

Umeå University  
Computing Science

May 24, 2022

5DV143 - Degree Project in Computing Science Engineering  
Spring 2022, 30p

---

# A Comparison of CI/CD tools on Kubernetes

version 1.0

---

## Student

**Name** William Johansson  
**E-mail** wijo0012@student.umu.se

## Internal Advisor

**Name** Paul Townend  
**E-mail** paul.townend@umu.se

## External Advisor

**Name** Mulugeta Tamiru  
**E-mail** mulugeta.tamiru@elastisys.com

## Abstract

Kubernetes is a fast emerging technological platform for developing and operating modern IT applications. The capacity to deploy new apps and change old ones at a faster rate with less chance of error is one of the key value proposition of the Kubernetes platform. A continuous integration and continuous deployment (CI/CD) pipeline is a crucial component of the technology. Such pipelines compile all updated code and do specific tests and may then automatically deploy the produced code artifacts to a running system.

There is a thriving ecosystem of CI/CD tools. Tools can also be divided into two types: integrated and standalone. Integrated tools will be utilized for both pipeline phases, CI and CD. The standalone tools will be used just for one of the processes, which needs the usage of two independent programs to build up the pipeline. Some tools predate Kubernetes and may be converted to operate on Kubernetes, while others are new and designed specifically for usage with Kubernetes clusters.

CD systems are classified as push-style (artifacts from outside the cluster are pushed into the cluster) or pull-style (CD tool running inside the cluster pulling built artifacts into the cluster). Pull- and push-style pipelines will have an impact on how cluster credentials are managed and if they ever need to leave the cluster.

This thesis investigates the deployment time, fault tolerance, and access security of pipelines. Using a simple microservices application, a testing setup is created to measure the metrics of the pipelines. Drone, Argo Workflows, ArgoCD, and GoCD are the tools compared in this study. These tools are coupled to form various pipelines.

The pipeline using Kubernetes-specific tools, Argo Workflows and ArgoCD, is the fastest, the pipeline with GoCD is somewhat slower, and the Drone pipeline is the slowest. The pipeline that used Argo Workflows and ArgoCD could also withstand failures. The other pipelines that used Drone and GoCD were unable to recover and timed out. Pull pipelines handles the Kubernetes access differently to push pipelines as the Kubernetes cluster credentials does not have to leave the cluster, whereas push pipelines needs the cluster credentials in the external environment where the CD tool is running.

---

## Acknowledgements

Throughout the thesis, communication with both supervisors helped and guided me. Paul Townend, my internal supervisor, first helped me organize the project and properly format my report. My external supervisor, Mulugeta Tamiru, advised me on which areas to focus on and how to evaluate CI/CD pipelines. Both components were essential to complete the report.

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Problem Definition	1
1.2	Aims & Objectives	2
1.3	Risk Register	3
1.4	Time Plan	4
1.5	Outline	5
<b>2</b>	<b>Literature Survey</b>	<b>6</b>
2.1	Cloud	6
2.1.1	Cloud Infrastructure Provisioning	6
2.1.2	Cloud Reliability	6
2.1.3	Cloud Native	7
2.1.4	Microservices	7
2.1.5	Virtual Machines	8
2.1.6	Containers	8
2.2	Container Platforms	9
2.2.1	Container Engine	9
2.2.2	Container Orchestrators	9
2.2.3	Kubernetes	10
2.3	Developing Containerized Applications	11
2.3.1	DevOps	11
2.3.2	DevOps Pipelines	11
2.4	Components of a DevOps Pipeline	12
2.4.1	Continuous Integration	12
2.4.2	Artifact Repository	13
2.4.3	Continuous Delivery/Deployment	13
2.4.4	GitOps	14
2.4.5	Push vs. Pull Style	14
2.4.6	Integrated vs. Standalone	14
2.4.7	Pipeline Security	15
2.4.8	Large Ecosystem Of Tools	16
2.5	Tools	16
2.5.1	Major Tools	17
2.5.2	Drone	18
2.5.3	Argo Workflows	19
2.5.4	ArgoCD	19
2.5.5	GoCD	19
2.6	Selecting The Best Tool	20
2.6.1	Evaluating Solutions	20
2.6.2	Access Security	20
2.6.3	Deployment time	21
2.6.4	Fault Tolerance & Fault Tolerance Time	21

---

2.7	Research Questions . . . . .	22
<b>3</b>	<b>Solution Design</b>	<b>23</b>
3.1	Components . . . . .	23
3.2	Example Use Of The System . . . . .	25
3.3	Evaluation Criteria . . . . .	25
3.3.1	Deployment Time . . . . .	25
3.3.2	Fault Tolerance . . . . .	26
3.3.3	Access Security . . . . .	26
3.4	Criteria For Success . . . . .	26
<b>4</b>	<b>Implementation</b>	<b>27</b>
4.1	Cluster Setup . . . . .	27
4.2	Motivations . . . . .	27
4.3	Code Repositories . . . . .	28
4.4	Application Setup . . . . .	29
4.5	CI/CD Tools Setup . . . . .	29
4.5.1	Drone . . . . .	29
4.5.2	Argo Workflows . . . . .	31
4.5.3	ArgoCD . . . . .	33
4.5.4	GoCD . . . . .	34
4.6	Testing Tool . . . . .	36
4.6.1	Fault Injection . . . . .	36
<b>5</b>	<b>Testing/Results</b>	<b>37</b>
5.1	Experiment Design . . . . .	37
5.1.1	Deployment-time Experiment . . . . .	37
5.1.2	Fault-tolerance Experiment . . . . .	37
5.2	Results . . . . .	38
5.2.1	Deployment Time . . . . .	38
5.2.2	Access Security . . . . .	40
<b>6</b>	<b>Evaluation</b>	<b>41</b>
6.1	Test Results . . . . .	41
6.1.1	Deployment Time Test . . . . .	41
6.1.2	Deployment Time With Faults Test . . . . .	42
6.2	Access Security . . . . .	43
6.3	Aims & Objectives . . . . .	43
<b>7</b>	<b>Conclusion</b>	<b>45</b>
7.1	Technical Issues Encountered . . . . .	45
7.2	Personal Reflections . . . . .	46
7.3	Future Work . . . . .	47

# 1 Introduction

Kubernetes is a rapidly growing technology platform that is used to develop and operate modern IT applications. The core value proposition of the Kubernetes platform is the ability to deploy new applications and modify existing ones at a higher pace with a lower risk of error. As part of that, a continuous integration and continuous deployment (CI/CD) pipeline is an important part of the tool. Such pipelines repeatedly (upon code commits, and/or recurring, i.e., nightly) compile all modified code and perform certain testing (unit, etc.), and can upon completion automatically deploy the generated code artifacts to a running environment.

