jamk

GitOps tool Argo CD in service management

Case: Conduit

Matti Korhonen

Bachelor's thesis
September 2021
Information and Communication Technology
Bachelor's Degree Programme in Information and Communication Technology

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Description

Korhonen, Matti

GitOps-työkalu Argo CD ja palveluhallinta
Case: Conduit

Jyväskylä: Jyväskylän ammattikorkeakoulu. Syyskuu 2021, 71 sivua

Tekniikan ala. Tieto- ja Viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: englanti

Verkkojulkaisulupa myönnetty: kyllä

Tiivistelmä

DevOps on osoittautunut tehokkaaksi lähestymistavaksi ohjelmistokehitykseen lähentäen ohjelmistokehittäjien ja operatiivisen puolen välisiä suhteita muokkaamalla Agilen käytäntöjä ja sitä ympäröivää työkulttuuria. DevOpsin myötä on myös kehittynyt GitOpsin kaltaisia menetelmiä uudistamaan ohjelmistokehityksen toimintamalleja entisestään. GitOps tarjoaa työkaluja ja periaatteita pilvisovellusten julkaisuun ja hallintaan. Samoin konttiteknologioiden hallinta on kehittynyt eri orkestrointijärjestelmien, kuten Kuberneteksen myötä. Sekä GitOps että Kubernetes ovat olleet läheisessä yhteydessä toisiinsa siitä lähtien, kun GitOps julkaistiin vuonna 2017. Kehittyvien teknologioiden hyödyntäminen voi tarjota yrityksille kilpailukykyä monilla eri markkinoilla mukaan lukien ohjelmistokehittämisen. GitOps on ehtinyt keräämään huomiota suuriltakin yrityksiltä, kuten Amazonilta, GitHubilta ja Microsoftilta. Suosittu tapa käyttää GitOpsia on asentaa ohjelmistoagentti varmistamaan, että järjestelmän reaaliaikainen tila vastaa aina konfiguraatiota. Pilvisovellusten konfiguraatio ja sovellusten toimitus tapahtuu ainoastaan Gitin kautta, joka vähentää palvelimille manuaalisesti tehtyjen muutoksien määrää. GitOpsin menetelmät tarjoavat auditoitavaa ja helposti hallittavaa konfiguraatiota ja voivat tarjota monia etuja yrityksille, jotka päättävät omaksua sen toimintamallit.

Ensimmäinen tavoite oli tutkia GitOpsia ympäröiviä keskeisiä käsitteitä ja sen eroa DevOpsiin. Toisena tavoitteena oli luoda suunnitelma ja toteuttaa se käyttöönottamalla Conduit -niminen verkkofoorumi GitOpsin jatkuvan julkaisun työkalun Argo CD:n ja GitLabin jatkuvan integroinnin avulla. Suunnitelmassa Conduitin frontend-, backend- ja tietokantakontit toimitettaisiin Kubernetes-klusteriin CSC cPouta -pilvialustalle. GitOps -käytäntöjä tutkittiin myös toteutuksen aikana vertailemalla ohjelmiston julkaisua Kuberneteksen manuaalisen ja Argo CD:n avulla suoritetun automatisoidun julkaisutavan välillä.
Havaittiin, että DevOps ja GitOps voivat osittain omata samanlaisia toimintamalleja pysyessään silti erillisinä käsitteinä. GitOps keskittyy suurelta osin menetelmissään jatkuvaan toimitukseen, kun toisaalta DevOps vaikuttaa myös yrityksen kulttuuriin laajemmassa mittakaavassa. Tämän lisäksi ohjelmistojen julkaiseminen GitOps-työkaluilla eroaa myös huomattavasti manuaalisesta toimitustavasta Kuberneteksella.

Avainsanat (asiasanat)

GitOps, Kubernetes, DevOps, Argo CD, Git, Docker, pilvipalvelut

Muut tiedot (salassa pidettävät liitteet)

jamk

Description

Korhonen, Matti

GitOps tool Argo CD in service management
Case: Conduit

Jyväskylä: JAMK University of Applied Sciences, September 2021, 71 pages

Engineering and technology. Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for web publication: Yes

Language of publication: English

Abstract

DevOps has proven to be an effective way to approach software development by removing borders between developers and operative people by modifying Agile practices and work culture as a whole. DevOps has also inspired methods such as GitOps to modify the workflow of modern software development even further with declarative cloud native applications. Similarly, containerization technologies have also seen an expansion to different options with orchestration systems such as Kubernetes. Both GitOps and Kubernetes have been closely associated with each other since the arrival of GitOps in 2017. The knowledge of emerging technologies can give an advantage to companies in a competitive market such as in software development, and GitOps has already been gathering attention from large companies such as Amazon, GitHub, and Microsoft. A popular way of utilizing GitOps is to have an agent ensuring that the current state of a system always matches the desired state declared in the configuration. The agent compares the application to the configuration stored in Git, which acts as the single source of truth for the systems state. Having an auditable and easily controlled configuration through Git, GitOps could offer varying benefits to companies that decide to adopt it.

The first goal was to study the key concepts surrounding GitOps and its differences with DevOps. The second was to form a plan and implement it by deploying a web forum called Conduit with the use of GitOps continuous delivery tool Argo CD and GitLab continuous integration. The plan included deploying Conduit with separate frontend, backend, and database containers into a single Kubernetes node on the CSC cPouta cloud platform. The differences between deploying to Kubernetes manually versus deploying with Argo CD were also explored by following GitOps practices during the implementation.
It was found observed that DevOps and GitOps can partially share similar principles while remaining distinct from each other. GitOps largely focuses its methods on continuous deployment when on the other hand DevOps also influences the culture of a company on a larger scale. In addition to this, deploying software with GitOps tools also differs notably from manual deployment to Kubernetes.

Keywords/tags (subjects)

GitOps, Kubernetes, DevOps, Argo CD, Git, Docker, cloud services

Miscellaneous (Confidential information)

Contents

Figures

1 Introduction

The purpose of the project was to explore and study GitOps tools and practices, and how it can be used to deploy and manage software in a controlled and automated way. Kubernetes holds a strong relation with GitOps and as such it was used with a set of preselected tools to achieve a running application built through a CI/CD pipeline. The subject was chosen because GitOps is a new and emerging concept that could prove to add value to already existing parts of software deployment by increasing reliability and automation.

DevOps is being valued highly across the IT-Industry with almost half of respondents answering that DevOps is extremely important, and a third thinking it as somewhat important to scaling software development in a survey done by Stack Overflow (Importance of DevOps to scaling software development worldwide as of 2020. 2021). On the other hand, GitOps builds on top of DevOps with tools and a focus to declarative containerization while not making too many changes to the philosophical aspect of DevOps. Compared to DevOps, GitOps is much more recent and has not had the time to fully standardize across the industry. Nonetheless, GitOps still has gathered interest and support from larger organizations such as Amazon, Github, and Microsoft because it could enable increased productivity, higher reliability, stability, and an enhanced developer experience if implemented. (GitOps working group n.d).

The substantial goal of the project was to have a working application deployed to Kubernetes with the GitOps continuous delivery tool Argo CD. The theory part of the covers the practices of DevOps and GitOps and their core technical concepts with a focus on the tools and techniques used in the implementation part of the project. The implementation and planning section on the other hand focus on planning the architecture and logical sequences between services as well as their implementation. The implementation part of the project consists of building a CI/CD pipeline and a Kubernetes cluster with MicroK8s to a cloud server with Argo CD acting as the continuous delivery tool and GitLab being the single source of truth for the application source code and Kubernetes configuration manifests. This was done to test configuring a Kubernetes cluster and running applications inside of it while using GitOps tooling and practices. An open-source web application Conduit was deployed to the cluster through the pipeline. Conduit is a clone of the online publishing site medium.com to act as an example application for demoing different frontend and backend frameworks in action. This means that there are blog posts that need to be saved into a

database, a frontend for the user to interact with, and a backend to handle the information moving around on the website. Choosing Conduit provided a way to get familiar with both the features of Kubernetes and Argo CD and the environments through the installation process. The infrastructure itself was built on the CSC-owned cPouta cloud platform that is based on the open-source cloud platform OpenStack.

2 Research methods

Research-based development was chosen as the research method because building a Kubernetes cluster would help to test the practices of GitOps and understand the deployment methods it provides for declarative application configuration by having a tangible environment where to explore the different tools. Three research questions were formed:

- How does GitOps differ from DevOps?
- How does deploying software with Argo CD differ from deploying to Kubernetes manually?
- How to deploy Conduit to a Kubernetes cluster using GitOps practices?

New technologies and practices can lead to improvements overall. GitOps is an emerging toolset built on DevOps practices but the actual benefits and differences of GitOps can be hard to understand initially. The first research question was planned to be answered by researching the basic concepts of both techniques. The second research question was planned to be answered along with the latter question by researching readily available literary sources but also through a demonstrative configuration of a system built on Kubernetes. The modernity of GitOps also means that related studies are somewhat hard to come by. The research questions would partly be answered through implementing a system while applying GitOps practices and observing the process. The third question was formed to test the actual workflow of GitOps while working with Kubernetes and see what it requires to achieve a working implementation of Conduit.

To narrow the topic the subject can be divided into two parts. Learning about the technologies and concepts related to software deployment and Kubernetes, while the latter would consist of understanding the tools and practices of GitOps. There was a focus on the actual workflow of GitOps by building the environment while using preselected tools of Argo CD, Kubernetes, Conduit, and GitLab CI. Understanding the principles and tools would already cover a large amount of information to explore so comparing tools between each other was decided to be left at a minimum

especially because the subjects covered were mostly unfamiliar at the start of the project. As a methodology, GitOps mostly provides tooling unlike DevOps, which also provides a much more defined set of principles and culture guidelines. Knowing these limitations, decisions were done to try and keep the topic easily understandable and narrow the topic into the tools chosen. The sources were chosen to provide a general understanding of the topics while not diving too deep into the different choices between tooling and practices because of the variety available. The sources mostly consist of online tool documentation and articles because of the recent nature of GitOps and the lack of standardization. While choosing reference material, the newest releases were prioritized while trying to get relevant sources closest to the original if any were available.

3 DevOps

3.1 History of DevOps

GitOps differs from DevOps in many ways but internalizing the values behind DevOps can bring insight to the decisions leading to the development of GitOps. DevOps is described as a combination of philosophies, practices, and tools to increase the pace of delivering applications and services by Amazon (What is DevOps? - amazon web services (AWS.). In a differing view by Freeman & Forsgren (2019) there is no official definition of DevOps, but instead suggest that it is just better described as a philosophy that prioritizes people over processes and tooling and as a culture that tries to take everyone involved into account. A standard for DevOps was released by the Institute of Electrical and Electronics Engineers (or IEEE) in 2021, where DevOps is described as a set of principles and practices for enabling better communication and collaboration with stakeholders to specify, develop, and operate software and products and improvements throughout the life cycle (IEEE standard for DevOps: Building reliable and secure systems including application build, package, and deployment. 2021).

DevOps was built upon the principles of Agile software development and was first brought to the public in 2008 in the form of an event called DevOpsDays in Belgium. It was arranged by Andrew Clay Shafer and Patrick Debois because they wanted to create an even more agile approach for software development and to solve the conflicts that arose between developers and operations specialists that still happened even within Agile principles (ibid.).

Since DevOps affects multiple areas of software development it can lead to improvements on a larger scale. According to Virmani (2015, 82) DevOps gives organizations the ability to save time, adapt to continuous feedback, balance cost and quality efficiently, have more predictability with their product releases, and increase the general efficiency of the organization.

The tools needed for the DevOps lifecycle for software development are known as the DevOps toolchain. The tools can be different based on the needs of the organization. It is often represented as an infinite figure of eight (See Figure 1.) where each part represents a set of actions or tools used during the DevOps software lifecycle. The wheel is usually depicted with the left side governing development activities such as continuous integration and delivery while the right side depicts the operational activities such as monitoring and incident management. (Krohn, R. n.d).

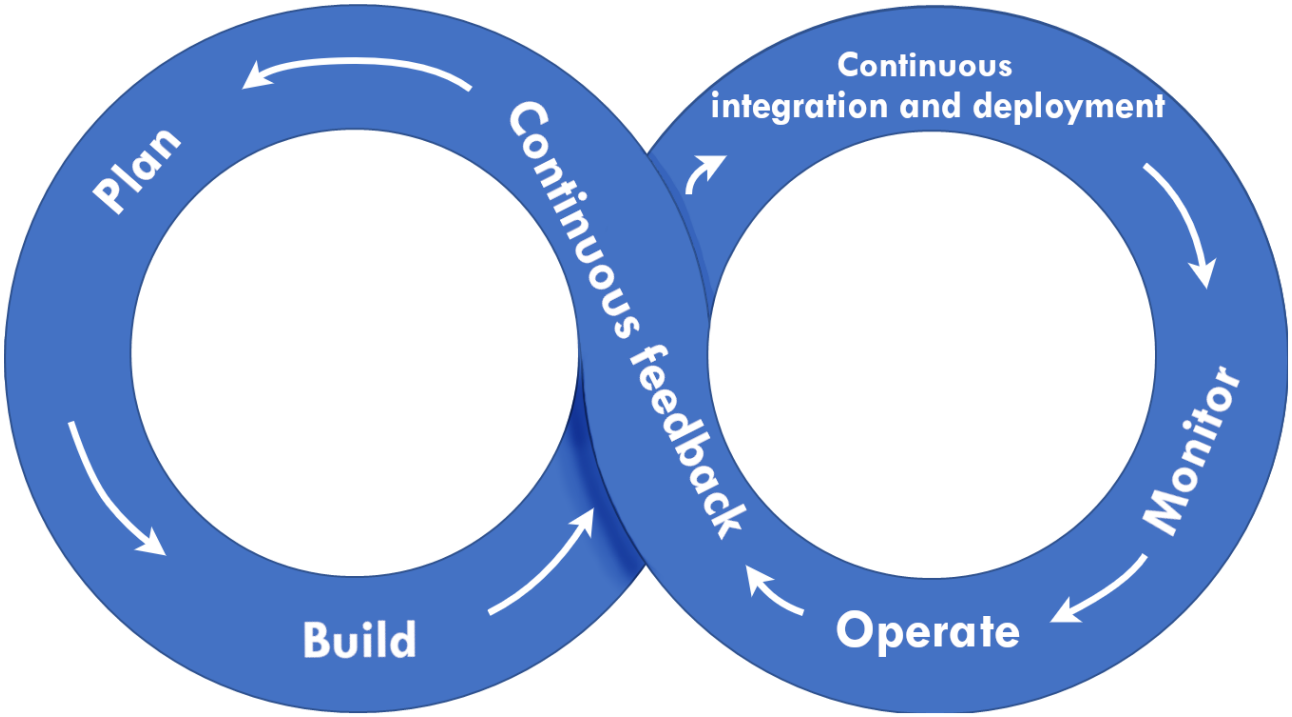

Figure 1. Devops infinity wheel (Adapted from Krohn, R. n.d)

3.2 Continuous Integration

Continuous integration (or CI) governs the part of development where developers merge their code into a central repository in a version control system after which the code is automatically built and tested with the end goal of reaching working code and being ready for production as fast as possible (What is Continuous Integration? – amazon web services. n.d).

CI requires a VCS (Version control system) to host the codebase. Bitbucket, Github, and GitLab are some of the most popular options. Some platforms host their own CI, such as GitLab (Rehkopf, M. n.d). Continuous integration encourages developers to integrate their code faster instead of developing a local version in their workspaces because the automated testing and building helps to validate the code continuously and merge even smaller changes into the code. (Virmani, M 2015, 79).

3.3 Continuous Delivery and Deployment

Continuous deployment is a follow-up to continuous integration where the previously built code is deployed to environments automatically. Continuous deployment does not require separate approval for the deployment to happen. On the other hand, Continuous delivery provides a way to automate the delivery of an application but requires additional approval before the actual deployment. An example of this would be a situation where the code is deployed to a test or staging environment automatically with continuous deployment, while continuous delivery would be reserved for deploying production ready code to a production environment by an authorized person giving approval for the deployment. Both concepts can be abbreviated with CD (What is Continuous Delivery? – amazon web services n.d). Combining continuous integration and continuous delivery is known as the CI/CD pipeline. Having continuous delivery lessens the complexity of deploying software and makes making smaller changes easier and as such encourages faster iterations (Pittet, S. n.d a).

3.4 Testing

Testing is a broad subject and can be done throughout the life cycle of the software, which is also better done earlier than later according to the left-shift principle of DevOps (IEEE standard for DevOps… 2021). Testing can be automated to save the developers from repeating tests by hand although manual testing can still be useful even if parts of it have been automated. The lower level of testing in terms of scale is called unit testing, which is usually done to individual functions of an application to test that the logic of the methods and functions match their original purpose. Some testing methods require that parts of the application are deployed to an environment. Integration tests could be done to test the communication between the parts of an application to confirm that they work together beyond being isolated modules. For example, an integration test could be used to test that a separate frontend, backend, and a database can communicate with each other in an

expected way. Additionally, it is possible to use functional testing for confirming that a product meets the functional requirements of the application. Functional tests try to confirm that the output of an application matches the expected result laid for it. Functional testing differs from integration by having a focus on the actual output of an action while integration tests might only focus on a database is communicating with another part of the application. It can also be useful to do rudimental checks to an application before using more extensive and resource demanding testing. This is called smoke testing. When the application is nearing the expected behaviour of a full application, end-to-end testing, performance testing, and acceptance testing become more relevant. End-to-end tests are done to simulate user behaviour in an application. It can be used to test that the application works as intended when performing actions that a user might do in the application. Additionally, performance testing simulates situations where an application is subjected to heavy traffic or load. This is done to check the reliability of an application. Acceptance testing comes into play when it is time to check that all the functionality of an application meets the requirements for it to be considered ready for release (Pittet, S. n.d b).

3.5 DevSecOps

Another recent development in the DevOps scene is DevSecOps. DevSecOps is a branch of DevOps that tries to ensure security throughout the software development cycle by adding the responsibility for security to a larger scope in the same way as DevOps lessens the gap between developers and operative people. DevOps philosophy and best practices still apply but DevSecOps applies them with security measures in addition. Best practices in DevSecOps also include Shifting left, where security is made a part of development from the start to the end which allows for security risks to be identified earlier with potentially lesser consequences. Security education should also cover the whole organization so everyone can be kept up to the same standards of security. DevSecOps also shares communication as a best practice with DevOps because it allows changes to happen dynamically and have teams being able to manage systems securely after understanding the requirements. To securely work through the lifecycle of a product, DevSecOps also applies traceability, auditability, and visibility to the workflow. Traceability is achieved by leaving marks of changes via configuration to allow tracking of the state of the product. Auditability is to have security-relevant details to be well-documented and auditable without lacking information. Visibility is to monitor the security and provide information via alerts and notifications to ensure awareness of risks and accountability for the lifecycle of a project (IBM Cloud Education, 2020).

