated Deployment of Monitoring Infrastructure**: Use GitOps pipelines to automatically deploy and update OpenTelemetry Collectors and other monitoring components across your Kubernetes clusters. This automation guarantees that monitoring infrastructure is consistently deployed across all environments.

- **Instrumentation of Applications**: Incorporate OpenTelemetry SDKs into your application code to capture detailed performance metrics and traces. Managing SDK configurations through Git allows for controlled updates and consistency across services.

OpenTelemetry provides a unified and vendor-neutral framework to capture, process, and export telemetry data, enabling developers and operators to achieve deep observability in their applications. By streamlining the collection of logs, metrics, and traces, and making this data easily exportable to analysis tools, OpenTelemetry facilitates a better understanding of software performance and behavior, ultimately improving the reliability and efficiency of applications.

OpenTelemetry offers a single, vendor-agnostic framework for collecting traces, metrics, and logs from applications and infrastructure. It simplifies the instrumentation of code and the deployment of agents, providing a standardized way to gather telemetry data that can be analyzed by various observability platforms.

The next part deals with the possible alerting strategies that can be integrated into a GitOps framework.

Looking at alerting strategies in a GitOps framework

Effective alerting is about notifying the right people with the right information at the right time. Within a GitOps framework, alerting rules and notification configurations are defined as code and managed alongside application and infrastructure configurations:

- **Define Alerting Rules as Code**: Store definitions for alerting rules within your Git repository, specifying the conditions under which alerts should be triggered. This approach enables version control and automated deployment of alerting rules, ensuring that they are consistently applied.
- **Integration with Notification Channels**: Configure integrations with notification channels (such as email, Slack, or PagerDuty) as part of your GitOps workflows. This ensures that alert notifications are reliably sent to the appropriate teams or individuals.
- **Feedback Loops for Continuous Improvement**: Implement feedback loops that use monitoring and alerting data to inform development and operations practices. Incorporating this feedback into your GitOps processes facilitates continuous improvement of both application performance and operational efficiency.

Figure 12.7 visualizes how GitOps can be used with Argo CD to deploy rules and notification channels as code across different clusters:

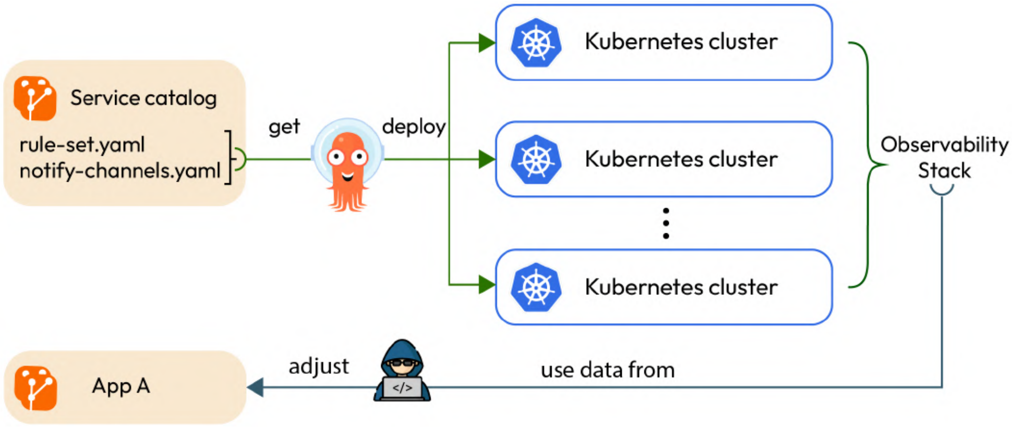


Figure 12.7: Continuous improvement with GitOps and observability

The developers or platform engineers can use the information from the observation in the form of a feedback lock to optimize their applications. This can then be used, for example, to define new rules if something has been overlooked and, thanks to the GitOps approach, it can be rolled out across an *N* number of clusters.

Some relevant alerting rules

Here are a few insights from different projects on how platform engineers' teams define rulesets and deploy Kubernetes clusters everywhere to help developers better understand their applications and live the SRE approach:

- **Dynamically Adjust Thresholds**: Implement rules that adjust thresholds based on historical data or current load to minimize false alarms and increase the relevance of notifications.
- **Monitor Dependencies**: Set up rules to monitor dependencies between services and components to proactively identify potential issues before they impact user experience.
- **Ensure Log Completeness**: Establish rules that check for the completeness and structuring of logs. This helps improve the effectiveness of troubleshooting and analysis.
- **Resource Utilization Alerts**: Create rules to monitor the utilization of resources such as CPU, memory, and disk space. Set alerts for when usage approaches critical thresholds, indicating potential overcommitment or resource exhaustion.
- **Latency Monitoring**: Implement rules to monitor the latency of critical operations or API calls. High latency can be an early indicator of system strain or overcommitment in processing resources.
- **Node Overcommitment in Kubernetes**: It's one of my absolute favorite alerting rules, which has already helped an enormous number of teams, especially those with many small, tailored clusters. It helps prevent performance degradation and ensure the reliability of applications running on Kubernetes by monitoring and alerting on node overcommitment. By setting up alerting rules for node overcommitment, teams can detect when the demand on a node exceeds its capacity, allowing them to take preemptive actions to prevent performance degradation and ensure that applications remain reliable. This approach not only improves system stability but also supports optimal resource utilization, making it a highly valuable practice for maintaining the health and efficiency of Kubernetes clusters.

Diving deeper into node overcommitment in Kubernetes

I'll break down the **node overcommitment in Kubernetes** rule a little further here so that it becomes clear why such a simple rule and the associated alerting are attached to it:

- **Sustainability in Resource Utilization**: Monitoring node overcommitment can lead to more efficient use of computational resources, reducing energy consumption and contributing to the sustainability goals of an organization. Efficient resource utilization minimizes unnecessary workloads and idle resources, aligning with eco-friendly practices.

> **Important note**
>
> **FinOps**, or **Financial Operations**, is a practice that combines systems, best practices, and culture to help organizations manage and optimize cloud costs more effectively. It focuses on creating a collaborative cross-functional team approach that brings financial accountability to the variable spend model of the cloud, enabling faster, more informed business decisions.

- **FinOps and Cost Optimization**: By preventing overcommitment and optimizing resource allocation, organizations can adhere to FinOps principles, ensuring that cloud spending is aligned with business value. Alerting on node overcommitment helps avoid over-provisioning and underutilization, leading to significant cost savings and more predictable cloud expenses.
- **Enhanced Application Performance**: Proactively managing node resources ensures that applications have access to the necessary computational power when needed, enhancing user experience and application performance.
- **Reliability and Availability**: Avoiding the overcommitment of nodes contributes to the overall reliability and availability of services, as resources are balanced, and potential points of failure are minimized.
- **Scalability**: Effective monitoring and management of node overcommitment prepare the infrastructure for scalability, allowing for smooth scaling operations that accommodate growing workloads without compromising performance or incurring unnecessary costs.

Integrating these considerations into Kubernetes resource management practices not only addresses immediate operational concerns but also positions organizations to better align their technical strategies with environmental sustainability, financial accountability, and long-term scalability.

Adopting monitoring and alerting strategies within a GitOps framework provides a systematic and automated approach to observability. Leveraging tools such as OpenTelemetry within this framework enhances the granularity and utility of telemetry data, driving more informed decision-making and operational resilience. This methodology not only ensures high levels of system performance and reliability but also fosters a culture of continuous improvement and operational excellence in cloud-native environments.

The last section of the chapter is about how scaling observability can be achieved with the help of GitOps.

Scaling observability with GitOps

As organizations grow and their technology stacks become more complex, ensuring effective observability at scale becomes a formidable challenge. Cloud-native architectures, microservices, and dynamic environments, all managed through practices such as GitOps, introduce a level of complexity that traditional observability strategies struggle to accommodate. This section explores the advanced practices, tooling, and organizational strategies necessary to achieve comprehensive observability at scale, ensuring that systems are not only observable but also manageable, regardless of their size and complexity.

Scaling observability components

The foundation of observability at scale lies in efficiently managing the three pillars: *logging*, *monitoring*, and *tracing*. Each of these components must be scaled thoughtfully to handle the vast amounts of data generated by large, distributed systems without compromising the speed or accuracy of insights derived from the data. Efficient data management is not only essential for technical performance but also for cost management, as the volume of data stored and analyzed can significantly impact project expenses.

In the following, we look at how logging, monitoring, and tracing at scale behave:

- **Logging at Scale**: Implement structured logging to standardize log formats across services, making them easier to aggregate and analyze. Utilize centralized logging solutions that can handle high volumes of data, providing powerful search and analysis tools to quickly derive insights from logs.
- **Monitoring at Scale**: Leverage scalable monitoring solutions that support high-frequency data collection and can dynamically adjust to the changing topology of cloud-native environments. Adopt service meshes such as Linkerd or Istio, which provide built-in observability features for Kubernetes clusters, reducing the overhead on individual services.
- **Tracing at Scale**: Distributed tracing becomes critical in microservices architectures to track the flow of requests across services. Solutions such as Jaeger, Zipkin, or those provided by service meshes, integrated with OpenTelemetry, offer scalable tracing capabilities. Implement trace sampling strategies to balance the granularity of trace data with the overhead of collecting and storing that data.

Advanced tooling for observability at scale

Adopting the right tools is crucial for managing observability at scale. Tools such as *Prometheus for monitoring*, *Elasticsearch for logging*, and *OpenTelemetry for instrumentation* are chosen because they are open source, follow OpenTelemetry guidelines, and provide robust, community-supported solutions. When integrated into a GitOps workflow, these tools ensure that observability infrastructure can be deployed, scaled, and managed as efficiently as the applications and services they monitor.

We can use GitOps practices to dynamically configure observability tools based on the current needs and scale of the system. This includes the automatic scaling of data storage, processing capabilities, and the deployment of additional monitoring or tracing agents as the system grows.

Another good idea is to incorporate AI and ML techniques for anomaly detection and predictive analytics, helping to sift through the noise in large datasets and identify emerging issues before they impact users.

In the next subsection, we will examine how to cultivate a culture of observability through cross-functional collaboration, continuous education, and strategic feedback loops.

Organizational strategies for effective observability

Observability at scale is not just a technical challenge but also an organizational one. Cultivating a culture of observability requires involvement from across the organization, from developers to operations to business stakeholders:

- **Cross-Functional Teams**: Encourage collaboration between development, operations, and business teams to ensure that observability goals align with business objectives and operational requirements. This collaboration fosters a shared understanding of what needs to be observed and why.
- **Education and Advocacy**: Invest in training and resources to ensure that teams understand the importance of observability and how to effectively leverage tools and practices at scale. Advocacy for observability as a fundamental aspect of system design and operation ensures its integration throughout the development life cycle.
- **Continuous Feedback Loops**: Establish feedback loops that bring observability data back into the development process, informing decision-making and driving continuous improvement. This includes using observability data to refine performance baselines, adjust alerting thresholds, and prioritize development efforts.

Achieving observability at scale requires a comprehensive approach that extends beyond just tooling to encompass organizational practices and culture. By integrating scalable observability tools with GitOps workflows, leveraging advanced data processing techniques, and fostering a culture of collaboration and continuous improvement, organizations can ensure that their systems remain observable, manageable, and performant, regardless of scale. This holistic approach not only addresses the technical challenges of observability at scale but also aligns observability practices with broader business objectives, driving value and competitive advantage in today's dynamic and complex technology landscape.

In the next part, I'll share insights to help you decide which tools might be useful for your setup.

Selecting the right observability tools for specific use cases

Choosing the right observability tools depends on your specific monitoring needs and desired outcomes. It's often not easy, as many use cases sound similar but have different requirements. Here are some insights to help you combine different tools for the optimal observability stack. The goal is not to find the perfect tool but to focus on the different layers of observability. To clarify the understanding and different requirements for observability, I've added possible stakeholders. This list is not exhaustive but includes key stakeholders and their interests based on various real projects. I hope these insights will help you get the most out of your observability setup.

> **Important note**
>
> This section focuses less on GitOps itself and more on when to use which tools, providing a comprehensive view of observability. Many questions may arise, such as, "*I understand GitOps with observability, but which tools should I use and when?*" By exploring various use cases, we hope to give you a sense of which tool is the right one for each specific scenario.

Let's explore some common scenarios and the tools that best fit each use case.

Monitoring the availability of applications and the expiry of certificates:

- **Use Case**: You want to ensure your application is available, assign SLAs, monitor SSL certificate expiry, and receive alerts.
- **Recommended Tool**: Uptime Kuma
- **Explanation**: Uptime Kuma is ideal for this scenario as it supports multi-protocol monitoring (HTTP(S), TCP, DNS), and provides customizable alerts for downtime and SSL certificate expiration. It is user-friendly and cost-effective, making it a good choice for straightforward uptime monitoring.

- **Stakeholders**:
  - **Service Owner**: Monitors overall service health to ensure that all services are running
  - **Developer**: Understands how changes impact the user experience and diagnoses issues in production
  - **Customer**: Ensures that the service meets the provided SLAs

Monitoring Resource Utilization and Application Logs:

- **Use Case**: You need to track metrics such as CPU, RAM, and storage usage, and analyze application logs. You also want to be notified when these metrics exceed certain thresholds.
- **Recommended Tools**: Prometheus + Grafana-Stack + Alertmanager
- **Explanation**: Prometheus excels at collecting and storing time-series data, which includes resource utilization metrics. Grafana-Stack not only provides robust visualization capabilities, allowing you to create detailed dashboards, but also offers the ability to collect and enrich logs. Alertmanager integrates with Prometheus to handle alerting based on the defined thresholds.
- **Stakeholders**:
  - **Site Reliability Engineer**: Monitors system health and resource usage to ensure reliability and performance
  - **Developer**: Uses logs and metrics to debug and optimize application performance
  - **DevOps Engineer**: Automates monitoring and alerting to streamline operations

Detecting Unusual Application Behavior on Host System:

- **Use Case**: You want to be notified if an application performs unauthorized actions on the host system, such as opening a shell.
- **Recommended Tools**: Falco + Prometheus + Alertmanager
- **Explanation**: Falco is a runtime security tool that detects anomalous behavior in your applications and host systems. It integrates with Prometheus for monitoring and Alertmanager for handling alerts, providing a comprehensive solution for detecting and responding to security threats.
- **Stakeholders**:
  - **Security Team**: Monitors and responds to potential security threats
  - **System Administrator**: Ensures system integrity and compliance

Tracing Packet Loss and Identifying Bottlenecks:

- **Use Case**: You need to understand why packets are being lost and where requests are experiencing delays, without modifying the application code.
- **Recommended Tools**: Linkerd + Jaeger
- **Explanation**: Linkerd is a lightweight service mesh that provides observability into service-to-service communication without requiring code changes. Jaeger is a distributed tracing system that integrates with Linkerd to trace requests through your microservices, helping you identify and optimize performance bottlenecks.
- **Stakeholders**:
  - **Network Engineer**: Diagnoses and resolves network-related issues
  - **Developer**: Identifies and fixes performance bottlenecks in the application
  - **Site Reliability Engineer**: Identifies and fixes performance bottlenecks to ensure system reliability

Customizing and Enriching Logs via an SDK:

- **Use Case**: You want to adjust and enrich application logs using an SDK.
- **Recommended Tool**: OpenTelemetry
- **Explanation**: OpenTelemetry provides comprehensive support for collecting, processing, and exporting telemetry data (logs, metrics, and traces). It allows for both manual and automatic instrumentation of your code, enabling detailed customization and enrichment of logs.
- **Stakeholders**:
  - **Developer**: Customizes and enriches logs for better debugging and performance monitoring
  - **Site Reliability Engineer**: Customizes and enriches logs to ensure system reliability and performance

All the tools mentioned are open source. This is important because using open source tools ensures that we avoid vendor lock-in, rely on a strong community, and have the flexibility to contribute and receive help as needed.

In the next section, let's understand how observability with GitOps affects our daily work in the company.

Enterprise-level best practices with observability and GitOps

I don't know whether these are really the best practices for enterprise. I can only say that what is shared in this section is good practice that works in many different projects for us and share these insights with you. In this section, I will provide detailed insights into how GitOps maximizes the efficiency and effectiveness of our observability stack.

In the following, we look at how different stakeholders use the GitOps approach to generate added value for themselves.

- **Service Owner**: GitOps allows service owners, responsible for multiple services across different clusters, to define their Grafana dashboards once and roll them out as *ConfigMaps* across all relevant clusters independently. This approach also applies to the alerts for their respective services.
- **Platform Teams**: GitOps enables us, as a platform team, to deploy our monitoring stack irrespective of the number of clusters. This capability allows us to efficiently monitor our infrastructure and the services provided, expand the stack as needed, and maintain it effortlessly.
- **Trainees**: For instance, our trainees can define their own Grafana dashboards to integrate sensors that measure the clearance height under bridges in Hamburg. These dashboards are defined once and can then be rolled out across all necessary clusters.
- **Service Providers**: These are responsible for services such as RabbitMQ (message broker) on multiple clusters and use the GitOps approach to deploy alert configurations across all clusters and integrate them into their external alerting systems.
- **Developers**: These use a similar approach as the service providers to deliver their software with the corresponding dashboards and alerts.
- **Security Teams**: An emerging but promising practice is involving security teams in observability processes. However, this does not work because, for example, security teams in our projects are used to regulating rules independently in the company's interests. To achieve this, they use their own tools, which cause additional overheads.
- **FinOps Departments**: This currently does not work because the observability topic and the Kubernetes platform are both technically too complex. For example, creating budget alerts over YAML manifests based on the calculated costs of a *Namespace* corresponding to a project is challenging.