There is a very rich ecosystem of such CI/CD tools, and one major classification of CD systems is push-style CD and pull-style CD. The push style involves pushing artifacts from the outside into the cluster, whereas the pull style involves a CD tool operating inside the cluster and pulling created artifacts in. Pull- and push-style pipelines will affect how the cluster credentials are handled and whether the cluster credentials ever need to leave the cluster.

Tools can also be classified as integrated or standalone. Integrated tools will be used for both steps of the pipeline, both CI and CD. The standalone tools will only be used for one of the steps, which requires two different tools to set up the pipeline. The tools can also predate Kubernetes and be retrofitted to also run on Kubernetes, other tools are very recent and purpose-built to be used exclusively with Kubernetes clusters.

## 1.1 Problem Definition

Given the wide selection of alternatives available, it can be very difficult for an organization that wants to adopt Kubernetes to decide on which CI/CD tools to adopt.

This thesis will investigate the advantages and disadvantages of some of the most popular tools, as well as different types of CI/CD pipelines in Kubernetes, using a combination of literature studies and experiments.

## 1.2 Aims & Objectives

The aims and objectives for the project are:

- A1** Create an infrastructure to test the different CI / CD tools.
  - O1** Identify and setup example Kubernetes applications that will be used for testing.
  - O2** Set up pipelines with tools that combine CI / CD, separate CI and CD tools, Kubernetes-specific tools and generic tools.
- A2** Establish evaluation criteria and use the testing infrastructure to evaluate CI / CD pipelines.
  - O3** Identify the appropriate evaluation criteria.
  - O4** Characterize each pipeline according to the criteria.
- A3** Evaluate which CI/CD technologies are most suitable for a microservices application on Kubernetes.
  - O5** Investigate Kubernetes-specific tools compared to generic CI/CD tools.
  - O6** Investigate integrated compared to standalone CI/CD tools.

### 1.3 Risk Register

#	Risk	Likelihood (L/M/H)	Impact (L/M/H)	Mitigation
1	Not enough time to compare all the different tools.	M	M	Narrow the scope of the project by focusing on a smaller amount of tools or projects.
2	A local Kubernetes cluster is not sufficient to test the tools.	M	L	A cluster will be set up on a cloud provider such as Google using free-tiers. If this is not enough, Elasticsys will set up accounts.
3	The tools supported by CNCF does not cover all categories (e.g. push, pull, integrated, standalone)	L	L	Look for other open-source tools not officially supported by CNCF.
4	Can not get in touch with external advisor	L	L	Will contact Henrik Enberg, the operations manager of Elasticsys.
5	Can not get in touch with internal advisor	L	L	Will contact Henrik Björklund, the person in charge of the course.
6	Covid or other illness prevents me from doing work.	M	M	Narrow the scope of the project by focusing on a smaller amount of tools or projects.
7	I fall behind schedule.	M	M	If early in the project some of the tools could be removed to narrow the scope. If later in the project reduce the complexity of the experiments, i.e. only perform the experiments with a single application.

Table 1: Risk analysis for the project.

## 1.4 Time Plan

Work will be scheduled from 08 to 17 on weekdays.