4 Kubernetes

Kubernetes was originally built at Google and then open-sourced in 2014. The initial need for Kubernetes came from wanting to bring a cloud-native application experience to a broader audience but also noticing that the recently released Docker was not going to be the complete solution because of its focus on a single machine and lesser scalability. An orchestrator would be needed to manage a large number of containers. (Burns 2018).

4.1 Containerization

Since Argo CD is made to work closely with Kubernetes, it is important to understand the core concepts of Kubernetes well before usage. Kubernetes is an open-source platform for orchestrating containerized workloads and services. Kubernetes deployments use computational resources effectively by taking advantage of the benefits of containerization because the containers are decoupled from the underlying resources unlike with virtual machines. (What is Kubernetes? n.d.).

Containerization is not limited to just Kubernetes. Kubernetes just uses containers as its method of delivering applications. Containers are a way to package applications and abstract them from the environment they are run in, unlike virtual machines that require a guest operating system to run with the applications. Virtualizing hardware is not needed and the saved memory from not running an OS per instance makes containers into a lightweight solution to running applications (Containers at google n.d; What is a Container? – Docker n.d). Docker is a popular containerization technology that surfaced around 2013 with the open-source release of Docker Engine. The clear benefits of containerization come from the reduced size required for running an application. For example, in Docker, the images can be measured in megabytes, when a virtual machine can be several gigabytes because of the full operating system running inside of them. The size difference allows for more containers being run per server compared to virtual machines. This does not mean that virtualization is not needed anymore. Containerization still relies on the containers sharing an operating system kernel and virtual machines can be used to create different types of operating system environments to run containers in (What is a Container? – Docker n.d).

4.2 Kubernetes features

Kubernetes makes use of small computational units called pods to host either singular or multiple containerized applications inside of them (Pods - Kubernetes n.d). Kubernetes as an orchestrator includes many features such as service discovery that allows the use services to point network traffic to relevant pods based on their labels even if they are replaced by another pod. This bypasses the problem of IP addresses being tied to the lifecycle of a pod if the pod gets terminated. Kubernetes is also able to automatically load balance between the pods based on services and the use of labels. Automated rollouts and rollbacks can be used to change the deployed containers to the desired state at a controlled rate. For example, by removing existing containers and adopting their resources to form new ones. During rollbacks, related services are only advertising the containers available at the time which helps with keeping downtime during maintenance lower (Service – Kubernetes n.d; What is Kubernetes? n.d; Performing a rolling update. n.d). Kubernetes is also capable is automatic bin packing by allocating CPU and RAM resources to each container and automatically distributing the containers between different nodes to save resources (What is Kubernetes? n.d).

A node can be either a virtual or a physical machine that is responsible for running the workload of a cluster (Kubernetes - Nodes n.d). Pods are also self-healing in Kubernetes since Kubernetes can automatically restart, replace, or kill containers that do not pass user-defined health checks. The rolling updates try to ensure that only the working containers are advertised during changes. Passwords and other sensitive information can be stored as secrets inside the cluster without exposing them in plaintext configuration. Pods can use these secrets to carry out tasks that need sensitive information to be passed without the information itself being written into the pod configuration. In addition to these features, Kubernetes is capable of automated storage mounting by having dynamic volume provisioning. It removes the need for manually creating volumes and allows automatic creation of storage when it is required. This can be done to local storages and public cloud providers among other options (What is Kubernetes? n.d.; Dynamic Volume Provisioning n.d).

4.3 The control plane components

The control plane in Kubernetes is a container orchestration layer that can expose the interfaces and the API to manage containers and their lifecycle (See Figure 2). It manages the clusters, nodes,

and pods. It spans multiple computers for high availability and fault tolerance. The components include the kube-apiserver, etcd, kube-scheduler, and finally the cloud-controller-manager. Kube-apiserver can be thought of as the front end of the Kubernetes control plane because it exposes the Kubernetes API (Kubernetes components n.d).

Etcd acts as the data store for Kubernetes. Applications read and write data from and to etcd, which distributes the configuration between nodes for incident tolerance (What is etcd? n.d). The kube-scheduler's role is to decide on what nodes to run pods on based on multiple different factors such as resource requirements, and constraints from hardware, software, or policies. The kube-controller-manager runs controller processes. A controller is a control loop that works towards getting the current state to the desired state of a cluster via the apiserver. There can be different types of controllers that take separate responsibilities like a node, job, endpoint, and service account and token controllers (Kubernetes components n.d). Every node runs Node components that are composed of the kubelet, kube-proxy, and the Container runtime. Kubelet is an agent that ensures that containers are running in pods. Kube-proxy on the other hand is responsible for network rules of connections to pods either from the inside or outside of the cluster. Addons are also available in Kubernetes. These include a DNS addon that can track DNS records for Kubernetes services, a dashboard for managing and troubleshooting the cluster, Container Resource Monitoring for monitoring container metrics through a UI, and cluster-level logging that saves a centralized log from the containers (ibid n.d).

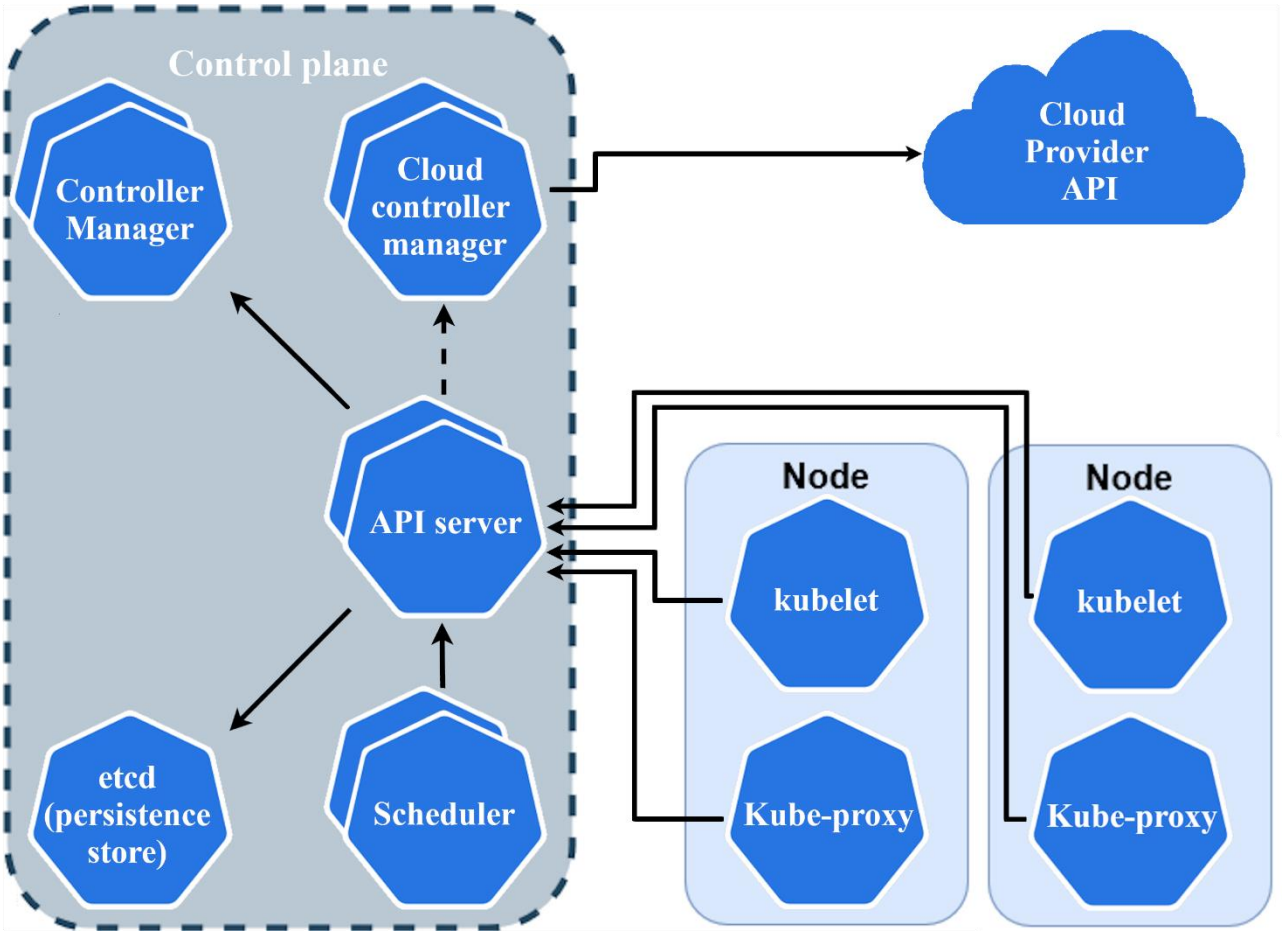

Figure 2. Kubernetes control plane (Adapted from Kubernetes components n.d)

4.4 Kubernetes workload resources

Kubernetes workloads are essentially containerized applications that are run inside sets of pods. Pods are the smallest units of application computing inside kubernetes. The pods have a specification that tells the pod how to run containers inside of them. Pods are designed to be ephemeral and are rarely created by hand, instead commonly created from deployments or jobs that specify what kind of pods to create. Pods also share networking and storage resources with each other (Pods - Kubernetes n.d). Deployments are used to declare the desired state of either a pod or a ReplicaSet. Deployment configurations are saved in YAML files and can be updated to change the definitions of pods or for example, scale up the number of pods to run through ReplicaSets. They can be rolled back to previous versions if the latest release ends up being unstable (Deployments – Kubernetes n.d). The function of a ReplicaSet is to keep an identical set of pods up and running. Since deployments can be used to manage ReplicaSets, managing ReplicaSets directly is not often needed (ReplicaSet – Kubernetes n.d).

Another type of specification for pods is a Job. For example, a Job can be used to create a set of pods to work in parallel until a goal is met or just a singular pod to run until a command has reached its goal. After the Jobs are completed, the pods are not automatically deleted by default for logging and diagnostic purposes. Jobs differ from deployments in that they are meant to achieve a goal after which new pods are not created, but in the case of deployments, the pods are usually meant for longer-lasting applications (Jobs – Kubernetes n.d).

4.5 Services and networking

In Kubernetes, each pod has an IP address but because pods can be replaced or destroyed, especially when using deployments to control them, accessing pods by making static references to IP addresses would be problematic. A service in Kubernetes is a configurable abstraction that can be used to define network policies for pods by using pod labels as means to organize them up to logical groups which also bypasses the previously mentioned problem with static IP definitions. These selectors are given to pods while defining deployments and then given again when defining the service to logically connect the objects (Service – Kubernetes n.d).
Ingress is a type of API object in Kubernetes that can manage external access to the cluster by having a routing rule to a service. HTTPS and HTTP routes to services are the specialties of Ingress, while the service types NodePort and LoadBalancer can be used for other protocols. Ingress also requires an Ingress controller to control the resource itself. Ingress has load balancing options present and can route traffic based on hostnames if the services are running on the same IP address (Ingress – Kubernetes n.d).

4.6 Storage

There are ephemeral Volumes for data storage as well as persistent volumes in Kubernetes. Ephemeral storage options share their lifecycle with a pod while Persistent Volumes allow the volumes to have independent lifecycles. Kubernetes also provides different types of volumes based on the cloud platform used such as AWS, Azure, OpenStack, and Google. A *hostPath* volume provides a simple way to mount local files and directories from a nodes filesystem but suffers from multiple flaws such as having the possibility to expose privileged credentials that can be used to create attacks that could affect the cluster (Volumes – Kubernetes n.d). PersistentVolumes allow

the data to be kept after a pod is terminated. To manage a persistent volume, two types of recourses need to be configured. First of these is called PersistentVolume or *PV* for short. It is the literal share of storage that is provisioned, and it is designed survive regardless of any related pods and their lifecycle. It can be provisioned by hand or dynamically with a Storage Class and a provisioner. Secondly, a PersistentVolume needs a PersistentVolumeClaim, which is often shortened to just PVC. It is used for requesting the storage for usage and specifying the read and write access modes as well as the amount of storage. PVs can be static or dynamic. Static volumes are simply created by hand, while dynamic PVs can be created automatically based on PVCs even if a PV declaration does not exist. This still requires a StorageClass object to be configured by an administrator and enabling the *DefaultStorageClass* admission controller before dynamic provisioning is usable (Persistent Volumes – Kubernetes n.d).

4.7 Configuration resources

Sometimes small amounts of data need to be used by the pods to access other services with data such as credentials or environmental variables. Secrets and ConfigMaps are solutions to this kind of demand. Sensitive data such as credentials and keys can be stored in Secrets. Secrets are not encrypted by default but instead are stored in base64 encoded strings that can be accessed with anyone with API access rights. It could still be considered safer than hardcoding plain text credentials to image or pod definitions and they also aim to help configuration since giving a pod access to a secret is easier in the long run than change a password by hand in multiple locations. The data inside a Secret can either be used with the Opaque type to handle user-defined data or by using the built-in variations for popular use cases such as Docker configs, SSH keys, or TLS certificate and key pairs (Secrets – Kubernetes n.d).

ConfigMaps are similar to Secrets in that they store small amounts of data inside them, but the data is meant to be non-confidential. A ConfigMap has two fields to store data in. The *data* and *binaryData* fields. Both are meant for key-value pairs, but *data* contains byte sequences of UTF-8, while the *binaryData* has binary data stored as base64-encoded strings. ConfigMaps are commonly used for environment variables, command-line arguments, and configuration files that pods can use to interact with application components (ConfigMaps – Kubernetes n.d).

4.8 MicroK8s

MicroK8s is a version of Kubernetes developed by Canonical who are also responsible for publishing Ubuntu. MicroK8s seeks to provide high availability and self-healing clusters by allowing it to be configured to work with multiple nodes that automatically promote each other if one node is lost. MicroK8s also aims to be easily usable and eliminate everyday administration of Kubernetes clusters by having a quick install procedure and providing automatic security updates while also being lightweight enough to even be installed on a Raspberry PI (MicroK8s - zero-ops Kubernetes for developers, edge and IoT | MicroK8s n.d).

5 GitOps

5.1 What is GitOps?

GitOps was pioneered by Weaveworks co-founder Alexis Richardson in 2017 originally to manage Kubernetes clusters and handle application delivery while using Git as the source of truth for the state of an application and declarative infrastructure. In practise, GitOps fundamentals are not limited to just Kubernetes and can be utilized with other cloud native technologies as well (What you need to know - Guide to GitOps. n.d). GitLab defines GitOps as a combination of IaC, merge requests to git, and the addition of continuous integration and continuous delivery where the merge requests act as the triggering change mechanism to infrastructure updates. (What is GitOps? n.d). Infrastructure as Code shares similarities with GitOps. IaC tools can be used to provision server infrastructure with declarative configuration and has a freedom of choice with the version control system. GitOps on the other hand can be used to add declarative configuration also to containerized applications on these servers and specifies that the configuration is held in Git. They can be used hand in hand because the technologies complement each other and allow managing the whole infrastructure stack from servers to applications (What you need to know - Guide to GitOps. n.d).

As a methodology, GitOps is recent and is not defined the same way across the industry, even though the original idea was coined by Weaveworks. GitOps builds upon DevOps best practices but applies the principles to declarative container orchestration. Configuring infrastructure has seen chances from IaC but applying the same declarative methods to containerized software is a more recent development.

The benefits of GitOps vary from being time saving and cost effective by having deployments that use automated tests or configuration that is codified, which reduces repeated manual tasks and provides accountability. Systems can also be less risky and less error-prone because the declarative configuration can be reviewed before being put into production. Easy rollbacks through git also provide a way to audit and track past changes and finally since all the changes go through the same merge request and approval processes in Git, the senior engineers get a more centralized way to review and contribute to the project that saves them time to focus on other tasks (Use-case: GitOps n.d).

Push-based deployment

Deployment strategies can be divided into two methods: Push-based and Pull-based deployments. In Push-based deployments, the strategy is executed by CI/CD tools with a pipeline starting from a trigger in the application repository (See Figure 3). Images are built and pushed to an image registry and the environment repository is updated with new deployment tags. In turn, the update to the environment repository triggers the deployment pipeline which deploys the changes to the environment. This also means that the deployment pipeline is only triggered from the environment repository changes and as such can't automatically monitor differences between the current state in the environment and the desired state. External monitoring needs to be configured for this to be ensured (Beetz, F., Kammer, A., Harrer, S. n.d).

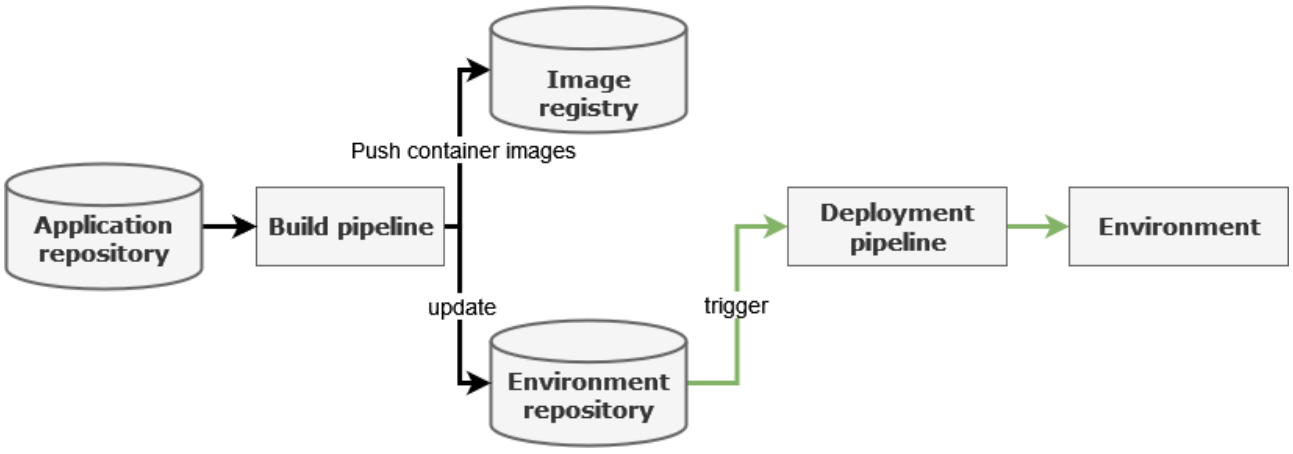

Figure 3. Push-based deployment (Adapted from Beetz, Kammer, & Harrer n.d)

Pull-based deployment

The other strategy is Pull-based deployment, which differs from Push-based deployment by having an operator working from inside the environment taking the role of the deployment pipeline while observing the environment repository and optionally the image registry (See Figure 4). The addition of an operator means that most pull-based deployment strategies are used with Kubernetes clusters although in theory, the requirements for GitOps can be filled without using a Kubernetes cluster. The pull-based GitOps has a key difference with the push-based approach in that the operator also compares the desired state in the repositories to the deployed current state for differences. This means that changes made to the cluster that are not written in the environment repository are automatically reverted. Since the operator is working from inside the cluster there is no need to give deployment credentials to CI/CD pipelines or external services, which makes this approach more secure. Operators usually support notification messages if environment states do not match but it should be noted that operators are wholly responsible for the deployment so monitoring the operator itself is a good idea in case of errors. Having multiple environments can be needed in a project for example for differentiated production and staging environments. This can be reflected by using branches to represent different environments in the environment repository and having the operator track these branches. (Beetz, F., Kammer, A., Harrer, S. n.d).