Currently, in most projects, platform teams handle security aspects by rolling out Falco rules and Prometheus alert configurations, for instance, to detect unwanted syscalls such as shell openings on a node, and trigger alerts accordingly. However, this often increases the responsibility burden and can result in alerts not being thoroughly investigated.

The GitOps approach significantly enhances our observability practices by saving time and costs, providing our stakeholders with the necessary autonomy, and boosting overall motivation. By creating an environment where teams can manage their observability configurations without the constant back-and-forth of tickets, we foster independence and a healthy error culture. Teams understand that if something goes wrong, a simple commit revert will restore the previous state, making the process more resilient and reliable. This approach transforms collaboration across different departments, ensuring that observability is seamlessly integrated into our development and operational workflows.

Intrinsic motivation drives a fundamental technical understanding of observability within the company, which is a significant advantage. This leads to better engagement and innovation. Empowering all employees to contribute to and improve the observability stack makes the organization more resilient, adaptable, and better prepared to tackle new challenges. This collaborative approach not only enhances team efficiency but also promotes a culture of continuous improvement and shared responsibility.

And to be honest, I really like the way the culture changes! This is a point that we could not achieve with traditional DevOps with CI/CD, although DevOps ironically describes exactly that of the culture.

Summary

This comprehensive chapter traversed the intricate landscape of observability within cloud-native applications, emphasizing its critical role across various dimensions of GitOps and Kubernetes environments. Starting with the foundational principles of SRE, we explored how these practices are seamlessly integrated into GitOps workflows, enhancing the reliability and performance of Kubernetes deployments. The distinction between internal and external observability was clarified, underscoring the importance of a balanced approach for comprehensive system insight. We further delved into the strategic implementation of SLO-driven performance metrics aligned with DORA indicators, offering a structured framework for continuous improvement. Through the lens of Linkerd, we examined the deployment of distributed tracing within GitOps, highlighting the enhanced visibility and diagnostic capabilities it brings to microservices architectures. Monitoring and alerting strategies, empowered by tools such as OpenTelemetry, were discussed to establish proactive incident management and system health monitoring. Finally, scaling observability to meet the demands of growing and complex systems was addressed, showcasing the necessity of advanced tooling, organizational strategies, and a culture that prioritizes observability. This chapter encapsulated a holistic view of implementing and scaling observability in modern cloud-native ecosystems, ensuring that systems are not only observable but also resilient and efficient.

But the most important thing to learn should hopefully be that observability is versatile and not just logs, metrics, and traces!

In the next chapter, we will look at the security part with GitOps and take a look at the attack possibilities with Argo CD and how these can be minimized.

References

- [*1*] `https://opentelemetry.io`
- [*2*] `https://github.com/grafana/loki`
- [*3*] `https://github.com/prometheus/prometheus`
- [*4*] `https://github.com/jaegertracing/jaeger`
- [*5*] `https://github.com/louislam/uptime-kuma`

13

Security with GitOps

Implementing GitOps offers several benefits for the security of software development processes. By using Git as a central source for configuration and code, the integrity and traceability of environments can be ensured. In this chapter, we will explore the various aspects of security in the context of GitOps.

We will begin by examining the well-known **Cockpit** and **Fleet** approaches and use them to highlight various security considerations when using Argo CD. Next, we will focus on **Kyverno** as a **policy engine** that acts as a gatekeeper, defining what is allowed on the cluster and which deployments are even permitted.

The topic of permissions plays a central role in the context of GitOps. Therefore, we will discuss the secure handling of secrets and introduce two tools that have already been successfully adopted in the industry.

From the perspective of the platform team, we will consider the provisioning of context information such as cert-manager, Ingress Controllers, and so on for developers. Here, we will use the proven **Kubernetes Service Catalog** principle and explore its secure application and updating.

Finally, we will take a look at the **KubeClarity** tool, which provides clarity about vulnerabilities in running applications on various levels, such as the following:

- **Configuration**: Misconfigurations of Kubernetes resources
- **Images**: Vulnerabilities in the container images used
- **Code**: Security vulnerabilities in the application code

In addition, we will briefly dive into the OS level or kernel level and look at the **Falco** tool. This tool can be used to detect suspicious activity on the host, such as when operations are executed that should not be executed.

The goal of this chapter is to illuminate the different perspectives on different layers and thus gain a better understanding of how GitOps can help teams improve security.

We will cover the following main topics in the chapter:

- Hardening declarative GitOps CD on Kubernetes
- Committing everything to Git? What about Secrets?
- Leveraging a policy engine for policy-as-code practices
- Automating security scanning and compliance
- Keeping your platform catalog up-to-date

Hardening declarative GitOps CD on Kubernetes

In this section, we'll delve into the practices essential for hardening declarative GitOps **continuous delivery** (**CD**) on Kubernetes, focusing on enhancing security and reducing the risk of misconfigurations – a leading cause of cloud breaches. The shift toward cloud-native technologies has simplified the complexity of systems, operational theories, and skillsets, offering a clearer and more secure framework for building and managing applications. However, the security of these systems extends beyond just the software development and supply chain aspects; it crucially involves addressing the configuration vulnerabilities.

Addressing configuration vulnerabilities

Cloud-native technologies streamline skillsets, operational theories, and system complexities, enhancing the understandability and security of system architectures. However, the primary cloud security risk remains misconfiguration, often overlooked amid the focus on developmental and supply chain security. The adoption of the GitOps pattern, particularly for progressive application delivery, has become widespread, offering a more secure alternative to traditional direct-to-production build servers. This method employs a Git repository for changes, enabling pre-deployment security assessments and minimizing privilege escalation and configuration drift.

The Cloud Native Computing Foundation commissioned **ControlPlane** [*1*] to conduct a detailed threat modeling analysis on Argo CD, a tool emblematic of the GitOps approach, focusing on its deployment in a multi-tenant Kubernetes environment. This analysis revealed 19 identified threats, with 6 classified as high priority, emphasizing the need for rigorous security measures. Recommendations for hardening include enhancing password management, integrating **single sign-on**, and applying strict **Role Based Access Contro** (**RBAC)** principles to limit access to sensitive information.

The report from ControlPlane [*1*] also provides visual attack trees for the most critical threats, aiding stakeholders in understanding and mitigating risks, alongside a comprehensive overview of the Argo CD deployment architecture. It includes Terraform code for replicating the setup, ensuring that security controls can be effectively validated. These measures, grounded in the report's findings, aim to fortify the security of Argo CD deployments, aligning them with organizational security standards and maximizing the benefits of using GitOps within cloud-native ecosystems.

We will integrate parts from the report and combine them with the well-known Cockpit and Fleet approach. The report in question is from 2023, featuring *Argo CD version 2.67*, and some issues may have already been resolved with newer Argo CD releases. At the time of writing, *Argo CD version 2.10** is considered stable. We will compare specific points from the report against *Argo CD Version 2.10**.

In this section, we will not go through the entire report but will select a few items to discuss. Each potential threat will be abbreviated as **ATM-ID**, with priorities defined from low to high. These will be divided into three categories. *Table 13.1* depicts the **Change Impact Assessment** (**CIA**) as follows:

Confidentiality	High	Cluster takeover (Operations, Tenant) due to leak of admin credentials
	Medium	Sensitive information related to application or cluster state is exfiltrated
	Low	Non-sensitive information leakage
Integrity	High	Compromise of source code repositories and application deployments
	Medium	Application sync fails due to misconfiguration/invalid configuration
	Low	Non-critical operation is blocked due to misconfiguration/invalid configuration
Availability	High	Cluster (Operations, Tenant) subject to **Denial of Service** (**DoS**)
	Medium	Managed applications are blocked for a significant period
	Low	Managed app synchronization is blocked for a short period

Table 13.1 – Change Impact Assessment [1]

Enhancing password management and RBAC

Let's now examine where we can identify parts of the threats within our Cockpit and Fleet approach. This will be indicated in *Figure 13.1* with an exclamation mark and an ID:

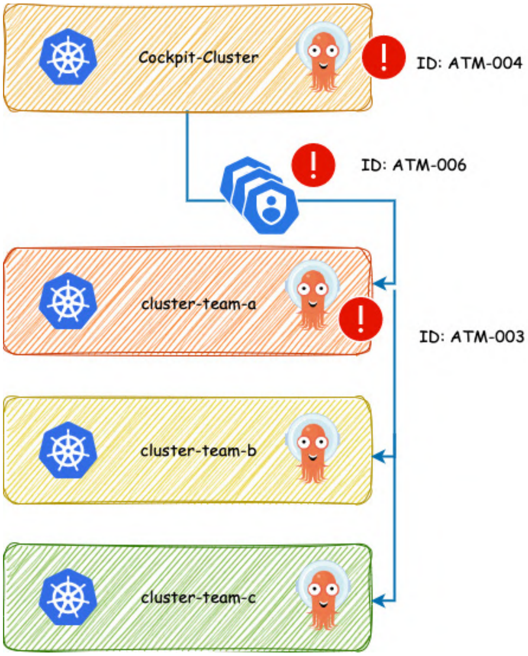


Figure 13.1 – Cockpit and Fleet approach combined with a threat model

Now, the first ID, *ATM-004*, is located in *Table 13.2* under the category *Argo CDs RBAC*:

ID	UID	Category	Priority
ATM-004	KR-AR-002	Argo CD RBAC	High

Table 13.2 – ATM-004: UI local users' credentials never expire, and don't have strong authentication

The risk here is that the Argo CD UI local users' credentials never expire and are solely based on a username and password without secondary authentication. The priority is already set too high. Considering the Cockpit, which has access to the remaining clusters, in my opinion, it becomes even more critical.

The danger is that unauthorized changes could occur if the credentials were to be compromised.

Important note

OAuth2 and **OpenID Connect** (**OIDC**) are protocols for secure authorization and authentication. OAuth2 allows secure resource access, while OIDC, built on OAuth2, adds user identity verification. They ensure secure data sharing without exposing credentials, enhancing application security through token-based authentication.

Dex is an open source identity service that uses OIDC to authenticate users and provide access to various applications and services.

Microsoft Entra ID, formerly **Azure Active Directory** (**AAD**), is a cloud-based identity and access management service provided by Microsoft.

It is therefore recommended to use the local admin account only for initial configuration and then switch to single sign-on through the provided Dex server and an OIDC provider that supports OAuth2, such as Microsoft Entra ID, allowing the mapping of roles to corresponding groups. Subsequently, the admin access should be disabled, as admin access does not provide more extensive permissions than the highest mapping of the admin group to an Azure group.

This can then be adjusted in the `argocd-rbac-cm` configmap as follows:

```
apiVersion: v1
data:
  policy.csv: |
    p, role:org-admin, applications, *, */*, allow
    p, role:org-admin, clusters, get, *, allow
    p, role:org-admin, repositories, get, *, allow
    p, role:org-admin, repositories, create, *, allow
    p, role:org-admin, repositories, update, *, allow
    p, role:org-admin, repositories, delete, *, allow
    g, "GROUP_ID", role:org-admin
  policy.default: role:readonly
kind: ConfigMap
metadata:
  labels:
    app.kubernetes.io/name: argocd-rbac-cm
    app.kubernetes.io/part-of: argocd
  name: argocd-rbac-cm
  namespace: argocd
```

Subsequently, the local admin can be disabled in `argocd-cm`:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: argocd-cm
  namespace: argocd
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
data:
  admin.enabled: "false"
```

If we also consider that the admin password (*ATM-003*) exists as a Kubernetes Secret, then hopefully, it's clear how dangerous it can be if an attacker gains access either to a Fleet cluster or, even worse, to the Cockpit, which enables access to the *ServiceAccounts* and thus to the remaining clusters.

ID	UID	Category	Priority
ATM-003	KR-AR-001	Argo CD RBAC	High

Table 13.3 – ATM-003: Initial admin password is stored as a Kubernetes Secret

If attackers gain access to a Fleet cluster or, even worse, to the Cockpit, which controls the *ServiceAccounts* and thus allows access to the remaining clusters, there is a significant risk involved. This situation could lead to *unauthorized changes* in the Argo CD configuration due to unauthorized reading of the Argo CD initial admin password from Kubernetes Secrets.

Therefore, it is recommended to rotate the Argo CD admin password and delete the secret, as Argo CD only creates the secret for access purposes but does not need the secret itself. According to Argo CD's documentation, the secret serves no other purpose than to store the initially generated password in clear text, and can safely be deleted at any time.

To modify the admin password, follow the subsequent steps. It's recommended to pass this guidance on to every user or team managing a fleet cluster. The optimal approach is to transition to OIDC and then deactivate the admin account as previously outlined. However, if you need to change the password, you should adhere to the following procedures:

Important note

These commands are run on a Unix-like operating system such as Ubuntu, Debian, or other Linux distributions, as well as on macOS.

1. First, forward to `argocd server`:

```
kubectl port-forward --namespace argocd service/argocd-server
8080:443
```

2. The password can be easily accessed using the Argo CD **command-line interface** (**CLI**):

```
argocd admin initial-password -n argocd
```

3. Using the previous `admin` as username and password, log in to Argo CD's IP or hostname:

```
argocd login <ARGOCD_SERVER>
```

4. Change the password using the following command:

```
argocd account update-password
```

5. As an alternative to the previous steps, you can use a shorter method by combining the commands into one line, as in the following:

```
kubectl port-forward svc/argocd-server -n argocd 8080:443
& argocd login localhost:8080 --username admin --password
$(kubectl -n argocd get secret argocd-initial-admin-secret -o
jsonpath="{.data.password}" | base64 -d) --insecure --grpc-web
```

6. Then, update the password as follows:

```
argocd account update-password
```

7. Now, kill the forwarding process running in the background as follows:

```
kill $(ps aux | grep 'kubectl port-forward svc/argocd-server -n
argocd 8080:443' | grep -v grep | awk '{print $2}')
```

Now that we've updated the password, let's examine the final part, *ATM-006* (*Table 13.4*), which becomes particularly significant in the context of the Cockpit and Fleet approach:

ID	UID	Category	Priority
ATM-006	KR-ASM-001	Argo CD Secrets Management	High

Table 13.4 – ATM-006: Never-expiring tenant cluster credentials are stored as Kubernetes Secrets

The risk here is that Argo CD tenant cluster credentials are stored as Kubernetes Secrets, and the Argo CD `argocd-manager` service account token on the tenant cluster is configured never to expire.

Consequently, an attacker could perform unauthorized actions on the tenant cluster due to unauthorized access to the never-expiring tenant bearer token from the Kubernetes Secret.

Depending on the provider and Kubernetes distribution, different mitigation strategies can be applied. It is advised to use **workload identities or managed identities**, especially in services such as Azure Kubernetes Service, allowing clusters to access Azure resources without secrets. Alternatively, consider leveraging an external **key management service** (e.g., **AWS Key Management Service**) to securely manage and expose Argo CD tenant cluster credentials. It's crucial to ensure that Argo CD's tenant cluster bearer tokens are rotated regularly, aligning with organizational security policies. For manual rotation, deleting the corresponding Kubernetes Secret in the tenant cluster will trigger the creation of a new token.

However, the aim of this section is not to address all security vulnerabilities, as I consider it unnecessary to cover them all due to their vast diversity. This section is intended to raise awareness of the potential security gaps that can arise when using GitOps and how to attempt to counteract them.

Figure 13.2 should now make it clear that we have only mitigated three out of many possible high-level attacks:

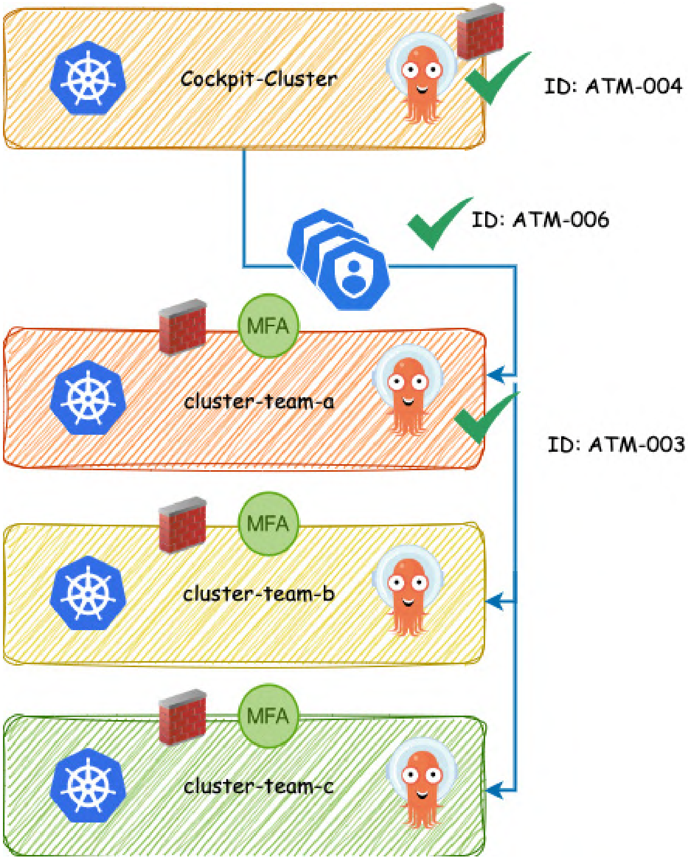


Figure 13.2 – Cockpit and Fleet approach combined with the threat model – the reduced attack vector view

Mechanisms commonly supported by most systems, such as firewall rules, should be implemented to restrict access to Cockpit clusters exclusively tospecified IP address ranges. Similarly, access to Fleet clusters can be limited using firewalls, potentially originating from the Cockpit cluster itself and confined to certain IP spaces. It's crucial to secure cluster access, particularly to the Cockpit, with **multi-factor authentication** (**MFA**) to significantly reduce the initial attack vector. Whenever feasible, perpetual tokens should be replaced with identity solutions to facilitate access without the need for secrets.

I strongly advise delving into the *Threat Model* document provided by **ControlPlane**. It includes detailed **attack trees** that illustrate how exploiting multiple vulnerabilities can have profound impacts. This comprehensive understanding is essential for effectively safeguarding your infrastructure.

The next section is about the fact that GitOps takes the approach that everything is in Git, but what about secrets? This section looks at how to store secrets securely in Git.

Committing everything to Git? What about Secrets?

In a GitOps workflow, the idea of committing every piece of configuration to Git repositories is central. It ensures that the entire state of your infrastructure is declaratively represented and can be versioned, audited, and reviewed. However, this approach presents a challenge when it comes to handling secrets, such as passwords, tokens, and private keys. Storing such sensitive information plainly in Git repositories is not secure. This is where tools such as Sealed Secrets and External Secrets come into play, providing secure mechanisms to manage secrets in a GitOps workflow.

Sealed Secrets

Sealed Secrets [*2*] is a Kubernetes controller and toolset, designed to encrypt secrets that can safely be stored in Git repositories. When you apply the sealed secret to your cluster, the Sealed Secrets controller decrypts it and creates a regular Kubernetes secret. This process allows you to manage your secrets' life cycle through Git without exposing the sensitive content.

Use it as follows:

1. **Deployment**: Deploy the Sealed Secrets controller in your Kubernetes cluster via GitOps and install the `kubeseal` CLI tool on your local machine, as follows:

```
helm repo add sealed-secrets https://bitnami-    labs.github.io/
sealed-secrets
helm install sealed-secrets sealed-secrets/sealed-secrets
#Install e.g. CLI on MacOS
brew install kubeseal
```

2. **Key-pair**: Generate a key-pair or use the certificate from the deployed Sealed Secrets Operator:

```
kubectl -n sealed-secrets get secret sealed-secrets-… -o json
-o=jsonpath="{.data.tls\.crt}" | base64 -d > sealed-secret.crt
```

3. **Sealing Secrets**: Use the kubeseal CLI to encrypt your secret. The CLI generates a *SealedSecret* resource, which you can commit to your Git repository:

```
kubectl create secret generic my-secret --from-
literal=password='myStrongPassword' --dry-run=client -o json |
kubeseal --cert sealed-secret.crt > mysealedsecret.yaml
```

4. **Applying Sealed Secrets**: Commit `mysealedsecret.yaml` to your Git repository. When you apply this file to your cluster, the Sealed Secrets controller decrypts it and creates a standard Kubernetes secret.
5. **Automation with GitOps**: Integrate this process into your GitOps workflows. Whenever you update your sealed secrets in Git, your CI/CD pipeline can automatically apply them to your cluster.

External Secrets

External Secrets [*3*] is an open source project that integrates external secret management systems such as **AWS Secrets Manager**, **Azure Key Vault**, and **Google Secret Manager** with Kubernetes. It allows you to securely inject secrets into your applications without having to expose them in your Git repositories.

Use it as follows:

1. **Deployment**: Deploy the External Secrets Operator in your Kubernetes cluster:

```
helm repo add external-secrets https://charts.external-
secrets.io
helm install external-secrets external-secrets/external-secrets
```

2. **Configuration**: Define an *ExternalSecret* resource that specifies the external secret store and the secret key. The operator fetches the secret from the external store and creates a Kubernetes secret in the cluster:

```
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: my-external-secret
spec:
  secretStoreRef:
    name: my-secret-store
    kind: SecretStore
  target:
    name: my-kubernetes-secret
```

```
      data:
      - secretKey: external-secret-key
        remoteRef:
          key: name-of-the-secret-in-external-store
```

3. **Applying External Secrets**: Commit the *ExternalSecret* resource to your Git repository. The External Secrets Operator will automatically create or update the Kubernetes secret in your cluster based on the external source.
4. **Integration with GitOps**: Incorporate External Secrets into your GitOps pipelines. Changes to the *ExternalSecret* definitions in your Git repo trigger the operator to sync the secrets, ensuring your cluster's secrets are always up-to-date.

The choice of tools depends on various factors. For instance, if you don't have a vault for storing secrets, keys, or certificates, then External Secrets might not be suitable. Initially, a connection from the External Secrets Operator to the Secret Store or **ClusterSecretStore** must be established. In our projects, we utilize managed identities to ensure this. Otherwise, you typically need an ID and a secret, which can be challenging to manage securely with GitOps at scale, as the ID and secret must be securely transferred to the cluster, possibly through a CI/CD pipeline. A significant advantage of External Secrets is its ability to fetch secrets at runtime without dependency on the cluster.

However, access to a secrets manager through a vault is not always available. Therefore, the Sealed Secrets Operator, particularly in on-premises environments, is currently considered the industry standard.

Teams need to decide which option is the better choice. In my opinion, both options are solid and integrate very well into the GitOps ecosystem. This not only allows for the secure storage of secrets, keys, and certificates while embracing the GitOps approach but also opens up entirely new possibilities with GitOps at scale, such as deploying a pull secret across a specific registry. The platform team, by combining External Secrets and Kyverno, can deploy a secret across all clusters, and Kyverno distributes it across the namespaces. As a result, every team has the pull secret necessary to pull images from a central image registry. This enhances security since every image can be subjected to scanning, and it provides an overview of the images in operation.

By leveraging Sealed Secrets and External Secrets, you can maintain the GitOps principle of storing all configurations in Git while securely managing your secrets. These tools help you automate the management of secrets, keeping your infrastructure secure and your deployments consistent.

Every tool that is added to the Kubernetes Service Catalog increases the likelihood that more security vulnerabilities will be introduced into the system. We will look at how to maintain your catalog in the next section.

In the following section, we will delve into enhancing security through GitOps and a **policy engine**. This approach enables proactive measures during deployment, determining what is permissible to deploy and identifying associated risks. This preemptive strategy ensures that only secure, compliant configurations make their way into production, thereby reinforcing your security posture.

Leveraging a policy engine for policy-as-code practices

In the contemporary landscape of software development, the way we ensure security and compliance within our systems has significantly evolved. A pivotal aspect of this evolution is the adoption of policy-as-code practices, which enable the integration of governance and security policies directly into the development and operational processes. Within this context, policy engines such as Kyverno [*4*] and **Open Policy Agent** (**OPA**) [*5*] play a crucial role. These tools empower organizations to define and enforce their security, compliance, and governance requirements as code, thereby ensuring consistent application and transparency across their infrastructures and applications.

Kyverno and OPA are both powerful, flexible tools designed specifically to implement policy as code in Kubernetes environments and beyond. Kyverno focuses on simplifying Kubernetes security by using a user-friendly, Kubernetes-native syntax that allows developers to define and enforce security policies directly within their CI/CD pipelines. On the other hand, OPA is a more general-purpose policy engine that employs a highly flexible, domain-agnostic language called **Rego**, enabling the creation and implementation of fine-grained, context-aware policies across a variety of software systems.

Integrating these policy engines into a GitOps-based workflow architecture significantly enhances security, as it ensures the continuous enforcement and validation of compliance. By adopting GitOps, teams can manage their infrastructure and application setups as code, creating a versioned, repeatable, and automatable environment. Incorporating Kyverno or OPA into this process means that changes to infrastructure or applications are automatically validated against predefined policies before being implemented. This ensures that all deployments comply with organizational security standards and that potential security vulnerabilities or configuration errors are proactively identified and remediated before they impact the production environment.

Integrating Kyverno and OPA

In this part, we explore how integrating tools such as Kyverno and OPA into the development and operational workflow not only enhances security and compliance but also promotes efficiency and speed in development cycles. *Automating* policy enforcement within the GitOps framework allows teams to deliver faster and more securely, leading to more robust, secure applications and infrastructures.

We will focus on Kyverno to facilitate a better understanding, as I believe it's a tool that users can comprehend more easily without the need to learn Rego. However, before delving into that, let's examine the layer at which Kyverno operates and its integration with GitOps. *Figure 13.3* will elaborate on these subjects:

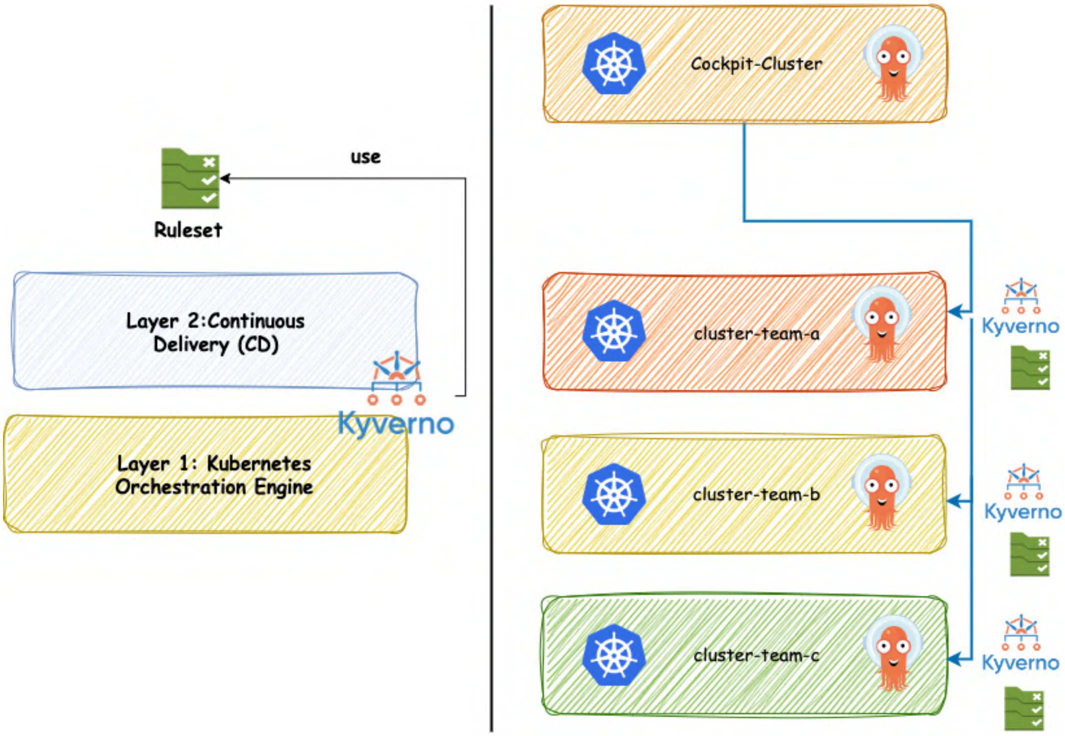


Figure 13.3 – Kyverno operation layer and GitOps

As illustrated, Kyverno operates across both layers, effectively acting as a gatekeeper. When you attempt to deploy something that violates its rules, Kyverno intervenes, causing the deployment to fail. However, Kyverno's capabilities extend beyond this; it can also validate already-running services within the Kubernetes cluster or mutate resources. This versatility justifies positioning Kyverno between these layers, making it an advantageous choice.

Leveraging GitOps, you can deploy the Kyverno instance across every cluster in the fleet, ensuring consistent policy enforcement. Additionally, the *define once, deploy everywhere* approach facilitates multi-cluster deployment, streamlining the management of policies across various environments. By collaborating with the security team, you can implement different rulesets tailored to specific environments, enhancing the security and compliance of your deployments. This synergy between Kyverno and GitOps not only simplifies governance but also fortifies the infrastructure's overall security posture.

Now, let's look at how it works in the praxis.

Hands on – let's put theory into practice *[6]*

First, we establish a streamlined folder structure for the Kubernetes Service Catalog, as in the following:

```
.
├── applicationsets
│   └── security
│       └── kyverno-applicationset.yaml
├── kustomize
│   └── security
│       └── kyverno
│           └── policies
│               └── base
│                   ├── disallow-latest-tag.yaml
│                   ├── kustomization.yaml
│                   └── restrict-nodeport.yaml
└── security
    └── kyverno
        ├── Chart.yaml
        └── values.yaml
```

By initiating the deployment of `kyverno-applicationset.yaml`, you will target every fleet cluster identified by the `env=prod` or `env=development` labels. This deployment not only sets up the Kyverno tool across these clusters but also applies a specific set of policies from the `kustomize` folder to each one. In our illustration, we utilize a basic base folder; however, you have the flexibility to configure distinct folders – for instance, for different stages. These can be aligned with the cluster labels, enabling you to deploy the appropriate ruleset to the corresponding cluster. Such an arrangement proves beneficial for implementing more lenient policies in development clusters and stricter ones in production environments.

Now, let's examine an example policy that will be deployed across every fleet cluster matching the specified labels. We'll focus on the `disallow-latest-tag.yaml` policy.

In the first part, we see the annotations, the `Best Practices` category, and then the version, the severity classification, which subject is affected, and the description of which negative consequences can result from it. I find it nice and compact, clear, and easy to configure:

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: disallow-latest-tag-example
  annotations:
    policies.kyverno.io/title: Disallow Latest Tag Example
    policies.kyverno.io/category: Best Practices
    policies.kyverno.io/minversion: 1.6.0
```

```
    policies.kyverno.io/severity: medium
    policies.kyverno.io/subject: Pod
    policies.kyverno.io/description: >-
      The ':latest' tag is mutable and can lead to unexpected errors
if the
      image changes.. This policy validates that the image
      specifies a tag and that it is not called `latest`.
```

In the subsequent section under specifications, we observe that the action is designated as `Audit` rather than `Enforce`, implying that it is not compulsory. Following this, two rules are established. The initial rule conducts a validation of the container's tag using the `*:*` pattern, ensuring it adheres to specific criteria. Subsequently, the second rule escalates to the resource level, applying a broader scope by targeting the pods directly. It stipulates a constraint that the pattern with the `latest` tag must not be used, reinforcing the governance over the deployment practices:

```
spec:
  validationFailureAction: Audit
  background: true
  rules:
    - name: require-image-tag
      match:
        any:
          - resources:
              kinds:
                - Pod
      validate:
        message: "An image tag is required."
        pattern:
          spec:
            containers:
              - image: "*:*"
    - name: validate-image-tag
      match:
        any:
          - resources:
              kinds:
                - Pod
      validate:
        message: "Using a mutable image tag e.g. 'latest' is not
allowed."
```

```
          pattern:
            spec:
              containers:
                - image: "!*:latest"
```

In practice, it looks like this: when the rule is applied and an attempt is made to deploy a pod and its contained container with the `latest` tag, the deployment is blocked. Let's try to deploy a Nginx pod with the `latest` image tag:

```
kubectl run nginx --image nginx:latest
```

Now you can get a Kyverno `policyreport` like the following:

```
kubectl get policyreport fbe08ffd-bee6-497f-aee8-f96e9cd022f8  -n
default -o yaml | grep image -A 8
```

The output should be like the following:

```
- category: Best Practices
  message: 'validation error: Using a mutable image tag e.g.
''latest'' is not allowed.
    rule validate-image-tag failed at path /spec/containers/0/image/'
  policy: disallow-latest-tag
  result: fail
  rule: validate-image-tag
  scored: true
  severity: medium
  source: kyverno
```

As you can see, we violated the best practices policy, specifically the `disallow-latest-tag` policy.

I refer to this setup as a gatekeeper because it allows you to dictate, through simple rules, what can and cannot be deployed into the cluster. However, the capabilities of this setup extend further. For instance, you can configure Kyverno policies to enforce image signing or to ensure that only images from certain registries are allowed to be pulled, among other things. A policy engine is a powerful tool. In contrast to the gatekeeper, Kyverno or OPA also operates on already-deployed resources or resources that passed the gate and are therefore placed between the two layers, as shown in *Figure 13.3*.