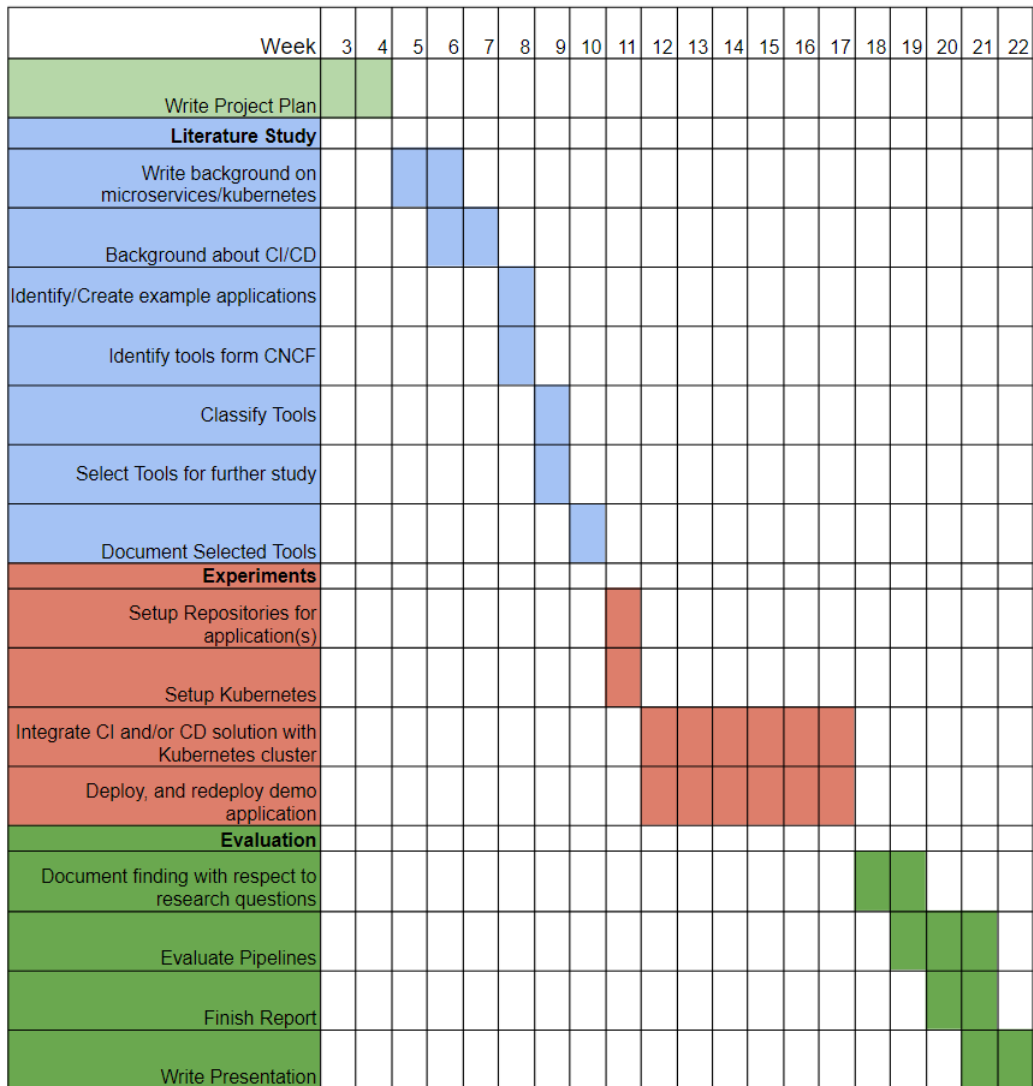


Figure 1: GANTT Chart for the project.

## 1.5 Outline

Chapter 2 of the thesis contains a literature review that explores the background and previous work in the topic. Scientific papers, journals, and other online resources are used to get an understanding of the Cloud, cloud-native, containers, orchestrators, and devops.

The next chapter is Chapter 3, which presents a solution design to the research questions outlined at the end of Chapter 2. The solution represents the architecture, defines the individual components, and includes an example run of the testing tool.

Chapter 4 discusses how the solution design of Chapter 3 is implemented. It explains which tools are utilized, why those tools were chosen, and how the tools are setup. Chapter 5 then describes the tests and findings for each of the tools to be evaluated.

Chapter 6 offers an evaluation of the results from Chapter 5 and a discussion of the outcomes in relation to the thesis's aims and objectives. Chapter 7 concludes the research with some thoughts and challenges observed during implementation, as well as future work to continue the research.

## 2 Literature Survey

### 2.1 Cloud

Simply described, cloud computing is the storage and access of data and applications over the Internet rather than our computer's hard disk. Cloud computing allows users to access data and run apps on any device that can connect to the Internet [30]. The main features that make the cloud interesting to enterprises are that users only have to pay for the resources they use and do not have to buy their own infrastructure. Applications on the cloud can also be flexible, agile and scalable. The flexibility of the cloud enables applications to scale up or down depending on the operations of the application. This could mean increasing or decreasing the available hardware by adding or removing servers or memory [34]. To utilize the cloud, infrastructure such as virtual machines must be provisioned.

#### 2.1.1 Cloud Infrastructure Provisioning

Infrastructure and software development teams are increasingly employing automated technologies to build and manage their cloud infrastructure, which is known as "*infrastructure as code*". These technologies demand the user to specify their servers, networking, and other infrastructure aspects in files that resemble software source code. These files are interpreted by the tools to determine which actions to take [25].

One of the most used tools for infrastructure as code is Terraform. Configuration files will be specified in a human-readable format and allows configuration of both cloud and on-premises resources. These configurations can be versioned, reused, and shared. Standardized procedures can be used to provide and manage all infrastructure throughout its life. Terraform can handle low-level components, such as computing, storage, and networking resources, as well as high-level components, such as DNS records and SaaS services<sup>1</sup>.

After the infrastructure has been provisioned, it can still be subject to different types of faults.

#### 2.1.2 Cloud Reliability

A failure in applications occurs when the given service deviates from the correct service. The cause of application failures can be errors. An error is the component of the system's state that can cause service failures. The cause of the error could be an internal or external fault. There is a very wide variety of faults that could affect software, from the development to the use of services. Examples of fault types are development faults, operational faults,

---

<sup>1</sup>Terraform. *What is Terraform?* URL: <https://www.terraform.io/intro> (visited on 03/07/2022).

hardware faults, and software faults. Fault tolerance is then the ability to avoid service failures in the presence of faults. [2].

One of the more common faults that causes errors for Kubernetes-running applications is restarting and terminating containers, and applications should expect host and container faults to be a probable occurrence and implement tolerance against them [6]. Since faults are unavoidable, the ability of a system to recover quickly from faults is a critical factor in the overall availability of the system. This has previously been done through human interventions, but modern systems, such as cloud native systems, exhibit self-healing properties to automatically heal from these types of faults [26].

### 2.1.3 Cloud Native

Cloud Native is an approach to designing, constructing, and operating workloads that are built to run in the cloud. Some properties that Cloud-native applications have in common include:

- *Cloud-native apps are designed to work on a global scale.* This means that user data and services could be replicated in various data centers around the world.
- *Cloud-native apps must also scale effectively with thousands of concurrent users.* This requires careful attention to synchronization and consistency.
- *Cloud-native apps are also developed with the idea that infrastructure may fail.*

The Cloud Native architectures helps applications take advantage of the cloud ecosystem and the first major design pattern that emerged for cloud native apps was that decomposing apps into very basic components, which we now regard to as microservices, was necessary to achieve scale and stability [15].

### 2.1.4 Microservices

Microservices is an architectural approach where a single application is made up of a number of loosely connected smaller services. These smaller services will be deployable independently and could have individual technology stacks. The microservices architecture has several advantages over the monolithic approach, such as updates to the code base will be easier as not all of the code base needs to be changed, and each microservice can be independently deployed, or different parts of the application can scale independently from others. This will reduce the cost of running applications where only some parts of the application experience high load.

To run microservices securely, isolated and reliably, some form of virtualization is needed [9].

### 2.1.5 Virtual Machines

One of the key enablers of the cloud is virtualization. Virtualization helps to use resources among cloud users and will increase security because different cloud users will be isolated from each other. Virtualization will also allow multiple users to share resources, as the resource can be fragmented into virtual resources. These virtual resources can also be dynamically allocated and de-allocated [21].