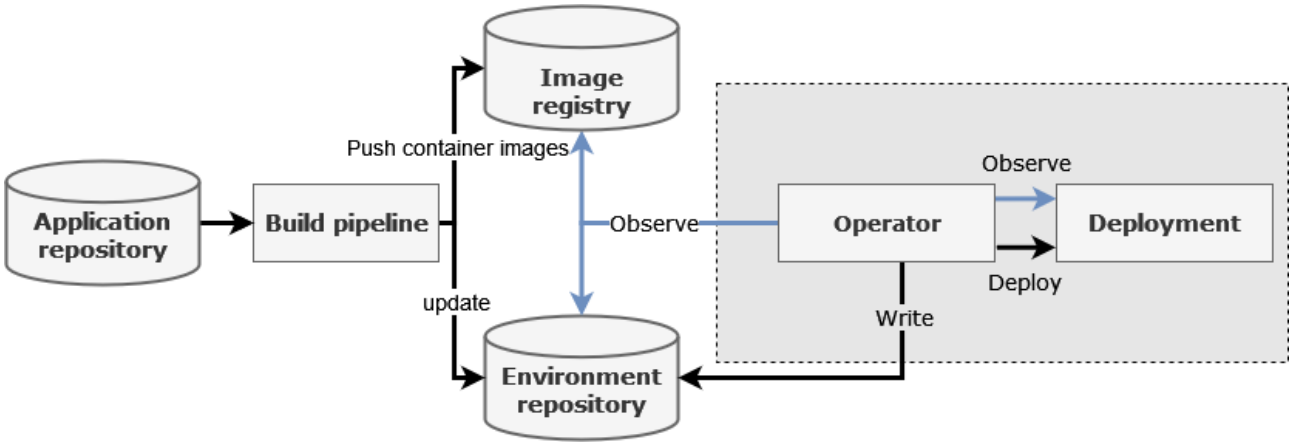

Figure 4. Pull-based deployment (Adapted from Beetz, F., Kammer, A., Harrer, S. n.d).

5.2 Four principles of GitOps

Weaveworks provides four principles that must be established while managing software with GitOps (What you need to know - Guide to GitOps. n.d).

- The entire system described declaratively
- The canonical desired system state versioned in Git
- Approved changes that can be automatically applied to the system
- Software agents to ensure correctness and alert on divergence

The entire system described declaratively

Configuration can be imperative or declarative. Imperative configuration means that a system is defined by telling the exact steps to an end goal. This is usually done through commands in a command line or by changes through a UI. Declarative configuration differs from imperative configuration by telling the system the desired end goal instead of a step-by-step list. An example of this could be a situation where Kubernetes is told to raise the number of the frontend pod replicas to two through a change in a configuration file. Kubernetes does so and tries to maintain that state without the administrator having to worry about the steps that lead to that state (Kurek, T. 2019)

The canonical desired system state must be versioned in Git.

System configuration needs to be declaratively stored in a version control system acting as the source of truth for configuration so that the changes are driven from the system and not done by hand. Since the configuration in Git is also the state of the application, rollbacks can be done by the command *Git revert* to go to previous application states. Because all the changes go through git, SSH keys can be used to sign commits and guarantee the authorship of changes (What you need to know - Guide to GitOps. n.d). If configuration is stored as code in a Git repository, it means that it inherits the benefits of Git itself. This means that the code is version controlled in a way that changes to the configuration can be audited and reviewed through the commit history to get a good view of the state of the application.

Approved changes that can be automatically applied to the system

The third principle is to allow the previously declared state in Git to be automatically applied to a system. Because the configuration is automatically applied, there is no need to have direct access to the cluster and because of that, no credentials to the cluster are needed to make a change into the current state. Having automated deployment removes human errors and by having a good deployment approval process can help eliminate them further (What you need to know - Guide to GitOps. n.d).

The software agents need to ensure correctness and alert on divergence

The fourth principle is where a software agent is monitoring and self-healing the system for differences between the desired and current state of a system. This is also done in case of human errors in the system and not to be confused with the separate form of Kubernetes self-healing capabilities that focus on the technical errors of the cluster to keep the containers running since they would be present even without GitOps. Having a software agent monitoring that the current state is matching the desired state also acts as a feedback and control loop for operations (What you need to know - Guide to GitOps. n.d).

5.3 Argo CD

Argo CD is a part of the Argo Project set of tools for Kubernetes developed by Applatix, which was acquired by Intuit in 2018 (Tessel 2018). It is an open-source declarative, GitOps continuous delivery tool for Kubernetes. The tool automatically deploys applications from declarative configuration files that are stored in Git to Kubernetes. This makes the configuration version-controlled and auditable through Git. Argo CD acts as a Kubernetes controller monitoring the state of both the source from Git and the deployed application. It looks for differences between the two and provides the means to update the deployed application manually or automatically if the Git version has been updated. It also provides a visualized report of the current state of differences via the web user interface. (Argo CD – Declarative GitOps CD for Kubernetes n.d.)

Argo CD can be divided into three different architectural components that run inside a Kubernetes cluster: API server, repository server, and the application controller. The gRPC/REST-based API server serves multiple purposes including the application management and reports through Web

UI or CLI, managing credentials for the repositories and cluster, and acts as the listener and forwarder for Git webhook events. The repository server keeps a local cache of the application manifests from the git repository and returns or generates application manifests if needed. The application controller is responsible for monitoring the state of the application and compares it to a Git repository looking for OutOfSync state and carries out corrective actions if configured so by the user (Architectural Overview n.d).

Argo CD is also capable of dividing access to configuring applications through its projects that can have access defined based on teams of developers with role-based access control. It also supports the use of Kustomize with the Kubernetes configuration as well as Helm charts, Ksonnet, Jsonnet, raw YAML/JSON, and other possible tools if they are added as plugins. It is also feature-rich with synchronization options such as automatic pruning of old resources and offering manual control to the synchronization with most of the controls being accessible through the UI. It still does not have continuous integration features available for building docker images, which still leaves the need for an external CI (Overview – Argo CD n.d.).

There are multiple alternatives to Argo CD that all differ in the way they handle the deployment workflow. One of these examples would be Flux. Flux is a simple GitOps agent that handles a single Git repository for the configuration. Multiple instances of Flux need to be installed if there is more than one environment. It does have a feature that scans an image registry for metadata changes and deploys new images that Argo CD currently lacks, although a separate operator Argo CD Image Updater is being developed. Currently, one way of updating images with Argo CD is to update image tags to the manifests in Git that the sync policy then deploys. Other examples could be Jenkins X, which also includes full CI capabilities but does not support Kustomize, or having multiple projects being separated to support multi-tenancy through one install of Jenkins X (Portela et al, 2020).

5.4 Conduit as an example of a service

Conduit is an open-source web application that can be used as a learning tool and to demonstrate full-stack applications. It was built to able developers to try different combinations of frontend and backend frameworks that can be combined with either private databases or the public Conduit database to allow focusing on the frontend and backend. The web application itself is a blog site

made in the style of medium.com, where users can have personal accounts and post blog posts and comment on them in real-time (See Figure 5). Conduit has over 100 implementations that offer varying frameworks and programming languages and provides multiple tutorials for the most popular ones such as Node, Rails, and Django for backends. Popular frontends that have tutorials include React / Redux, Angular 2+, and AngularJS 1.x (Simons n.d).

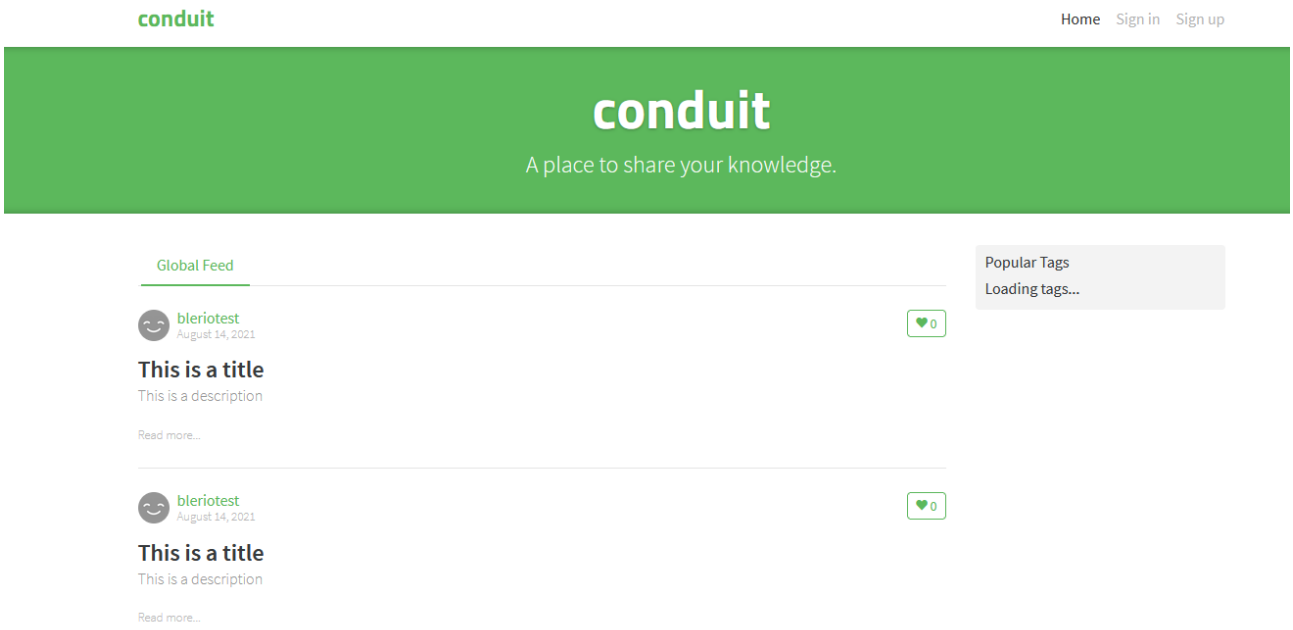

Figure 5. Conduit landing page

6 Planning and implementation

6.1 Planning

The starting plan for the implementation was to have a CI/CD pipeline deploying Conduit to a single node Kubernetes cluster with GitLab CI handling the continuous integration and Argo CD doing continuous delivery. The example program Conduit would be divided into a frontend, backend, and a database. The most popular frameworks were chosen for the project to be sure that they would not have too many bugs from lack of development. The frontend of Conduit would be based on React + Redux, the backend on Node / Express, and the database would be made using a MongoDB database. These parts of Conduit would be run inside MicroK8s that would act as the version of Kubernetes used and would be installed on Ubuntu 20.04 LTS in the cPouta cloud platform (See Figure 6).

The code for the backend and frontend would be held in separate repositories for clarity and manageability with the Kubernetes configuration files being held in a third repository in YAML format to separate configuration from source code according to Argo CD best practices. The application would be deployed to development and production environments separated with namespaces in Kubernetes. On the configuration side they would be divided by creating a separate branch for development and the master branch for production. It is technically possible to hold both inside one branch but separating them would provide clarity to the configuration without being too hard to implement.

The user would connect to the application over the internet with an internet browser. Because DNS was not planned to be configured separately to save time and keep the focus on the deployment itself it was decided that a combination of binding the IP address of the server in cPouta to hostnames for the dev and prod environments through the use of a local *hosts* file and an ingress configuration that would route the traffic to the right frontend and backend in their respective namespaces. The frontend and backend pods would have the possibility of using multiple scaling replicas that Kubernetes would load balance between them automatically.

The Kubernetes cluster required a machine and a cloud platform to run it on. The not-for-profit state-owned company CSC provides multiple different cloud services for higher education students and researchers and varying institutes (Customers – CSC. n.d). One of these services, the CSC cPouta was chosen to be used as the platform to host the Linux server in. cPouta is Infrastructure as a Service and is based on the OpenStack software that can be used to host private cloud infrastructure. The cPouta servers are hosted in Finland and they provide the basic resources to host and manage cloud services with features such as internet-accessible virtual machines, networking, floating IP addresses, storage, and changeable computing resources per instance. Managing the resources can be done through a web interface or a set of APIs (cPouta n.d).

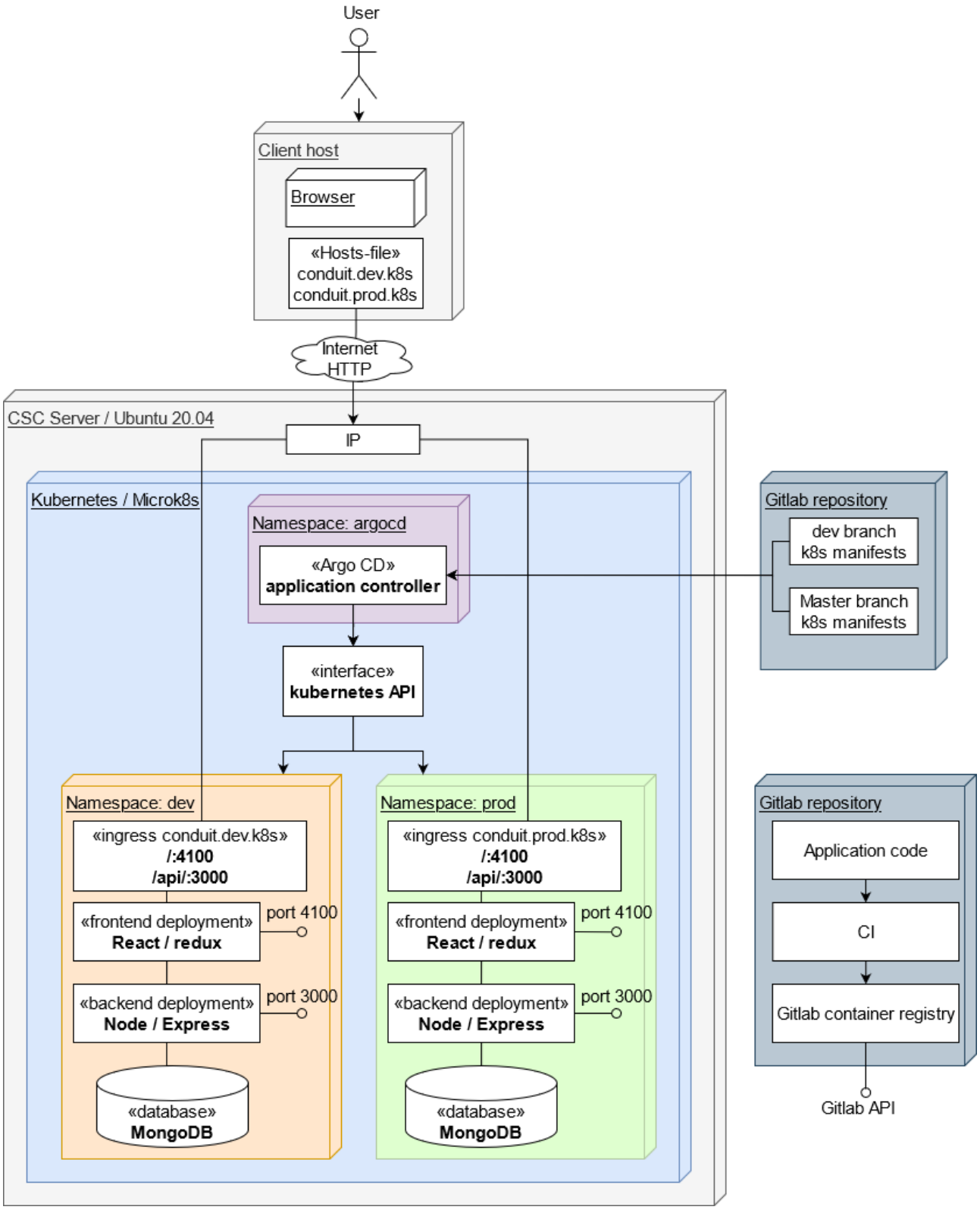

Figure 6. Deployment diagram

6.2 Creating and configuring the node

The implementation was started by installing an Ubuntu server and establishing an SSH connection to it. The cPouta web interface gives an option to create SSH key pairs from the *Compute* menu. Keys were generated from the site and saved on the local Windows 10 computer to be used with PuTTY to connect to Conduit-Node. The private key was saved into *C:/users/User/.ssh/cdc-keys.pem.* For the key to be used with PuTTY, the private key also needed to be opened in the PuTTY key generator, while adding a passphrase, and saved with *Save Private Key* to converted it to a PPK file format. The private key was saved to the previously used folder and was ready to be used. The path to the private key was given in PuTTY in the Connection - SSH – Auth menu.
From the cPouta web interface an instance was created by clicking *Launch Instance* from the menu path *Compute – Instances.* The name "Conduit-Node" was given to the instance and a flavor of *standard.medium,* which determines the amount of computing power and recourses the instance will have. Ubuntu version 20.04 was chosen as the operating system before moving to the next tab (See Figure 7). In the next tab named *Access & Security*, the previously made SSH key pair is chosen from the drop-down menu and the default security policy is enabled with the checkbox. Creating the keys before launching an instance seemed to be the way to proceed because other ways of associating a key pair to a machine after it was created were not found unless there was already some way to get access to that machine. The rest of the settings were left on their default settings and the instance was launched.

