Esa Paavola

Managing Multiple Applications on Kubernetes Using GitOps Principles

Metropolia University of Applied Sciences

Bachelor of Engineering

Information and Communication Technology

Bachelor's Thesis

30 August 2021

Abstract

Author:	Esa Paavola
Title:	Managing Multiple Applications on Kubernetes Using GitOps Principles
Number of Pages:	41 pages
Date:	30 August 2021
Degree:	Bachelor of Engineering
Degree Programme:	Information and Communication Technology
Professional Major:	Software engineering
Supervisors:	Auvo Häkkinen, Principal lecturer Pasi Juvonen

Organizations are combining development and operations teams and principles to develop products with more excessive speed and efficiency. Combing development and operations teams increases product quality and value to the customer as a single team develops, tests, deploys and monitors the software. This specific method is called DevOps. The term DevOps comes from the combination of development (Dev) and operations (Ops). DevOps has become a prevalent philosophy across the software industry. DevOps principles enable continuous software delivery, faster lead time for changes, and faster resolution of problems.

GitOps represents a technology-focused approach to DevOps. GitOps term comes from the combination of a version control system called Git and IT operations (Ops). GitOps uses a version control system as the single source of truth, which means that everything that is part of a given system should be stored in the version control system. Using Git workflows and describing infrastructure as a code, GitOps brings developer experience to application management. Besides application management, GitOps takes DevOps best practices such as collaboration and continuous deployment tooling and applies them to infrastructure automation. GitOps also introduces operators for intelligent automation. Operators compare the running state of a system with the desired state and keep the system always in sync.

This thesis aims to introduce the benefits of using GitOps methods in application management on the Kubernetes cluster. The study contains a practical proof-of-concept, where multiple cloud applications are managed with the GitOps best practices. First, the GitOps principles and related technologies are discussed, and two open-source tools are compared. Requirements for implementing GitOps in application management are then studied based on the proof-of-concept.

Keywords: GitOps, DevOps, Kubernetes

Tiivistelmä

Tekijä:	Esa Paavola
Otsikko:	Managing Multiple Applications on Kubernetes Using GitOps Principles
Sivumäärä:	41 sivua
Aika:	30.8.2021
Tutkinto:	Insinööri (AMK)
Tutkinto-ohjelma:	Tieto- ja viestintätekniikka
Ammatillinen pääaine:	Ohjelmistotuotanto
Ohjaajat:	Yliopettaja Auvo Häkkinen Pasi Juvonen

Yritykset ja organisaatiot yhdistävät sovelluskehityksen ja IT-operaatioiden prosesseja kehittääkseen tuotteita nopeammin ja tehokkaammin. Kehitys- ja operaatiotiimien yhdistäminen lisää tuotteiden laatua ja arvoa asiakkaalle, kun yksi tiimi vastaa tuotteen kehityksestä, testauksesta, käyttöönotosta sekä monitoroinnista. Kyseistä menetelmää kutsutaan nimellä DevOps, ja siitä on tullut yleinen filosofia koko ohjelmistoalalla. DevOps periaatteet mahdollistavat jatkuvan ohjelmistotoimituksen, nopeammat ohjelmistojen päivitykset ja nopeamman ongelmien ratkaisun.

GitOps edustaa teknologiakeskeistä lähestymistapaa DevOpsiin. Kuten nimestä voi päätellä, GitOps käyttää Git-versionhallintajärjestelmää sovellusten ja järjestelmien määritysten hallintaan. GitOps tuo sovelluskehittäjille Gitistä tutut prosessit ja toiminnot käytettäväksi sovellusten hallinnassa. Sovellusten hallinnan lisäksi GitOps hyödyntää DevOpsin parhaita käytäntöjä, kuten yhteistyökäytäntöjä sekä jatkuvan koonnin työkaluja, ja soveltaa niitä infrastruktuuriautomaatioon. GitOpsin yksi periaatteista on kaiken automatisointi ja tämän ratkaisemiseksi GitOpsissa käytetään älykkäitä operaattoreita. Operaattorit vertaavat järjestelmän nykyistä tilaa haluttuun tilaan versionhallinnassa ja pitävät järjestelmän aina synkronoituna.

Tämän opinnäytetyön tarkoituksena on tutkia GitOps menetelmien etuja sovellusten hallinnassa Kubernetes-klusterissa. Tutkimus sisältää konseptitodistuksen, jossa useita pilvisovelluksia hallitaan GitOps käytäntöjen avulla. Ensin esitellään GitOps periaatteet ja niihin liittyvät teknologiat ja verrataan kahta avoimen lähdekoodin GitOps-operaattoria. GitOpsin käyttöä sovellusten hallinnassa tutkitaan konseptitodistuksen perusteella.

Avainsanat: GitOps, DevOps, Kubernetes

Contents

List of Abbreviations

API	Application programming interface. Provides connection between computers or computer softwares.
CD	Continuous delivery. Automated software release process.
CI	Continuous integration. Development process where code changes in repository trigger automated testing and building of software.
CPU	Central processing unit. An essential computer hardware part.
CRD	Custom resource definition. An extension of Kubernetes resource API, which gives developers an easy way to customize Kubernetes installation.
DevOps	Set of practices that combines development and operations to achieve more agile software production.
GitOps	Technological approach to DevOps. The term comes from the words Git (version control system) and IT-operations.
JSON	JavaScript object notation. Human-readable file format for transferring data objects.
OS	Operating system. Software for managing computer hardware.
SDK	Software development kit. Collection of tools for developing software provided in one package.
SHA-1	Secure hash algorithm 1. A cryptographic hash function.
SSH	Secure shell. A cryptographic network protocol.

VM	Virtual machine. Virtualization of a computer system.
YAML	YAML ain’t markup language. Data serialization standard for any programming language with a human-friendly layout.

1 Introduction

Most people's lives get affected by software every day. Usually, the software works somewhere hidden and without notice. Many processes and systems are often controlled enterally by software. Demand for fast and reliable software delivery is high. Organizations are combining development and operations principles to develop products with more excessive speed and efficiency. This specific method is called DevOps. The term DevOps comes from the combination of development (Dev) and operations (Ops). DevOps has become a prevalent philosophy across the software industry. DevOps principles enable continuous software delivery, faster lead time for changes, and faster resolution of problems. GitOps represent a technology-focused approach to DevOps. As the name suggests, GitOps uses a version control system as the single source of truth, which means that everything that is part of some system should be stored in the version control system. Using Git workflows and describing everything as code, GitOps brings developer experience to application management. Besides application management, GitOps takes DevOps best practices such as collaboration and CI/CD tooling and applies them to infrastructure automation. GitOps also introduces operators for intelligent automation. Operators, sometimes called software agents, are software-based systems that control the cloud environment. Operators compare the running state of a system with the desired state and keep the system always in sync.

This thesis was done for Nokia Solutions and Networks Oy for the Edge Cloud Platform unit. The study was conducted in a team that develops Edge Automation Tool. The Tool automates both hardware and cloud edge infrastructure workflow. It offers users more accessible and efficient service operations through automation. Nokia Edge Automation Tool was used in the proof-of-concept where it was deployed using GitOps principles.

GitOps methods can be applied for almost any cloud-native stack combined with any version control system. This thesis aims to introduce the benefits of

using GitOps methods in application management on the Kubernetes cluster. Using GitOps methods with infrastructure automation tools such as Ansible or Terraform is not in the scope of the present study. Also, Git is the only version control system, and Kubernetes is the only container orchestration system this study focuses on. This thesis introduces GitOps principles and methods along with tools and technologies. The study also compares two Cloud Native Computing Foundation's GitOps operator projects called Argo CD and Flux CD. The study contains a practical proof-of-concept, where multiple cloud applications are managed with the GitOps best practices.

This thesis contains six chapters. The first chapter provides an introduction of the project, objectives, and scope. Chapter 2 describes the theoretical background. In Chapter 3, GitOps tools and technologies are discussed. Chapter 4 contains the comparison of the open-source tools. Chapter 5 covers the proof-of-concept, and finally, chapter 6 summarises the study and assesses the results.

2 GitOps Principles

GitOps is a set of best practices for managing applications and infrastructure in a cloud environment. The term GitOps comes from the words git (version control system) and operations.[1] The term was introduced by Alexis Richardson in 2017 [2]. The core idea of GitOps is to use Git as the single source of truth, which means that everything that is part of some system should be stored in Git. For example, a single source of truth can consist of application source code, test cases, deployment definitions, infrastructure definitions and documentation. Git repositories that work as a single source of truth contains a declaratively described state of some system. The state described in Git is the desired state of the system. GitOps introduces procedures that enable synchronization between the desired state in Git and the actual state in the system. GitOps combines infrastructure as a code (IaC), continuous deployment (CD), and Git practices. Using Git as the single source of truth, GitOps introduces developer experience for operational tasks. In GitOps, pull requests (or merge requests) are used to

make changes to the infrastructure and applications [3]. A version control system is efficient for collaboration, change tracking, and making rollbacks.
GitOps principles have become popular in the cloud-native community and especially in the Kubernetes community [1 p.7]. GitOps is an approach that could be used with many different version control systems and cloud environments, but primarily GitOps methods are utilized with Git and container orchestrator platform Kubernetes.