One method of virtualization is to use a hypervisor to host Virtual Machines. This will have multiple virtual operating systems used by different users that share the same hardware. The different Virtual Machines will provide isolation from each other and the resources of the underlying server can be split and shared between the virtual operating systems [4].

However, one drawback of using Virtual Machines is the overhead of provisioning or restarting. When a new virtual machine is started, a new OS must be started, which takes some time. This delay is also noticeable when a Virtual Machine must be restarted [17]. Therefore, containers are often used as a lighter alternative.

### 2.1.6 Containers

A container is a standard unit of software that encapsulates everything necessary to run an application, including code, runtime, system tools, system libraries, and settings. All of this will be packaged into an image, which then can be stored in a registry and pulled when needed. This means that each microservice will be isolated from others and that each container environment can be preconfigured with libraries when the image is built. This ensures that an application can run quickly and reliably between different computing environments and will also separate the microservice from the underlying infrastructure [28].

When using containers instead of virtual machines, the different applications can share both operating systems and possibly libraries. This makes containers smaller in size than hypervisor solutions and enables more containers to run on each machine. Container restart times will also be much shorter, since only the container needs to be restarted, not the entire operating system [4].

However, containers also have some drawbacks. The container isolation mechanism is weaker compared to VM solutions. Enterprise security managers cannot always implement the various, and often fine-grained security policies that are required to follow. This somewhat limits the widespread of containerized solutions in industrial applications [35].

Unlike virtual machines, containers cannot run directly using a hypervisor and can instead run on any server that provides a container engine.

## 2.2 Container Platforms

Container platforms are software that runs a container and manages the containers that are running. These platforms can be elementary and simply run containers, but can also be complemented with other tools that manages these running containers [7].

### 2.2.1 Container Engine

A container engine is the software that will accept requests from users, pull the corresponding image and will run the image as a container<sup>2</sup>. There exists several tools to build and run container applications, but Docker is the most used and has become the industry standard. With Docker, container images can be built from the application and will contain all dependencies, runtime, libraries, and settings. These images can then be run as containers using the Docker Engine<sup>3</sup>.

Tools such as Docker can build and run images, but this still leaves the need for tools to manage the running containers.

### 2.2.2 Container Orchestrators

Container orchestrating tools are used to automate large parts of the container lifecycle and environments. These areas include deployment, management, scaling, and networking. These tools enable companies to manage thousands of containers. There exists several different container orchestrators, such as Kubernetes, OpenShift, Docker Swarm and Apache Mesos, but Kubernetes has become the industry standard for cloud-native applications [1].

When using container orchestration tools, the state of a container will generally be described using the YAML or JSON format. This file will instruct the orchestrator on how to manage a container, which could include which image to run, how the network should be established, and where to store the logs.

When a new container is deployed, the container orchestrator will schedule the deployment to a cluster automatically. The orchestrator will find the correct host while considering any restrictions or requirements from the user. When the container has been deployed, it will be managed by the orchestrator according to the specifications in the deployment file [1].

---

<sup>2</sup>RedHat. *A Practical Introduction to Container Terminology*. URL: <https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction> (visited on 02/03/2022).

<sup>3</sup>Docker. *Use containers to Build, Share and Run your applications*. URL: <https://www.docker.com/resources/what-container> (visited on 02/03/2022).

### 2.2.3 Kubernetes

Kubernetes, also known as k8s, was created in 2014 by Google and later entrusted to the Cloud Native Computing Foundation (CNCF). Kubernetes is a complex system for automating the deployment, planning, and scaling of container-based applications. Kubernetes provides Containers as a Service (CaaS), which considerably accelerates the application deployment process. Kubernetes has a wide range of functionality, a large and growing ecosystem of open source tools, and has great portability between different cloud providers [24].

In Kubernetes, *pods* are the smallest deployable computing units that can be created and controlled. A pod consists of one or more containers that share storage and network resources. A single pods contents will be run in a shared context<sup>4</sup>. Pods are usually not scheduled directly and instead will often be part of a *deployment*. When using deployments the desired state should be described and Kubernetes will take care of making changes to the current state to reach the desired state. The deployments can specify the desired number of containers, which Kubernetes will then ensure are up and running<sup>5</sup>.

Since pods in Kubernetes can be created and destroyed dynamically, the IP address of a pod in a deployment can change if the pod is destroyed and recreated. This will be a problem if different pods wants to communicate with one another. To solve this Kubernetes uses *Services*, which defines a specified way to access pods that will be persistent in the cluster. If the pods can be reached from outside the cluster, an ingress could be used. This will receive incoming traffic and could route it to the specified pods. Both the ingress and service will also provide load-balancing between the different pods the receive the incoming requests<sup>6</sup>.

Kubernetes provides the *HorizontalPodAutoscaler* (HPA) to automatically scale a workload depending on the demand. This means that the response to high load will be to deploy more pods to handle the load. If the load then decreases the number of pods will scale back down to save on resources<sup>7</sup>.

If a container fails, Kubernetes will also try to restart the container or replace it with a new one to ensure high availability for deployment<sup>8</sup>.

The access security to a Kubernetes cluster is established in multiple steps. Initially the

---

<sup>4</sup>Kubernetes. *Pods*. URL: <https://kubernetes.io/docs/concepts/workloads/pods/> (visited on 03/02/2022).

<sup>5</sup>Kubernetes. *Deployments*. URL: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/> (visited on 03/02/2022).

<sup>6</sup>Kubernetes. *Service*. URL: <https://kubernetes.io/docs/concepts/services-networking/service/> (visited on 03/02/2022).

<sup>7</sup>Kubernetes. *Horizontal Pod Autoscaling*. URL: <https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/> (visited on 03/02/2022).

<sup>8</sup>Kubernetes. *Production-Grade Container Orchestration*. URL: <https://kubernetes.io/> (visited on 02/04/2022).

traffic to the cluster can be protected using Transport Layer Security (TLS). Once TLS has been established, the request will be authenticated. Client certificates, password, plain tokens, and JSON Web Tokens (used for service accounts) are examples of authentication modules. If the request cannot be authenticated, it will be rejected with status 401, otherwise the request must be authorized. A request must include the requester's username, the requested action, and the object affected by the action. The request is authorized if an existing policy declares that the user has permissions to complete the requested action<sup>9</sup>.