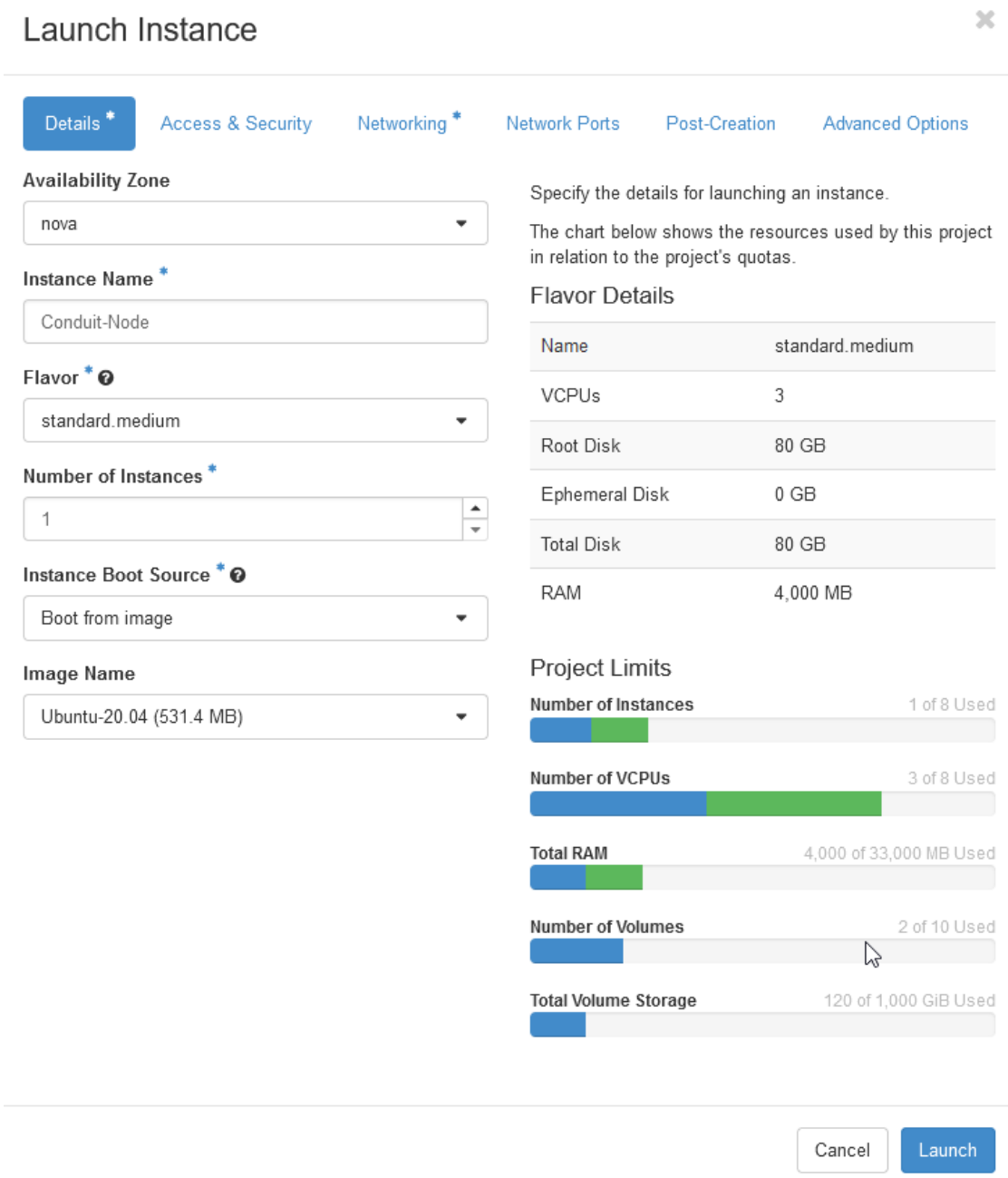

Figure 7. cPouta - Launch instance window

Connecting to the virtual machine

From the Actions menu in the Instances view, a floating IP address could be associated with the machine for connectivity by choosing *Associate floating IP address* and adding an address from the plus sign and finishing by clicking *Associate.* Next, the default security group rules were changed to

allow traffic from the local Windows 10 machine. This was done from *Network – Security Groups – Manage Rules – Add Rule* by choosing SSH from the dropdown menu and inputting the public IP address of the local machine that could be checked from a service like whatismyip.com. A subnet mask of /32 which only includes one IP address. It should be noted that if the IP address is dynamic, the IP address might change, and the rule must be updated with the new address or by allowing a larger pool of addresses. Finally, the floating IP address of the virtual machine was given in the Session menu in PuTTY, making an SSH connection to the virtual machine possible.

Setting up the virtual machine and MicroK8s

CSC images use different pre-set usernames for the default users. Ubuntu uses a user called "ubuntu". Connecting to the machine with PuTTy asked for a username and the passphrase previously provided while converting the SSH key to PuTTy's format. Once the SSH connection to the virtual machine was established, MicroK8s was set up according to the official documentation. First, MicroK8s was installed with the command *sudo snap install microk8s --classic --channel=1.21*. Ubuntu user was added to the MicroK8s group to allow easier management with admin privileges with the command *sudo usermod -a -G microk8s $USER*. In addition, rights to the *~/.kube* directory were added for visibility with *sudo chown -f -R $USER ~/.kube.*
To prepare for using the Argo CD web UI, another user without admin rights was created and Ubuntu desktop was installed so that the web UI would be viewable through the remote console viewer in CSC if needed. The user "*testerino*" was created with *sudo adduser testerino.* A password was assigned with *sudo passwd testerino* and a home directory to the user with *mkhomedir_helper testerino*. Ubuntu desktop was installed with *sudo apt-get install ubuntu-desktop.* These were done because viewing the Argo CD web UI over the internet would have either been more unsafe or required considerable amounts of work that was not essential to the goal of the project.

Installing Argo CD

Argo CD was installed through Kubernetes kubectl commands and providing a manifest for the program according to the official documentation. Argo CD would be run inside MicroK8s in its own namespace called *argocd*. First, the namespace needed to be created with *microk8s.kubectl create namespace argocd.* The *microk8s.* prefix for kubectl is only relevant when using MicroK8s because

the developers of MicroK8s wanted to avoid clashing with other installations of Kubernetes on the host machines (Working with kubectl n.d). Argo CD manifest was applied to the namespace with *microk8s.kubectl apply -n argocd -f https://raw.githubusercontent.com/argoproj/argo-cd/stable/manifests/install.yaml*.

Configuring Argo CD was mostly done in a declarative manner to comply with the GitOps best practices although some parts still had to be done imperatively such as adding the Git repository. Argo CD best practices even encouraged to leave room for imperativeness in the configuration (Best Practices – Argo CD n.d). An admin account for Argo CD called "admin" was generated during the installation with a secret named "argocd-initial-admin-secret" containing the password for the account. To change the password, the secret could be retrieved with *kubectl -n argocd get secret argocd-initial-admin-secret -o jsonpath="{.data.password}" | base64 -d.* This outputted the password stored in the secret in plaintext to the terminal by decoding it from base64. To login to the argocd-server, the command *microk8s.kubectl get pods -n argocd* was run to get the full name of the server, which includes a randomly generated end part associated with the pod during creation. In addition to this, the server pod information was displayed to get the IP address for the argocd-server: *microk8s.kubectl describe pod argocd-server859b4b5578-439hp* (See Figure 8).

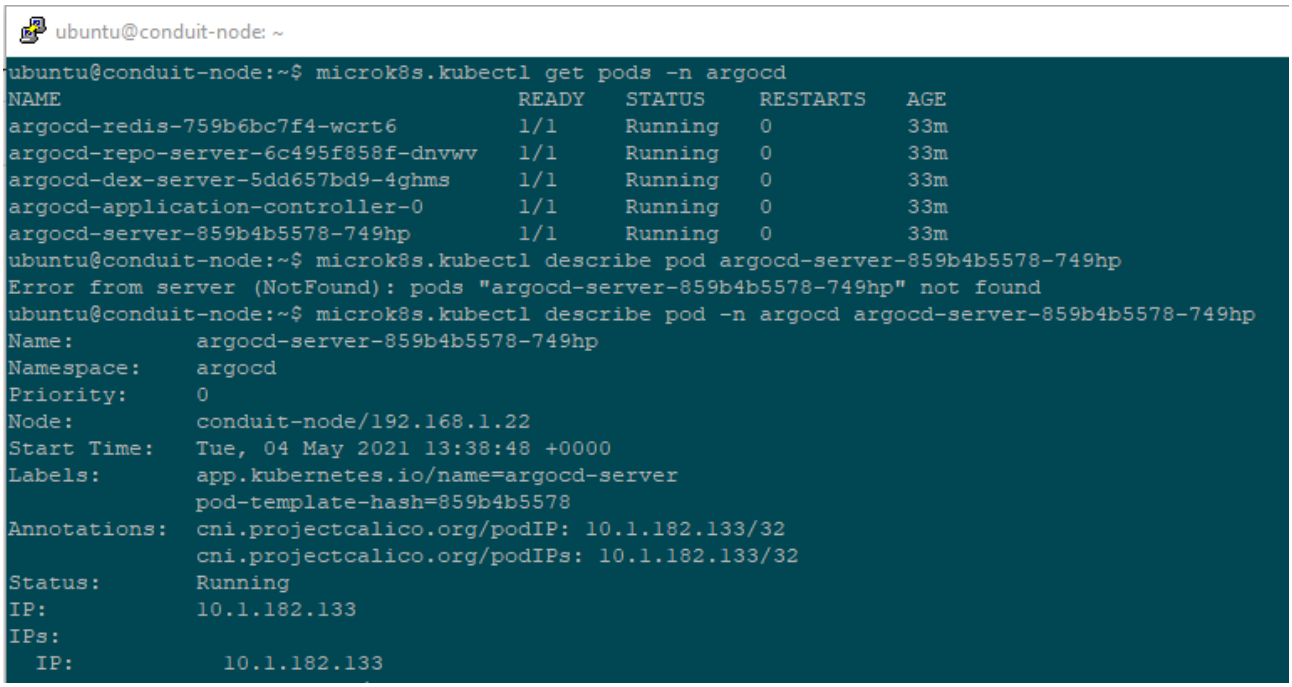

Figure 8. Kubectl commands to get information about pods

A port forward was opened for the argocd-server to make it accessible from the node with the in another PuTTY window with *microk8s.kubectl port-forward svc/argocd-server -n argocd 8080:443*.

This makes going inside the container possible with the IP address retrieved previously when used with the port 8080 that is listening for traffic. Port forwarding in Kubernetes lasts as long as the command is active in the terminal, so another terminal was needed. After connecting to the pod with *microk8s.kubectl exec -it -n argocd argocd-server859b4b5578-749hp*, changing the password was possible with the command *argocd account update-password*. The command asked for the password specified in the *argocd-initial-admin-secret* before allowing to change it to a new one (See Figure 9). Returning to the node could be done by inputting *exit.*

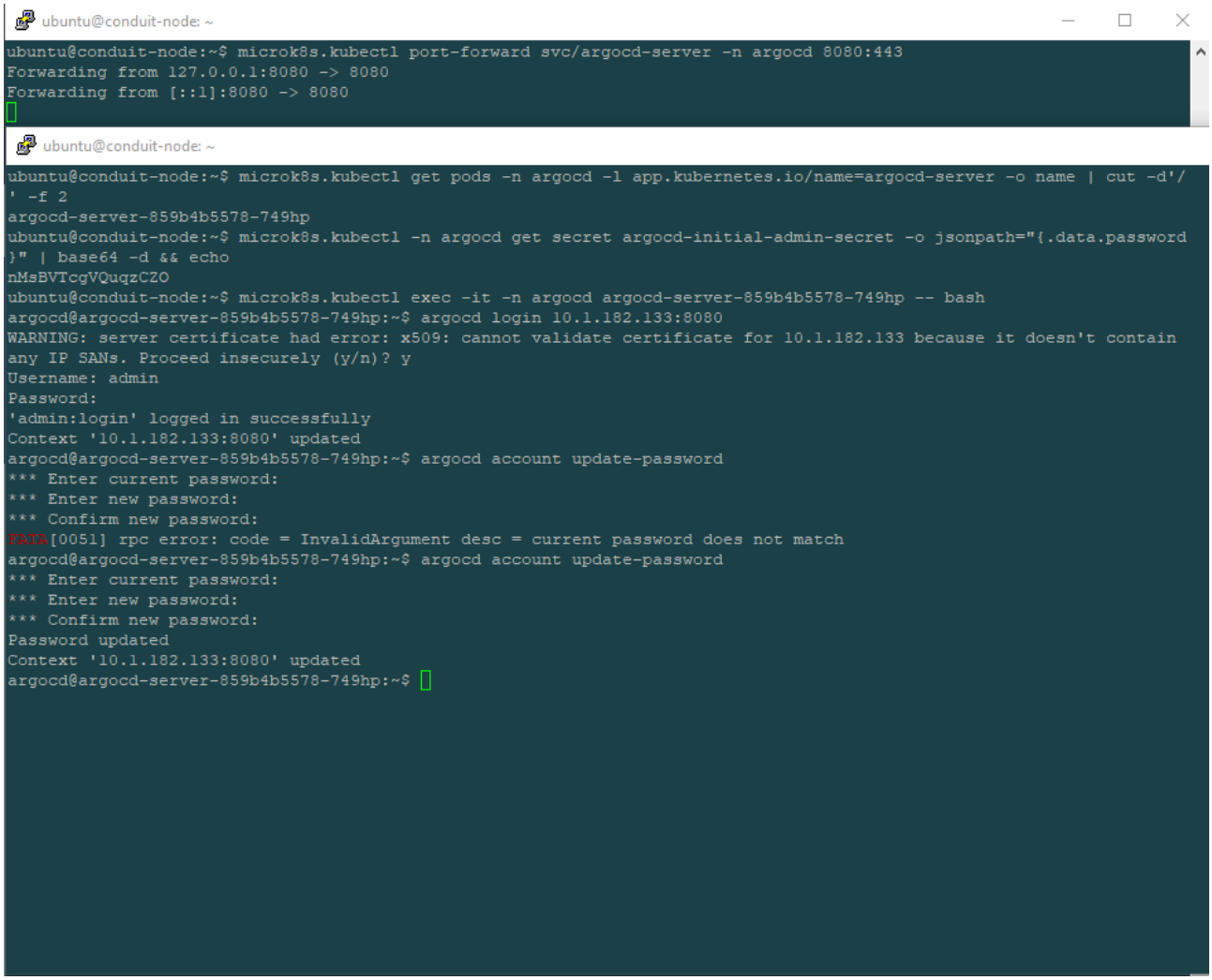

Figure 9. Changing the default admin password in Argo CD

Because the initial password in argocd-initial-admin-secret was not used anymore, it was deleted by inputting *microk8s.kubectl delete secret -n argocd argocd-initial-admin-secret.*

6.3 Configuring GitLab and Argo CD

The configuration for the Kubernetes cluster was mostly going to be held in GitLab with some configuration files such as Kubernetes secrets and a single Argo CD application CRD manifest being exceptions to this. The secrets were not managed by any external credential manager but were kept inside Kubernetes after configuration and the application CRD being held inside Conduit-Node.

Configuring GitLab

Three repositories were created to hold the respective parts needed for the application. The GitLab server used was hosted by the Jyväskylä university of applied sciences in *gitlab.labranet.jamk.fi*. A group called Conduit *Gitops* was created by going to *Groups – Create group* in GitLab. Then the three repositories needed were created by choosing "New project" while inside the group view of GitLab. A blank project *gitops* was created to host the Kubernetes YAML manifests that Argo CD uses as the single source of truth for the Kubernetes configuration. Afterward the two remaining application source projects were added as new projects but choosing *Import project* instead of a blank one. Choosing Repo by URL and inputting https://github.com/gothinkster/node-express-realworld-example-app for the *backend-source* project and https://github.com/gothinkster/react-redux-realworld-example-app for *frontend-source*. Separate repositories for the backend and frontend of the Conduit app were used so that separate program images could be built later with the GitLab CI while separating the Kubernetes configuration from the application code according to Argo CD best practices. were named *backend-source* and *frontend-source*. Git was installed with default options on the Windows 10 machine from https://git-scm.com/download/win.

From the Gitlab web IDE, the *gitops* repository was split into a development and production branch by committing a *prod* folder and a temporary file to the master branch that would act as the *prod* branch. Then renaming the folder into *dev*, changing the name of the temporary file, and committing to a new branch named *dev* for clarity. The two different test files were created with random text to test syncing later. Since the project was left on private visibility, Argo CD needs to have credentials to read the repository for changes. The access token *argo_pull* was added from the *gitops repository settings – Access Tokens* with a *read_repository* scope and store the token value for later usage because it could not be read again after creation.

On the Windows machine, GIT bash *ssh-keygen* command was used to generate an SSH key pair to connect to Gitlab. After providing the passphrase, the public key was provided to GitLab under the user settings – SSH keys by copy-pasting the contents of the *id_rsa.pub* file to the key field.
After creating a local folder *gitops-dev* for cloning the dev branch and using right-click in file explorer to open Git Bash, the dev repository was cloned with *git clone --single-branch --branch dev git@gitlab.labranet.jamk.fi:conduit-gitops/gitops.git.* Same command was repeated inside a folder named *gitops-prod* while replacing *dev* with *master* in the command. Additionally, the email and username were provided in Git to show the information in git commits with the syntax of *git config –global user.email "email@example.com" and git config –global user.name "name"*.