2.1 Infrastructure as Code

Infrastructure as Code (IaC) is a way to manage IT infrastructure through code. The code consists of configuration files that define the infrastructure. The configuration files are infrastructure specifications that are easy to edit and distribute. Automating the process of setting up IT infrastructure with IaC removes the need for manual provisioning and managing servers and other infrastructure components. There are two different approaches for defining configuration files in IaC - declarative and imperative. The declarative approach defines the desired state, resources and specifications of the system, and a specific tool will handle the configuration. On the other hand, the imperative approach is used to define the steps needed to achieve the configuration, and the steps are executed in a specific order. GitOps procedures are leaning on a declarative approach, and therefore it is the recommended way to define infrastructure as a code. [4]

In GitOps, infrastructure as code principles are used to define everything as code. The considerable benefits of IaC are repeatability, reliability, efficiency, savings, and visibility. The repeatability means that codified infrastructure can be automatically deployed to multiple environments. The repeatable process allows faster recovery time, and it is less error-prone than a manually provisioned infrastructure. Reliability means that a codified process reduces the change of human errors significantly. The next benefit is efficiency, which means that IaC increases the productivity of the team. IaC increases efficiency because engineers can use the tools they already know, such as Git, APIs, SDKs, and text editors. Implementation of IaC requires time and effort, but the savings come in

the long run. Using IaC, the environments can be deployed automatically on demand instead of manually provisioning each environment. The last notable benefit is visibility. Visibility means that codified infrastructure or application configuration can be easily viewed in a version control system. Anyone who has access to that version control system can see the desired state of the system with ease. [1 p.9]

2.2 Desired System State in Git

Everything that is part of the single source of truth of a system should be stored in Git. GitOps is all about defining everything as a code and storing that code in a version control system. The code in a version control system is the desired state of the system.

GitOps embraces standard revision control processes such as code review, pull requests and merges to master for reviewing and approving changes to the desired state [1 p.7]. Code review is a process where one or several persons check code changes for quality problems. The author of the code should not be the one who reviews the changes. Although code review is good practice to reduce errors in code, developers can use it for sharing knowledge and giving credit to team members. [1 p.11]

Git offers promising features that fit in GitOps principles very well. Git stores all the changes (commits) made to the repository, which creates a history for the system definition. Git uses SHA-1 hash to make checksums of every change made in Git, so it is not possible to make changes without leaving a mark behind. The state of the system stored in Git can be audited when everything is tracked, and proper access control is set up. [5 pp.3-8] Git makes the rollbacks of the system trivial. Each commit in Git represents a particular state of the system. Git history provides access to the previous states of the system. The previous state can be recovered quickly with the Git command called git revert. Fast rollbacks reduce mean recovery time significantly. [1 pp.7-8]

As Git works as a single source of truth for the system, everything should be driven from there. Ideally, in GitOps, Git should be the only place where engineers make changes. In extreme view of GitOps software agents, or GitOps operators, are the only ones with permission to make changes directly to the system (for example, Kubernetes cluster). [1 pp.7-8] This kind of approach has significant benefits from the security point of view. Engineers do not need credentials to the Kubernetes cluster when everything is done in the version control system. Also, Git tracks every change and log every operation. [6 p.80]

2.3 Synchronise System Automatically

GitOps leans heavily on automation. Instead of pushing imperative commands directly to the system, the GitOps operator automates this process with declarative definitions. One of the fundamental principles is that the GitOps operator should automatically apply all the approved changes in Git to the system. GitOps operator will constantly watch the configuration repository for changes and reconciles the content of the Git repository with the resources running in the system. [7 pp.446-448]

GitOps operator has a key role in a continuous deployment model. Operator automation works between continuous integration pipeline and cloud environment (for example, Kubernetes cluster). GitOps workflow for continuous deployment is illustrated in Figure 1.

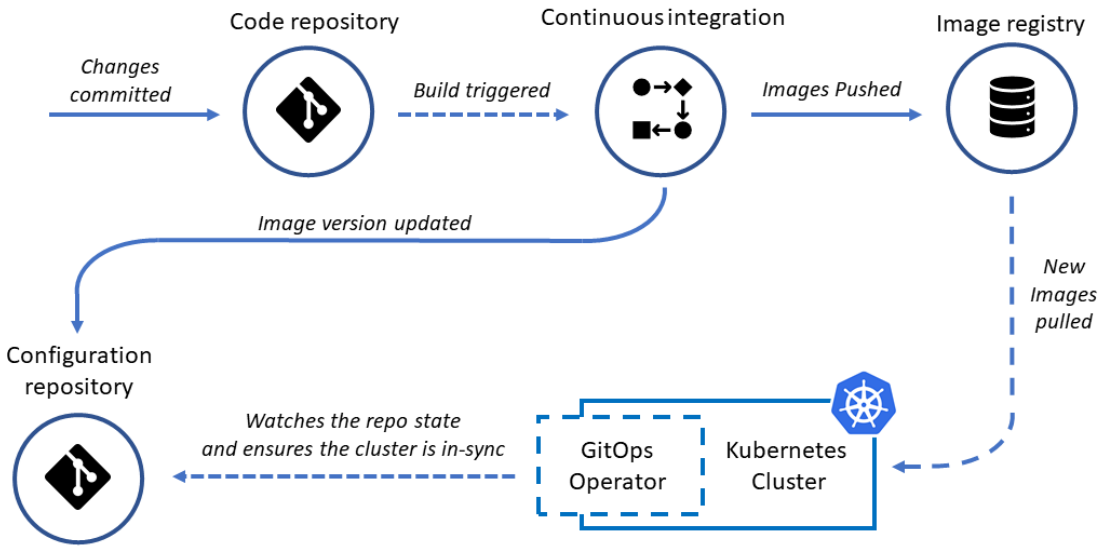

Figure 1. GitOps workflow for continuous deployment.

GitOps workflow for continuous deployment starts from making changes to the code repository. Changes in the code repository will trigger the continuous integration pipeline, which runs tests, make a new build of the software, and push the new images to the image registry. After creating new images, the continuous integration pipeline will update the image versions to the configuration repository. Configuration repository contains the declarative state of the system, which the GitOps operator watches for the changes. Any change made to the desired state in the configuration repository will trigger the operator to apply those changes to the running system. GitOps workflow for continuous deployment has many benefits. Firstly, it is declarative in nature, so any failures in deployments will result in reconciliation to a healthy state if needed. Secondly, the GitOps operator will ensure that cluster is in sync with the desired state in the Git repository. It means that the operator will roll back any changes made directly to the Kubernetes cluster, and it will also automatically delete any unknown resources found in the cluster. This feature of GitOps operators is very positive, but it is good to be mindful of situations where changes may intentionally break the model. [7 pp.446-448]

2.4 Observability

Observability means that the current running state of a system is known. It means that authorized users can describe the state of the system, inspect what is running on that system, and view its configuration. Software agents observe the current state of the system and alert when the state does not match the desired state. Observability is one of the critical things in GitOps, and it brings this ability to detect divergence from the desired state. [8] Observability should be thought of as a property of a system, as scalability and availability are system properties. Observability is a source of truth for the current running state of a system in the same way that Git is the source of truth for the system's desired state. [9 pp.4-6]

3 GitOps Tools and Technologies

In this chapter, GitOps tools and technologies are discussed. GitOps tools consist of tools needed for managing applications in a cloud-native environment. Some of the most important technologies for GitOps are explained. The technologies presented in this chapter are version control systems, GitOps operators, continuous integration, virtualization, container ecosystem, monitoring, and secretes management. Version control systems are presented with a popular tool called Git. Container orchestration system Kubernetes and package manager Helm are discussed alongside virtualization and container ecosystem. Also, a metrics-based monitoring system called Prometheus and a secrets management tool called Bitnami Sealed Secrets are presented.

3.1 Version Control Systems

A version control system (VCS) is a tool that manages and tracks changes of code, documents, or any content. A revision control system (RCS) and a source code manager (SCM) are generically also referring to the same tools as a version control system. There are some slight differences between these terms. However, all the systems resolve the same problems: tracking all the changes,

maintaining a repository of content, and providing a version history. [10] In this thesis, the term VCS is preferred over RCS and SCM.

Git has become the most popular version control system in the software industry. In principle, any version control system works with GitOps, but Git is the recommended system as the name suggests. GitHub, GitLab and Bitbucket are examples of Git providers. Many organizations and companies set up and manage their own Git servers. Git can be installed on-premises, or it can be used in the cloud.

3.2 Operators

The term operator is often mixed up with the term controller. This study uses the term GitOps operator over GitOps controller when referring to a continuous delivery tool. One way to think about the terms operator and controller differences is that all operators are controllers, but all controllers are not operators. An operator can be, for example, a domain or application-specific controller. [1 pp.38-40]

A basic GitOps operator for Kubernetes needs a continuous reconciliation loop in order to work. The loop consists of at least three steps illustrated in Figure 2. The loop starts from cloning the git repository that contains the latest configurations. Configuration is cloned to local storage. The second step is to discover manifests from cloned repository. Manifests are Kubernetes API object descriptions (for example, Kubernetes deployment or service) written in YAML or JSON. The third step is to apply all the manifests to the cluster. [1 p.45]

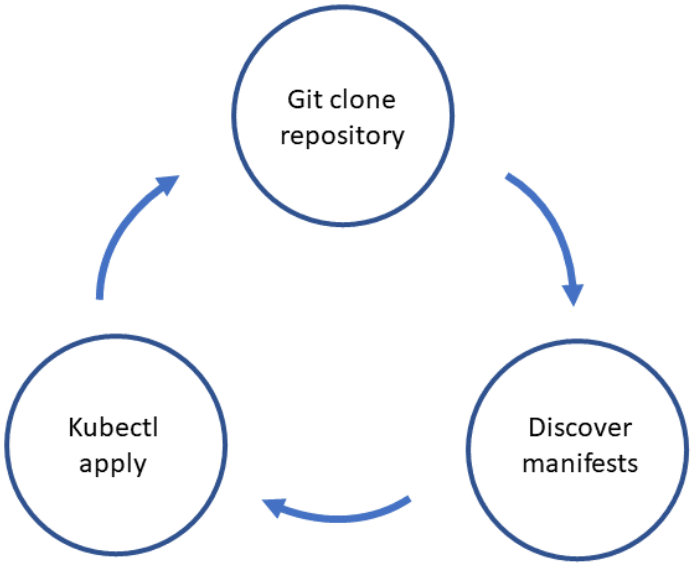

Figure 2. Simple GitOps operator reconciliation loop.