When combined with the GitOps approach, it opens up new possibilities that transcend team boundaries, enhancing the security of projects. This integration not only streamlines the deployment processes but also ensures that the operational standards are consistently met, thereby bolstering the overall security posture of the infrastructure.

In the next section, we will look at how to increase security on two different layers using two different tools.

Automating security scanning and compliance

In today's fast-paced and security-conscious IT environment, maintaining continuous security and compliance is crucial. Automated security scanning and compliance are integral components of a robust cybersecurity strategy, especially in Kubernetes clusters where applications and services are dynamically scaled and updated. Here, we'll discuss how the **KubeClarity** [*7*] or **Falco** [*8*] tool can be utilized to enhance your security posture in Kubernetes environments. The following image illustrates the level at which KubeClarity operates:

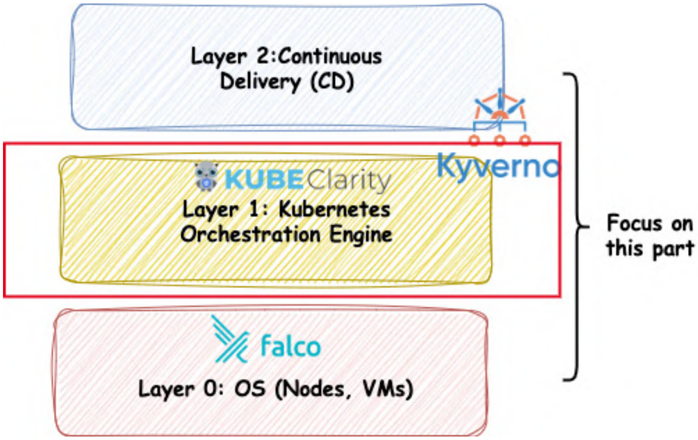


Figure 13.4 – KubeClarity operation layer

KubeClarity is an open source tool designed for security scanning and compliance analysis in Kubernetes environments. It provides a comprehensive overview of the potential vulnerabilities in your container images and Kubernetes configurations.

KubeClarity

Let's first understand how KubeClarity enhances security:

- **Vulnerability scanning**: KubeClarity scans container images for known vulnerabilities, utilizing various databases and vulnerability tracking sources. It provides detailed insights into the security flaws within your images, along with their severity levels, enabling you to prioritize fixes.
- **Compliance assurance**: The tool checks your Kubernetes configurations against best practices and compliance standards, ensuring that your deployments adhere to industry and organizational security policies.
- **Continuous monitoring**: KubeClarity continuously monitors your containerized applications for new vulnerabilities, providing real-time alerts and updates. This ongoing vigilance helps maintain a secure Kubernetes environment.

By integrating KubeClarity within a GitOps framework, an organization can significantly enhance its security and compliance posture, ensuring that its Kubernetes clusters are fortified against evolving threats. This adoption of GitOps principles allows for the automation of security scans and compliance checks, making these processes more consistent, efficient, and scalable. The proactive approach to security facilitated by KubeClarity not only protects your infrastructure but also fosters trust with customers by demonstrating a commitment to maintaining a secure and compliant environment. The continuous monitoring and automatic alignment with security policies enabled by GitOps ensure that any deviations are quickly detected and rectified, maintaining a robust defense against potential vulnerabilities.

Falco

Falco is an open source tool for real-time security monitoring and analysis, specifically designed for Kubernetes environments. It functions as an **intrusion detection system** (**IDS**), enabling teams to detect anomalous activities in their applications and infrastructures that could indicate security breaches or other threats. The following picture illustrates the level at which Falco operates.

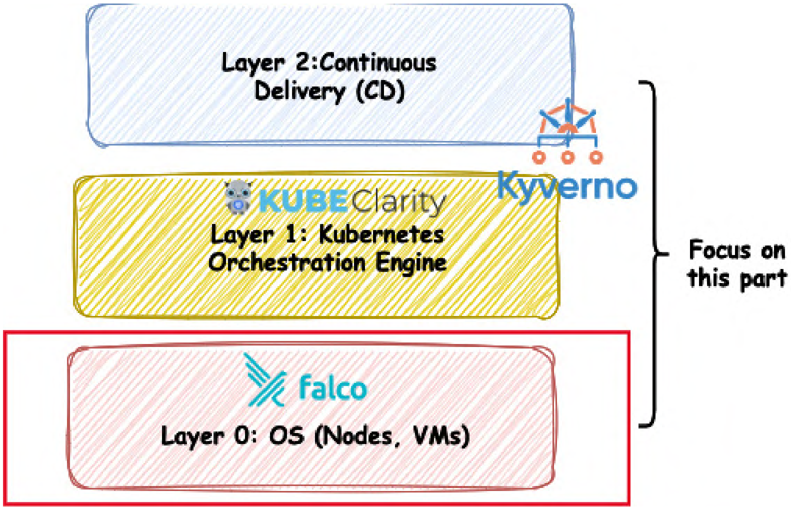


Figure 13.5 – Falco operation layer

Falco operates at the system level (*Figure 13.5*), monitoring the underlying Linux kernel functionality, or more precisely, the system-level activities of container orchestration platforms. It utilizes Linux kernel capabilities, particularly **extended BPF** (**Berkeley Packet Filter**) or traditional **system calls** (**syscalls**) monitoring through a kernel module, to observe and analyze system-wide events in real time. Falco can capture and evaluate system calls from applications running inside containers, identifying unusual or undesirable behavior.

The advantages of integrating Falco with GitOps are as follows:

- **Automated security monitoring**: By integrating Falco into a GitOps pipeline, security policies can be defined as code and automatically applied to every change in the infrastructure or applications, enabling continuous and automated monitoring of security standards.
- **Real-time alerts**: Falco can be configured to send real-time alerts when a defined rule is violated. In a GitOps environment, these alerts can facilitate swift responses to potential security incidents.
- **Consistency and compliance**: With GitOps, Falco configurations can be versioned, reviewed, and automatically deployed, ensuring consistent enforcement of security policies across different environments.
- **Transparency and traceability:** All changes to Falco policies are recorded in Git, providing a clear audit trail. This promotes transparency regarding security monitoring and facilitates compliance with regulatory requirements.

Integrating Falco into the GitOps strategy allows teams not only to enhance their security posture but also to adopt a proactive approach to identifying and responding to security threats in their Kubernetes environment.

However, every tool also brings potential risks with it, even if it is intended to increase security, which is why maintenance should not be neglected and attention should be paid to this when designing or selecting the tools. This is exactly what we will discuss briefly in the next section.

Keeping your platform catalog up-to-date

The content of this section has been extensively covered already in *Chapter 5* under the *Bonus – maintenance with GitOps at scale and KSC* section. It serves as a reminder that with every tool added, the catalog grows, and a keen focus on security should be maintained.

When a **Common Vulnerabilities and Exposures** (**CVE**) threat is revealed and you opt for the Cockpit and Fleet approach, adopting GitOps at scale also facilitates a vulnerability-management-at-scale strategy. Therefore, this aspect should be considered particularly early in the development of a concept.

Implementing such a strategy ensures that as your infrastructure expands, your approach to handling vulnerabilities scales accordingly. This proactive stance helps in quickly addressing security issues across the entire fleet, maintaining the integrity and security of your systems. Early integration of these practices into your conceptual framework is crucial to establishing a robust, scalable, and secure infrastructure.

To conclude the chapter, *Figure 13.6* illustrates the stages or gates that the packet (application or service) must pass through, highlighting where various security mechanisms can intervene to scan the packet as thoroughly as possible:

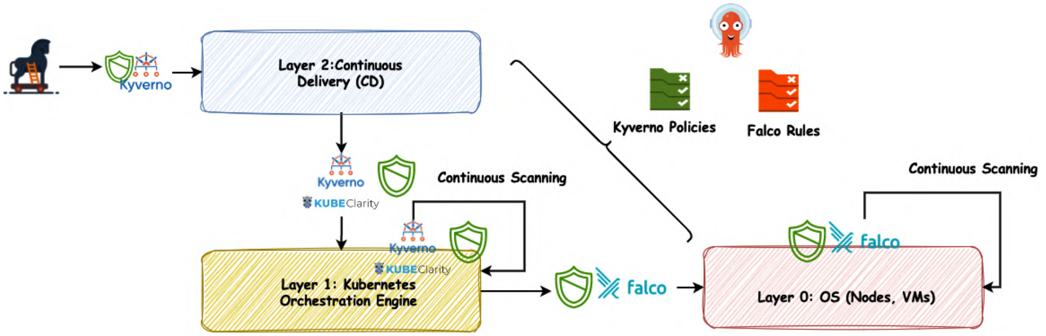


Figure 13.6 – Security on multiple layers

Nevertheless, a packet might pass through certain stages, hence the necessity for tools such as Falco and Kyverno to enable continuous scanning. These tools thrive on defined rules, which, fortunately, can be easily deployed with GitOps.

Summary

In this chapter, it should have become clear that security doesn't operate on just a single layer and shouldn't be enforced by just one team. Security is a cultural issue that can be expanded with approaches such as GitOps and tools such as Argo CD at scale, which can include defining rulesets or policies. Security doesn't stop with tools; it also encompasses how one handles sensitive data such as secrets and keeps third-party tools in the ecosystem up-to-date and secure. Effective security practices require collaboration across teams, continuous monitoring, and proactive management to ensure a robust and resilient infrastructure.

In the next chapter, we'll explore the integration of FinOps with GitOps to achieve sustainable, cost-effective operations. We'll cover core FinOps principles, cost forecasting, optimization, and understanding the carbon footprint. Additionally, we'll look at future trends in GitOps, including AI-driven automation.

References

- [1] `https://github.com/argoproj/`
- [2] `https://github.com/bitnami-labs/sealed-secrets`
- [3] `https://external-secrets.io/latest/`
- [4] `https://kyverno.io`
- [5] `https://www.openpolicyagent.org`
- [6] `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes/tree/main/chapter13`
- [7] `https://github.com/openclarity/kubeclarity`
- [8] `https://falco.org`

14

FinOps, Sustainability, AI, and Future Trends for GitOps

In this chapter, we venture beyond the conventional boundaries of GitOps to explore its extensive scope and multifaceted impact across various domains not covered in the main sections of the book. Our journey is structured into distinct yet interconnected blocks, each shedding light on different dimensions where GitOps extends its influence, providing a comprehensive insight into the potential and versatility of this practice.

We commence with the fundamentals of **FinOps** and **cost management**, where the fusion of financial acumen and operational expertise comes to the forefront. This segment elucidates how the FinOps framework, integrated with GitOps methodologies, empowers organizations to wield granular control over their cloud expenditures. By leveraging tools such as **OpenCost** and **Kubecost** in a GitOps-driven environment, businesses can achieve unprecedented transparency and efficiency in managing costs for clusters, projects, and beyond, ensuring that every dollar spent is an investment toward innovation and growth.

Transitioning to the realm of sustainability and green operations, we delve into how GitOps can be a catalyst for eco-friendly IT practices. This block emphasizes the significance of sustainable operations, not only from an economic standpoint but also in contributing to environmental stewardship. We discuss practical strategies, such as the automated shutdown of development clusters, illustrating how GitOps can align technological advancements with ecological responsibility, thereby fostering a culture of sustainability within the tech industry.

The exploration expands to include GitOps and **artificial intelligence** (**AI**)-driven automation, a segment that bridges the advanced realms of AI and GitOps. This fusion is revolutionizing the way organizations deploy, monitor, and manage their IT landscapes, transcending the hype to deliver tangible, impactful automation solutions. Through real-world examples and case studies, we examine the dual benefits of AI in enhancing DevOps practices and streamlining the deployment of inter-company services, all within the GitOps framework.

Finally, we reflect on the evolving landscape of GitOps, contemplating the continuous evolution of GitOps principles and their transformative influence across various sectors. This concluding block forecasts the future trajectory of GitOps, speculating on its role amid emerging technologies and its potential to redefine industry standards. We delve into how GitOps is **pivotal** in forecasting and monitoring cloud costs, thereby enabling organizations to navigate the complexities of modern cloud environments with confidence and strategic foresight.

Each of these blocks, while distinct in focus, collectively underscores the expansive reach of GitOps, illustrating its role as a cornerstone in modern IT strategy. This chapter aims to broaden readers' perspective, encouraging them to envision GitOps not merely as a tool for operational efficiency but as a holistic approach that harmonizes technological innovation with financial wisdom, environmental consciousness, and forward-thinking adaptability.

We will cover the following main topics in this chapter:

- Covering the fundamentals of FinOps
- Forecasting and monitoring costs with GitOps
- Optimization techniques for cloud spend
- Assessing carbon footprint and promoting green operations
- Looking at GitOps and AI-driven automation
- Future challenges and opportunities in GitOps
- The role of GitOps in emerging technologies

Covering the fundamentals of FinOps

FinOps, an operational framework blending finance and DevOps principles, is fundamentally a cultural practice aimed at maximizing the business value of cloud investments. It fosters a collaborative environment where engineering, finance, and business teams work together to facilitate data-driven decisions, enhancing financial accountability and optimizing cloud costs.

The essence of FinOps lies in its ability to bridge the gap between traditionally siloed departments, promoting a shared responsibility model where all stakeholders are vested in the cloud's cost-effectiveness and operational efficiency. This cross-functional synergy is crucial for enabling faster product delivery while ensuring financial transparency and control.

Key aspects of FinOps include real-time monitoring of cloud expenses, enabling organizations to make informed decisions that balance cost, speed, and quality. The approach goes beyond mere cost-cutting; it's about leveraging the cloud's variable spending model to drive innovation, revenue growth, and strategic investments in the tech infrastructure.

Organizations looking to adopt FinOps can start by exploring resources offered by the FinOps Foundation, which provides education, best practices, and community engagement to support different maturity levels in the FinOps journey. The **Crawl, Walk, Run** maturity model advocated by FinOps allows organizations to evolve their financial operations progressively, enhancing their cloud architecture and investment strategies continuously.

The Crawl, Walk, Run model

The **Crawl, Walk, Run** model is a framework that describes the progressive stages of learning and implementation. In the **Crawl** stage, individuals or organizations focus on understanding basic principles with limited scope and high reliance on guidance. The **Walk** stage involves greater familiarity and confidence, with more complex tasks and increased efficiency. Finally, in the **Run** stage, there is full mastery and independent, innovative use of the new concept, process, or technology. This model helps manage expectations and provides a structured growth path to ensure a solid foundation before advancing to more complex levels.

Central to the FinOps framework are principles guiding collaborative efforts, personal ownership of cloud usage, and the strategic alignment of cloud investments with business value. These principles ensure that every team member, from executives to engineers, understands and contributes to the efficient and innovative use of cloud resources.

Incorporating FinOps into an organization's operational model not only promises enhanced cost management and optimization but also aligns cloud expenditure with business outcomes, ensuring that investments translate into tangible value. This strategic alignment is crucial in today's dynamic cloud environments, where adaptive planning and cost-effective resource utilization are key to sustaining competitive advantage and fostering long-term growth.

In the next section, we will use OpenCost [*1*] and Kubecost [*2*] to determine the costs for a cluster or for a namespace, which could, for example, represent a project.

Forecasting and monitoring costs with GitOps

GitOps can be seamlessly integrated with FinOps to enhance the financial governance of cloud resources while **maintaining agility** and operational efficiency. The combination of GitOps and FinOps practices enables organizations to manage their cloud infrastructure and costs more effectively through automation, version control, and continuous monitoring.

How GitOps complements FinOps

First, let's understand how GitOps complements FinOps:

- **Automated cost optimization**: GitOps allows for the automation of deployment and scaling of cloud resources, which can be integrated with FinOps strategies to ensure that resource utilization is optimized for cost without sacrificing performance. This means infrastructure can scale up or down automatically, aligning with financial objectives and operational demands.
- **Version-controlled spending**: With GitOps, every change to the infrastructure is version-controlled in Git repositories, providing an audit trail of what changes were made, who made them, and why. This aligns with FinOps principles by adding transparency to the financial impact of operational changes, enabling better budget tracking and cost allocation.
- **Collaboration and visibility**: GitOps fosters collaboration among development, operations, and finance teams by using Git as a **single source of truth** (**SSOT**). This collaborative environment ensures that financial considerations are integrated into the earliest stages of infrastructure planning and application development, promoting a cost-aware culture.
- **Policy enforcement**: Integrating **policy-as-code** (**PaC**) tools within a GitOps workflow can enforce financial governance, ensuring that resources are utilized efficiently and within budget. Policies can be set to prevent overspending, enforce the use of cost-optimized resources, or ensure adherence to budgetary constraints.
- **Continuous cost monitoring and feedback**: GitOps enables continuous monitoring and feedback loops, allowing teams to quickly identify and address inefficiencies in cloud resource utilization. This constant vigilance helps in maintaining a balance between operational efficiency, cost, and speed, in line with FinOps goals.
- **Proactive financial management**: By using GitOps, organizations can proactively manage their cloud expenses. **Infrastructure-as-code** (**IaC**) templates stored in Git repositories can be analyzed to forecast costs and understand the financial implications of planned changes before they are executed.