## 2.3 Developing Containerized Applications

The increased complexity, amount of moving parts and independent technology stacks of a microservices architecture will require investments into deployment, monitoring and lifecycle automation to successfully develop and deploy applications. Microservices are good for activities such as DevOps, continuous integration, and continuous deployment (CI/CD), since the architecture consists of small services that can be deployed frequently<sup>10</sup>.

### 2.3.1 DevOps

DevOps is about developing and supplying business processes quickly and easily. It combines development, delivery, and operations. This allows for a lean, seamless connection between these formerly isolated divisions. Integrating DevOps into a project will also require organizational shifts into less divided teams and more frequent feature updates. This will help organizations produce value more quickly and consistently [39].

DevOps can be used in a variety of deliver models, but it must be customized to the environment and product architecture. A high degree of automation is important for high-quality deliveries with short cycle times. Therefore, DevOps automation requires the use of tools, and picking the right tools for the environment or the project is crucial [11]. The DevOps cycle can be divided into different categories, such as source code management, build processes, continuous integration, deployment automation, and monitor and logging [19].

### 2.3.2 DevOps Pipelines

The basic approach to creating a reliable and frequent delivery process is automation of the deployment pipeline. An automated deployment pipeline can be set up in which any software submitted to the repository must be a production-candidate version. The software

---

<sup>9</sup>Kubernetes. *Controlling Access to the Kubernetes API*. URL: <https://kubernetes.io/docs/concepts/security/controlling-access/> (visited on 02/27/2022).

<sup>10</sup>IBM Cloud Education. *Microservices*. 2021-03-30. URL: <https://www.ibm.com/cloud/learn/microservices> (visited on 02/01/2022).

will go through several stages and could then be pushed to production. Each step in a pipeline is evaluated for success before the next step is started.

A pipeline can be triggered in a variety of ways. The most popular ones are to either continuously poll the source code repository for changes, or to be notified of changes through webhooks [16].

Artifacts are pipeline intermediate or end products. These could be container images for testing or for usage in the final production environment [16].

While each organizations DevOps pipeline is unique, it often comprises build, continuous integration, automation testing, validation, and reporting. It may also have one or more manual gates that require human interaction before code may be executed<sup>11</sup>.

## 2.4 Components of a DevOps Pipeline

A devops pipeline consists of multiple steps before the application can be deployed to a running environment. These could include a build stage, followed by testing, merging to a code repository, and then releasing to an environment.

### 2.4.1 Continuous Integration

Continuous integration are a collection of principles that apply to the everyday work-flow of development teams. To use continuous integration, the source code must be kept in a shared source code repository. All developers should constantly integrate their code changes into the current main branch of the repository. This is done so that any conflicts between the various developers are quickly discovered and resolved.

When the code is committed to a repository an automated system will pick up the change and perform a series of commands to ensure that the changes are okay and did not break any part of the system. To work, the build must be automated and should also include detailed tests to ensure that the system works as expected with every change. The test should have high coverage, but should also be fast as to not hinder the productivity of the developers. These types of server could be either self-hosted or hosted by different providers.

The practice of making smaller changes and integrating them to the main branch often will lead to benefits beyond the development process. These smaller changes that can be sent to production more frequently will be easier to debug when something breaks [22].

The CI step produces a deployment artifact, such as a built executable or an image. This

---

<sup>11</sup>Tom Hall. *DevOps Pipeline*. URL: <https://www.atlassian.com/devops/devops-tools/devops-pipeline> (visited on 02/08/2022).

artifact will then be stored in a artifact repository [20].

### 2.4.2 Artifact Repository

An artifact registry, or container registry, is a centralized storage of artifacts used in a project. This allows the developers to easily manage and publish new images that will be accessible for other parts of the system. The registries can be configured for public or private access. These systems can also be hosted by a cloud provider such as Google Artifact Registry<sup>12</sup> or Amazon Elastic Container Registry<sup>13</sup>, or they can be cloud-independent like DockerHub<sup>14</sup>, or they can be manually configured and self-hosted like Harbor<sup>15</sup> [12].

### 2.4.3 Continuous Delivery/Deployment

After the changes to the code have been integrated into the code base, the next step of the release pipeline can start, this is the deployment. When using continuous deployment, the changes that are integrated to the code base can be further tested and then automatically released to staging or production environments. Changes can be deployed within hours of the software change.

During the continuous deployment process, changes to the software should be made in small increments that are easily deployable. The responsibility for software updates is placed on the developers who created them. This also means that the developers must be involved in the entire process of making software changes to releasing the software, this includes testing, staging, providing configurations, and supporting the updates once they have been deployed.

There are several stages to a deployment pipeline, these could include testing, code review, release engineering, and deploying. The software is tested using unit and module tests as it is being developed. The next step would be integration testing, where the new changes are tested with the entire system with the updated changes. These can be setup using virtual environments such as VMs or containers. These tests can also be automated with an artificial load that simulates the users of the system. Next could be performance tests to catch performance errors as early as possible. It is the responsibility of developers to create effective tests for the software they create [31]. The deployment can either be done traditionally by executing a set of commands to deploy the application or declarative by having a repository stating the configurations.

---

<sup>12</sup>Google. *Artifact Registry*. URL: <https://cloud.google.com/artifact-registry> (visited on 03/15/2022).

<sup>13</sup>Amazon. *Elastic Container Registry*. URL: <https://aws.amazon.com/ecr/> (visited on 03/15/2022).

<sup>14</sup>Docker. *DockerHub*. URL: <https://hub.docker.com/> (visited on 03/15/2022).

<sup>15</sup>Harbor. *Trusted cloud native repository for Kubernetes*. URL: <https://goharbor.io/> (visited on 03/15/2022).

#### 2.4.4 GitOps

GitOps is a relatively new technique for implementing continuous deployment in cloud native applications. The primary concept of GitOps has a Git repository containing declarative descriptions of the desired infrastructure, as well as an automated procedure for the environment to match the defined state. By updating the repository, the cluster will automatically be updated [3].