This kind of primary GitOps operator can be implemented in numerous ways. One way to implement a simple operator is by defining Kubernetes CronJob. CronJob is a Kubernetes object that runs a job periodically with a given schedule.

As GitOps is becoming more popular, so are GitOps operators. There are multiple open-source projects currently developing GitOps tools. Jenkins X, Argo CD, and Flux CD are one of the most popular GitOps tools.

3.3 Image Registries

Container images are stored in places called image registries. Image or a container image is a standalone, executable package of software, which contains everything needed to run an application. Registries provide easy access to the images. Image registries contain one or multiple image repositories, which in turn contains one or multiple images. Image registry overview can be seen in Figure 3. [11 pp.54-55]

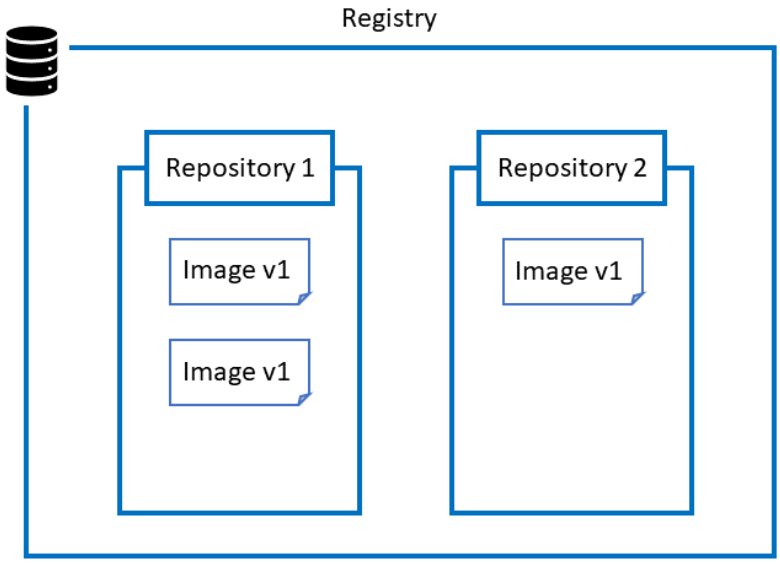

Figure 3. Container images can be organized into multiple repositories inside the registry.

Image registries are offered as a service by multiple providers. Docker Hub is one of the most popular public image registries. Many users start with Docker Hub because it is simple and easy to use. Public registries are considered to be great for individuals and small teams. For larger teams and organizations, private container image registries are often a better solution. Private registries can be hosted remotely or installed on-premises. The main reason to use a private registry over a public registry is increased efficiency and security.

3.4 Continuous Integration

Continuous integration (CI) is a software development practice where all the developers of a team regularly integrate code in the central repository. Every merged code change triggers build-and-test pipelines automatically for the software. Regularly and continuously deployed code offers notable benefits. Firstly, automation saves time compared to the manual integration process. Secondly, the whole team gets instant feedback and reports on integration failures, which increases transparency. Thirdly, the team detects defects and issues early, leading to a faster build cycle. [12 p.25]

GitOps embraces automation, and continuous integration has a vital role in GitOps flow. As seen in Figure 1, the CI pipeline creates the container images for the system and then updates the corresponding values to the configuration repository. Different CI tools are available as a service, or they can be installed on-premises or hosted remotely. Cloud build, Circle CI, Jenkins X, Travis CI, GitLab CI are popular tools for building continuous integration pipelines.

3.5 Virtualization and Container Ecosystem

Virtualization is a technology that creates an abstraction layer on top of hardware resources. The level of abstraction depends on the methods used for virtualization. Virtual machines (VMs) and containers are the two main virtualization techniques. [13] An architectural overview of these two techniques can be seen in Figure 4.

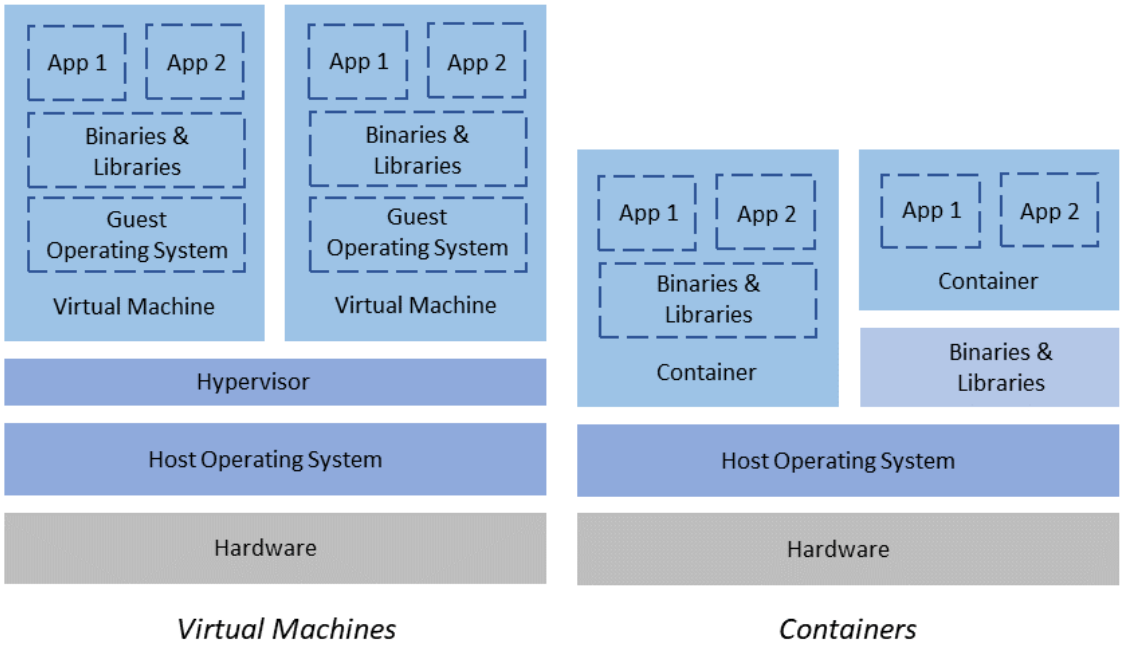

Figure 4. Comparison of container-based and hypervisor-based virtualization.

Virtual machines use a technology called a hypervisor. A hypervisor is software or hardware that enables VMs to run on a host machine. The host machine is the computer where the hypervisor runs virtual machines, and the virtual machines are called quest machines. VM is an abstraction of physical

hardware, and each VM has a full hardware stack which includes virtual CPU, network adapter and storage. Having full hardware stack virtualization means that each VM needs to have a full-blown operating system. [14]

Containers are a virtualization technique where virtualization of resources happens on the operating-system level. Each virtual machine needs an operating system, but containers share the host operating system, and in some cases, containers also share binaries and libraries. A shared operating system makes container deployment significantly smaller than hypervisor VMs. Small deployments make it possible to run hundreds of containers on a physical host. One great benefit of containers is that restarting the container does not require a reboot of the whole operating system. This benefit makes container deployments fast and flexible. [15 p.82] Containers provide a standard way to package applications and configurations into a single resource. This ability ensures that the application will run properly on any machine.

3.5.1 Container Orchestration

Container orchestration automates the deployment and management of containers on multiple hosts. Organizations that manage hundreds or thousands of containers dramatically benefit from container orchestration systems. The most popular container orchestration tools are Kubernetes, Docker Swarm, and Apache Mesos. These tools can automate configuration, scheduling, networking, scaling and resource allocation of containers. Container orchestration tools give a framework to run containers and microservice systems at scale. Orchestrator can operate a cluster of containers without any user input using policies. Orchestrator handles error detection and automatically fixes the system state. For example, the orchestrator can re-create failed containers. [16]

GitOps principles are mainly used alongside Kubernetes. Kubernetes, also known as K8s, is an open-source system that Google initially developed. Kubernetes was released in 2014, and it is based on Google's internal container

cluster management system called Borg [17]. Even though Google initially developed Kubernetes, it is not influenced by any single company. [1 p.22]

Running Kubernetes means running Kubernetes cluster. Kubernetes cluster is a set of node machines where the containerized applications will run. At minimum Kubernetes cluster has a control plane, which is the brains of the cluster and at least one compute machine (node). The control plane manages the desired state of the cluster; for example, it takes care of which applications are running and where they are running. The computing machine or worker node is responsible for actually running the application. Containerized applications run on pods. Pods are the most miniature Kubernetes objects. One pod can run one or multiple containers [18 p.56]. Deploying applications in different namespaces gives logical grouping. Namespaces are virtual clusters within the physical cluster. Namespaces help isolate different projects. Kubernetes uses a key-value database to store the desired state of the cluster. This highly available and consistent store is called etcd. [19 p.34]

Kubernetes has extendable architecture and open API, which have fuelled the growth of the community around it. Kubernetes API has an endpoint called resource, and it stores a collection of API objects. A custom resource (CRD) is an extension of this API, and it gives developers an easy way to customize Kubernetes installation.