Configuring an application spec for Argo

A YAML-file named *argocdsetup.yaml* was created to configure the application environments inside Argo CD using the text editor Nano on Conduit-Node with the command *sudo nano argocdsetup.yaml*. This application CRD file can be edited to tell Argo CD where the configuration files are held and what environment to apply them to. The file was divided into two parts: a development, and production environment. Both would a have similar configuration with the difference of pointing the targetRevision and path parameter to the respective branches in the GitOps GitLab repository and separate names into the metadata (See Figure 10). The Kubernetes namespace was separated to have the applications in their own virtual environments while the sync policy Prune was kept the same for both environments. The policy removes applications from the environment if they cannot be found from Git. In addition to this, the self-healing option was enabled because Argo CD does not do it by default. The option triggers an automatic synchronization when changes to the live cluster are made (Automated Sync Policy – Argo CD n.d). The *dev* and *prod* namespaces were added imperatively to the cluster with the syntax *microk8s.kubectl create namespace dev* although later it was found out that namespaces can also be created declaratively through the previous configuration file through a *syncPolicy* option *CreateNamespace=true*. The configuration was applied with *microk8s.kubectl apply -n argocd -f argocdsetup.yaml.*

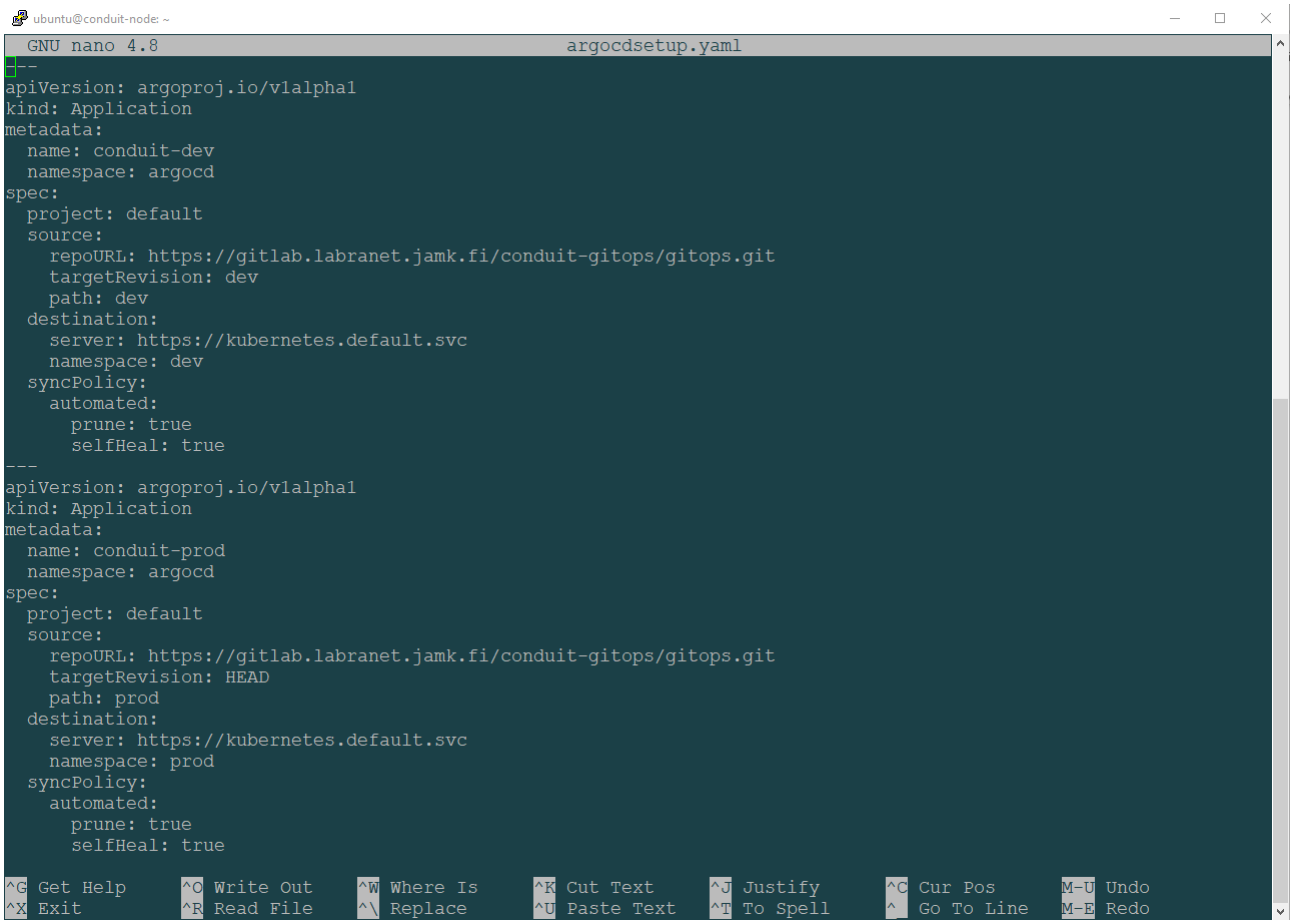

Figure 10. Argocdsetup.yaml

Adding the repository credentials was done imperatively to avoid having plain text passwords on the machine itself. This, like most things with Argo CD, can either be done with the CLI or the web UI. Argo CD does provide an option to have the credentials as a Kubernetes secret but a declarative definition of one still has the contents unencrypted and as such, can't be stored in Git. Secret managers can still be used with Argo CD, but it was decided to handle the repository credentials imperatively instead to avoid further dependencies and configuration.

The credentials were added by port forwarding the Argo CD server *argocd-server859b4b5578-749hp* and connected to again and inputting the command *argocd repo add* while providing the GitLab repository URL, a username that can be anything, and a password parameter where the value is the project access token *argo_pull* previously created for the GitOps project in GitLab (See Figure 11).

Figure 11. Repository credentials for Argo CD

6.4 CI configuration

Since the application code was ready in the repositories and the environment was starting to form up, the CI/CD would be the next logical step to go through. GitLab CI would trigger image building stages from code changes and write the updated image tags to the related files in the *gitops* repository, triggering Argo CD to deploying the images to production (See Figure 12).

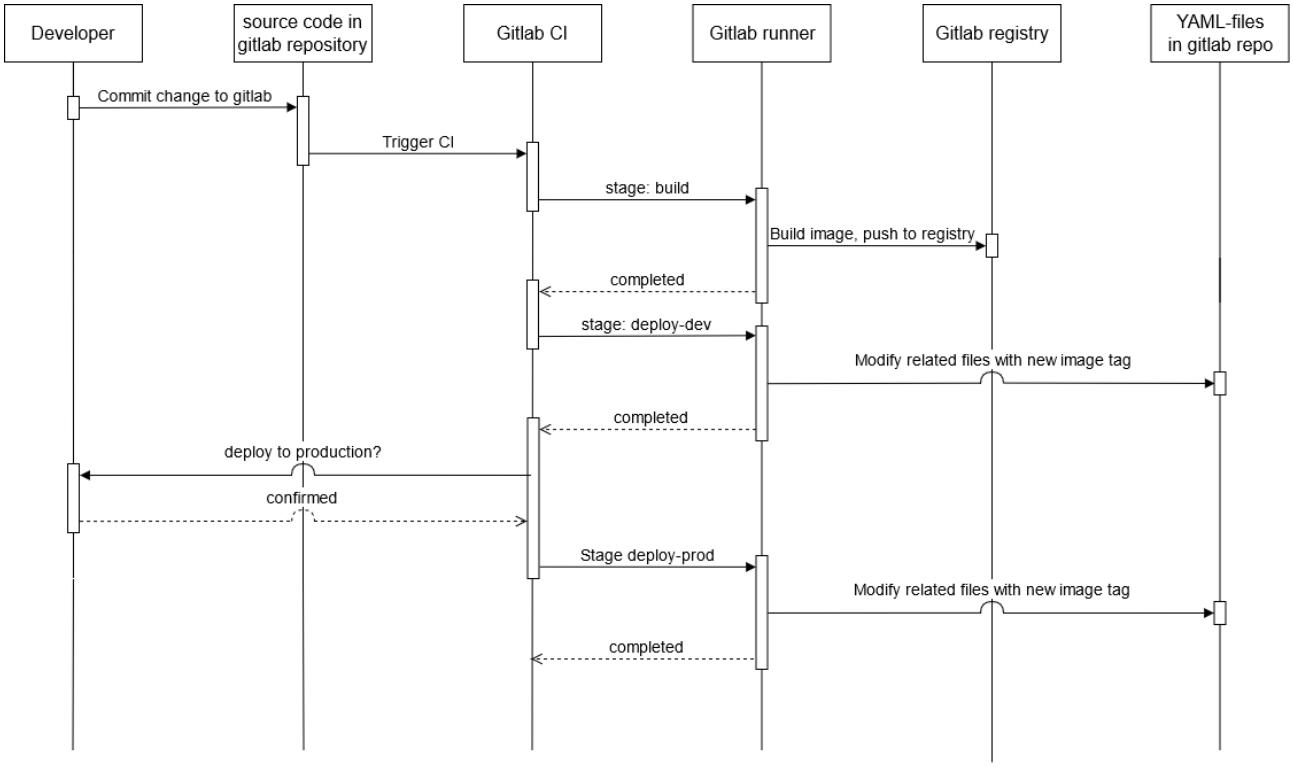

Figure 12. CI/CD sequence

In both the frontend and the backend projects, the shared CI/CD runner was assigned from Settings – CI/CD – Runners – Enable shared runners.

Because the CI/CD pipeline would modify the contents of the *gitops* repository, deploy keys needed to be added. Another pair of SSH keys were created with ssh-keygen and the public key text was pasted to the *gitops* repository from Settings – Repository – Deploy keys and given a name *SSH_DEPLOY*. The private key was instead added both the frontend and backend CI/CD variables from Settings – CI/CD – Variables by pasting the value and giving the name *SSH_PRIVATE_KEY*. The keys would be used later in the CI/CD as environment variables to connect to the *gitops* repository.
The output of *ssh-keyscan gitlab.labranet.jamk.fi* was also copied and added as a CI/CD variable *SSH_KNOWN_HOSTS* to verify that the key would connect to the right address and safeguard from man-in-the-middle attacks (Using SSH keys with GitLab CI/CD n.d).

After this a file *.gitlab-ci.yml* was added to both application project's root path. The pipelines would consist of four stages: *build*, *test*, *deploy-dev*, and *deploy-prod*. In the configuration file, they were listed first before their actual definitions (See Figure 13). The application image would be built using a modified Kaniko script from GitLab documentation that allows building docker images without installing Docker (Use Kaniko to build Docker images n.d). The script uses a Dockerfile specified in the root of the repository to build the image and tags and pushes it to the project's image registry. The build stage triggers only if the master branch is updated.
Specifying the CI/CD runner tag *general* was required for every stage so that the runner knows what tasks belong to it.

```
stages:
  - build
  - test
  - deploy-dev
  - deploy-prod

variables:
  # Save manifest repository URL to a variable for clarity
  MANIFEST_REPOSITORY: git@gitlab.labranet.jamk.fi:conduit-gitops/gitops.git

build:
  stage: build
  image:
    name: gcr.io/kaniko-project/executor:debug
    entrypoint: [""]
  script:
    # build from dockerfile with Kaniko
    - echo "{\"auths\":{\"$CI_REGISTRY\":{\"username\":\"$CI_REGISTRY_USER\",\"password\":\"$CI_REGISTRY_PASSWORD\"}}}" >
    /kaniko/.docker/config.json
    - /kaniko/executor --context $CI_PROJECT_DIR --dockerfile ./Dockerfile --destination $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
  only:
    - master
  tags:
    #assign shared runner
    - general
```

Figure 13. .gitlab-ci.yml build stage

The next stage in order was test. It was written with just as an example without any actual tests implemented due to time constraints, but it still needed the runner assigned to it for the pipeline to not fail during the stage.

The next real stage *deploy-dev* was used to modifying the new image tags to the configuration manifests the residing in the *gitops* repository. First in the before script, the tool Kustomize was installed on the Alpine virtual machine to modify the Kubernetes YAML-files (See Figure 14). SSH key was imported from *SSH_PRIVATE_KEY* to get access to the private repository *gitops*. In the script itself the *gitops* repository branch dev was cloned. A Kustomize command was used in the frontend folder to update the image tags by using environment variables saved from the build phase. Finally, the changes were committed and pushed back to the dev branch. Passing the stage would update the *gitops* repository, which would trigger a deployment to the dev environment in the cluster from Argo CD.

```
test:
  stage: test
  script: echo "insert tests here..."
  tags:
    # assign shared runner
    - general

deploy-dev:
  stage: deploy-dev
  image: alpine:latest
  before_script:
    # kustomize install
    - apk add --no-cache git curl bash
    - curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install_kustomize.sh"  | bash
    - mv kustomize /usr/local/bin/
    - git config --global user.email "gitlabbu@gitloponer.com"
    - git config --global user.name "GitLab CI"

    # add SSH key to runner from CI/CD variables so that it can push to repository
    - apk add --no-cache --update openssh-client
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
    - echo "$SSH_KNOWN_HOSTS" >> ~/.ssh/known_hosts
    - chmod 400 ~/.ssh/known_hosts
    - chmod 400 ~/.ssh/id_rsa
  script:
    # clone manifest repository - update image tag with kustomize - push changes to same repository
    - git clone --single-branch --branch dev $MANIFEST_REPOSITORY
    - ls -l
    - cd gitops/dev/frontend/
    - kustomize edit set image $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - cat kustomization.yaml
    - git commit -am '[skip ci] DEV image updated'
    - git push origin dev
  only:
    - master
  tags:
    - general
```

Figure 14. Test and deploy-dev stages of the pipeline

The final part of the CI definition was the *deploy-prod* stage, which would act very similarly to the *deploy-dev* stage but to change the running *prod* environment images. The configuration itself is mostly the same as *deploy-dev* but the branch cloned in the script section was the master branch of the manifest repository (See Figure 15). Since the folder naming of the repository was different from the *dev* branch, the path *gitops/prod/frontend* was specified. After this, the changes would be pushed to the master branch hosting the production manifests. A key difference to the deployment to the development environment is that the production deployment would be done only after a manual confirmation would be done in GitLab. This could be specified with the keyword and parameter *when: manual.*

```
deploy-prod:
  stage: deploy-prod
  image: alpine:latest
  before_script:
    # install kustomize
    - apk add --no-cache git curl bash
    - curl -s "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hack/install_kustomize.sh"  | bash
    - mv kustomize /usr/local/bin/
    - git config --global user.email "gitlabbu@gitloponer.com"
    - git config --global user.name "GitLab CI"

    #add SSH key to runner from CI/CD variables so that it can push to repository
    - apk add --no-cache --update openssh-client
    - mkdir -p ~/.ssh
    - chmod 700 ~/.ssh
    - echo "$SSH_PRIVATE_KEY" > ~/.ssh/id_rsa
    - echo "$SSH_KNOWN_HOSTS" >> ~/.ssh/known_hosts
    - chmod 400 ~/.ssh/known_hosts
    - chmod 400 ~/.ssh/id_rsa
  script:
    # clone manifest repository - update image tag with kustomize - push changes to same repository
    - git clone --single-branch --branch master $MANIFEST_REPOSITORY
    - ls -l
    - cd gitops/prod/frontend/
    - kustomize edit set image $CI_REGISTRY_IMAGE:$CI_COMMIT_SHA
    - cat kustomization.yaml
    - git commit -am '[skip ci] PROD image updated'
    - git push origin master
  only:
    - master
  tags:
    # assign shared runner
    - general
  # requires manual approval to deploy to production
  when: manual

```

Figure 15. Deploy-prod stage

The basic framework for the GitLab CI was now formed and the file was copied to the *backend-source* repository. Only minor tweaks needed to be made between the files. The path to the directory where the Kustomize command would be run would be changed to *gitops/dev/backend* for the *deploy-dev* stage and *gitops/prod/backend* for the *deploy-prod* stage.

6.5 Conduit configuration

The configuration for the continuous integration was finished but to let the pipeline build images for Kubernetes, the docker image building instructions were still required. Some preparation was needed. In the root of the *backend-source* repository, a file called *Dockerfile* was created. The file would be used as instructions for the pipeline to build an image from the application files. In the file, the *package.json* file was copied to the image built on an *lts-alpine* image to reduce image size. The command *npm install* was run to install the node program package and dependencies required and to expose port 3000 for traffic. A *.dockerignore* file was also added to exclude logs and *node_modules* files from the image when built to save space (See Figure 16).

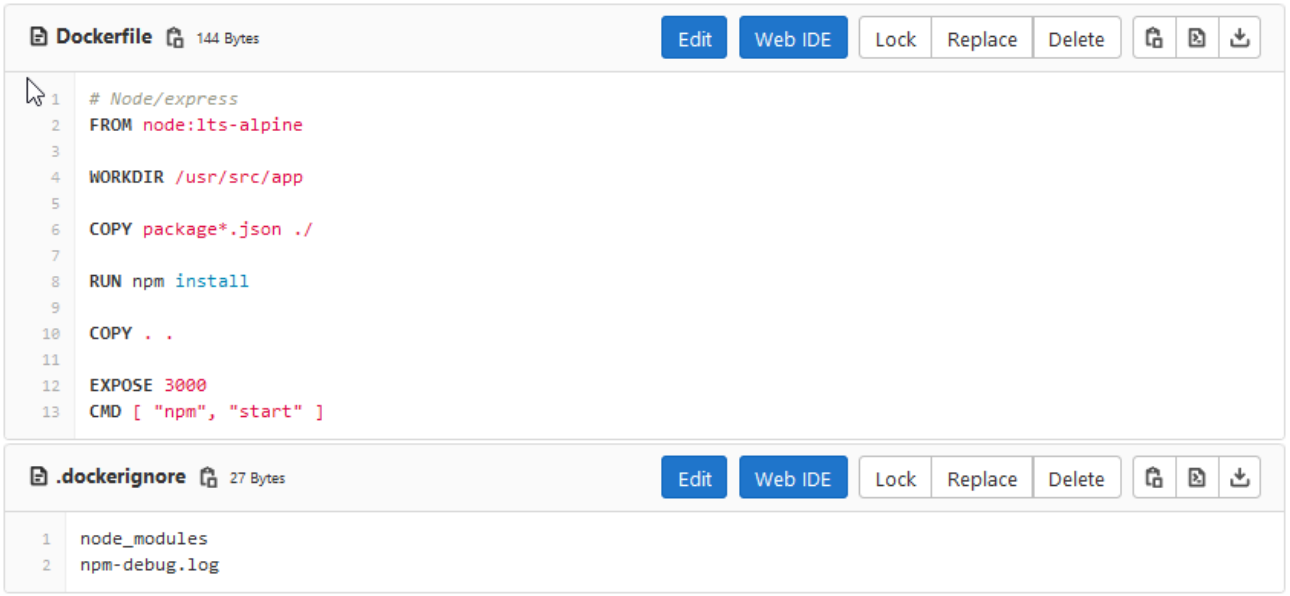

Figure 16. Backend Dockerfile and .dockerignore

The files were then copied to the *frontend-source* repository with the difference that the exposed port was changed from 3000 to 4100 because Reacts default port of 3000 would have conflicted with the port of the backend that was based on Node. To allow the multiple parts of Conduit to communicate with each other after deployment, the actual code still required small tweaks.

Changes to the application configuration

In *frontend-source/src/agent.js* file the API URL was changed from the default to a path of */api* which later would be routed using ingress files (See Figure 17). The default pointed to an actual API server that could be used for testing purposes, but a local database would be configured for the same purpose, which meant that a change was needed.