Utilizing GitOps with FinOps

The following points describe how integrating cost monitoring, budget enforcement, resource optimization, and financial operations automation into a GitOps pipeline can enhance cost efficiency and ensure financial governance:

- **Implementing cost monitoring tools**: Integrate cloud cost monitoring tools into the GitOps pipeline to provide real-time feedback on the financial impact of code changes and infrastructure updates
- **Enforcing budget policies**: Use GitOps to enforce budget policies through IaC, ensuring that deployments conform to predefined financial constraints and operational budgets

- **Optimizing resource allocation**: Leverage GitOps to automate the deployment of cost-optimized resources, such as spot instances, and to shut down underutilized resources, ensuring efficient cloud spend
- **Streamlining financial operations**: Automate financial operations tasks such as cost reporting, budget alerts, and resource tagging through GitOps, ensuring that financial governance is consistently applied across all cloud resources

By integrating GitOps with FinOps practices, organizations can ensure that their cloud infrastructure is not only operationally efficient but also aligned with their financial objectives, delivering maximum value from their cloud investments. This holistic approach enables a more agile, transparent, and cost-effective cloud management strategy.

Now, let's take a look at how it works in practice. For this, we will examine two tools: OpenCost as an open source option and Kubecost, which is built on OpenCost but offers many additional features on top in the next part.

OpenCost versus Kubecost with GitOps

OpenCost and Kubecost are tools used in the realm of Kubernetes cost monitoring and analysis, providing insights into resource utilization and assisting in managing costs associated with Kubernetes environments in the cloud. Let's cover a brief description of both tools and the differences between them.

OpenCost

OpenCost is an open source tool that brings **transparency** to the costs and usage associated with Kubernetes clusters. It enables teams to monitor, analyze, and optimize their Kubernetes costs by providing detailed insights into resource utilization and associated expenses. OpenCost offers features such as the following:

- Cost breakdown at the pod, deployment, namespace, or cluster level
- Support for multi-cluster environments
- Integrations with cloud providers for more accurate cost estimation
- Forever free and open source, supported and maintained by experts

How does this all fit with GitOps, or, more specifically, what value does GitOps add in the FinOps context? OpenCost sets up on-premises pricing configurations based on the **Google Cloud Platform** (**GCP**) `us-central-1` region—for instance, `"CPU": "0.031611"$`, and `"RAM": "0.004237"$` on an hourly basis. Nevertheless, OpenCost also allows for the customization of pricing. This is where GitOps comes into play, enabling not just the deployment of the OpenCost tool across various clusters (*Figure 14.1*) but also the capability to configure prices accordingly. For example, if you have three data centers in Europe, located in three different countries, the CPU and RAM prices will vary due to factors such as electricity prices, procurement, depreciation, and so on.

There are globally defined values under `optimization/opencost`, and then there's a custom pricing model for each specific country. Visually, it looks as follows:

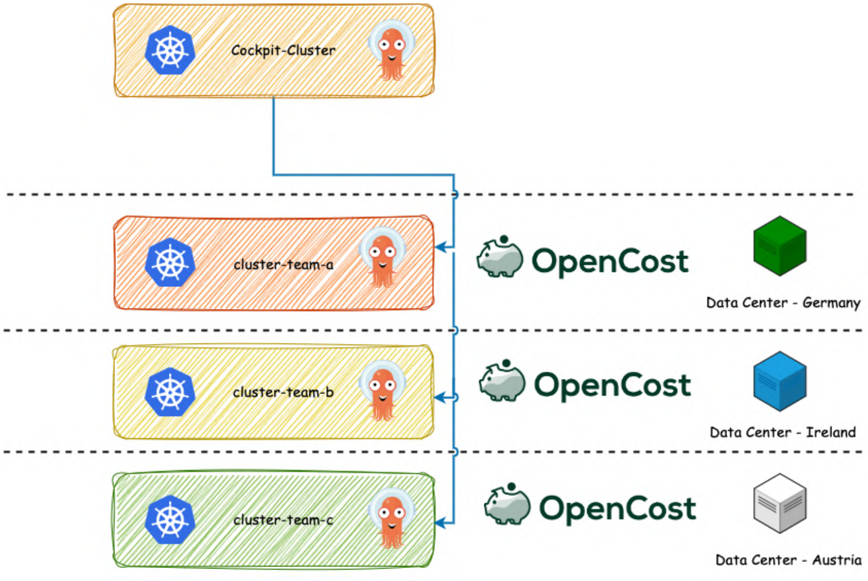


Figure 14.1 – OpenCost with GitOps and on-premise pricing configuration

For the scenario in *Figure 14.1*, we establish the following folder structure [3]:

```
.
├── applicationsets
│   └── optimization
│       └── opencost-applicationset.yaml
├── cluster
│   ├── in-cluster-austria
│   │   └── optimization
│   │       └── opencost
│   │           └── values.yaml
│   ├── in-cluster-germany
│   │   └── optimization
│   │       └── opencost
│   │           └── values.yaml
│   └── in-cluster-ireland
│       └── optimization
│           └── opencost
```

```
|                   └── values.yaml
└── optimization
     └── opencost
          ├── Chart.yaml
          └── values.yaml
```

It can also be combined with **hybrid cloud** and **multi-cloud**; then, it might look like this:

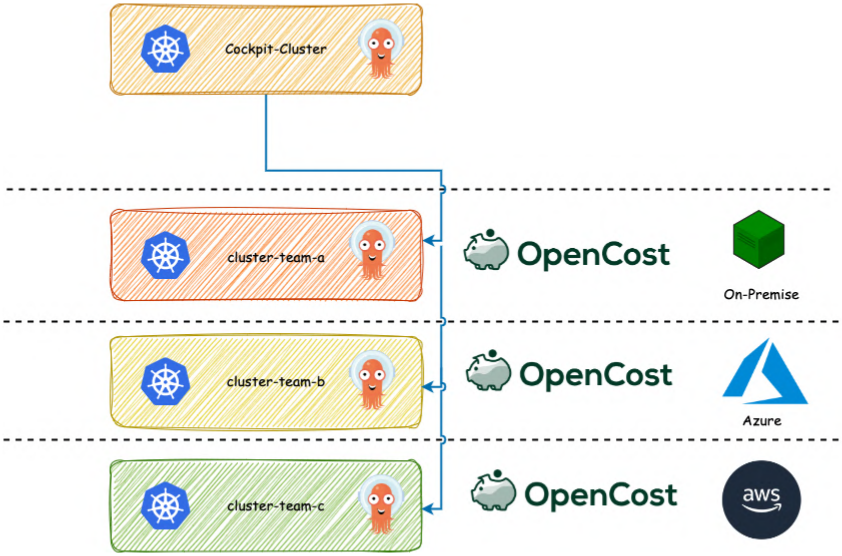


Figure 14.2 – OpenCost with GitOps and hybrid-setup pricing configuration

GitOps, combined with OpenCost, opens up new possibilities in FinOps, enabling more user- and infrastructure-defined cost visibility across distributed infrastructures, thus allowing for enhanced financial oversight, precise cost allocation, and strategic budget optimization in cloud environments.

However, if you require functionalities such as team collaboration, budget planning, alerting, estimated monthly savings, and more, then you should consider exploring Kubecost.

Kubecost

Kubecost is a product that builds on OpenCost. It's a commercial solution that offers additional features and support for businesses that go beyond the core functionalities of the OpenCost project. Kubecost includes all the features of OpenCost, along with the following:

- Advanced budgeting and cost forecasting
- Personalized optimization recommendations

- Additional security and compliance features
- Professional support and consultation

Additionally, you have the flexibility to utilize GitOps not just for deploying Kubecost across various clusters but also to **empower teams** to set **budgets** based on namespaces, clusters, or labels, along with specifying actions (*Figure 14.3*) to take when budget thresholds are exceeded:

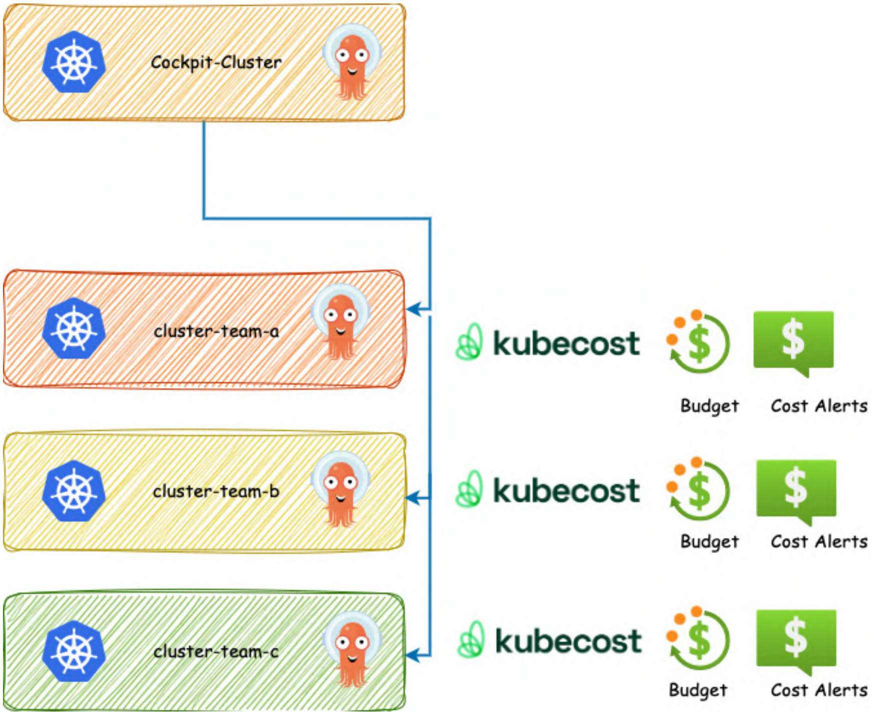


Figure 14.3 – Kubecost with GitOps: defining budget and cost alerts

By integrating GitOps with Kubecost, organizations empower their teams with enhanced control over their projects or clusters, streamlining the deployment of this powerful tool. This synergy allows teams not only to deploy Kubecost efficiently at scale but also to define essential parameters that align with their budgetary needs. Such integration is pivotal in fostering a collaborative environment for cost management, enabling various teams within the organization to share best practices, insights, and strategies to collectively optimize resources.

This collaborative atmosphere is further enriched by the capabilities of Kubecost combined with GitOps, which facilitate proactive cost management. With the ability to set predefined thresholds and configure automated alerts, teams are equipped to actively monitor and manage their cloud spending. This proactive stance ensures that any potential budget overruns are swiftly identified and addressed, and overall cloud expenditure is optimized effectively.

The confluence of GitOps and Kubecost transforms the approach to cloud cost management, moving from a reactive to a proactive model. Teams are no longer in the dark about their spending trends and are instead provided with a clear, actionable framework that enables them to maintain financial efficiency while capitalizing on the scalability and flexibility of cloud resources. This integration ensures that cost optimization is not just an afterthought such as the end of the month but a fundamental aspect of **daily operations**, empowering teams to leverage cloud resources judiciously and economically.

Differences between OpenCost and Kubecost

Here are the most important differences between the two tools:

- **Open source versus commercial**: OpenCost is an open source project inviting the community to freely use and adapt it, whereas Kubecost is a commercial version offering enhanced features and support for business customers.
- **Feature set**: Kubecost builds upon OpenCost, extending it with additional, commercial features specifically designed for enterprise customers. For example, while OpenCost provides basic cost monitoring for Kubernetes clusters, Kubecost offers advanced features such as automated cost optimization recommendations and detailed cost allocation reports for different teams and projects.
- **Support**: Kubecost provides professional support and consulting services that go beyond what is typically expected from an open source project.

Both tools aim to improve transparency and control over costs associated with Kubernetes usage, but they cater to different user groups and needs, ranging from the open source community to large enterprises.

This combination of GitOps and OpenCost or Kubecost not only streamlines operational efficiency but also significantly enhances the strategic financial management of Kubernetes environments.

In the next part, we look at how to optimize cloud costs and how GitOps with Kubecost contributes to this.

Optimization techniques for cloud spend

Optimization techniques for cloud spending are crucial for organizations looking to maximize their **cloud investment** while minimizing unnecessary costs. Integrating GitOps with tools such as Kubecost can significantly streamline this process, providing a systematic approach to managing and reducing cloud expenses. Here's how this combination can be leveraged to enhance cloud spend optimization.

Combining GitOps and Kubecost for cloud spend optimization

GitOps, by design, brings automation, predictability, and transparency to cloud operations. It establishes a Git repository as the SSOT for the entire infrastructure, which means every change is version-controlled, traceable, and reversible. This methodical approach is instrumental in cloud spend optimization.

Important note

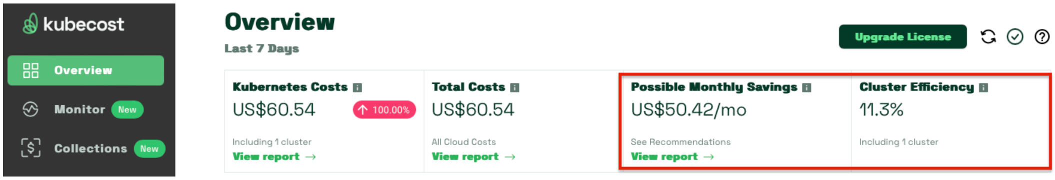
An example on GitHub under *Chapter 14*, `...optimization/kubecost`, demonstrates how to configure alerts in the Kubecost UI and via Helm Chart deployment using the `values.yaml` file.

When GitOps and Kubecost are used together, they provide a powerful framework for ongoing cloud spend optimization:

- **Automated resource optimization**: GitOps can automate the deployment of Kubecost's recommendations, ensuring that cost-saving measures are promptly and consistently applied across the infrastructure
- **Continuous monitoring and adjustment**: The combination allows for continuous monitoring of cloud spend and automatic adjustments based on predefined policies, ensuring that the cloud environment is always running in the most cost-effective manner
- **Enhanced collaboration**: By integrating these tools, financial and operational teams can collaborate more effectively, with GitOps providing the operational framework and Kubecost offering financial insights, leading to more informed decision-making

Let's imagine a new project comes in. The platform team provides the Kubernetes platform, but the FinOps team, which has a fixed budget of 500 USD per project, is responsible for managing the costs. The FinOps Team can create a budget with alerts in the UI or let the GitOps team deploy the alert via GitOps budgets for the project namespaces and clusters and set up **alerts**.

If the FinOps team, for example, wonders why the cumulative budget across the namespaces is between 70-100 USD, but the total spent budget is around 600 USD, they can use the alerts and UI to investigate the root cause. The Kubecost UI (*Figure 14.4*) reveals that the cluster efficiency, for example, is below 12%, and savings of about 50 USD are already possible with a 7-day retrospective:



Overview
Last 7 Days

Kubernetes Costs US$60.54 ↑ 100.00%
Including 1 cluster
View report →

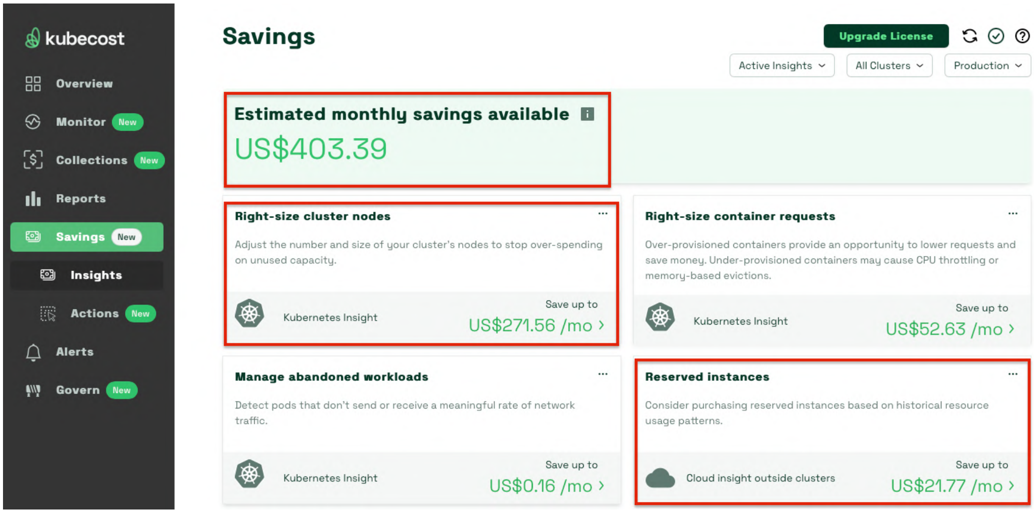
Total Costs US$60.54
All Cloud Costs
View report →

Possible Monthly Savings US$50.42/mo
See Recommendations
View report →

Cluster Efficiency 11.3%
Including 1 cluster

Figure 14.4 – Kubecost cluster efficiency ~12% for 7 days

The FinOps team then examines the potential savings in *Figure 14.5* to understand the origins of the costs. It quickly becomes apparent that the cluster for the project is oversized. Together with the platform team and by utilizing Kubecost, the teams involved in the project can implement sizing adjustments and mechanisms to modify actions accordingly. At this juncture, it's crucial that collaboration between the teams is effective and that all teams act in the interest of the company, sustainability, and the project. This is how cloud costs can be sensibly optimized:



Savings

Estimated monthly savings available
US$403.39

Right-size cluster nodes
Adjust the number and size of your cluster's nodes to stop over-spending on unused capacity.
Kubernetes Insight — Save up to US$271.56 /mo ›

Right-size container requests
Over-provisioned containers provide an opportunity to lower requests and save money. Under-provisioned containers may cause CPU throttling or memory-based evictions.
Kubernetes Insight — Save up to US$52.63 /mo ›

Manage abandoned workloads
Detect pods that don't send or receive a meaningful rate of network traffic.
Kubernetes Insight — Save up to US$0.16 /mo ›

Reserved instances
Consider purchasing reserved instances based on historical resource usage patterns.
Cloud insight outside clusters — Save up to US$21.77 /mo ›

Figure 14.5 – Kubecost savings recommendation

In conclusion, leveraging GitOps alongside Kubecost can transform cloud spend optimization from a reactive to a proactive endeavor. This integration not only provides granular insights into cloud usage and expenses but also automates the application of cost-saving strategies, ensuring that cloud resources are utilized efficiently and economically, and aligning cloud expenditure with organizational budgetary goals and operational requirements.