3.5.2 Package Manager for Kubernetes

Package managers make platforms such as Kubernetes more accessible for their users. In order to use a container orchestrator such as Kubernetes, there needs to be software that can be installed on the Kubernetes cluster. The package manager's job is to make it easy to install, update and remove software on platforms. [20 p.12]

Helm is a package manager that is an integral part of the Kubernetes ecosystem. Helm is open-source software that is part of the Cloud Native

Computing Foundation. Helm removes the need for figuring out how to make an application run on Kubernetes. Helm packages the software into an easy-to-use format called chart. Helm does not call itself a configuration management tool, but that is how it is often used. Helm files are defined in .yaml format, and the charts can be parameterized. Parameterising Helm files makes it easy to produce charts, for example, for different environments. [1 pp.72-73]

3.6 Monitoring

Monitoring is one part of observation discussed earlier in Chapter 2.4. Monitoring as a term lacks consensus on what it means, but in this study, monitoring is thought of as a source of information for managing healthy systems production-wise. Monitoring systems collect and analyze metrics. Today's cloud-native software consists of many small parts, as microservice architecture is often the preferred solution. It is impractical to try to monitor the health of each component of an infrastructure or application manually. Monitoring plays a critical role in observing applications current running state, and it should be taken into account while developing software, not afterwards. [21 pp.7-8]

Monitoring of software systems has three main purposes. The first one is alerting, which is one of the most popular use cases of the monitoring systems. The monitoring system tells the developer that something is wrong, and the developer can look at the issue. The second purpose is debugging. After alerting, the developer needs to figure out what causes the issue, which is what debugging is about. Monitoring systems provide data that is used to resolve the root cause of the issue. The third purpose is trending. Alerting and debugging are more or less urgent events, but monitoring systems provide historical information to spot trends in the system. Trending is about monitoring how the system evolves. This information may help make enhancements to the system. [22 pp.15-16]

There are a lot of tools available for monitoring software systems in cloud environments. One of the most popular open-source tools for monitoring used with Kubernetes is Prometheus. Prometheus was initially developed by SoundCloud but donated afterwards to the Cloud Native Computing Foundation. Prometheus is a metrics-based monitoring system that lets developers and operators analyze how the applications are performing. All the data can be queried using PromQL query language and visualize easily on a web-based dashboard. Prometheus does not provide a dashboard, so a dashboard system such as Grafana is often used for visualizing the data. [22 pp.13-14]

3.7 Secrets Management

One of the key ideas in GitOps is to store everything in a version control system. The idea has many benefits, but also one big challenge and that is secrets. Managing secretes the GitOps way does not work out of the box. Storing secrets in Git or any version control system as plain text is not a good idea from the security point of view. Secrets can be stored in the Kubernetes cluster or somewhere outside the cluster, but in order to store secrets in Git, they need to be encrypted. [1 pp.185-186] Encrypting secrets in Git have gained popularity, especially one particular tool called Bitnami Sealed Secrets. Sealed Secrets offers a solution that makes storing secrets in Git safer. The idea of Sealed Secretes is to encrypt the sensitive data so it could be stored in Git and then decrypt the encrypted data inside the Kubernetes cluster. Bitnami Sealed Secrets helps to automate the encrypting and decrypting process by providing a controller and CLI tool. [1 p.194]

4 Comparison of Open-source Tools

In this chapter, two open-source tools called Argo CD and Flux CD are compared. Argo CD and Flux CD are both Cloud Native Computing Foundation (CNCF) incubation projects. CNCF is part of the Linux Foundation. Both projects offer the

same ability to connect multiple Git repositories to the Kubernetes cluster and sync the content between Git and cluster declaratively.

Flux is a set of continuous delivery solutions for Kubernetes. Weaveworks initially developed Flux, but it is now an open-source CNCF project. Flux version 2 consist of controllers and APIs which each serves own purpose. Flux calls this set of tools GitOps toolkit, and it is the runtime of Flux v2. The toolkit includes Source Controller, Helm Controller, Kustomize Controller, Notification Controller and Image automation controllers. [23]

Argo CD is a GitOps and continuous delivery tool for Kubernetes. Argo CD offers a way to do continuous delivery declaratively. The Tool consists of components that leverage GitOps principles. Two main ideas describe why Argo CD exist. Firstly, version control should be a single source of truth where applications, environments and configurations are defined declaratively. Secondly, deployment of applications and lifecycle management should be automated, auditable and in an understandable format. [24]

Argo CD monitors the state of the Kubernetes cluster and compares that to the desired state defined in version control. Argo CD will keep the cluster synchronized with the desired state. [24]

Argo CD runs inside the Kubernetes cluster. All Argo CD components should be in the same Kubernetes namespace. Argo CD consist of an API server, repository server, application controller, Dex server for identity management and Redis in-memory database for storing cached data. Notification controller and image update controller are additional Argo CD components. [24]

4.1 Repository Controller

GitOps tools watch version control system for declaratively described cluster state. The repository controller is part of the GitOps operator and is responsible for pulling the desired state from the remote Git repository. The repository

controller creates a copy of the Git repository locally inside the Kubernetes cluster. Figure 5 shows how the repository controller interacts within the GitOps model.

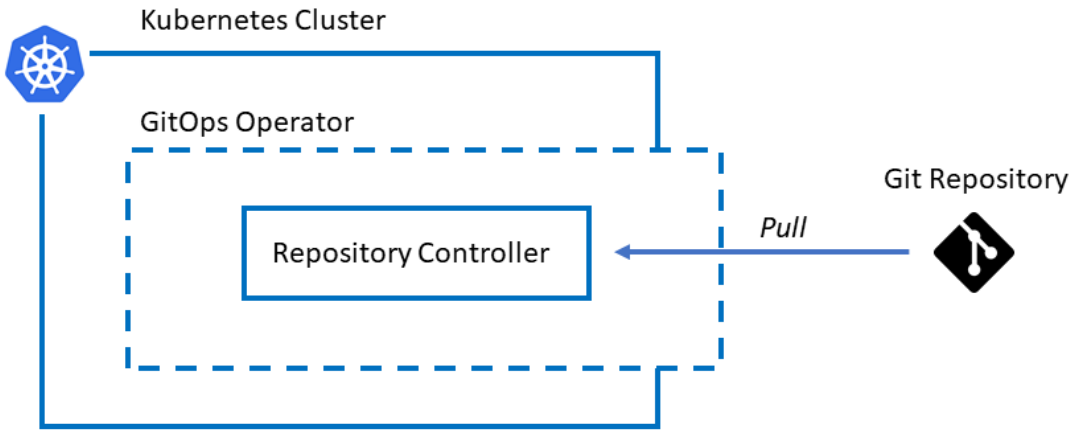

Figure 5. The repository controller pulls the content from the remote repository to a local cache inside a cluster.

In Flux CD, the Source Controller is used to watch different repositories. The source controller is responsible for validating the different sources, authenticating them, and watching for changes in sources based on constraints and fetch resources on-demand or scheduled. Flux CD source controller keeps a local copy of the source (for example, Git repository) and serves the fetched resources in a standard format (tar.gz, YAML) for other controllers inside a cluster. [25]

Argo CD Repository Server is responsible for watching changes in a version control system. Repository Server makes a local cache of the Git repository, which holds the application manifests. Application controller uses the manifests Repository Server stores in the local cache. [26]

Source Controller in Flux CD and Repository Server in Argo CD can handle multiple Git repositories. Both controllers support on-demand fetching and scheduled polling. The polling method is the default option for both controllers. There are some differences between these two tools. Firstly, deployments can be defined as Helm chart, Kustomize or YAML for Flux Source Controller,

whereas Argo CD Repository Server can also handle Ksonnet, Jsonnet and JSON files. Secondly, new deployments added to version control (for example, Git repository) are automatically registered when using Flux's controller, but Argo CD registering new deployments is manual.

4.2 Multi-tenancy

Multi-tenancy is an architecture where a single instance of software manages multiple customers or tenants. Multi-tenancy architecture brings operational advantages and costs savings. [27] Multi-tenancy is a valuable feature for GitOps operators when it comes to managing multiple clusters.

Organizations are moving to use multiple clusters to achieve better scalability, isolation, and availability. Multiple Kubernetes clusters are a helpful solution when development and production environments need to be separated. Also, when a single data centre is not recommended, multiple clusters can be deployed in different regions and availability zones. [28]

Flux CD is capable of deploying applications in multiple clusters. Flux CD is an operator which acts inside the Kubernetes cluster, and in a multi-cluster setup, Flux CD needs to be in each cluster. Flux configuration for multiple clusters can contain single or multiple git repositories, and all the clusters synchronize the state from a specified repository. [1 p.284]

Argo CD is capable of deploying and managing applications in multiple clusters. Argo CD needs to be installed only on one of the clusters. One Argo CD instance can manage multiple clusters. [29] The destination cluster for application deployment can be defined in Application CRD.

4.3 Image Automation

Container applications are distributed using container images. A container image is a static file that contains code that can be executed. Container images contain

system libraries, tools and settings to be able to run on a containerization platform. [30 pp.61-62] Images are stored in a registry, which can be a public or private repository. When an application is updated, a new version of the image is built and uploaded to the image registry. GitOps tools can be used to automate the deployment of the new image to the cluster.

Flux CD has image automation controllers which update the clusters state in the git repository when a new image is uploaded to the image registry. Image Reflector controller fetches the image tags from the registry by polling them regularly. Image Automation controller updates the manifests and commits the changes to a given Git repository. Image Automation controllers have three CRDs, which are used to set up the automation. ImageRepository CRD is used to specify the source for the image and how often metadata is fetched. ImagePolicy CRD is used to define the policies for updating manifests. Image policies offer different strategies for image updates. *SemVer* policy updates image to highest version according to constraints. *Alphabetical* policy chooses the last image tag when versions are listed alphabetically. *The numerical* policy chooses the last image tag when versions are listed numerically. [31]

Argo CD Image Updater automatically updates images of apps that Argo CD manages. Image Updater polls the image registry for updates and commits the changes to the Git repository. Applications images can be updated using different strategies. *Semver* strategy updates image to highest allowed version according to policy settings. *The latest* strategy updates the image to the most recent version. *Name* strategy updates image to the most recent image in alphabetical order. [32]

Both GitOps tools have very similar abilities in image automation solutions. Both tools support joint public image registries and private registries through configuration. There are few differences between the tools. Firstly, Flux CD image automation supports webhooks, whereas Argo CD can only use the polling method. Secondly, Argo CD can only update applications defined with Kustomize

or Helm compared to Flux CD, where every YAML file can be updated using an image automation controller.