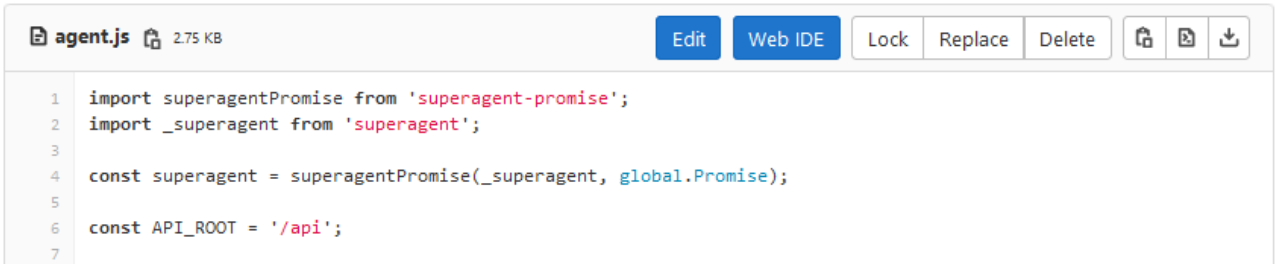

Figure 17. API URL change

Another minor tweak was to downgrade the frontend react-scripts to version 0.9.5 from version 1.1.1 because the application encountered problems with the default setting (See Figure 18).

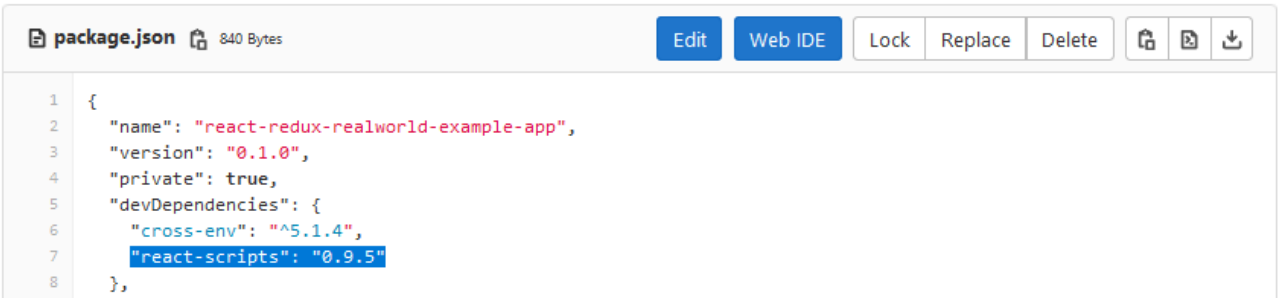

Figure 18. Downgrading the react-scripts version

Kubernetes configuration manifests

The folder structure of the *gitops* repository was organized to make working with the CI easier. The contents of the master branch *prod* folder were divided into a backend, frontend, and a database folder. Each folder including the *prod* folder required a *kustomization.yaml* file so that the

images could be updated via the CI. All required Kubernetes manifests related to each part of Conduit were also placed for later configuration (See Figure 19).
The structure was kept the same for the *dev* branch except for the *prod* folder being named *dev*.

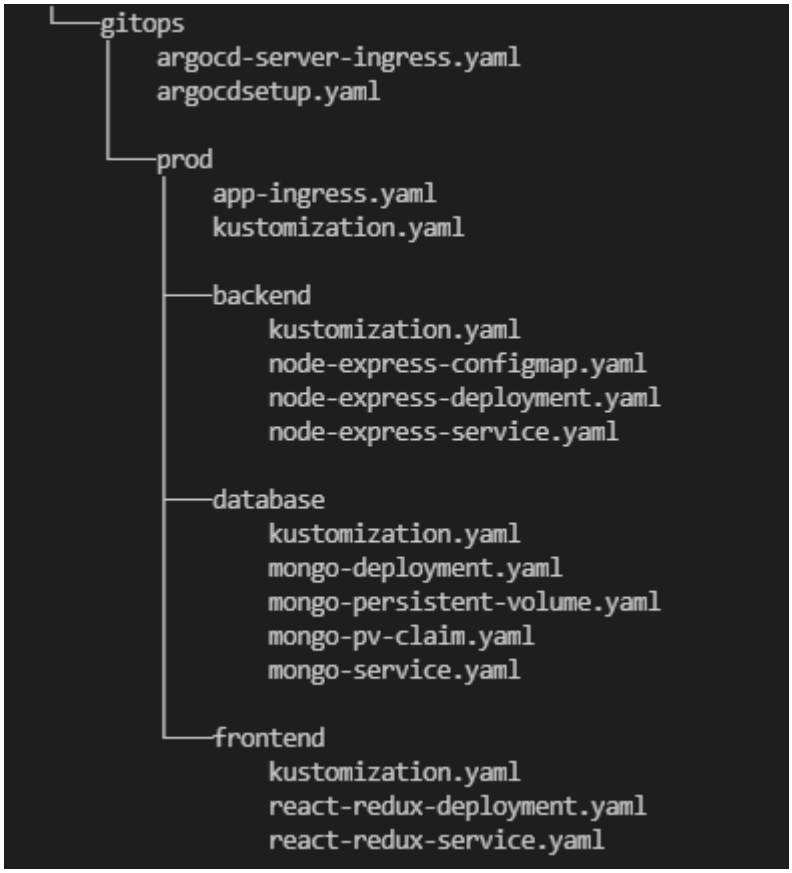

Figure 19. Folder and file structure of the gitops repository

The *kustomization.yaml* files in each directory define a list of Kubernetes configuration files in their respective folder as resources. In the case of the *prod* folders *kustomization.yaml* file, it also lists the folders inside of itself so that a Kustomize command can be used to modify them. If the folder had deployment manifests, lines were added to point to the path and name of the image in the respective container registry. GitLab CI would update the image tags through these files as they would be built (Appendices 1-4). With the preparations finished, the main part of Kubernetes configuration would start from defining the backend deployment manifest, service, secrets, and a ConfigMap file.

Backend configuration

The backend deployment file defines the specification for the backend pods that would contain the Node / Express backend image of Conduit. Selector labels *component: app* and *role: backend* were given to link the services to the pods (See Figure 20). These would be matched later in the service only if both were present since the selector uses the AND operator by default. The used image is specified in the manifest but the previously written *kustomization.yaml* specifies the actual tagged version of the image. The image path could be found from GitLab *backend-source* repository from the menu *Packages & Registries – Container registry.* An environment variable *NODE_ENV* is given with a value of *production* that allows for the custom database definition for the variable *MONGODB_URI* which would otherwise default to *localhost*. Then the value is remapped by pointing to the ConfigMap *node-express-configmap.yaml*. So that the backend can talk to the database, a secret also needed to be created to provide credentials to MongoDB. Port 3000 was opened for traffic so that the frontend can talk to the backend. For the deployment to be able to pull images from the private image registry, *imagePullSecrets* were pointed with a name of *backend-regcred*. Both the secrets and the ConfigMap needed to be created by hand. The number of replicas was set to 2. This would create two identical pods and load balance between them automatically.

node-express-deployment.yaml 911 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: node-express-deployment
spec:
  replicas: 1
  selector:
    matchLabels:
      component: app
      role: backend
  template:
    metadata:
      labels:
        component: app
        role: backend
    spec:
      containers:
        - name: node-express
          image: gitlab.labranet.jamk.fi:4567/conduit-gitops/backend-source
          env:
            - name: NODE_ENV
              value: "production"
            - name: MONGODB_URI
              valueFrom:
                configMapKeyRef:
                  name: node-express-configmap
                  key: mongodb_uri
            - name: SECRET
              valueFrom:
                secretKeyRef:
                  name: node-express-secret
                  key: node_express_secret
          ports:
            - containerPort: 3000
      imagePullSecrets:
      - name: backend-regcred
```

Figure 20. Backend deployment manifest

The private container registry pull credentials were created from the backend repository menus *Settings – Access Tokens* by giving the name *backend_regcred* and giving the *read_registry* scope to the access token. This provided a password string that could only be viewed once. The frontend deployment also needed the same kind of token to allow images to be pulled from the registry, so it was created in the same way from the respective repository.

To create the secret on the cluster, an SSH connection was taken again to Conduit-Node. The two secrets needed to be created to both the *dev* and *prod* environment with a total of four modified versions of the same command:

microk8s.kubectl create secret docker-registry backend-regcred -n prod --docker-server=gitlab.labranet.jamk.fi:4567 --docker-username=gittulabbu --docker-password= rSYYmyiEcMoFB2uxh4_K --docker-email=gittulabbu@example.fi.

In the command the namespace needed to be changed from *-n prod* to *-n dev* to create the secret to both environments. As for the frontend, the name *backend-regcred* was changed to *frontend-regcred* and the *–docker-password* was changed to the frontend equivalent token value from GitLab. Since the authentication with the token does not need a username, the email values were dummies.

The secret *node-express-secret* for the was created by generating a random password with the command *date +%s | sha256sum | base64 | head -c 32 > ./node_express_secret.txt.* Then the text file was used to create a secret with *microk8s.kubectl create secret generic node-express-secret --namespace=dev --from-file=node_express_secret=./node_express_secret.txt.* And again, the same command needed to be used but this time changing *–namespace=dev* to *–namespace=prod* to have the secret in both namespaces. The text file could be deleted afterward.

The ConfigMap for the variable *MONGODB_URI* was created by filling the *node-express-config-map.yaml* with the information of the service responsible for the MongoDB networking so that the backend gets the IP address of the database (See Figure 21).

node-express-configmap.yaml 192 Bytes Edit Web IDE Lock Replace Delete

```
1  apiVersion: v1
2  kind: ConfigMap
3  metadata:
4    name: node-express-configmap
5  data:
6    mongodb_uri: mongo-service
7  #mongodb_uri = mongodb IP address. passed to the backend in node-express-deployment
```

Figure 21. Node-express-configmap.yaml

The last part of the backend configuration was creating a service to make the components talk with each other. Selector labels *component: app* and *role: backend* were given to match the service to the backend pods and the TCP port 3000 was exposed (See Figure 22).

node-express-service.yaml 192 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: v1
kind: Service
metadata:
  name: node-express-service
spec:
  selector:
    component: app
    role: backend
  ports:
    - protocol: TCP
      port: 3000
      targetPort: 3000
```

Figure 22. Node-express-service.yaml

Frontend configuration

The frontend deployment *react-redux-deployment.yaml* and the service react-redux-service.yaml were simpler compared to the backend configuration although very similar in their logic (See Figure 23). For the deployment, the image information, image registry secret, and labels were given along with the frontend respective information including port 4100 for connecting to the frontend. for the service, the matching labels and port information were provided.

react-redux-deployment.yaml 494 Bytes — Edit | Web IDE | Lock | Replace | Delete

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: react-redux-deployment
spec:
  replicas: 2
  selector:
    matchLabels:
      component: app
      role: frontend
  template:
    metadata:
      labels:
        component: app
        role: frontend
    spec:
      containers:
        - name: react-redux
          image: gitlab.labranet.jamk.fi:4567/conduit-gitops/frontend-source
          ports:
            - containerPort: 4100
      imagePullSecrets:
      - name: frontend-regcred
```

react-redux-service.yaml 192 Bytes — Edit | Web IDE | Lock | Replace | Delete

```yaml
apiVersion: v1
kind: Service
metadata:
  name: react-redux-service
spec:
  selector:
    component: app
    role: frontend
  ports:
    - protocol: TCP
      port: 4100
      targetPort: 4100
```

Figure 23. Frontend deployment and service

Database configuration

The database deployment required a volume to allow persistent data to the database. A persistentVolumeClaim was specified with *claimName: mongo-pv-claim-prod (*See Figure 24*)*. MongoDB did not require an image to be built considering it was readily available in the Docker Hub container registry. A hostPath volume was given a path */data/db* with the name *mongo-pv-storage-prod* and the container port 27017 was exposed for the pods. The hostPath volume was chosen because of its simplicity and the readily available Kubernetes documentation. In a real-world production environment, using a hostPath volume could potentially be a security risk and is recommended to be only mounted with *readOnly* permissions. Considering that the database would not

host any sensitive data in this project, it was chosen to be used regardless (Volumes – Kubernetes n.d).

mongo-deployment.yaml 596 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-deployment
spec:
  selector:
    matchLabels:
      component: app
      role: database
  replicas: 1
  template:
    metadata:
      labels:
        component: app
        role: database
    spec:
      volumes:
        - name: mongo-pv-storage-prod
          persistentVolumeClaim:
            claimName: mongo-pv-claim-prod
      containers:
      - name: mongo-database
        image: mongo:4.4
        volumeMounts:
        - mountPath: /data/db
          name: mongo-pv-storage-prod
        ports:
        - containerPort: 27017
```

Figure 24. Mongo-deployment.yaml

The PersistentVolume mongo-persistent-volume.yaml specified the name *manual-prod* for the *storageClassName* that would be matched in the PersistentVolumeClaim to bind them together (See Figure 25). The storage capacity was set to 2Gi and the access mode to ReadWriteOnce because of the single node nature of the cluster. Other options are for multi node clusters. Finally, the location where the data would be stored was specified with the path *"/mnt/data/prod"*. The PersistentVolumeClaim mongo-pv-claim.yaml consisted of mostly the same information to match the PersistentVolume but was still needed for the nature of how volumes work in Kubernetes.

mongo-persistent-volume.yaml 289 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongo-persistent-volume-dev
  labels:
    type: local
spec:
  #storageClassName: microk8s-hostpath
  storageClassName: manual-dev
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data/dev"
```

mongo-pv-claim.yaml 237 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pv-claim-dev
spec:
  #storageClassName: microk8s-hostpath
  storageClassName: manual-dev
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```

Figure 25. Mongo-persistent-volume.yaml

The database service *mongo-service* configuration followed the already established way of configuring simple Kubernetes services (See Figure 26). Database related labels were specified as selectors the port 27017 was exposed for accessing the database.

mongo-service.yaml 188 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: v1
kind: Service
metadata:
  name: mongo-service
spec:
  selector:
    component: app
    role: database
  ports:
    - protocol: TCP
      port: 27017
      targetPort: 27017
```

Figure 26. Mongo-service.yaml

Differences between the master (prod) and the *dev* branch were required to have two separate databases. The claim and volumes were created for both the *dev* and prod environments separately because otherwise, both environments would use the same database. The differences in the configuration were still minimal because the naming convention could be followed by just replacing *"prod"* to "*dev*" in most cases except for services that remained identical (See appendices 5-7).

6.6 Deployment

Argo CD would deploy the programs after an image tag is modified by the CI in the *gitops* repository (See Figure 27). In addition to this, any changes made manually to the environment would be automatically reverted if they differed from the declared state in Git. This would be tested later.

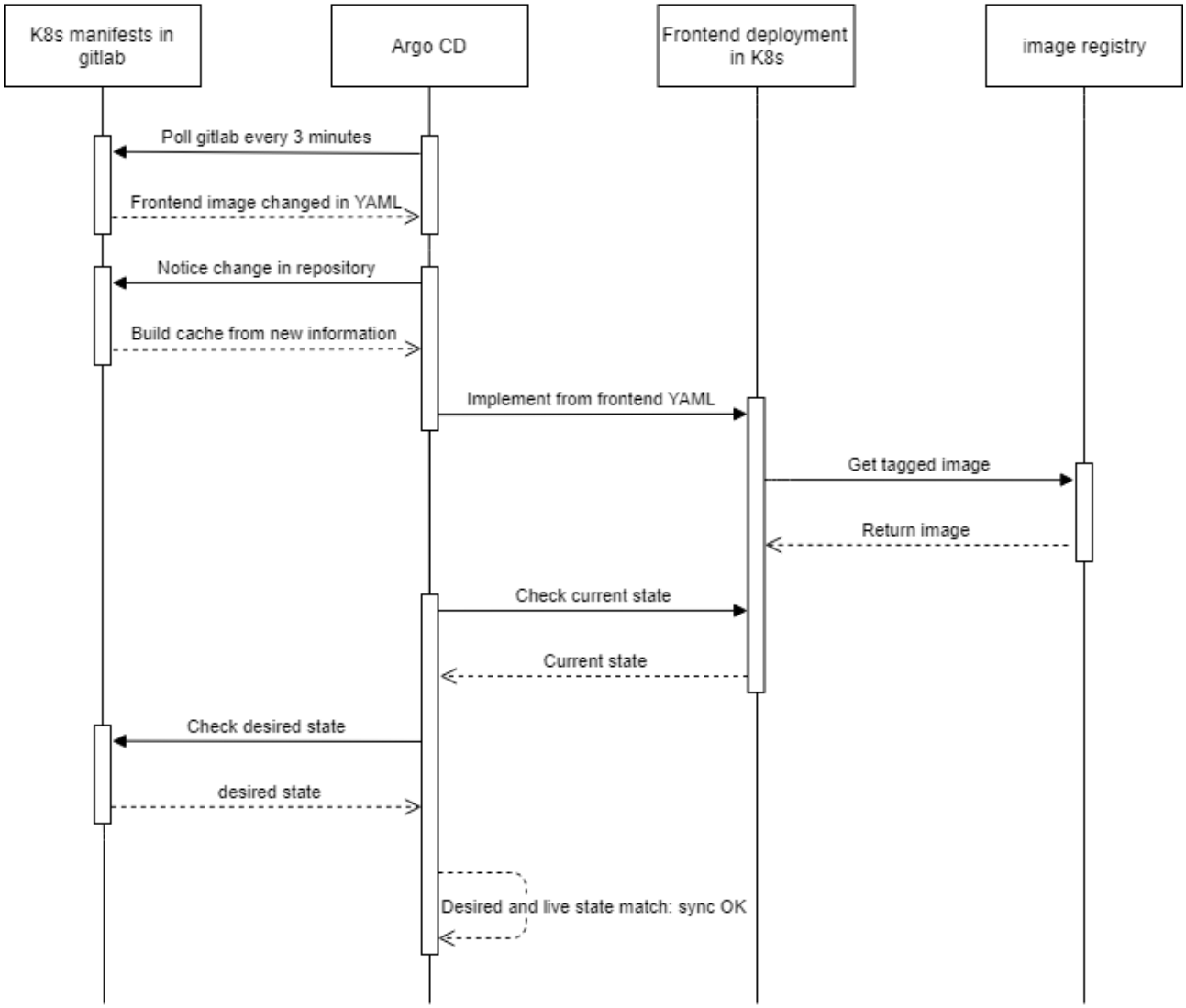

Figure 27. Deployment after an image tag change

For the sake of testing, the title field of the application was changed from the index.html to have a smiley face in the master branch of *frontend-source* repository and was committed to trigger the pipeline. The pipeline ran through the build, test, and *deploy-dev* phase successfully in under a minute, which could be followed from the *GitLab CI/CD – Pipelines* section. This could also be verified from the *gitops* repository because the *deploy-dev* stage committed changes to the frontend *kustomization.yaml* (See Figure 28). This triggered the deployment to the *dev* environment from Argo CD, which could be either checked from the Conduit-Node with *microk8s.kubectl get pods -n dev* or through the Argo CD web UI.