In the following section, we will explore the utilization of GitOps along with diverse tools to progress toward sustainability.

Assessing carbon footprint and promoting green operations

Assessing the carbon footprint and promoting green operations in cloud environments are critical steps toward achieving sustainability in IT operations. Tools such as `kube-green` [4] and **Armada** [5] offer innovative approaches to managing and reducing the environmental impact of cloud computing.

Assessing carbon footprint with kube-green

The `kube-green` tool focuses on optimizing resource usage in a way that correlates directly with energy consumption, thereby minimizing the environmental impact. Here's how it contributes to green operations:

- **Workload scheduling**: The `kube-green` tool can intelligently schedule workloads to run during times when the energy grid is supplied by renewable sources, thus promoting the use of green energy
- **Resource optimization**: It helps in fine-tuning the allocation of resources, ensuring that applications use only what they need and reducing the overall energy consumption of the data center
- **Idle resource management**: The tool can automatically scale down or turn off idle resources, significantly cutting down energy waste and associated carbon emissions

Implementing `kube-green` allows organizations to move toward carbon-neutral computing, aligning IT operations with broader environmental sustainability goals.

Promoting green operations with Armada

While not a direct tool like `kube-green`, Armada represents the concept of a fleet management approach in Kubernetes, which can be adapted to promote green operations. By managing clusters efficiently, Armada can help in the following ways:

- **Cluster consolidation**: Optimizing the number of active clusters and nodes based on demand, **reducing energy consumption** by avoiding the over-provisioning of resources
- **Energy-efficient deployments**: Facilitating the deployment of applications in an energy-efficient manner, potentially integrating with tools that forecast the availability of green energy
- **Monitoring and reporting**: Providing insights into the energy usage and efficiency of clusters, enabling informed decisions about how to reduce the carbon footprint

Adopting a strategy such as Armada allows organizations to oversee their Kubernetes clusters with an eye toward sustainability, ensuring that the infrastructure is as energy-efficient as possible.

Assessing carbon footprint by integrating with GitOps

Integrating tools such as `kube-green` and concepts such as Armada into a GitOps workflow can further enhance their effectiveness:

- **Automated implementation**: GitOps can automate the deployment of `kube-green` policies and Armada strategies across all clusters, ensuring uniform adherence to sustainability practices

- **Continuous optimization**: With GitOps, the continual adjustment and optimization of green policies can be maintained across the life cycle of applications, keeping sustainability a priority in every deployment
- **Transparency and accountability**: The declarative nature of GitOps provides a clear, version-controlled history of all changes made to promote green operations, fostering transparency and accountability in sustainability efforts

By assessing the carbon footprint and promoting green operations, organizations not only contribute to the environmental sustainability of their cloud infrastructure but also align with global efforts to reduce carbon emissions. The integration of these practices with GitOps ensures a systematic, scalable, and effective approach to sustainable cloud computing.

As AI is becoming increasingly important and there are already tools that can provide teams with useful support, we will take a look at these in the next section.

Looking at GitOps and AI-driven automation

As the IT landscape becomes increasingly complex, especially when dissecting all aspects of Kubernetes in detail without the aid of GitOps or other tools, many layers accumulate, spanning from the operating system level through the network and up to **public key infrastructures** (**PKIs**). I notice more frequently that new employees in this field are finding it increasingly difficult to get started and to know where exactly to begin, especially when they are thrust into a project. Hence, there are practical AI tools available that can provide support in this area.

In this chapter, we will explore two tools. The first tool, **Robusta.dev**, serves as a **troubleshooting co-pilot** for developers, providing them with real-time insights and solutions. The second tool is a **retrieval-augmented generation** (**RAG**) model, a self-implemented middleware solution from **iits**, which can be used as an enhancement to support developers with a knowledge base. By combining these two tools and deploying them via GitOps in the relevant projects, I believe they can be extremely beneficial not only for developers new to the field but also for those who are already experienced. This integration is likely to result in a productive boost.

Robusta.dev

Robusta [6] is an open source platform designed to enhance the observability and debugging capabilities of Kubernetes. It's a tool that helps DevOps teams, **site reliability engineers** (**SREs**), and developers to get more insights into their Kubernetes clusters, facilitating better monitoring, alerting, and troubleshooting.

Key features of Robusta include the following:

- **Enhanced observability**: Robusta provides detailed insights into the Kubernetes environment, offering rich, actionable alerts and notifications. It transforms plain log files into more interactive, enriched data that helps in quicker understanding and resolution of issues.
- **Automated troubleshooting**: The platform can automate the troubleshooting process for common Kubernetes problems, reducing the manual effort required to diagnose and resolve issues.
- **Customizable playbooks**: Users can create custom playbooks to automate responses to specific incidents or alerts. This allows for a tailored response mechanism that can evolve with the needs of the infrastructure.
- **Integration with existing tools**: It integrates well with the existing ecosystem of DevOps tools, providing seamless connectivity with monitoring solutions, alerting tools, and messaging platforms such as Slack.
- **Open source community**: Being open source, it allows developers and users to contribute to the tool, fostering a community-driven approach to enhancing Kubernetes observability and management.

We will use the enhancement **ChatGPT** bot [*7*] with a customizable playbook to make the Robusta tool even more effective. But let's follow the next steps in the guide. We will use Prometheus and ChatGPT in our following example.

Important note

Prometheus is an open source monitoring and alerting toolkit widely used for its powerful querying language and ability to handle multi-dimensional data such as metrics from cloud and containerized environments, while **ChatGPT** is an advanced AI language model developed by **OpenAI**, capable of generating human-like text, engaging in conversation, answering questions, and providing information across a vast range of topics.

Let's get started:

1. Install the Robusta Python CLI:

   ```
   python3 -m pip install -U robusta-cli --no-cache
   ```

2. Generate a values file for Helm:

   ```
   robusta gen-config
   ```

   You will be guided through all necessary steps:

   ```
   Configure Slack integration? This is HIGHLY recommended. [Y/n]:
   Y
   If your browser does not automatically launch, open the below
   ```

```
url:
https://api.robusta.dev/integrations/slack?id=05b9c718-7f2e-
4749-82fe-c3f545266692
You've just connected Robusta to the Slack of: poc-chatgpt-
kubernetes
Which slack channel should I send notifications to? # pocs
Configure MsTeams integration? [y/N]: N
Configure Robusta UI sink? This is HIGHLY recommended. [Y/n]: Y
Enter your Google/Gmail/Azure/Outlook address. This will be used
to……
```

3. In the newest version, they validate the Helm over Helm lint, so you have to add `clusterName`:

```
clusterName: "aks-excelsior-development-2"
globalConfig:
  signing_key: ea657a0b******
  account_id: 7935371f******
sinksConfig:
- slack_sink:
    name: main_slack_sink
    slack_channel: pocs
    api_key: xoxb******
enablePrometheusStack: true
enablePlatformPlaybooks: true
runner:
  sendAdditionalTelemetry: true
rsa:
  private: ******
  public: ******
```

Then, you can modify the `generated_values.yaml` file to create multiple triggers. For the showcase, I will keep it simple and apply it over helm in the next step.

4. Install Robusta with Helm:

```
kubectl create ns robusta
helm repo add robusta https://robusta-charts.storage.googleapis.
com && helm repo update
helm install robusta robusta/robusta -f .generated_values.yaml
```

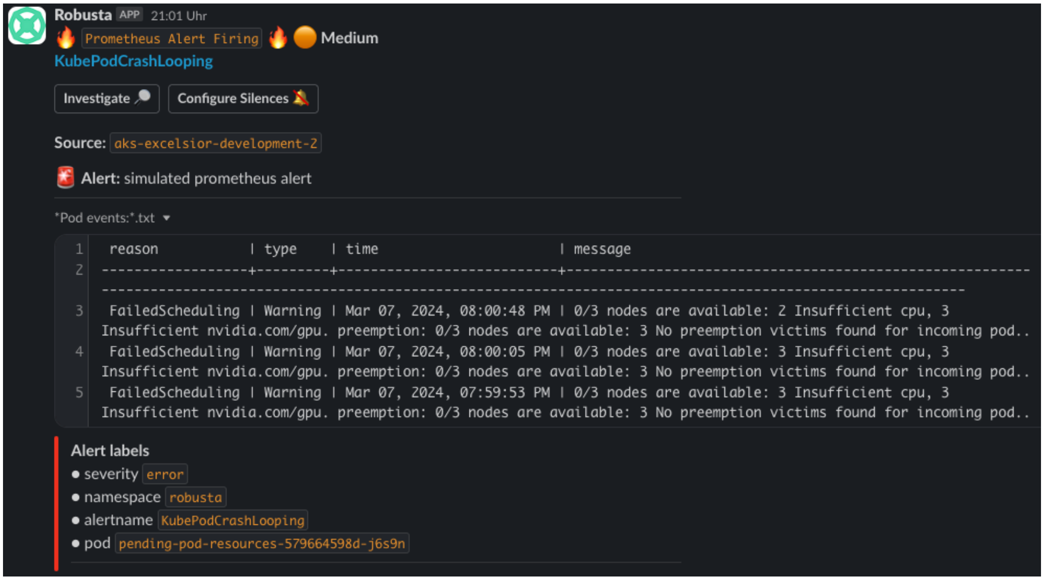
5. Run a demo to see if the trigger (**Prometheus**), action (**Logs-Enricher**), and sink (Slack) work:

```
k apply -n robusta -f https://raw.githubusercontent.com/robusta-
dev/kubernetes-demos/main/pending_pods/pending_pod_resources.
yaml
```

Now, trigger the Prometheus alert or wait 5-10 minutes:

```
robusta playbooks trigger prometheus_alert alert_
name=KubePodCrashLooping namespace=robusta pod_name=pending-pod-
resources-579664598d-j6s9n
```

You should receive a notification in Slack, as illustrated in *Figure 14.6*, containing metainformation along with the pod's logs:



Robusta 21:01 Uhr

Prometheus Alert Firing Medium

KubePodCrashLooping

Source: `aks-excelsior-development-2`

Alert: simulated prometheus alert

```
 reason           | type    | time                        | message
-----------------+--------+----------------------------+--------------------------------------------------------------
--------------------------------------------------------------------------------------------------------------
 FailedScheduling | Warning | Mar 07, 2024, 08:00:48 PM | 0/3 nodes are available: 2 Insufficient cpu, 3
Insufficient nvidia.com/gpu. preemption: 0/3 nodes are available: 3 No preemption victims found for incoming pod..
 FailedScheduling | Warning | Mar 07, 2024, 08:00:05 PM | 0/3 nodes are available: 3 Insufficient cpu, 3
Insufficient nvidia.com/gpu. preemption: 0/3 nodes are available: 3 No preemption victims found for incoming pod..
 FailedScheduling | Warning | Mar 07, 2024, 07:59:53 PM | 0/3 nodes are available: 3 Insufficient cpu, 3
Insufficient nvidia.com/gpu. preemption: 0/3 nodes are available: 3 No preemption victims found for incoming pod..
```

Alert labels
- severity `error`
- namespace `robusta`
- alertname `KubePodCrashLooping`
- pod `pending-pod-resources-579664598d-j6s9n`

Figure 14.6 – Robusta with Slack as sink without ChatGPT

6. We now need to include the playbook repository and integrate our custom playbook tailored for the ChatGPT action, as follows:

```
clusterName: "aks-excelsior-development-2"
globalConfig:
  chat_gpt_token: sk-dw******
  signing_key: ea657a******
  account_id: 7935371f******
sinksConfig:
- slack_sink:
    name: main_slack_sink
    slack_channel: pocs
    api_key: xoxb******
```

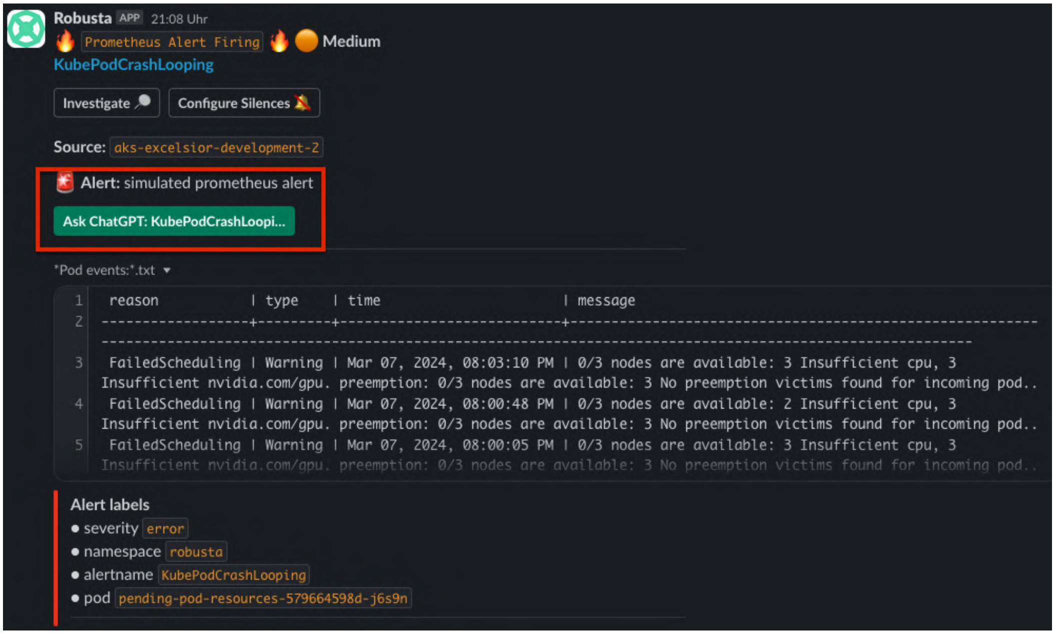
```
- robusta_sink:
    name: robusta_ui_sink
    token: eyJhY2NvdW******
enablePrometheusStack: true
# This part is added to the default generated_values.yaml
enablePlatformPlaybooks: true
runner:
  sendAdditionalTelemetry: true
rsa:
  private: ******
  public: ******

# This part is added to the default generated_values.yaml
playbookRepos:
  chatgpt_robusta_actions:
    url: "https://github.com/robusta-dev/kubernetes-chatgpt-bot.
git"

# This part is added to the default generated_values.yaml
customPlaybooks:
# Add the 'Ask ChatGPT' button to all Prometheus alerts
- triggers:
  - on_prometheus_alert: {}
  actions:
  - chat_gpt_enricher: {}
```

The highlighted section is the additional part.