4.4 Deployment Strategies

A deployment strategy is a way to update an application. The most common advanced strategies are blue-green, canary and A/B deployment. These strategies have different purposes and use cases, but the main goal is to ensure upgrades without downtime. [33] Kubernetes does not offer blue-green or canary deployments, and therefore a separate controller or operator is needed.

In this comparison of open-source tools, Argo Rollouts were used with Argo CD and Flagger was used with Flux CD. Both tools are part of CNCF. Flagger is part of GitOps Toolkit, and Argo Rollouts is part of the Argo Project.

4.4.1 Blue-green

The blue-green deployment strategy consists of two environments called blue and green. One of those environments serves live traffic for production (for example, green), and the other one (in this case, blue) is ready to be upgraded. After the new version of the application is tested in a blue environment, the production traffic can be shifted from green to blue. While the blue environment is serving production traffic, the green environment can be used to test the next release. Blue-green deployment gives developers and operators the capability to test an application or service in a similar environment as production. One of the key advantages of the blue-green strategy is rolling the application back to the previous version without downtime.

4.4.2 Canary

The idea of canary deployment is to test how the new release of an application works in a production environment. Canary deployment strategy consists of the primary production environment and the environment for an upgraded

application. The environment for a new version of an application is scaled up at the beginning of the upgrade process. [34] Canary deployment strategy is a progressive model where the upgrade of an application is done iteratively. Traffic to the new version of the application is gradually increased during the process. Traffic shifting between the two environments can be automated or provisioned manually. [35]

4.4.3 A/B deployment

A/B deployment strategy is similar to the canary, where changes are slowly rolled to the small part of users before making a new version available for everyone [36]. Specific conditions can be used during the release to test the application with particular users [37]. A/B and canary strategies sound similar, but they are used for different purposes. A/B is mostly used to test how the users respond to a change in the application. In A/B testing it is already known that the application works in the production environment.

4.4.4 Flagger

Flagger is an open-source project that Weaveworks initially developed in 2018. Flagger brings progressive delivery methods for Kubernetes. Flagger supports blue-green, canary and A/B deployments. [38] Flagger works with ingress controllers (Contour, Gloo, NGINX, Skipper and Traefik), service meshes (App Mesh, Istio and Linkerd) and metrics providers (Prometheus, Datadog, New Relic and CloudWatch). Flagger can be configured to send notifications and alerts for Slack, MS Teams, Discord, and Rocket using webhooks. [39] The Flagger has a custom Kubernetes resource definition called Canary. Canary is used to define a release process for an application. [40]

4.4.5 Argo Rollouts

Argo Rollouts enables different deployment strategies to be used with Argo CD. Intuit Inc. initially developed Argo Rollouts in 2019, and it brings the blue-green,

canary and A/B deployment strategies to the Kubernetes. [38] [41] Argo Rollouts can be integrated with ingress controllers (Nginx, ALB), service meshes (Istio, Linkerd and SMI) and metrics providers (Prometheus, Wavefront, Kayenta, Web, Kubernetes Jobs). [42] A Kubernetes workload resource called Rollout is used to create a deployment with an advanced deployment strategy. Rollout resource replaces Deployment object. Another two custom Kubernetes resources used with Argo Rollouts is Analysis and AnalysisRun. The Analysis is a template that defines what metrics are queried for the Rollout. The Analysis template can be used globally with multiple Rollouts. On the other hand, AnalysisRun is a resource that is attached to a single Rollout. The Analysis defines a threshold for specific metrics which determines if a rollout is successful or not. [43]

4.4.6 Project Comparison

Both projects, Argo Rollouts and Flagger, offer canary, blue-green and A/B deployment strategies. Flagger comes with a little more comprehensive support for different ingress controllers than Argo Rollouts. Both projects support three different service meshes and multiple metrics providers. There are some differences between these two projects. Firstly, Argo Rollouts benefits from the graphical user interface (GUI) of Argo CD. Figure 6 illustrates how the GUI offers a user-friendly way to observe deployments and traffic flow. Flux CD does not offer a GUI, and therefore Flagger used alongside Flux CD is based on command-line tools. Secondly, Flagger has vast support for sending alerts and notifications for third-party applications such as Slack, MS Teams and Discord using webhooks. Argo Rollouts does not support sending notifications to external applications using webhooks. Thirdly, there are some differences in how Argo Rollouts and Flagger defines the rollouts. Argo Rollouts uses a Kubernetes workflow resource called Rollout, which is used to replace Deployment objects. Flagger uses a custom resource called Canary, which does not replace the Deployment object but references it.

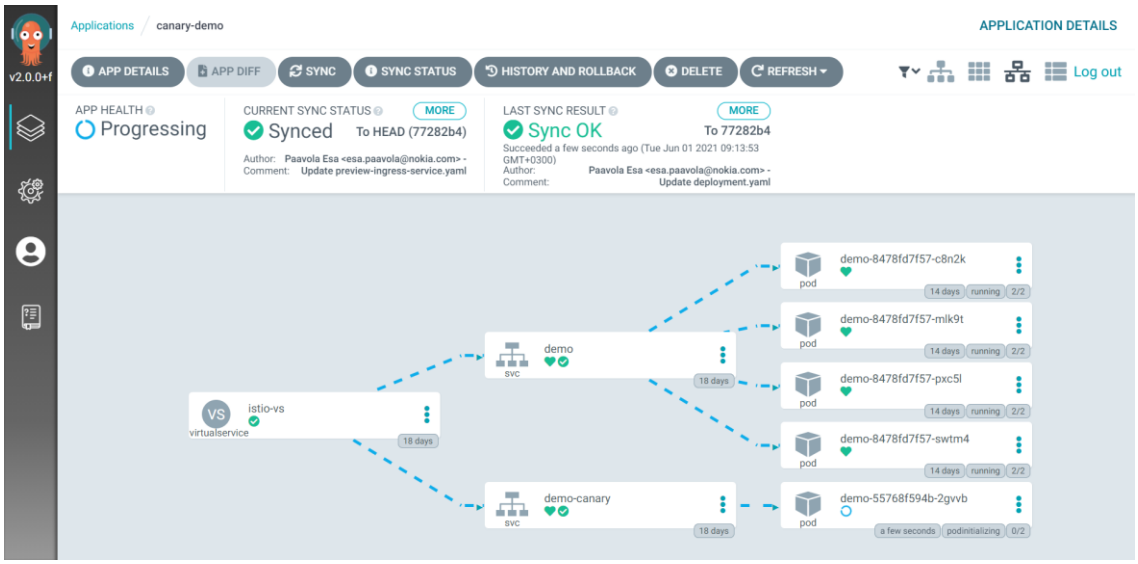

Figure 6: Traffic flow of a canary deployment illustrated in Argo CD dashboard.

Argo Rollouts and Flagger can be configured to do similar progressive deliveries using common strategies. The most significant differences are the lack of a graphical user interface in Flagger and the lack of support for external notifications in Argo Rollouts.

4.5 Summary

Both Argo CD and Flux CD deliver GitOps core features. A summary of the comparison can be seen in Table 1.

Table 1. Comparison of Argo CD and Flux CD.

	Argo CD	Flux CD
Unlimited number of git repositories	Yes	Yes
Detect changes in Git	Pull-based (every 3 min.), push-based (Webhook)	Pull-based (every 5 min.), push-based (Webhook)

Use repositories, branches and folders to define environments in Git	Yes	Yes
Application definitions	Application CRD	HelmRelease CRD, Kustomization CRD
Kubernetes resource definitions	Helm charts, Kustomize, YAML, Ksonnet, Jsonnet, JSON	Helm charts, Kustomize, YAML
Watch for changes in image registries	Yes	Yes
Multi-cluster support	Yes	Yes
Multi-tenancy with multiple clusters	Yes	No
Graphical user interface	Yes	No
SSO integrations for GUI and CLI	Yes	No

Argo CD and Flux CD are trying to solve the same problem, and thus they have many similarities. Both tools are easy to integrate, and they are very actively developed. The communities around both projects are also active.

5 Managing Applications on Kubernetes Using GitOps Principles

This chapter describes how the GitOps principles discussed earlier were used in a practical proof-of-concept. The objective of the proof-of-concept was to run the

Nokia Edge Automation Tool in a demo environment using GitOps methods. First, the Edge Automation Tool and the environment are discussed. Then, the implementation and configuration of the GitOps setup are introduced. Finally, application deployment, management and monitoring are presented.

5.1 Nokia Edge Automation Tool

Nokia provides edge cloud solutions to the customers, including automation tools for edge hardware and cloud infrastructure. Edge cloud environments consist of a more significant number of data centres compared to centralized cloud environments, so manual operations are more complex to execute. Automation is crucial for managing thousands of edge sites and enabling 5G cloud-based Communication Service Providers (CSPs) offerings. Automation brings operating expenses (OPEX) savings compared to today's semi-automation of central data centres, where manual operations are often required.