#### 2.4.5 Push vs. Pull Style

The deployment method can be performed either through push or pull strategies. The distinction between them is where the cluster is updated.

Popular CI/CD technologies, such as Jenkins<sup>16</sup> and CircleCI<sup>17</sup>, are typically setup using the push-based deployment method. The application's source code, as well as the Kubernetes YAMLs required to deploy the app, is stored in the application repository. The pipeline is located outside the cluster and is initiated when the application code is updated, and it builds container images before updating the deployment configuration repository with new deployment descriptions.

The pull-based pipeline method employs the same ideas as the push-based alternative, but the deployment step is instead located inside the cluster where the application is being deployed. Some tools that are typically setup using pull-style are Argo<sup>18</sup> and Flux<sup>19</sup>.

Traditional CI/CD tools run images and execute a set of commands, whereas the newer Kubernetes-specific alternatives instead deploy an operator inside the cluster that deploys new versions of the application and monitors the running application in the cluster. The operator can also be scheduled to periodically check the Git repository and update the cluster accordingly [3]. However, most CD tools can be setup using either push or pull style.

#### 2.4.6 Integrated vs. Standalone

Tools used in CI/CD can be specialized for a particular stage of the pipeline (*standalone*), such as integration or deployment, or can combine both phases into a single tool that can be used for the entire pipeline (*integrated*). Many of the popular tools used today are primarily CI solutions that are now being retrofitted to also focus on cloud-based deployments. These types of tool will likely provide the base level of attributes, but might

---

<sup>16</sup>Jenkins. *Build great things at any scale*. URL: <https://www.jenkins.io/> (visited on 03/14/2022).

<sup>17</sup>CircleCI. *Continuous Integration and Delivery*. URL: <https://circleci.com/> (visited on 03/14/2022).

<sup>18</sup>ArgoProj. *ArgoCD*. URL: <https://argo-cd.readthedocs.io/en/stable/> (visited on 02/25/2022).

<sup>19</sup>FluxCD. *Flux - the GitOps family of projects*. URL: <https://fluxcd.io/> (visited on 03/15/2022).

not provide the scalability, extensibility, and community that tools purposely built for the cloud and specific parts of the pipeline have. However, the use of two different tools for the integration and deployment steps could be a hindrance for businesses that attempt to adopt CI/CD practices or transition to new tools that focus on a single phase. When managing multiple tools, there will be greater overhead and complexity. Instead, the developers may end up resorting to a familiar tool that can handle both stages, but not optimally [37, p. 80].

#### 2.4.7 Pipeline Security

The concept of security in cloud environments is a vast field with many different aspects. This could include automating tests to detect noncompliance, continuous monitoring, dynamic security scans, securing containers, securing and minimizing authentication secrets, and securing the deployment pipeline to ensure that correct and secure images are deployed [23].

The main impact of pipeline types (push or pull) on security is the management of authentication secrets and which systems require access to other systems [10].

The CI/CD agent in a typical push pipeline must have write access to the cluster, the artifact registry, and read access to the source code repository. If an attacker gains access to the cluster's credentials they can execute harmful operations, such as removing deployments, deploying pods, or accessing data. When deployed from a push pipeline, the security of the entire cluster depends on the CI / CD system that has access to the cluster [10].

In pull-pipeline, the CI server will not have access to the cluster directly, which reduces the attack surface. Because the cluster is independent of the build agent, no path between the two systems is required, and the cluster is not required to offer any management interface at all. Furthermore, each modification would be logged in the git history. This still enables malicious users to change the cluster in similar ways as to the push pipeline, however, these changes will be much easier to detect as they are logged in Git. The security issues can then be mitigated in different ways [36]:

- *Having manually approved pull-requests.* This will not be fully automatic and requires human action before changes are implemented.
- *Operator detects only specified images that are deployed to cluster.* Write access is not established in the pipeline but by a separate operator from within the cluster that does not interact with users. An attacker could only modify the image that will be deployed, not the deployment structure or databases.

### 2.4.8 Large Ecosystem Of Tools

The marketplace for CI/CD is vast, as seen by the Cloud Native Computing Foundation's landscape overview [13]. Some tools, such as Jenkins, are generic and well established outside of the cloud-native ecosystem. Other tools are newer, purpose-built, and tailored to cloud-native applications. Some of these tools are open-source and hence available to anybody, whilst others tools may be hosted by a company requiring paid memberships or fees. These might also provide additional services, such as support or server hosting. Some tools will incorporate CI/CD into a single tool, whilst others will focus on a single phase of the CI/CD pipeline.

## 2.5 Tools

The Cloud Native Computing Foundation provides a comprehensive list over available integration and deployment tools. CNCF also provides a maturity level for the available projects. A high maturity level signals that they have adoption, a healthy rate of change, and committers from multiple organizations. This study will only consider tools that are open-source, and therefore free to setup and use. This study will also choose tools that fully implement CI, CD, or both steps. Tools that supplements existing tools or simplify small areas of CI or CD will be excluded.

There are several aspects to consider when selecting open source projects. The popularity of the project can be determined by the number of stars on Github. This measurement is readily available for all open source tools, but might not accurately represent the actual adoption rate of the tool in industry. The project documentation and community are important for new adopters of the project to get up and running. The project release history provides information on the activity of the project and how often it will receive updates [33].

The Open Source Security Foundation defines a set of open source software best practices. Projects can voluntarily apply for a badge to demonstrate that they adhere to these practices. The practices include documentation, change-control, reporting, quality, security, and analysis. Projects that adhere to these practices are more likely to produce higher-quality secure software. However, not all projects that meet the criteria for a badge may have applied [14].