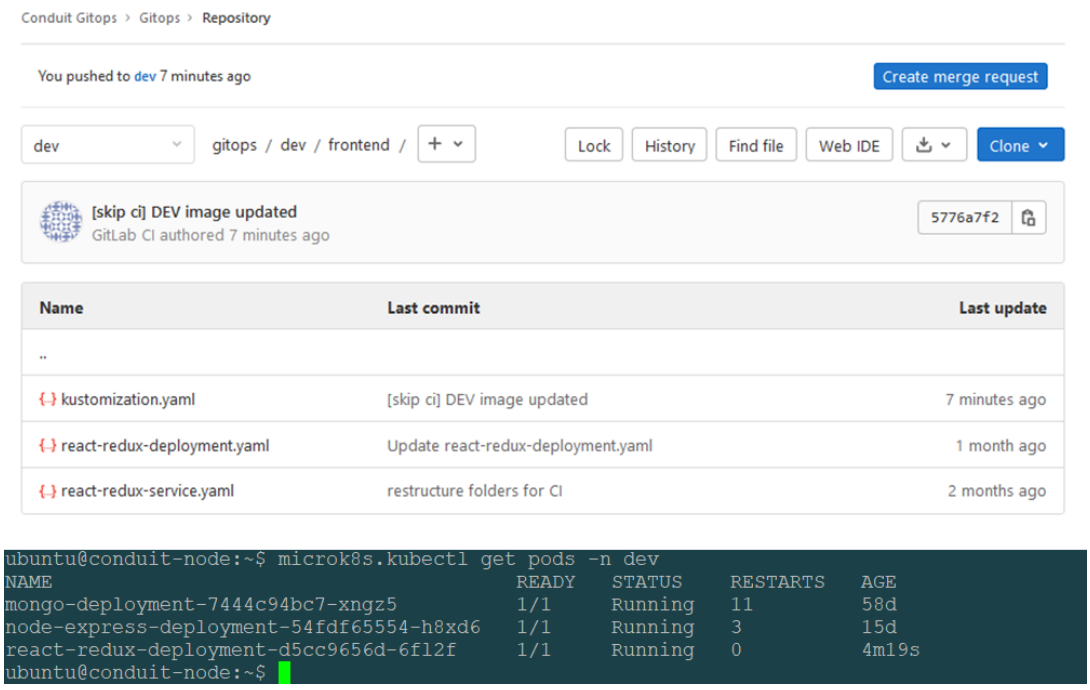

Figure 28. Frontend kustomization updated

The Argo CD web UI could be accessed through the internet if it was enabled through configuration but for simplicity's sake, it was accessed through the CSC console window that could be used to view the desktop of Conduit-Node. Checking the IP address of argocd-server was required again because it would be used to connect to the UI through a web browser. This was done with the command *microk8s.kubectl describe pod argocd-server-859b4b5578-749hp -n argocd | grep IP*. Accessing the web UI still required the port forward of the argocd-server which was done again

with *microk8s.kubectl port-forward svc/argocd-server -n argocd 8080:443 in another PuTTY terminal.* With the IP address copied from the terminal, a browser was opened through the CSC console view of Conduit-Node and pasted with the addition of port 8080. After logging in with Argo CD admin credentials and choosing *conduit-dev* from the applications a view opened that allowed control to the deployment of the application (See Figure 29). The web UI allows filtering based on objects and their health, changing sync policies and application settings. Rolling back deployments is also possible if auto-synchronization is set off. The synchronization to the cluster had went through successfully.

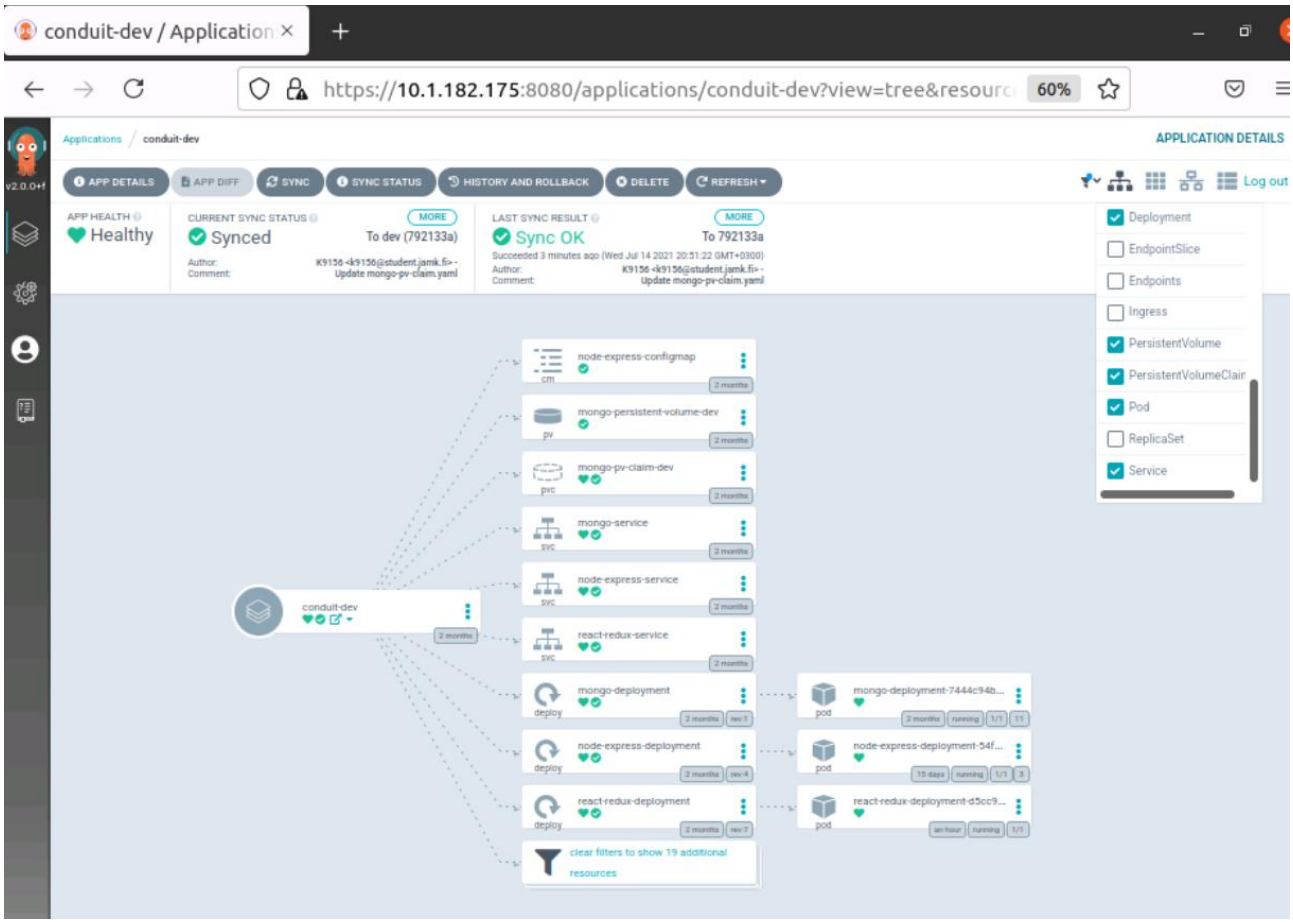

Figure 29. Argo CD web UI application view

To check the actual running applications, Ingresses were configured to both environments which allowed accessing them from the node. HTTPS enforcement was set to false through the *ssl-redirect* annotation because of the lack of certificates and the hostname was set to *conduit.dev.k8s*. All the traffic arriving to the paths in the end of the hostname was directed to the frontend service

and its port 4100. An exception to this was to set by having the requests that come to the path */api/.* These were redirected to the backend service and port 3000. The change from the *frontend-source/src/agent.js* API URL from before was needed for this to work. The production version of the ingress is identical apart from the hostname that is *conduit.prod.k8s* instead.

app-ingress.yaml 535 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: app-ingress-dev
  annotations:
    ingress.kubernetes.io/ssl-redirect: "false"
spec:
  rules:
  - host: "conduit.dev.k8s"
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: react-redux-service
            port:
              number: 4100
      - path: /api/
        pathType: Prefix
        backend:
          service:
            name: node-express-service
            port:
              number: 3000
```

Figure 30. Ingress for the development environment

It was decided that both applications would also be made accessible over the internet for easier testing. DNS was not configured but instead, the *hosts* file on the Windows 10 was modified to include the floating IP of Conduit-Node from CSC and the hostnames set in the ingresses. It could be found at *C:\\windows\System32\drivers\etc\hosts*. The lines *128.214.252.46 conduit.dev.k8s* and *128.214.252.46 conduit.prod.k8s* were added. This alone did not allow the traffic to the cluster because the network rules of the CSC Security group did not allow it. This could be changed from *CSC Network – Security Groups – Manage rules*. An Ingress rule for HTTP was added with a /32 network mask for the public IP address of the windows 10 computer. This would allow HTTP traffic to go inside the instances only from this one specific IP address. After the changes, *http://conduit.dev.k8s* was accessed through a browser and a post was posted to test the functionality of the site (See Figure 31)

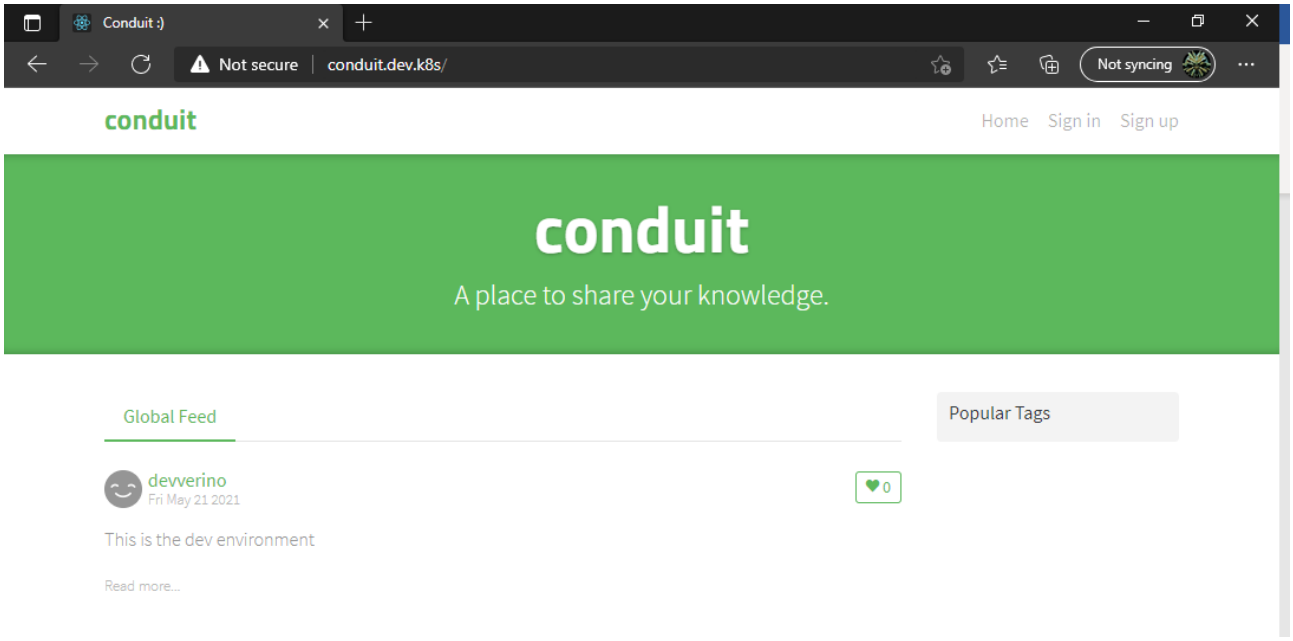

Figure 31. Testing Conduit

The smiley was visible in the page title and confirmed that the deployment had gone through. Deploying to the production environment still required a manual confirmation from GitLab. (See Figure 32). After the confirmation which triggered the *deploy-prod* stage, the production site was also accessible within minutes via the Windows 10 computers browser.

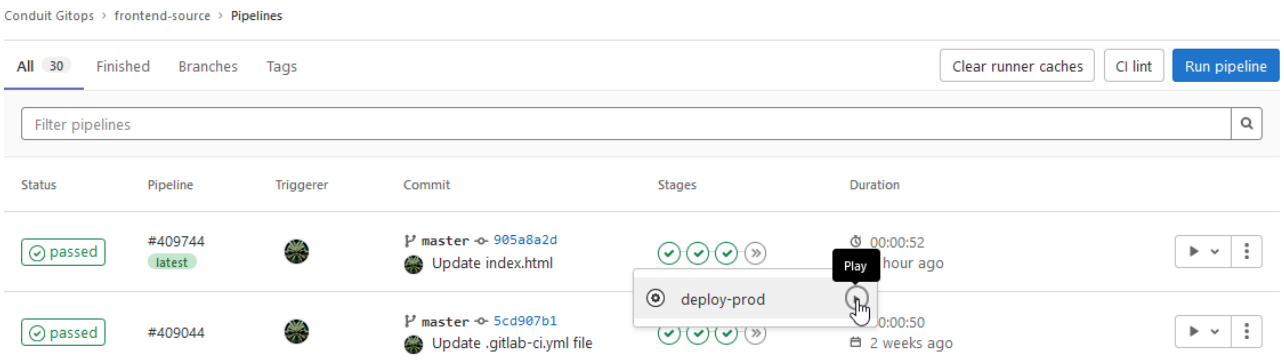

Figure 32. Deploy-prod stage confirmation

Now that the application was deployed, it was time to test Argo CD synchronization and the idea that the current state of the application would be ensured to match the version-controlled code. This could be tested by simply deleting a pod from an Argo CD-managed namespace (See Figure 33). The deletion was still possible but after deleting the pod Argo CD recreated a new one to match the declared state. The same applies to changes made into a running pod where the pod is then removed and a new one is created in its place.

```
ubuntu@conduit-node:~$ microk8s.kubectl get pods -n dev
NAME                                     READY   STATUS    RESTARTS   AGE
react-redux-deployment-74d7cb94f9-ddkgk  1/1     Running   3          30d
mongo-deployment-7444c94bc7-xngz5        1/1     Running   14         89d
node-express-deployment-54fdf65554-h8xd6 1/1     Running   7          46d
ubuntu@conduit-node:~$ microk8s.kubectl delete pod react-redux-deployment-74d7cb94f9-ddkgk -n dev
pod "react-redux-deployment-74d7cb94f9-ddkgk" deleted
^[[A^[[A^[[Aubuntu@condmicrok8s.kubectl get pods -n dev
NAME                                     READY   STATUS    RESTARTS   AGE
mongo-deployment-7444c94bc7-xngz5        1/1     Running   14         89d
node-express-deployment-54fdf65554-h8xd6 1/1     Running   7          46d
react-redux-deployment-74d7cb94f9-9qdll  1/1     Running   0          17s
ubuntu@conduit-node:~$
```

Figure 33. Deleting a pod

Another test was done by creating a new deployment by copying the frontend manifest from GitLab but applying it by hand with kubectl (See Figure 34). Immediately after applying the deployment, Argo CD could be seen terminating the pods because they did not match the wanted state. The project was declared to be finished with all the environments running in a manageable way.