Nokia Edge Automation Tool consists of workflows and centralized edge cloud stack management. The overview of the edge cloud solution can be seen in Figure 7. [44]

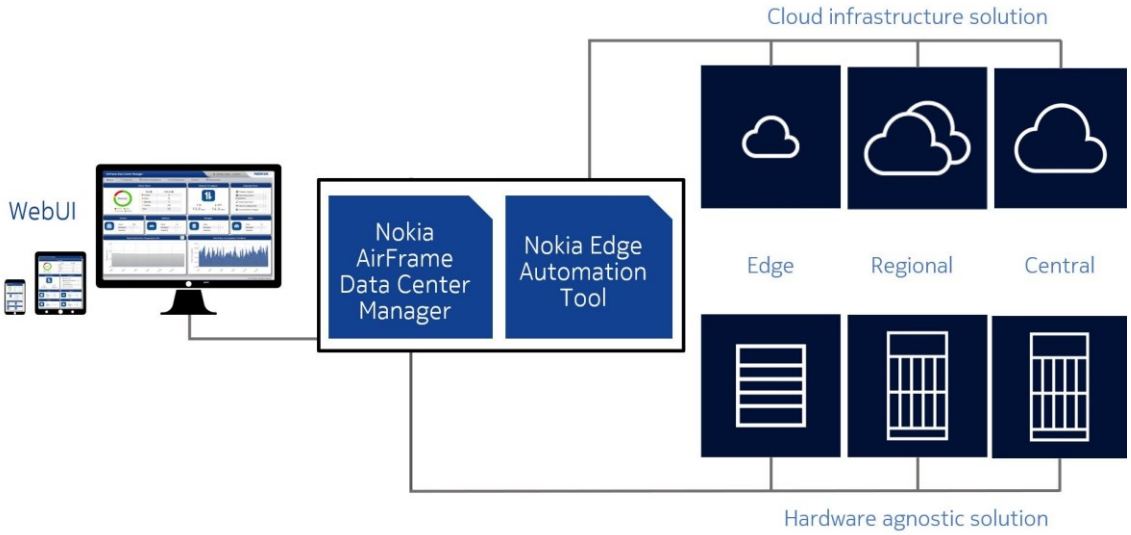

Figure 7. Nokia Edge cloud solution overview [45].

The central piece of the Nokia edge cloud solution is the Nokia AirFrame Data Center Manager and Nokia Edge Automation Tool. The Edge Automation Tool manages the cloud infrastructure and the AirFrame Data Center Manager is used for hardware management. These tools provide a web user interface for its users. [44] Nokia AirFrame Data Center Manager was not deployed in this proof-of-concept using GitOps principles.

5.2 Environment

This proof-of-concept was used to provide continuous deployment for Nokia Edge Automation Tool inside the company's internal demo environment. The demo environment is built on a virtual machine (VM) in the OpenStack private cloud. VM has CentOS 7.9 Linux distribution as an operating system. The Kubernetes cluster was installed on top of CentOS. Nokia provided an on-premise installed Git as a version control system for this PoC. The Kubernetes cluster had Nokia AirFrame Data Center Manager installed in advance.

5.3 Implementation

The implementation had two phases, installing and configuring Argo CD and defining the desired state in the Git repository. Setting up the environment was not in the scope of this proof of concept. GitOps architecture overview is illustrated in Figure 8.

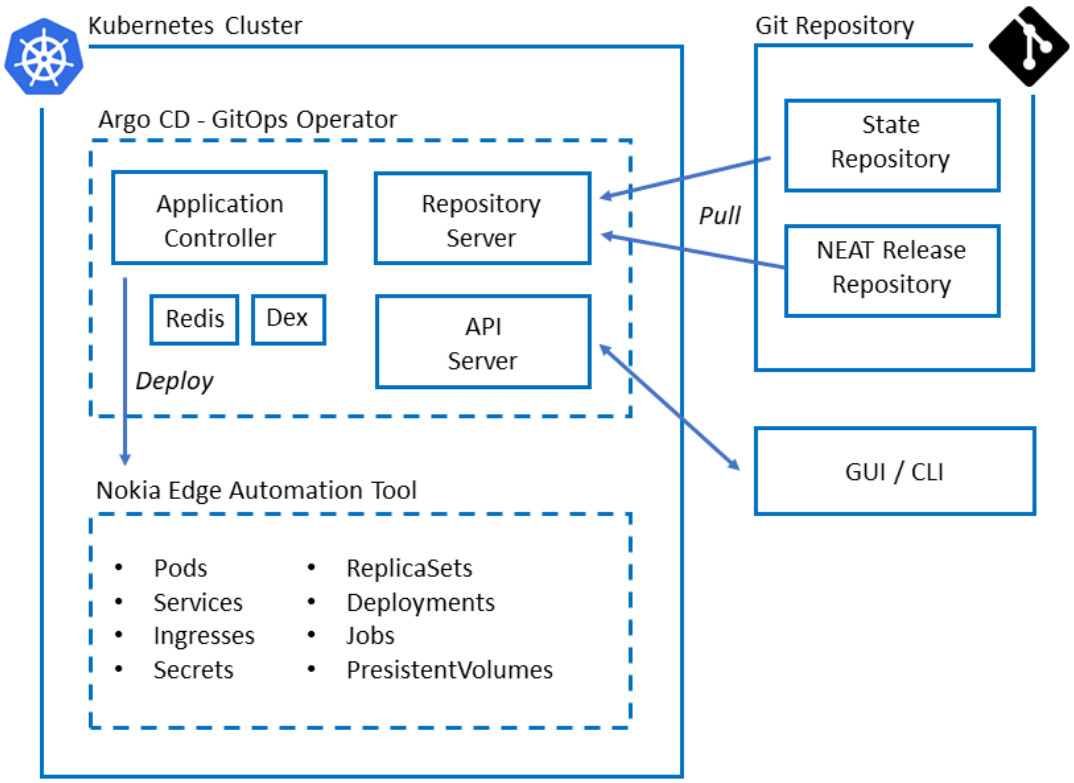

Figure 8. GitOps architecture of proof-of-concept.

The implemented GitOps architecture consists of a Kubernetes cluster and multiple Git repositories. The state repository was used to define the Argo CD Application CRDs and configuration files. The NEAT release repository had the actual application definition files. The Kubernetes cluster had multiple namespaces for isolating the tools and applications from each other. Argo CD and all of its components including Application Controller, Repository Server, API Server, Redis and Dex were installed on its own namespace. Nokia Edge Automation Tool was deployed to its own namespace. The deployment consists of multiple Kubernetes API objects including Pods, Services, Deployments, Ingresses, Secrets, Replica Sets and Jobs. In this proof-of-concept, Argo CD was accessed using a graphical user interface (GUI) and command-line interface (CLI). Argo CD API Server provides an access point where GUI and CLI connects.

5.3.1 Installation and Configuration of Argo CD

Argo CD was chosen as a GitOps operator for keeping the system synchronized with the desired state. Argo CD was selected because it is easy to use and provides both command-line interface (CLI) and graphical user interface (GUI), as mentioned in Chapter 3. Argo CD is one of the most popular GitOps tools, and it is trusted by a large number of companies worldwide. Also, as stated in comparison of open-source tools (cf. Chapter 3), Argo CD is compatible with other Argo projects such as Argo Rollouts and Argo Workflows.

The installation started by getting the CLI for Argo CD. The CLI was downloaded using the commands below.

```
VERSION=$(curl --silent "https://api.github.com/repos/argoproj/argo-cd/releases/latest" | grep '"tag_name"' | sed -E 's/.*"([^"]+)".*/\1/')

curl -sSL -o /usr/local/bin/argocd https://github.com/argoproj/argo-cd/releases/download/$VERSION/argocd-linux-amd64
```

The first command fetches the latest version tag for the Argo CD CLI, and the *curl* command was used for downloading the Tool. After the Tool was downloaded using output path */usr/local/bin/argocd*, the file was changed to executable using the following command:

```
chmod +x /usr/local/bin/argocd
```

Chmod +x command allows a file to be executed as a program. Argo CD was installed to the Kubernetes cluster used for the demo use-case. The installation was done by applying Argo CD install manifest from the GitHub of the Agro project. Before installing Argo CD, a namespace called argocd was created. It is recommended to use the default name argocd for the namespace. The list of commands executed for the installation is provided below.

```
kubectl create namespace argocd

kubectl apply --namespace argocd --filename
https://raw.githubusercontent.com/argoproj/argo-
cd/stable/manifests/install.yaml
```

The first Kubectl command creates a namespace and the second command applies Argo CD to the cluster. Argo CD installation was verified by watching for changes in the argocd namespace. After successfully installing all the required components of Argo CD, the resource view can be seen in Figure 9.

```
NAME                                    READY   STATUS    RESTARTS   AGE
pod/argocd-application-controller-0     1/1     Running   2          9d
pod/argocd-dex-server-9dc558f5-42s98    1/1     Running   0          9d
pod/argocd-redis-759b6bc7f4-p6th7       1/1     Running   0          9d
pod/argocd-repo-server-5fbf484547-lf4lg 1/1     Running   1          9d
pod/argocd-server-6d4678f7f6-nbzj2      1/1     Running   0          9d

NAME                              TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)                      AGE
service/argocd-dex-server         ClusterIP   10.98.65.173     <none>        5556/TCP,5557/TCP,5558/TCP   9d
service/argocd-metrics            ClusterIP   10.104.168.213   <none>        8082/TCP                     9d
service/argocd-redis              ClusterIP   10.103.71.95     <none>        6379/TCP                     9d
service/argocd-repo-server        ClusterIP   10.110.128.213   <none>        8081/TCP,8084/TCP            9d
service/argocd-server             ClusterIP   10.98.225.101    <none>        80/TCP,443/TCP               9d
service/argocd-server-metrics     ClusterIP   10.98.118.61     <none>        8083/TCP                     9d

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/argocd-dex-server   1/1     1            1           9d
deployment.apps/argocd-redis        1/1     1            1           9d
deployment.apps/argocd-repo-server  1/1     1            1           9d
deployment.apps/argocd-server       1/1     1            1           9d

NAME                                           DESIRED   CURRENT   READY   AGE
replicaset.apps/argocd-dex-server-9dc558f5     1         1         1       9d
replicaset.apps/argocd-redis-759b6bc7f4        1         1         1       9d
replicaset.apps/argocd-repo-server-5fbf484547  1         1         1       9d
replicaset.apps/argocd-server-6d4678f7f6       1         1         1       9d

NAME                                          READY   AGE
statefulset.apps/argocd-application-controller   1/1     9d
```

Figure 9. Pods, services, deployments, replicasets and statefulsets of Argo CD inside argocd namespace.