This study will choose to study tools based on:

- **Popularity.** Measured by the amount of GitHub stars.
- **Maturity & Activity.** Measured by CNCF project status and time since the last commit.
- **Open-Source community health metrics.** If a project follows the best practices this will be a positive, however, since not all projects may have applied for a badge

Name	Style	K8s specific	Github Stars	OSSF Practices	CNCF Status	Commit 23/2 - 2022
Drone	CI/CD	No	24419	No	Non-member	15/2 - 2022
Jenkins	CI/CD	No	18500	No	Non-member	22/2 - 2022
ArgoWorkflows	CI	Yes	10500	Yes	Incubating	23/2 - 2022
ArgoCD	CD	Yes	8500	Yes	Incubating	23/2 - 2022
Travis CI	CI/CD	No	8300	No	Non-member	23/2 - 2022
Spinnaker	CD	Yes	8276	No	Non-member	9/12 - 2021
Akuity	CI/CD	Yes	8100	Yes	Member	23/2 - 2022
Tekton	CI/CD	Yes	6817	No (19%)	Non-member	23/2 - 2022
GoCD	CD	No	6339	No	Non-member	19/2 - 2022
Concourse	CI/CD	No	6121	No	Member	23/2 - 2022
JenkinsX	CI/CD	Yes	4071	No	Non-member	21/2 - 2022
werf	CD	Yes	2900	No (72%)	Member	22/2 - 2022
Flux	CD	Yes	2700	Yes	Incubating	23/2 - 2022
Agola	CI/CD	No	939	No	Non-member	22/2 - 2022
Screwdriver	CI/CD	No	920	Yes	Non-member	8/2 - 2022
Woodpecker CI	CI/CD	No	668	No (25%)	Non-member	23/2 - 2022
Buildkite	CI/CD	No	607	No	Non-member	15/2 - 2022
PipeCD	CD	Yes	542	No	Non-member	22/2 - 2022
Razee	CD	Yes	397	No	Member	2/2 - 2022

Table 2: Major CI/CD tools.

the once that do not might still have good practices [14].

The selection will be based on the metrics while still choosing one tool by combining the different categories discussed in Section 2.4: integrated, standalone, generic and Kubernetes-specific. Due to the project’s time limits, only four tools will be chosen.

### 2.5.1 Major Tools

The major open-source tools gathered from the CNCF landscape can be seen in Table 2

- **Drone**<sup>20</sup> will be the *integrated, non Kubernetes specific CI/CD* tool of choice. It is not a CNCF member and has not applied for an OSSF badge, but it does have active committers and is the most popular tool according to Github stars.
- **ArgoWorkflows**<sup>21</sup> was chosen as the *standalone, Kubernetes specific CI* tool. It is the most widely used standalone CI tool and is both an incubating CNCF project and an OSSF badge holder.

<sup>20</sup>Drone. *Automate Software Build and Testing*. URL: <https://www.drone.io/> (visited on 03/14/2022).

<sup>21</sup>ArgoProj. *Argo Workflows*. URL: <https://argoproj.github.io/argo-workflows/> (visited on 02/25/2022).

- **ArgoCD**<sup>22</sup> will be the *standalone, Kubernetes specific CD* tool of choice. It is the most popular standalone pull tool and is both an incubating CNCF project and a CNCF badge holder. A close competitor would be Spinnaker, which holds almost the same number of stars, but has not applied for OSSF practices and is not a CNCF member project.
- **GoCD**<sup>23</sup> will be the *standalone, non Kubernetes specific CD* tool of choice. It is not a CNCF member project, and it has not applied for an OSSF badge, yet it is the most widely used tool for push continuous delivery.

### 2.5.2 Drone

Drone<sup>24</sup> is a continuous integration and deployment platform that enables teams to automate their build, test and release workflows using a cloud-native pipeline engine<sup>25</sup>. Drone uses servers and runners. Servers connect to the repository and accept jobs, and a runner is a standalone daemon that polls the server for pipelines pending to execute<sup>26</sup>.

Drone can connect with various Source Control Management systems, such as Github, Gitlab, or Bitbucket. The execution of the drone pipeline can be triggered by a push of code to the repository, a pull request being opened, or a tag being created. The source control management system automatically sends a webhook to Drone, which in turn triggers the execution of the pipeline<sup>27</sup>.

Pipelines are defined through a configuration file called *.drone.yml*. This file can define multiple integration or delivery pipelines and should be stored in the repository root<sup>28</sup>.

Drone pipelines can be run in different ways, such as Docker Containers, as containers in a pod in a Kubernetes cluster, via ssh on remote machines, or directly on the host machine without containers<sup>29</sup>.

---

<sup>22</sup>ArgoProj. *ArgoCD*. URL: <https://argo-cd.readthedocs.io/en/stable/> (visited on 02/25/2022).

<sup>23</sup>GoCD. *Open Source CI/CD Server*. URL: <https://www.gocd.org/> (visited on 03/14/2022).

<sup>24</sup>Drone. *Automate Software Build and Testing*. URL: <https://www.drone.io/> (visited on 03/14/2022).

<sup>25</sup>Drone. *Drone Documentation*. URL: <https://docs.drone.io/> (visited on 02/25/2022).

<sup>26</sup>Drone. *Drone Server*. URL: <https://docs.drone.io/server/overview/> (visited on 02/25/2022).

<sup>27</sup>Drone. *Drone Server*. URL: <https://docs.drone.io/server/overview/> (visited on 02/25/2022).

<sup>28</sup>Drone. *Drone Configuration*. URL: <https://docs.drone.io/pipeline/configuration/> (visited on 02/25/2022).

<sup>29</sup>Drone. *Pipelines Overview*. URL: <https://docs.drone.io/pipeline/overview/> (visited on 02/25/2022).

### 2.5.3 Argo Workflows

Argo Workflows<sup>30</sup> is an open source container-native workflow engine for orchestrating parallel jobs on Kubernetes. It is implemented as a Kubernetes CRD (Custom Resource Definition). Argo was designed from the ground up for containers, with none of the overhead and constraints associated with older VM and server-based setups<sup>31</sup>.

Using Argo Workflows, each step in a pipeline will be executed in a container, and integration jobs can be run natively on Kubernetes. The workflows in Argo consists of one or more templates. The templates are a collection of steps or a Directed Acyclic Graph (DAG). The DAG can be used to model the dependencies of different steps of a pipeline<sup>32</sup>.