7. Now, you should activate the Prometheus alert again using the following code:

```
robusta playbooks trigger prometheus_alert alert_
name=KubePodCrashLooping namespace=robusta pod_name=pending-pod-
resources-579664598d-j6s9n
```

You should now be able to view the **Ask ChatGPT** button, as depicted in *Figure 14.7*:



Robusta APP 21:08 Uhr
🔥 `Prometheus Alert Firing` 🔥 🟠 Medium
KubePodCrashLooping

Source: `aks-excelsior-development-2`

Alert: simulated prometheus alert

Pod events:.txt

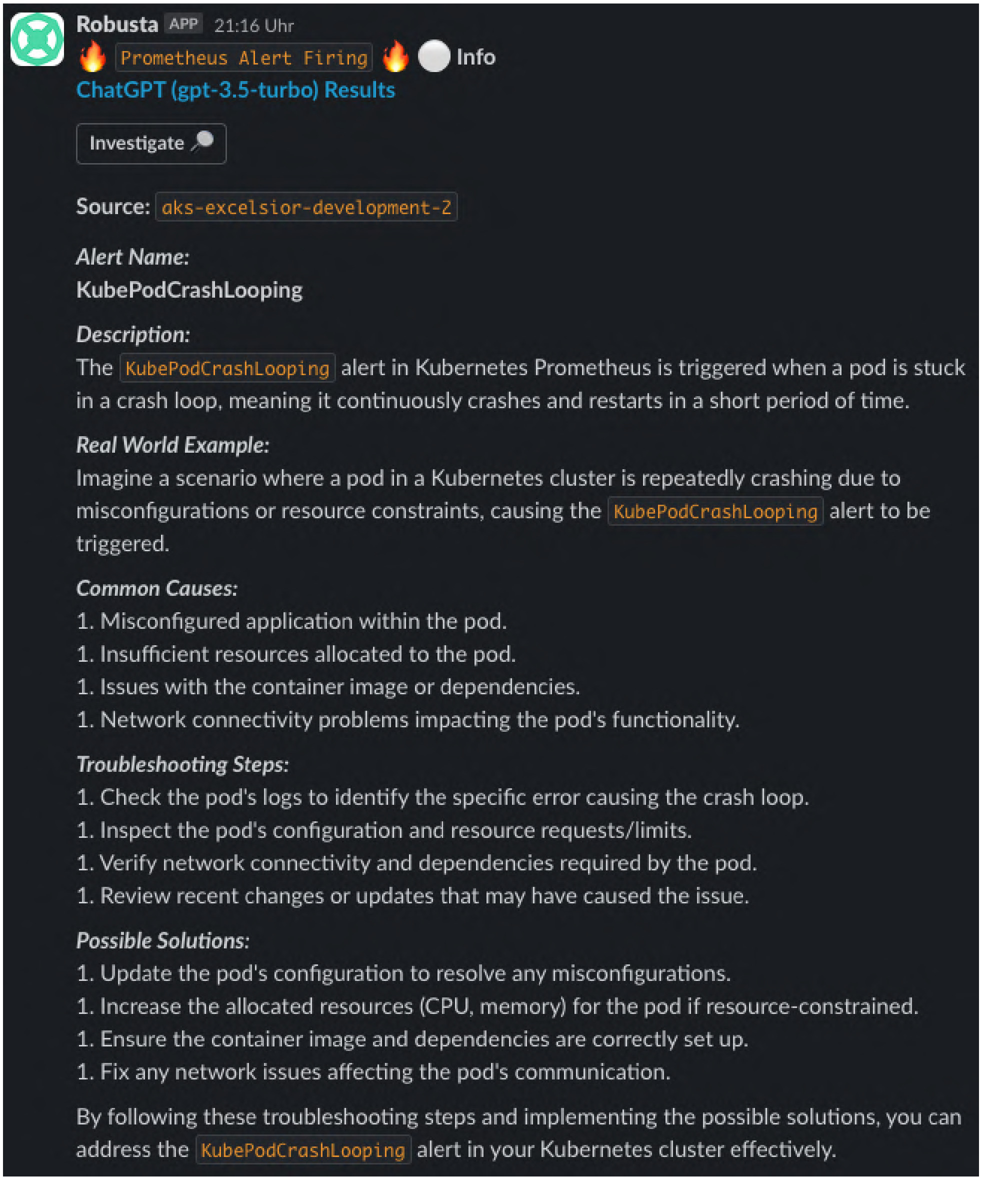
```
 reason           | type    | time                    | message
-----------------+---------+-------------------------+--------------------------------------------------------
------------------------------------------------------------------------------------------
 FailedScheduling | Warning | Mar 07, 2024, 08:03:10 PM | 0/3 nodes are available: 3 Insufficient cpu, 3
Insufficient nvidia.com/gpu. preemption: 0/3 nodes are available: 3 No preemption victims found for incoming pod..
 FailedScheduling | Warning | Mar 07, 2024, 08:00:48 PM | 0/3 nodes are available: 2 Insufficient cpu, 3
Insufficient nvidia.com/gpu. preemption: 0/3 nodes are available: 3 No preemption victims found for incoming pod..
 FailedScheduling | Warning | Mar 07, 2024, 08:00:05 PM | 0/3 nodes are available: 3 Insufficient cpu, 3
Insufficient nvidia.com/gpu. preemption: 0/3 nodes are available: 3 No preemption victims found for incoming pod..
```

Alert labels

- severity `error`
- namespace `robusta`
- alertname `KubePodCrashLooping`
- pod `pending-pod-resources-579664598d-j6s9n`

Figure 14.7 – Robusta with Slack as sink with ChatGPT

Upon clicking the button, you should receive a detailed explanation regarding the occurrence, common causes, potential troubleshooting steps, and viable solutions, as illustrated in *Figure 14.8*. This feature significantly aids developers by saving them considerable time and reducing the hassle involved in troubleshooting issues. Utilizing GitOps with Argo CD, you can deploy not only Robusta but also the extension across multiple clusters, as we have repeatedly demonstrated throughout this book:



Robusta APP 21:16 Uhr

🔥 `Prometheus Alert Firing` 🔥 ⚪ **Info**

ChatGPT (gpt-3.5-turbo) Results

Source: `aks-excelsior-development-2`

Alert Name:
KubePodCrashLooping

Description:
The `KubePodCrashLooping` alert in Kubernetes Prometheus is triggered when a pod is stuck in a crash loop, meaning it continuously crashes and restarts in a short period of time.

Real World Example:
Imagine a scenario where a pod in a Kubernetes cluster is repeatedly crashing due to misconfigurations or resource constraints, causing the `KubePodCrashLooping` alert to be triggered.

Common Causes:
1. Misconfigured application within the pod.
1. Insufficient resources allocated to the pod.
1. Issues with the container image or dependencies.
1. Network connectivity problems impacting the pod's functionality.

Troubleshooting Steps:
1. Check the pod's logs to identify the specific error causing the crash loop.
1. Inspect the pod's configuration and resource requests/limits.
1. Verify network connectivity and dependencies required by the pod.
1. Review recent changes or updates that may have caused the issue.

Possible Solutions:
1. Update the pod's configuration to resolve any misconfigurations.
1. Increase the allocated resources (CPU, memory) for the pod if resource-constrained.
1. Ensure the container image and dependencies are correctly set up.
1. Fix any network issues affecting the pod's communication.

By following these troubleshooting steps and implementing the possible solutions, you can address the `KubePodCrashLooping` alert in your Kubernetes cluster effectively.

Figure 14.8 – Robusta with Slack as sink with ChatGPT: support

Robusta aims to simplify the operational complexity of managing Kubernetes clusters, making it easier for teams to maintain **high availability** (**HA**), performance, and reliability of their containerized applications. By integrating with GitOps, Robusta enhances automation, ensures consistent environments through declarative configuration, and facilitates swift recovery and scalability, thereby streamlining the deployment process and reinforcing infrastructure resilience.

Private LLM solution with iitsAI

With tools such as **Robusta.dev**, we already have good support for observability and **incident management** (**IM**) in Kubernetes environments, but how can we further enhance productivity, onboard new developers, and improve existing ones? We are introducing an additional tool called **iitsAI**, from the **iits-consulting** company, operating under **iitsAI**. This tool allows developers to supplement their own data sources on top of a pre-trained module such as **Mistral 7B** and leverage multiple **large language models** (**LLMs; mixed models**).

> **Important note**
>
> **Ollama** is a cross-platform framework that lets you use LLMs such as **Mistral 7B** locally on your computer.
>
> An LLM is a type of AI program that can recognize and generate text, translate languages, write different kinds of creative content, and answer your questions in an informative way.
>
> RAG is a technique that improves the accuracy and reliability of **generative AI** (**GenAI**) models by incorporating factual information from external sources.
>
> **Mistral 7B** is a 7.3 billion-parameter **neural network** (**NN**) model trained on a massive dataset of text and code that can perform various tasks, including generating text, translating languages, writing different kinds of creative content, and answering your questions in an informative way.
>
> **iitsAI** offers a private LLM solution that runs on Kubernetes within your **own infrastructure**, enriched with your **own data**, providing you with complete end-to-end control. Alternatively, you can choose the hosted version by iits, available through a **sovereign cloud provider** in Germany. Both solutions include **single sign-on** (**SSO**) and fine-grained access control to the data through role mapping defined by you.

The functionality of LLMs is simplified as follows:

- The helpers of LLMs are large NNs trained on massive amounts of data
- Each LLM processes its part of the prompt and generates a response based on its knowledge and abilities
- The responses from the LLMs are then merged to create a comprehensive and informative response for the user

This way, complex tasks can be broken down into smaller parts and processed in parallel by multiple LLMs. The result is faster and more efficient processing of the prompt. Imagine a prompt that gets split into pieces and sent to different **AI helpers**. Each helper works on its piece, then all the answers are combined for a final response.

The tool built by iits provides the following architecture:

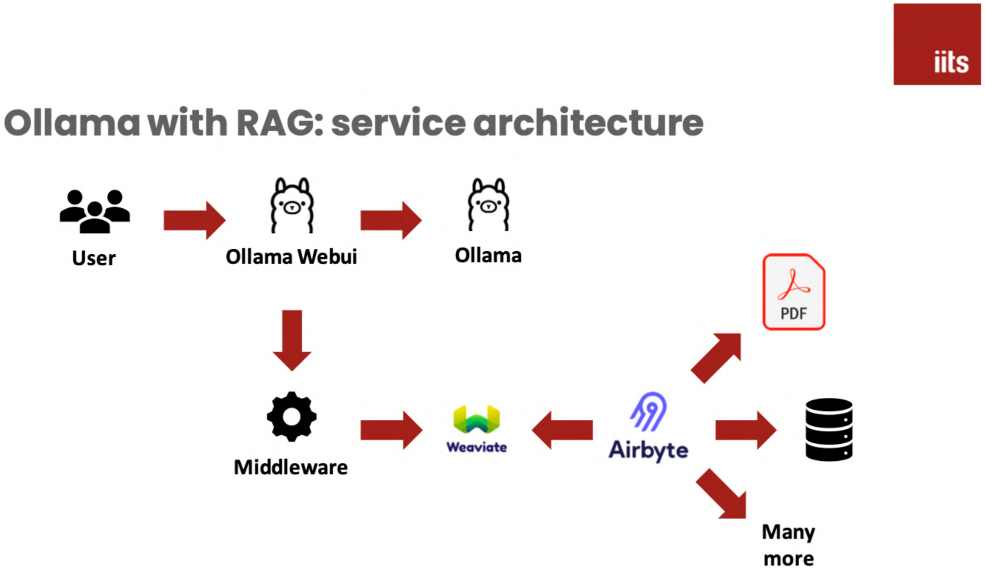
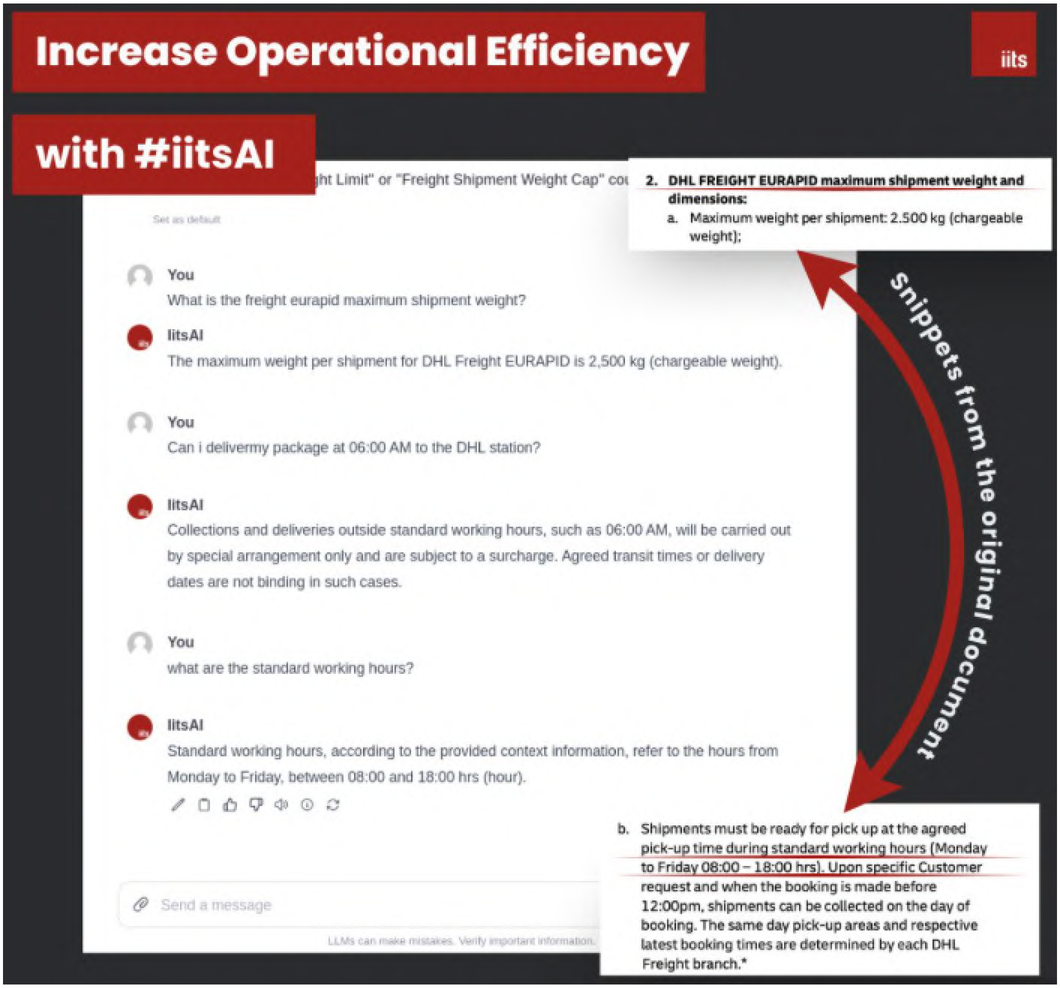


Figure 14.9 – Ollama with RAG: service architecture [8]

> **Important note**
>
> **Weaviate** is an open source vector database that enables you to store data objects and vectors.
>
> **Airbyte** is an open source data integration platform that helps you move data between different sources and destinations.

The user inputs a prompt into a web UI, as they are accustomed to with **ChatGPT**, which references a model via **Ollama**, such as **Mistral 7B**, and provides the base LLM model in the background. Then, **iitsAI** developed a middleware to conserve resources and extract the best possible outcome from the prompt. The prompt is divided into slices, undergoes several iterations, and is processed through multiple **LLMs**. Here, the LLMs act as a vector database (**Weaviate**), which connects to **Airbyte** through a custom-developed connector or uses Airbyte to load documents from various sources and provide them to the LLM agents. In the end, the user receives a response to their inquiry based on the provided documents, which looks like this:



Increase Operational Efficiency

with #iitsAI

iits

ht Limit" or "Freight Shipment Weight Cap" co

Set as default

You

What is the freight eurapid maximum shipment weight?

IitsAI

The maximum weight per shipment for DHL Freight EURAPID is 2,500 kg (chargeable weight).

You

Can i delivermy package at 06:00 AM to the DHL station?

IitsAI

Collections and deliveries outside standard working hours, such as 06:00 AM, will be carried out by special arrangement only and are subject to a surcharge. Agreed transit times or delivery dates are not binding in such cases.

You

what are the standard working hours?

IitsAI

Standard working hours, according to the provided context information, refer to the hours from Monday to Friday, between 08:00 and 18:00 hrs (hour).

Send a message

LLMs can make mistakes. Verify important information.

2. **DHL FREIGHT EURAPID maximum shipment weight and dimensions:**
   a. Maximum weight per shipment: 2.500 kg (chargeable weight);

Snippets from the original document

b. Shipments must be ready for pick up at the agreed pick-up time during standard working hours (Monday to Friday 08:00 – 18:00 hrs). Upon specific Customer request and when the booking is made before 12:00pm, shipments can be collected on the day of booking. The same day pick-up areas and respective latest booking times are determined by each DHL Freight branch.*

Figure 14.10 – iitsAI and logistics industry develop an internal knowledge base

The intriguing aspect of the solution is that it operates on Kubernetes and is deployed using GitOps. Consequently, the entire solution, from the UI to loading the base model and provisioning the files, is managed by the companies themselves on their own infrastructure. This autonomy is crucial for European businesses when employing technologies such as AI, Kubernetes, and GitOps.

Let's now explore how it can be integrated with GitOps and the opportunities this combination presents.