Argo CD API server is not exposed outside the Kubernetes cluster by default. Argo CD service called argocd-server was changed to the type LoadBalancer in order to access the API server. The change was made by executing the kubectl command presented below.

```
kubectl patch service argocd-server --namespace argocd -p '{"spec": {"type":
"LoadBalancer"}}'
```

Service can be updated using the patch command or by editing the whole manifest.

The initial admin password for Argo CD is stored as plain text in secret called argocd-initial-admin-secret in the same namespace as Argo CD. The following command shows how the password was retrieved and decoded:

```
kubectl --namespace argocd get secret argocd-initial-admin-secret --output
jsonpath="{.data.password}" | base64 -d && echo
```

The generated password was retrieved using kubectl and base64 decoding. The initial admin password was changed by logging into the API server with the Argo CD CLI tool and updating the credentials of the admin user. After the API server is exposed, the Argo CD GUI can be accessed with the host's IP address and port generated by Kubernetes.

Repositories that are begin used for application deployments need to be defined in Argo CD general configuration. General configuration is stored in ConfigMap called argocd-cm. Repositories can be added to ConfigMap using the graphical user interface or by editing configuration with kubectl. GitOps principles were used as the configuration file was defined as a code and stored in Git. The configurations can be seen in Listing 1.

```
apiVersion: v1
data:
  repositories: |
    - sshPrivateKeySecret:
        key: sshPrivateKey
        name: repo-gitops
      type: git
      url: git@git-host.com:user/fleet-repo.git
    - sshPrivateKeySecret:
        key: sshPrivateKey
        name: repo-gitops
      type: git
      url: git@ git-host.com:user/app-1.git
kind: ConfigMap
metadata:
  annotations:
  labels:
    app.kubernetes.io/name: argocd-cm
    app.kubernetes.io/part-of: argocd
  name: argocd-cm
  namespace: argocd
```

Listing 1. Argo CD general configuration file argo-cd-cm.yaml

All the manifests used in this proof-of-concept were held in private repositories. Private repositories were configured using SSH private key as a credential. As

seen in Listing 1, the SSH private key was held in Kubernetes secret called repo-gitops. Kubernetes secret was created from the private key using the command below.

```
kubectl create secret generic repo-gitops -n argocd --from-
file=sshPrivateKey=/$USER/.ssh/id_rsa
```

Argo CD has Custom Resource Definition (CRD) called AppProject, which was used to create some basic logical grouping for deployments. AppProject was configured to restrict the destination of applications. AppProject definition was written in YAML format and held in Git. The project called neat-demo was used for the Edge Automation Tool. The definition can be seen in Listing 2.

```
apiVersion: argoproj.io/v1alpha1
kind: AppProject
metadata:
  name: neat-demo
  namespace: argocd
  finalizers:
    - resources-finalizer.argocd.argoproj.io
spec:
  description: Neat demo project
  sourceRepos:
  - '*'
  destinations:
  - namespace: 'neat'
    server: https://kubernetes.default.svc
  clusterResourceWhitelist:
  - group: '*'
    kind: '*'
  namespaceResourceWhitelist:
  - group: '*'
    kind: '*'
```

Listing 2. AppProject definition represents a logical group of applications.

AppProject created for this setup allows any repository to pull manifests from, but the destination is restricted to a local cluster, and a specific namespace called neat. AppProject definition was saved to a file called projects.yaml.

5.3.2 Defining Desired State of Applications

Argo CD Application definitions were held in one repository, and the Nokia Edge Automation Tool deployment files in its repository in this proof-of-concept. The deployment files of Tool were defined using Helm charts, which was discussed in

Chapter 3.5.2. Argo CD supports Helm files, so charts cloud be used without any modifications. In Argo, CD applications are defined using Application CRD. In this proof-of-concept, two Application definitions were created. The first Argo CD Application references Nokia Edge Automation Tool and the second Application defines the first Application. The one that references the other application uses a pattern called app of apps. This pattern was used to make the deployment of other Applications easier in the future. Application definition that references to other Applications can be seen in Listing 3.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: demo
  namespace: argocd
  finalizers:
    - resources-finalizer.argocd.argoproj.io
spec:
  project: neat-demo
  source:
    repoURL: git@git-host.com:user/gitops.git
    targetRevision: HEAD
    path: apps
  destination:
    server: https://kubernetes.default.svc
    namespace: neat
  syncPolicy:
    automated:
      selfHeal: true
      prune: true
    syncOptions:
    - CreateNamespace=true
```

Listing 3. Demo Application's source points to the folder apps in the Git repository.

The Application CRD defines the source of the deployment files and the destination of the deployment. In addition to that, a synchronization policy can also be defined. If it is automated, Argo CD will automatically apply changes from Git to the cluster. In the Application definition called demo (Listing 3), the sync option CreateNamespace was set as *true*. This means that Argo CD will create the namespace specified in the destination spec if it does not exist.

The release files of the Nokia Edge Automation Tool are defined using Helm charts and Application definition was created to deploy those charts. Argo CD Application definition for the Edge Automation Tool can be seen in Listings 4.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: neat
  namespace: argocd
  finalizers:
    - resources-finalizer.argocd.argoproj.io
spec:
  project: neat-demo
  source:
    path: helm/neat
    repoURL: git@git-host.com:user/release.git
    targetRevision: HEAD
    helm:
      valueFiles:
      - values.yaml
destination:
    server: https://kubernetes.default.svc
    namespace: neat
  syncPolicy:
    automated:
      selfHeal: true
      prune: true
```

Listing 4. Argo CD Application definition referring to the Nokia Edge Automation Tool Helm charts.

Application definition that refers to the Helm charts of Edge Automation Tool has Git source, a local cluster as destination and an automated sync policy. In addition, the definition has some Helm parameters specified.

finalizers were added to the Application definition. This finalizer is a common Argo CD finalizer. When the Application definition gets removed from the Git repository, the finalizer will tell Argo CD to delete all the resources it has created base on the definition.

Argo CD Application definitions and configurations were held in the state repository to define the system's desired state. The content of the repository is presented in Listing 5.

```
Apps.yaml
apps/
     neat.yaml
argocd/
     argocd-cm.yaml
     projects.yaml
```

Listing 5. In the root of the state repository is the Apps.yaml file, which references the application inside the folder called apps.

Application file inside the folder apps reference to the actual application manifests in the Nokia Edge Automation Tool release repository.

5.4 Application Management

After installing Argo CD and defining the Applications and configurations as code in the Git repository, the Nokia Edge Automation Tool was deployed to the Kubernetes cluster. Before applying the Application files, the configuration and project files created earlier were applied. The following command was executed inside the folder of a local copy of the state repository in order to apply the configurations:

```
kubectl apply --filename ./argocd
```

After configuration files were applied, the Applications were applied to the cluster using the kubectl command below.

```
kubectl apply --filename Apps.yaml
```

These commands were the last thing that needed to execute using write permissions to the cluster. The GitOps principles explained in Chapter 2.2 were followed by letting the GitOps operator Argo CD manage the applications and the cluster. After this point, making changes to the Git repository would be the only way to update the demo environment.

After the Apps.yaml was applied to the cluster Argo CD synchronized the desired state from the Git repository and deployed the Nokia Edge Automation Tool to its own namespace. As seen in Figure 10, the successfully synchronized and deployed applications are marked as *healthy* and *synced*.

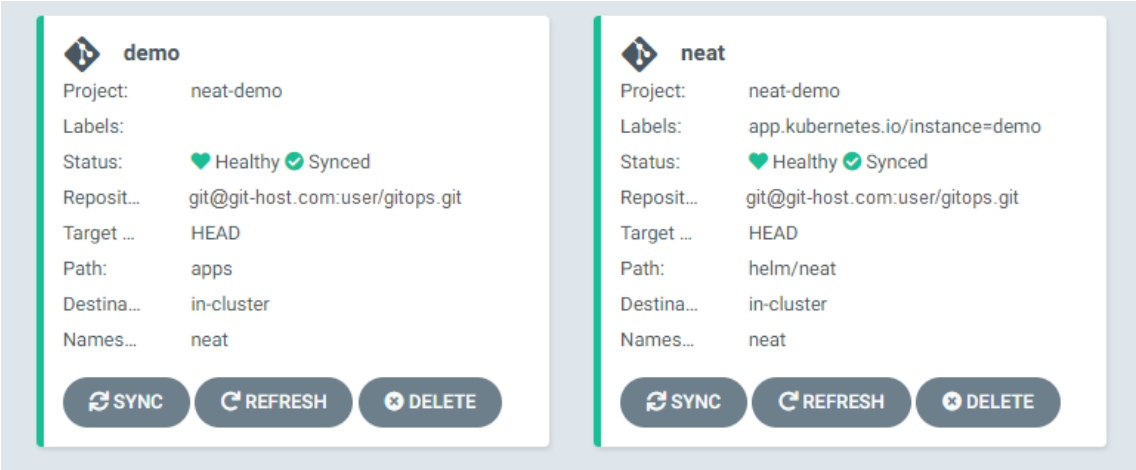

Figure 10. Argo CD graphical user interface lists all the applied applications on the front page.

Any commit to state repository or Nokia Edge Automation Tool Helm release repository will trigger a synchronization process that deploys the changes automatically to the cluster. The state of the applications can also be observed with the CLI tool.