```
ubuntu@conduit-node:~$ microk8s.kubectl apply -f test.yaml -n dev
deployment.apps/react-redux-deployment configured
ubuntu@conduit-node:~$ microk8s.kubectl get pods -n dev
NAME                                     READY   STATUS        RESTARTS   AGE
mongo-deployment-7444c94bc7-xngz5        1/1     Running       14         89d
node-express-deployment-54fdf65554-h8xd6 1/1     Running       7          46d
react-redux-deployment-74d7cb94f9-9qdll  1/1     Running       0          4m33s
react-redux-deployment-5649bdbfcc-trgct  0/1     Terminating   0          6s
react-redux-deployment-74d7cb94f9-tmzq2  1/1     Terminating   0          7s
ubuntu@conduit-node:~$ microk8s.kubectl get pods -n dev
NAME                                     READY   STATUS    RESTARTS   AGE
mongo-deployment-7444c94bc7-xngz5        1/1     Running   14         89d
node-express-deployment-54fdf65554-h8xd6 1/1     Running   7          46d
react-redux-deployment-74d7cb94f9-9qdll  1/1     Running   0          4m50s
ubuntu@conduit-node:~$
```

Figure 34. Creating a pod

7 Results and findings

The purpose of the project was to study the concepts and tools of GitOps and how they can be used to deploy and manage software. Answers to the research questions were sought by researching related documentation and by forming a project to get familiar with GitOps and its related technologies.

How does GitOps differ from DevOps?

The difference between GitOps and DevOps is notable, but they are not opposing practices as they can be used hand in hand. DevOps aims to lessen the gap between developers and operative people left from the principles of Agile by offering cultural guidelines and philosophy for modern software development. DevOps practices are still present in GitOps and can still be utilized because the GitOps practices are mostly an expansion and not a rewrite of DevOps guidelines. DevOps is much more fledged out in its entirety compared to GitOps because of its broader focus on the company culture. On the other hand, GitOps is a much more recent philosophy without an official standard, and it has a tighter focus in its way of software deployment. GitOps was also originally meant to work with Kubernetes when DevOps is much more technology agnostic with its approach to implementing a working system. In theory, GitOps principles can still be fulfilled without using Kubernetes but the pairing between the two is an easy one because Kubernetes allows the configuration to be declarative and allows running operators like Argo CD inside itself to ensure that the desired state matches the current state. In short, GitOps offers a way to handle continuous deployment in cloud environments via Git when DevOps offers a set of practices, tools, and culture guidelines to solve conflicts between developers and the operative side. GitOps shares more similarities with Infrastructure as Code than DevOps as both can be used for declarative configuration, but the attention is drawn more towards applications in GitOps. The conclusions were drawn by comparing the general principles of the methods from the most original possible sources.

How does deploying with Argo CD differ from deploying to Kubernetes manually?

Comparing a manual deployment to deploying with Argo CD is in essence the same as comparing manual deployment to deploying with CI/CD because Argo CD is responsible for continuous de-

ployment while also applying the GitOps methods to it. Deploying software to Kubernetes manually is mostly done declaratively even without Argo CD. For example, applying changes to deployments can be done from the command line on a node by applying a declarative configuration manifest. This would still require an administrator to apply the command instead of Argo CD applying the manifest automatically from a Git repository. Without Argo CD, running an application in Kubernetes would also require the administrator to take a remote connection to the node and manually change the configuration manifests, and apply them from the command line. Performing changes to the cluster requires much fewer actions after the initial setup with Argo CD when compared to manual Kubernetes management because Argo CD automates large parts of the deployment when paired with continuous integration. If Argo CD is used, the configuration manifests would not reside on the Node as they would always be in a Git repository. This means that apart from the initial setup of Argo CD, access to the node is not needed for administrators that work with the manifests. It could be argued that this would improve security, although problems or situations might arise that still would require manual configuration. The self-healing abilities of Argo CD also mean that changes made by hand inside the cluster are reversed immediately if Argo CD finds there to be a mismatch between configuration states. This can also be thought of as a security advantage.

Credential management is also different from a regular CI/CD. When deploying with a push-based CI/CD pipeline, credentials to the environment need to be given for the pipeline to allow it to deploy to the cluster. With Argo CD, the operator itself needs credentials to read the Git repository and pull images from an image registry but the credentials are kept inside the cluster. Ensuring that the Git repository commits are valid becomes the security concern in this sense. Deploying with Argo CD can be done without auto-sync and too that requires manual deployment but to get the most out of the program would mean to leave the auto-sync enabled and instead choose what to deploy by controlling what is committed or merged to the branch in Git that is polled by Argo CD. The self-healing capabilities that Argo CD provides for the pods can also be turned off to provide more manual control to the live cluster. These kinds of options provide more possibilities for deployment than just doing it manually with Kubernetes although it does require additional configuration in the initial setup and adds more complexity to the cluster. Just having another system running on a server always provides more possibilities for failure.

How to deploy Conduit to a Kubernetes cluster using GitOps practices?

The GitOps practices followed were the four principles laid out by Weaveworks on their GitOps website (What you need to know - Guide to GitOps n.d). The first principle requires that the system is described declaratively. All the Kubernetes configuration was written in declarative YAML-manifests that state the end goal of the application instead of imperatively creating an application. The Conduit application was divided into a frontend, backend, and database. Frontend and backend source code was kept in their separate Git repositories while the database only needed a PVC, PV, and a deployment configuration manifest because a default MongoDB image was used from the Docker Hub image registry. All the Kubernetes configuration for the deployments, ConfigMaps, and services were kept declarative but the secrets for image registries needed to be configured separately through the command line and stored inside the cluster because holding leftover configuration for plaintext passwords on the node or Git did not seem like a secure way to handle them. External secret managers were available and compatible with Argo CD but were not used in the end.

The second principle requires that the canonical system state is versioned in Git. This was done by keeping the Kubernetes manifests in a GitLab repository and in addition keeping the Conduit source code separate from the Kubernetes manifests according to Argo CD best practices. This also makes handling pipelines clearer and allows authorizing parts of the project to relevant developers in larger projects. Most of the configuration could be done through Git after the initial setup of Argo CD on the cluster.

The third principle requires that approved changes are applied automatically to the system. This was achieved by allowing Argo CD to deploy the frontend, backend, and database of Conduit by allowing it to read the GitLab repositories and automatically deploy them when there was a change in the files.

The fourth principle requires a software agent to ensure correctness and alert of divergence. In this case, the software agent used was Argo CD. Argo CD handled the continuous deployment of Conduit but also ensured that the application was always up to date when compared to the one stored in Git. The state of the application still needed to be monitored from Argo CD UI because

no webhooks were configured for alerts. Argo CD still provides ways to send notifications to administrators by defining triggers for statuses. Deploying Conduit to Kubernetes is possible with GitOps principles. It requires declarative configuration to be held in Git and an agent like Argo CD to monitor the state of the application and provide automatic deployment of Conduit.

8 Discussion

8.1 Discussion of the main results

The project was formed with the goal to learn more about cloud native applications like Kubernetes and become familiar with GitOps because of its cutting-edge reputation. Before the project CI/CD pipelines were familiar only on a conceptual level but not in practice. Taking these in to consideration, the subject was an adventurous one to choose for an IT engineering student specializing in networking. The subject was vast and narrowing it down to a digestible portion required that the technologies were chosen beforehand without too much research put into researching the differences between different options. The focus of the project was on understanding the main concepts of GitOps and building infrastructure to deploy Conduit to a Kubernetes cluster through a CI/CD pipeline. The research question concerning the differences between DevOps and GitOps could have been changed to study the differences between IaC and GitOps because they are also closely related and can be confused with each other. The research question “How does deploying software with Argo CD differ from deploying to Kubernetes manually?” relied on the knowledge of manual Kubernetes configuration even when the focus of the project was on leaning on the GitOps way of deploying software. The question could still be answered through the knowledge gathered through the initial research of the subject and the tests done outside of the implementation itself that were not documented in the theory part itself. It could have also been reworded to better fit the project. The project was successful in its implementation even if the unfamiliarity slowed down the initial configuration. The theory part did not include many statistics or comparisons to the effectiveness of the technologies because most of the sources used were the original documentation of the software used. Some differing sources could have been sought out but the recentness of GitOps also meant that most publications were web articles and good statistics were not found.

8.2 Conclusions and development proposals

The project was successful in its implementation even if the unfamiliarity slowed down the initial configuration. Some parts of the project could be improved upon if the environment would be used in an actual production environment. Because of time constraints and narrowing the implementation, some decisions were made to streamline the process. Separating the development environment from the production environment was done by having a domain name for each written to a hosts file on the Windows 10 machine mapped with the IP address of the cloud server. An Ingress file routed the traffic to each environment based on the URL requested and firewall rules were made to point to a single IP address of the Windows 10 machine. This was done to not have to configure a DNS for the environments. SSL certification for the sites was also skipped even though using self-signed certificates is also a possibility. To test the scaling properties of the whole cluster would have benefitted from having multiple nodes to be used. This would have also provided high availability in a real environment.

Pairing GitOps with IaC tools like terraform and ansible would be a logical step to make the full infrastructure stack declarative. This was not done to keep the scope of the implementation more manageable but considering the context, it would have made sense. The benefit of using GitOps comes from the ability to ensure the state of the application in an environment while providing auditability by having the configuration in Git. Implementing GitOps still would require changes to pipelines if a “normal” CI/CD setup was used. Just by having Argo CD in a cluster also means that it needs to be monitored too since it is solely responsible for deploying the cluster. No problems were encountered with Argo CD itself, but the responsibility of the application is still large. Separate monitoring was not set up during the project because of time constraints. Kubernetes offers its own easy to install dashboard which is also packed with MicroK8s. It was not needed during the project and most status checks were done through the Argo CD UI or by manual kubectl pod queries. To access the Argo CD UI still required the server to have a desktop installed. It could have been avoided by exposing the Argo CD server over the internet but after some testing, it was deemed too cumbersome to set up for the sake of the project.

The way the database was mounted locally to two directories on the node was also used for simplicity. It should be created outside of the cluster for fault tolerance. Using other options than hostPath would also be advisable because it has security issues. Kubernetes does provide cloud

platform-specific options as well as other network volume options.
Exploring separate secret managers could have allowed for more control on the secrets used in the cluster. Managing secrets required that they were created by hand and applied to every Kubernetes namespace one by one from the node command line. Committing the secrets to Git was not an option.

All in all, even with multiple possible improvements, the results were as expected, and the implementation stayed true to the original plan. The subject covered many different technologies and as such offered a great opportunity to gather knowledge of previously unfamiliar topics.

References

Argo CD – Declarative GitOps CD for Kubernetes. n.d. Accessed on 23 April 2021. Retrieved from https://argo-cd.readthedocs.io/en/latest/

Architectural Overview. n.d. Accessed on 23 April 2021. Retrieved from https://argo-cd.readthedocs.io/en/latest/operator-manual/architecture/

What is Kubernetes? n.d. Accessed on 26 April 2021. Retrieved from https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

Dynamic Volume Provisioning. n.d. Accessed on 26 April 2021. Retrieved from https://kubernetes.io/docs/concepts/storage/dynamic-provisioning/

Burns, B. 2018. The History of Kubernetes & the Community Behind It. Accessed on 26 April 2021. Retrieved from https://kubernetes.io/blog/2018/07/20/the-history-of-kubernetes-the-community-behind-it/

Tessel, M. 2018. Welcome Applatix to the Intuit Team. Accessed on 27 April 2021. Retrieved from https://www.intuit.com/blog/technology/welcome-applatix-to-the-intuit-team/

What is DevOps? - amazon web services (AWS). n.d. Accessed on 27 April 2021. Retrieved from https://aws.amazon.com/devops/what-is-devops/

Freeman, E., Forsgren, N. 2019., DevOps1st edition p. Hoboken, New Jersey: For Dummies. Accessed on 27 April 2021. Retrieved from https://janet.finna.fi, Skillsoft Books ITPro

What is GitOps? n.d. Accessed on 19 May 2021. Retrieved from https://about.gitlab.com/topics/gitops/

Importance of DevOps to scaling software development worldwide as of 2020. 2021. Accessed on 19 May 2021. Retrieved from https://www-statista-com.ezproxy.jamk.fi:2443/statistics/1127211/devops-importance-to-scaling-software-development/

Krohn, R. Considerations for your DevOps toolchain. n.d. Accessed on 19 May 2021. Retrieved from https://www.atlassian.com/devops/devops-tools/choose-devops-tools

What is Continuous Integration? – amazon web services. n.d. Accessed on 19 May 2021. Retrieved from https://aws.amazon.com/devops/continuous-integration/

What is continuous delivery? – amazon web services. n.d. Accessed on 20 May 2021. Retrieved from https://aws.amazon.com/devops/continuous-delivery/

Pittet, S. Continuous integration vs. continuous delivery vs. continuous deployment. n.d. Accessed on 25 May 2021. Retrieved from https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

Pittet, S. The different types of software testing. n.d. Accessed on 20 May 2021. Retrieved from https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing

Virmani, M. Understanding DevOps & bridging the gap from continuous integration to continuous delivery. Paper presented at the - Fifth International Conference on the Innovative Computing Technology (INTECH 2015), 78-82. doi:10.1109/INTECH.2015.7173368. Accessed on 24 May 2021. Retrieved from https://ieeexplore-ieee-org.ezproxy.jamk.fi:2443/document/7173368, IEEE Xplore

IEEE standard for DevOps: Building reliable and secure systems including application build, package, and deployment. 2021. doi:10.1109/IEEESTD.2021.9415476. Accessed on 25 May 2021. Retrieved from https://ieeexplore-ieee-org.ezproxy.jamk.fi:2443/document/9415476, IEEE Xplore

Kurek, T. Declarative vs imperative: DevOps done right. 2019 Accessed on 26 May 2021. Retrieved from https://ubuntu.com/blog/declarative-vs-imperative-devops-done-right

What you need to know - Guide to GitOps. n.d. Accessed on 25 May 2021. Retrieved from https://www.weave.works/technologies/gitops/

Beetz, F., Kammer, A., Harrer, S. GitOps. n.d. Accessed on 26 May 2021. Retrieved from https://www.gitops.tech/

Performing a Rolling Update. n.d. Accessed on 27 May 2021. Retrieved from https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/

Kubernetes - Nodes. n.d. Accessed on 27 May 2021. Retrieved from https://kubernetes.io/docs/concepts/architecture/nodes/

Usecase: GitOps. n.d. Accessed on 28 May 2021. Retrieved from https://about.gitlab.com/handbook/marketing/strategic-marketing/usecase-gtm/gitops/

MicroK8s - zero-ops kubernetes for developers, edge and IoT | MicroK8s. n.d. Accessed on 6 June 2021 Retrieved from http://microk8s.io

GitOps working group. n.d. Accessed on 8 June 2021. Retrieved https://github.com/gitops-working-group/gitops-working-group-old

Kubernetes Components. n.d Accessed on 15 June 2021. Retrieved from https://kubernetes.io/docs/concepts/overview/components/

What is etcd? n.d Accessed on 15 June 2021. Retrieved from https://www.redhat.com/en/topics/containers/what-is-etcd

Containers at Google. n.d. Accessed on 15 June 2021. Retrieved from https://cloud.google.com/containers

Customers. n.d. Accessed on 21 June 2021. Retrieved from https://www.csc.fi/en/customers

cPouta. n.d. Accessed on 21 June 2021. Retrieved from https://research.csc.fi/-/cpouta

Working with kubectl. n.d. Accessed on 23 June 2021. Retrieved from https://microk8s.io/docs/working-with-kubectl

Overview – Argo CD. n.d. Accessed on 23 June 2021. Retrieved from https://argoproj.github.io/argo-cd/

Best Practices – Argo CD. n.d. Accessed on 23 June 2021. Retrieved from https://argoproj.github.io/argo-cd/user-guide/best_practices/

Automated Sync Policy – Argo CD. n.d. Accessed on 28 June 2021. Retrieved from https://argoproj.github.io/argo-cd/user-guide/auto_sync/

Using SSH keys with GitLab CI/CD. n.d. Accessed on 29 June 2021. Retrieved from https://docs.gitlab.com/ee/ci/ssh_keys/

Use Kaniko to build Docker images. n.d. Accessed on 29 June 2021. Retrieved from https://docs.gitlab.com/ee/ci/docker/using_kaniko.html

Pods – Kubernetes. n.d. Accessed on 29 June 2021. Retrieved from https://kubernetes.io/docs/concepts/workloads/pods/

What is a Container? – Docker. n.d. Accessed on 29 June 2021. Retrieved from https://www.docker.com/resources/what-container

Deployments – Kubernetes. n.d. Accessed on 29 June 2021. Retrieved from https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

ReplicaSet – Kubernetes. n.d. Accessed on 29 June 2021. Retrieved from https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

Jobs – Kubernetes. n.d. Accessed on 29 June 2021. Retrieved from https://kubernetes.io/docs/concepts/workloads/controllers/job/

Service – Kubernetes. n.d. Accessed on 1 July 2021. Retrieved from https://kubernetes.io/docs/concepts/services-networking/service/

Ingress – Kubernetes. n.d. Accessed on 1 July 2021. Retrieved from https://kubernetes.io/docs/concepts/services-networking/ingress/

Volumes – Kubernetes. n.d. Accessed on 9 July 2021. Retrieved from https://kubernetes.io/docs/concepts/storage/volumes/

Persistent Volumes – Kubernetes. n.d. Accessed on 9 July 2021. Retrieved from https://kubernetes.io/docs/concepts/storage/persistent-volumes/

Secrets – Kubernetes. n.d. Accessed on 9 July 2021. Retrieved from https://kubernetes.io/docs/concepts/configuration/secret/

ConfigMaps – Kubernetes. n.d. Accessed on 11 July 2021. Retrieved from https://kubernetes.io/docs/concepts/configuration/configmap/

Github – Realworld. n.d. Accessed on 14 August 2021. Retrieved from https://github.com/gothinkster/realworld

Mastering Fullstack Development: Learn How to Build Modern Web Apps, E. Simons Accessed on 14 August 2021. Retrieved from https://thinkster.io/tutorials/fullstack

Portela, R., Sándor, Á., Cosbuc, M., Landry Tene, S. (2020). FluxCD, ArgoCD or Jenkins X: Which Is the Right GitOps Tool for You? Accessed on 14 August 2021. Retrieved from https://github.com/gothinkster/realworld

IBM Cloud Education. (2020). DevSecOps. Accessed on 14 August 2021. Retrieved from https://www.ibm.com/cloud/learn/devsecops

Appendices

Appendix 1. gitops/prod/kustomization.yaml

kustomization.yaml 66 Bytes Edit Web IDE Lock Replace Delete

```
---
resources:
- backend
- database
- frontend
- app-ingress.yaml
```

Appendix 2. gitops/prod/backend/kustomization.yaml

kustomization.yaml 290 Bytes Edit Web IDE Lock Replace Delete

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- node-express-configmap.yaml
- node-express-deployment.yaml
- node-express-service.yaml
images:
- name: gitlab.labranet.jamk.fi:4567/conduit-gitops/backend-source
  newTag: 307c70430e3c88c9573f3be6ae6c1b855ee5ff7c
```

Appendix 3. gitops/prod/database/kustomization.yaml

kustomization.yaml 211 Bytes Edit Web IDE Lock Replace Delete

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- mongo-deployment.yaml
- mongo-persistent-volume.yaml
- mongo-pv-claim.yaml
- mongo-service.yaml
images:
- name: mongo
  newTag: "4.4"
```

Appendix 4. gitops/prod/frontend/kustomization.yaml

kustomization.yaml 259 Bytes — Edit | Web IDE | Lock | Replace | Delete

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
- react-redux-deployment.yaml
- react-redux-service.yaml
images:
- name: gitlab.labranet.jamk.fi:4567/conduit-gitops/frontend-source
  newTag: 8302086cd0e4090802feda04f5f3137c32151311
```

Appendix 5. gitops/dev/database/mongo-deployment.yaml

mongo-deployment.yaml 593 Bytes — Edit | Web IDE | Lock | Replace | Delete

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mongo-deployment
spec:
  selector:
    matchLabels:
      component: app
      role: database
  replicas: 1
  template:
    metadata:
      labels:
        component: app
        role: database
    spec:
      volumes:
        - name: mongo-pv-storage-dev
          persistentVolumeClaim:
            claimName: mongo-pv-claim-dev
      containers:
      - name: mongo-database
        image: mongo:4.4
        volumeMounts:
        - mountPath: /data/db
          name: mongo-pv-storage-dev
        ports:
        - containerPort: 27017
```

Appendix 6. gitops/dev/database/mongo-persistent-volume.yaml

mongo-persistent-volume.yaml 289 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: mongo-persistent-volume-dev
  labels:
    type: local
spec:
  #storageClassName: microk8s-hostpath
  storageClassName: manual-dev
  capacity:
    storage: 2Gi
  accessModes:
    - ReadWriteOnce
  hostPath:
    path: "/mnt/data/dev"
```

Appendix 7. gitops/dev/database/mongo-pv-claim.yaml

mongo-pv-claim.yaml 237 Bytes Edit Web IDE Lock Replace Delete

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: mongo-pv-claim-dev
spec:
  #storageClassName: microk8s-hostpath
  storageClassName: manual-dev
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 2Gi
```