In use case A (*Figure 14.11*), a product provided to various customers can be delivered by the development team. **Simultaneously**, the team can supply a knowledge base to the customers on how to use the product. Users can then resolve all necessary queries using simple language requests without the need to sift through external documentation:

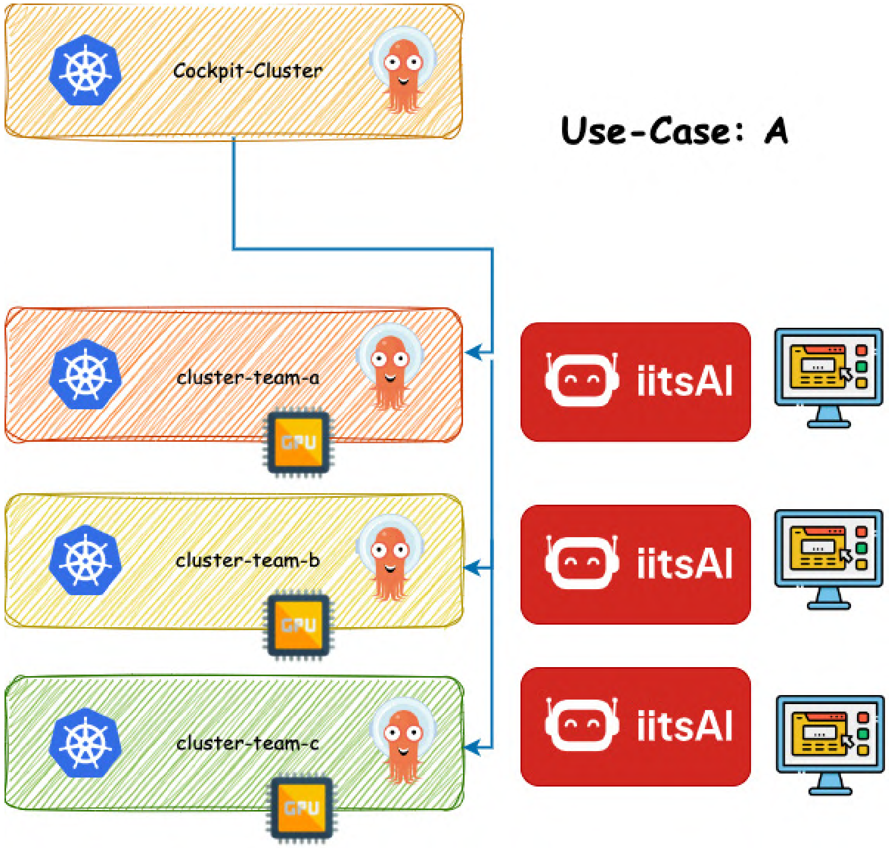


Figure 14.11 – iitsAI and custom app knowledge base

For use case B (*Figure 14.12*), as the platform team deploys the platform, they can also create an internal knowledge base about all the delivered tools, potential issues, releases, response times, FAQs, and so on, which can be accessed through a chat assistant in **natural language** (**NL**). This not only boosts efficiency but also eliminates several iterative steps, such as contacting the service desk, which typically involves searching the documentation provided and assisting at the first level of support:

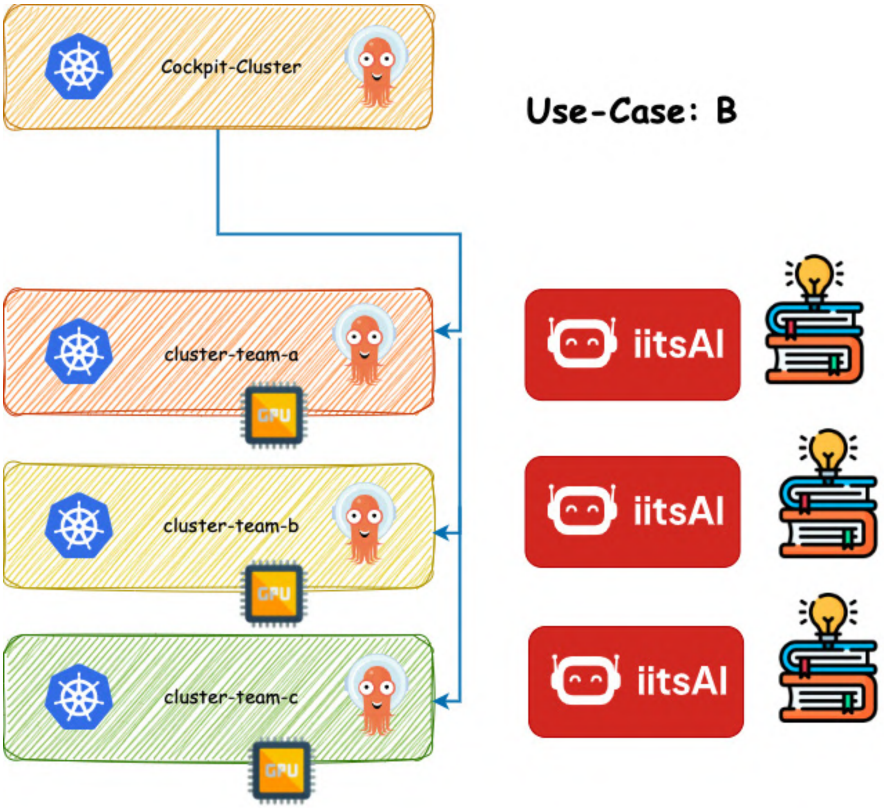


Figure 14.12 – iitsAI and platform knowledge base

Thanks to GitOps, the deployment is straightforward and can be scaled across clusters. Moreover, the GitOps approach aids in conserving resources since LLMs are **GPU-intensive**. It supports setups such as those shown in *Figure 14.13*, where only the UI is deployed on the workload cluster, while the middleware, the vector database, and the base model can operate on a service cluster that provides **GPU node pools**:

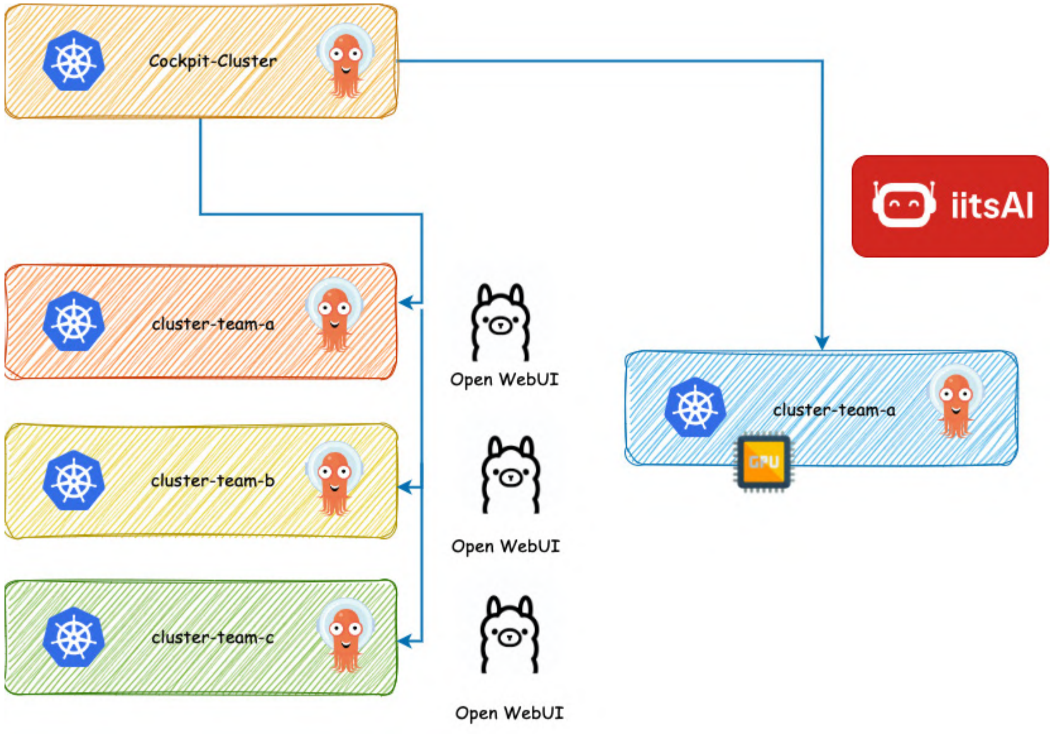


Figure 14.13 – iitsAI split UI and LLMs backend to save resources

As we can see, GitOps offers us tremendous flexibility at this juncture, reminding us of our **Kubernetes Service Catalog** (**KSC**) from previous chapters and how one can control their stack using labels. This allows for even more finely-grained control of the deployments. By now integrating AI, as just demonstrated, we can proactively assist with troubleshooting and provide an interactive knowledge base, hopefully making the added value clear.

In the next section, we will explore potential challenges and opportunities associated with the future use of GitOps.

Future challenges and opportunities in GitOps

In this section, it's crucial to address the collaborative dynamics and the overarching strategy required when multiple teams converge within the same operational framework, especially when they are required to share resources such as a repository. The key is to establish clear protocols from the outset, defining what is permissible and what needs coordination, much like setting contribution guidelines that dictate the sequence and considerations for changes.

This approach is reflective of the **open source software** (**OSS**) model, where contributors worldwide collaborate effectively on a project without it being owned by a single entity. Such a model underscores the potential of GitOps to harmonize efforts across diverse teams, transcending the traditional boundaries of DevOps. GitOps aims to unify development, operations, finance, sustainability, privacy, security, and other disciplines, enabling them to deliver a cohesive and robust product collectively.

Sustainability is poised to play a significant role in the realm of GitOps. Unlike traditional **continuous integration/continuous deployment** (**CI/CD**) pipelines that may run based on code changes, GitOps frequently reconciles the actual state of the system with the desired state defined in Git. While this ensures consistency and **reliability**, it also means that as the number of applications grows, resource consumption might increase correspondingly. This necessitates a thoughtful approach to how GitOps practices can be optimized to mitigate environmental impact without compromising the efficiency and reliability of the system. The future of GitOps will likely see innovations that allow it to scale sustainably, managing resource consumption wisely while maintaining **high standards** of automated, consistent, and declarative infrastructure management.

The role of GitOps in emerging technologies

In the rapidly evolving landscape of technology, GitOps stands out as a pivotal methodology, particularly in the realm of emerging technologies. It represents a paradigm shift, emphasizing the use of Git as an SSOT and automating the deployment process, thereby intertwining development and operations more closely than ever before. This methodology is becoming increasingly vital as organizations navigate the complexities of modern software deployment, especially in cloud-native environments and when dealing with innovative technologies. This list summarizes the role of GitOps in emerging technologies:

- **Enhancing cloud-native ecosystems**: As organizations increasingly adopt cloud-native technologies, the complexity of managing these environments grows. GitOps provides a structured, predictable method of managing this complexity, using version control to manage the entire state of the cloud-native infrastructure. This is particularly beneficial for Kubernetes, serverless architectures, and service mesh technologies, where configuration and state are critical. GitOps not only simplifies the management of these technologies but also enhances security, auditability, and transparency, which are paramount in cloud-native ecosystems.

- **Facilitating edge computing**: With the rise of edge computing, managing numerous remote environments consistently and securely has become a challenge. GitOps offers a way to deploy and manage applications across various edge locations reliably. By keeping configurations in Git, organizations can ensure that changes are traceable, verifiable, and automatically deployed, reducing the potential for human error and increasing operational efficiency.
- **Accelerating AI and machine learning (ML) operations**: The integration of GitOps in AI and ML operations streamlines the deployment of complex AI models and the continuous delivery of ML infrastructure. It ensures that data scientists and ML engineers can focus on model development and experimentation without worrying about the underlying infrastructure. GitOps automates the deployment process, ensuring consistent, repeatable, and reliable delivery of AI applications.
- **Supporting Internet of Things (IoT) deployments**: IoT involves managing a vast number of devices, each potentially running different software versions. GitOps can play a crucial role in automating the deployment and management of software across these devices, ensuring consistency, reliability, and security at scale. The declarative approach of GitOps means that the desired state of the IoT infrastructure can be version-controlled and automatically applied, reducing the complexity and increasing the scalability of IoT operations.
- **Enhancing security in DevSecOps**: In the DevSecOps world, integrating security into the **development and operations life cycle** is crucial. GitOps supports this by ensuring that all changes are reviewed, approved, and traceable through Git. This facilitates **rigorous audit trails**, quick rollbacks in case of issues, and a more secure infrastructure deployment pipeline, thereby enhancing the overall security posture of the organization.

GitOps is not just a trend but a fundamental shift in how we manage emerging technologies. It bridges the gap between development and operations, ensuring faster, more secure, and more reliable software delivery. As we continue to embrace new technologies and face new challenges, the principles of GitOps provide a solid foundation for managing the complexity of modern IT environments, making it an indispensable tool in the arsenal of today's technology leaders.

Summary

In this chapter, we hopefully gained clarity on how GitOps, through the use of various tools, enables both economic and ecological action. The importance of cultural change in the adoption of GitOps should also have become evident, as it allows numerous teams to collaborate and autonomously deploy tools, policies, and so on that align with the company's interests. While interpreting future trends isn't straightforward, the potential unlocked by combining the discussed AI tools should now be apparent. Ultimately, GitOps serves to deploy, expand, and maintain various tools or policies from different teams across a distributed cluster landscape or logically grouped clusters at scale.

In *Implementing GitOps with Kubernetes*, we have taken a hands-on approach to explore scalable and straightforward solutions grounded in real-world scenarios. From foundational concepts to advanced implementations, this book has aimed to equip you with the knowledge and tools necessary to harness the full potential of GitOps within Kubernetes environments. Through practical examples, insights from industry experts, and detailed explanations of best practices, we hope you now feel prepared to implement GitOps strategies effectively.

As you conclude this book, the next step is to apply what you've learned to real-world projects. Experiment with different tools, refine your workflows, and continually seek ways to optimize and secure your deployments. Stay engaged with the community, keep up with the latest trends, and never stop learning.

We wish you the best of luck on your GitOps journey ahead. May you achieve new heights in operational excellence, scalability, and innovation. Thank you for joining us on this journey, and we wish you success in all your GitOps endeavors. Wishing you all the best, Pietro and Artem.

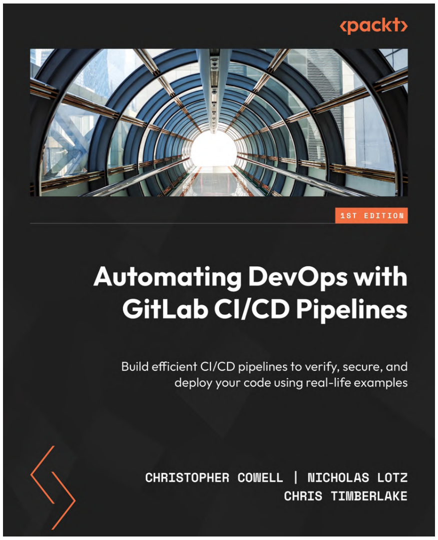
References

- [*1*] `https://github.com/opencost/opencost`
- [*2*] `https://github.com/kubecost`
- [*3*] `https://github.com/PacktPublishing/Implementing-GitOps-with-Kubernetes`
- [*4*] `https://github.com/kube-green/kube-green`
- [*5*] `https://github.com/armadaproject/armada`
- [*6*] `https://home.robusta.dev`
- [*7*] `https://github.com/robusta-dev/kubernetes-chatgpt-bot`
- [*8*] `https://github.com/iits-consulting/otc-terraform-template`

Index

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Automating DevOps with GitLab CI/CD Pipelines

Christopher Cowell, Nicholas Lotz, Chris Timberlake

ISBN: 978-1-80323-300-0

- Gain insights into the essentials of Git, GitLab, and DevOps
- Understand how to create, view, and run GitLab CI/CD pipelines
- Explore how to verify, secure, and deploy code with GitLab CI/CD pipelines
- Configure and use GitLab Runners to execute CI/CD pipelines
- Explore advanced GitLab CI/CD pipeline features like DAGs and conditional logic
- Follow best practices and troubleshooting methods of GitLab CI/CD pipelines
- Implement end-to-end software development lifecycle workflows using examples

Modern DevOps Practices

Gaurav Agarwal

ISBN: 978-1-80512-182-4

- •Explore modern DevOps practices with Git and GitOps
- Master container fundamentals with Docker and Kubernetes
- Become well versed in AWS ECS, Google Cloud Run, and Knative
- Discover how to efficiently build and manage secure Docker images
- Understand continuous integration with Jenkins on Kubernetes and GitHub Actions
- Get to grips with using Argo CD for continuous deployment and delivery
- Manage immutable infrastructure on the cloud with Packer, Terraform, and Ansible
- Operate container applications in production using Istio and learn about AI in DevOps

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.`packtpub.com` and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Share Your Thoughts

Now you've finished *Implementing GitOps with Kubernetes*, we'd love to hear your thoughts! If you purchased the book from Amazon, please click here to go straight to the Amazon review page for this book and share your feedback or leave a review on the site that you purchased it from.

Your review is important to us and the tech community and will help us make sure we're delivering excellent quality content.

Download a free PDF copy of this book

Thanks for purchasing this book!

Do you like to read on the go but are unable to carry your print books everywhere?

Is your eBook purchase not compatible with the device of your choice?

Don't worry, now with every Packt book you get a DRM-free PDF version of that book at no cost.

Read anywhere, any place, on any device. Search, copy, and paste code from your favorite technical books directly into your application.

The perks don't stop there, you can get exclusive access to discounts, newsletters, and great free content in your inbox daily

Follow these simple steps to get the benefits:

1. Scan the QR code or visit the link below

`https://packt.link/free-ebook/9781835884225`

2. Submit your proof of purchase
3. That's it! We'll send your free PDF and other benefits to your email directly