5.5 Summary

This chapter discussed the practical implementation of cloud-native applications using GitOps principles. Kubernetes was selected as a container orchestrator system, and Argo CD was selected as a GitOps operator. A continuous deployment environment was created for Nokia Edge Automation Tool. Deployments were defined using Argo CD custom resource definition called Application. In addition, Argo CD configurations were defined using GitOps principles.

The proof-of-concept was made for the Nokia edge platform unit and it got positive feedback. GitOps principles are gaining more interest inside Nokia, but the future is still open. GitOps methods demonstrated in this chapter cloud be extended for many use-cases in telecom solutions. Building GitOps workflow for an existing application may require a lot of work, but many cloud-native applications usually have some of the building blocks of GitOps workflow

already in place. For example, Nokia Edge Automation Tool was defined as Kubernetes API objects using Helm charts.

6 Conclusions

The need for continuously delivered good quality software is high, and will only keep growing in the future. In this thesis, GitOps was introduced and presented as a possible solution for continuous deployment and application management in a cloud-native environment. Technology-focused GitOps principles were discussed, and the most common tools were presented. Open-sourced GitOps operators were compared, and one of the tools was chosen for the setup of proof-of-concept. The proof-of-concept was conducted for the team that develops Nokia Edge Automation Tool.

GitOps has many benefits, and it drives DevOps culture wherever its principles are applied. DevOps can be seen as part of the remarkable story of the agile revolution in the software industry. GitOps embraces the DevOps culture and brings a set of best practices that shape the way how software systems are defined and deployed. For example, GitOps offers developers a natural way to do operational tasks through Git. GitOps operators remove the need for making changes directly to the cloud environment (for example, Kubernetes cluster) as everything is driven from Git. Another benefit of GitOps is that the operators keep the system always in sync with the desired state in Git. This enables continuous deployment, which is faster than traditional CI/CD tools and more developer-friendly.

Everything as code is an idea that GitOps is driving forward. In the proof-of-concept part of this thesis, the applications and configurations were defined as code. Everything as code is a principle that can be applied to any part of the system that is being defined. Moving towards declaratively defined systems can be time-consuming, but it offers possibilities and benefits that imperatively defined systems cannot offer. For example, declaratively defined infrastructure can be automatically deployed to multiple environments. The repeatable process allows faster recovery time, and it is less error-prone compared to manually provisioned infrastructure. Also, declarative definitions increase

visibility as the desired state of the system can be easily seen in a version control system.

The proof-of-concept demonstrates how a cloud-native application can be deployed to the Kubernetes cluster. This GitOps methodology offers a relatively fast way to build a continuous deployment environment. This kind of environment can be helpful for both testing and production. Using GitOps principles, developers can easily define and manage similar environments only with a version control system.

References

1 Matyushentsev, Alex; Yuen, Billy; Ekenstam, Todd & Suen, Jesse. 2021. GitOps and Kubernetes. Shelter Island: Manning Publications.

2 Richardson, Alexis. 2017. GitOps - Operations by pull-request. <https://www.weave.works/blog/gitops-operations-by-pull-request> Accessed: 1.6.2021.

3 GitLab website. What is GitOps?. <https://about.gitlab.com/topics/gitops/>. Accessed: 10.5.2021.

4 Red Hat articles. What is Infrastructure as Code (IaC)?. <https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac>. Accessed: 1.6.2021.

5 Chacon, Scott & Straub, Ben. 2014. Pro Git, Second Edition. New York City: Apress Media.

6 Rice, Liz. 2020. Container Security. Sebastopol: O'Reilly Media.

7 Harris, John; Lander, Rich; Rosso, Josh & Brand, Alex. 2021. Production Kubernetes. Sebastopol: O'Reilly Media.

8 Weave Works article. Guide To GitOps. <https://www.weave.works/technologies/gitops/>. Accessed: 10.5.2021.

9 Majors, Charity; Fong-Jones, Liz & Miranda, George. 2022. Observability Engineering. Early release E-book. Sebastopol: O'Reilly Media.

10 Loeliger, Jon & McCullough, Matthew. 2012. Version Control with Git, 2nd Edition. Sebastopol: O'Reilly Media.

11 Poulton, Nigel. 2020. Docker Deep Dive. Birmingham: Packt Publishing.

12 Vadapalli, Sricharan. 2018. DevOps: Continuous Delivery, Integration, and Deployment with DevOps. Birmingham: Packt Publishing.

13 Jain, Shashank Mohan. 2020. Linux Containers and Virtualization: A Kernel Perspective. New York City: Apress Media.

14 Anderson, Jason; Agarwal, Udit; Li, Hongda; Hu, Hongxin; Lowery, Craig & Apon, Amy. 2016. Performance Considerations of Network Functions Virtualization using Containers. International Conference on Computing, Networking and Communications (ICNC). IEEE.

15 Bernsteinm, David. 2014. Containers and cloud: From lxc to docker to Kubernetes. IEEE Cloud Computing.

16 Rodriguez, Maria & Buyya, Rajkumar. 2020. Container orchestration with cost-efficient autoscaling in cloud computing environments. Handbook of research on multimedia cyber security. IGI Global, pp. 190-213.

17 Burns, Brendan; Grant, Brian; Oppenheimer, David; Brewer, Eric & Wilkes, John. 2016. Borg, Omega, and Kubernetes. Communications of the ACM, pp. 50-57.

18 Lukša, Marko. 2017. Kubernetes in Action. Shelter Island: Manning Publications.

19 Arundel, John & Domingus, Justin. 2019. Cloud Native DevOps with Kubernetes. Sebastopol: O'Reilly Media.

20 Butcher, Matt; Farina, Matt & Dolitsky, Josh. 2021. Learning Helm. Sebastopol: O'Reilly Media.

21 Bastos, Joel & Araujo, Pedro. 2019. Hands-On Infrastructure Monitoring with Prometheus. Birmingham: Packt Publishing.

22 Brazil, Brian. 2018. Prometheus: Up & Running. Sebastopol: O'Reilly Media.

23 Flux documentation. 2021. GitOps Toolkit components. <https://fluxcd.io/docs/components/>. Accessed: 23.5.2021.

24 Argo CD documentation. Overview. <https://argoproj.github.io/argo-cd/>. Accessed: 18.5.2021.

25 Flux documentation. 2021. Source Controller. <https://fluxcd.io/docs/components/source/>. Accessed: 23.5.2021.

26 Argo CD documentation. Argocd repo server. <https://argoproj.github.io/argo-cd/operator-manual/server-commands/argocd-repo-server/>. Accessed: 25.5.2021.

27 Stackowiak, Robert; Romano, Carla & Nath, Shyam. 2017. Architecting the Industrial Internet. Birmingham: Packt Publishing.

28 Arbezzano, Gianluca & Palesandro, Alex. 2021. Simplifying multi-clusters in Kubernetes. <https://www.cncf.io/blog/2021/04/12/simplifying-multi-clusters-in-kubernetes/>. Accessed 29.7.2021.

29 Matyushentsev, Alexander. Hassle-free multi-tenant K8S clusters management using Argo CD. <https://blog.argoproj.io/hassle-free-multi-tenant-k8s-clusters-management-using-argo-cd-7dd35619046a> Accessed: 25.5.2021.

30 Schenker, Gabriel N.; Saito, Hideto; Lee, Hui-Chuan Chloe & Ke-Jou Carol Hsu, Ke-Jou Carol. 2019. Getting Started with Containerization. Birmingham: Packt Publishing.

31 Flux documentation. 2021. Image reflector and automation controllers. <https://fluxcd.io/docs/components/image/>. Accessed: 23.5.2021.

32 Argo CD documentation. Argo CD Image Updater. <https://argocd-image-updater.readthedocs.io/en/stable/>. Accessed: 23.5.2021.

33 Red Hat OpenShift documentation. Using deployment strategies. <https://docs.openshift.com/container-platform/4.7/applications/deployments/deployment-strategies.html>. Accessed: 10.6.2021.

34 Sullivan, Dan. 2019. Official Google Cloud Certified Professional Cloud Architect Study Guide. Hoboken: Sybex.

35 Golowinski, Orit. DevOps institute article. 2020. What is Progressive Delivery?. <https://devopsinstitute.com/progressive-delivery/>. Accessed: 10.6.2021.

36 Hepburn, Mike; O'Connor, Noel & Picozzi, Stefano. 2017. DevOps with OpenShift. Sebastopol: O'Reilly Media.

37 Red Hat OpenShift documentation. Advanced Deployment Strategies. <https://docs.openshift.com/container-platform/3.11/dev_guide/deployments/advanced_deployment_strategies.html>. Accessed: 1.6.2021.

38 Flagger website. <https://flagger.app/>. Accessed: 10.6.2021.

39 Flagger documentation. <https://docs.flagger.app/>. Accessed: 28.5.2021.

40 Flagger documentation. How it works. <https://docs.flagger.app/usage/how-it-works>. Accessed: 28.5.2021.

41 Thomson, Danny. Argo project blog. 2019. <https://blog.argoproj.io/introducing-argo-rollouts-59dd0fad476c>. Accessed: 11.6.2021.

42 Argo Rollouts documentation. <https://argoproj.github.io/argo-rollouts/>. Accessed: 28.5.2021.

43 Argo Rollouts documentation. Architecture. <https://argoproj.github.io/argo-rollouts/architecture>. Accessed: 28.5.2021.

44 Nokia. Nokia Edge Automation Tool - Executive Summary. <https://onestore.nokia.com/asset/210359>. Accessed: 5.7.2021.

45 Nokia. Edge Automation. <https://www.nokia.com/networks/portfolio/automation/edge-automation/>. Accessed: 5.7.202