### 2.5.4 ArgoCD

ArgoCD is a declarative, GitOps continuous delivery tool for Kubernetes. It follows the GitOps paradigm of specifying the intended application state using Git repositories as the source of truth. Kubernetes manifests can be defined in a variety of methods, including the following: Kustomize applications, helm charts, or plain yaml/json<sup>33</sup>.

Argo CD is built as a Kubernetes controller that constantly monitors running apps and compares the current live state to the desired goal state (as specified in the Git repository). OutOfSync refers to a deployed program whose live state deviates from the goal state. Argo CD reports and visualizes the differences, while also allowing you to sync the live state back to the intended target state automatically or manually. Any changes made to the intended target state in the Git repository may be applied and reflected in the chosen target environments automatically<sup>34</sup>.

### 2.5.5 GoCD

GoCD is a continuous delivery tool. It consists of a GoCD Server and one or multiple GoCD Agents. The server controls the workflow and the agents do the actual work of executing the pipelines<sup>35</sup>.

GoCD builds workflows using different modeling constructs, such as Tasks, Jobs, and

---

<sup>30</sup>ArgoProj. *Argo Workflows*. URL: <https://argoproj.github.io/argo-workflows/> (visited on 02/25/2022).

<sup>31</sup>ArgoProj. *Argo Workflows*. URL: <https://argoproj.github.io/argo-workflows/> (visited on 02/25/2022).

<sup>32</sup>ArgoProj. *Core Concepts*. URL: <https://argoproj.github.io/argo-workflows/core-concepts/> (visited on 03/02/2022).

<sup>33</sup>ArgoProj. *ArgoCD*. URL: <https://argo-cd.readthedocs.io/en/stable/> (visited on 02/25/2022).

<sup>34</sup>ArgoProj. *ArgoCD*. URL: <https://argo-cd.readthedocs.io/en/stable/> (visited on 02/25/2022).

<sup>35</sup>GoCD. *Introduction To GoCD*. URL: <https://www.gocd.org/getting-started/part-1/> (visited on 02/28/2022).

Stages. Tasks is an action to be performed, usually a single command. A job is a collection of multiple tasks that should be run in order. A stage consists of multiple jobs that can be run independently of the others. Using these blocks complex workflows can be modelled and GoCD will parallelize the execution of jobs in a stage<sup>36</sup>.

The triggers of a GoCD pipeline are called materials. Typically, these triggers will be a source control repository, and any new commits will be a trigger for the pipeline to run<sup>37</sup>.

## 2.6 Selecting The Best Tool

As can be seen from CNCF, there is an overabundance of CI and CD tools to choose from. These tools have various characteristics, such as push or pull style, integrated or standalone, and so on. Given all of these options, it can be difficult for firms seeking to adopt DevOps and CI/CD to select appropriate solutions for their project.

Developing an CI/CD pipeline to deploy software in an effective, timely, and sustainable manner is not an easy undertaking. The benefits of a CI / CD pipeline are many, but implementing continuous integration and continuous deployment takes time [29, 38].

### 2.6.1 Evaluating Solutions

There is no globally accepted method for evaluating CI / CD pipelines, as the optimal pipeline can be subjective and varies between projects and environments. Other studies have compared different aspects of the pipeline, such as how the pipeline is triggered, how the configurations are managed, and how the artifacts are stored [16]. This study will instead focus on deployment times, the security of the pipeline (Section 2.4.7), and how the pipeline achieves fault tolerance in the event of failures of pods. The following are the evaluation criteria that will be used in the project.

### 2.6.2 Access Security

As discussed in Section 2.4.7 the security of a CI/CD pipeline is a vast field and the main security difference when choosing different styles of CI/CD is the authentication and amount of access. Therefore, the aspect of security in focus for this study will be the amount of access a tool has and to which systems a tool has access. A tool should strive for the least amount of access possible as this will decrease the attack surface if secrets or tools are compromised.

---

<sup>36</sup>GoCD. *Concepts in GoCD*. URL: [https://docs.gocd.org/current/introduction/concepts\\_in\\_go.html](https://docs.gocd.org/current/introduction/concepts_in_go.html) (visited on 02/28/2022).

<sup>37</sup>GoCD. *Introduction To GoCD*. URL: <https://www.gocd.org/getting-started/part-1/> (visited on 02/28/2022).

Investigating how tools are configured and creating a diagram detailing the various accesses required by tools is one method of determining which systems have access to the cluster and the artifact repository [16].

### 2.6.3 Deployment time

As one of the core values of DevOps and using pipelines is to have short iteration and to often integrate new software to a repository (Section 2.4.1), it is important that the time it takes to integrate the new software is short as to not hinder the productivity of developers and the new software can then quickly reach the users.

The time it takes to deliver a new feature to production once its implementation is complete is termed deployment time. According to one study, this measure is divided into two dimensions. The first is the execution time of the tools required for the deployment, and the second is any decision-making process for deploying the features, such as acceptance testing.

Deployment time is the time it takes to deploy a new feature to production when its implementation has been completed. One study divides this metric into two dimensions. One is the execution time of the tools needed for the actual deployment, and the other is the decision process to deploy the features if additional product management activities, for example acceptance testing, are associated with the deployment.[18]

Another study simply measured the deployment time as the total time it took for the CI server to build, test and deploy the new images to the server [32].

### 2.6.4 Fault Tolerance & Fault Tolerance Time

As discussed in section 2.1.2 faults are unavoidable in cloud environments and common faults in Kubernetes systems include the restarts of pods and nodes.

Chaos engineering is the process of inserting faults into a production system to detect and remedy unforeseen weaknesses before they cause difficulties, such as downtime or outages. It focuses on systematic and well-defined experiments designed to depict the behavior of the systems during turbulent periods. Chaos Engineering tools such as Chaos-Mesh are excellent for injecting various faults into the system. These faults could be causing pods or nodes to die or restart, simulating network delays, or partitions. However, Chaos-Mesh does not provide the means for assessing the impact of a fault. This task is usually left to other third-party tools [27, 8].

A chaos experiment could be implemented by defining the state or outcome and then perform the chaos experiment and verify or falsify whether the wanted state could be reached or if a faulty state has been reached [5].

## 2.7 Research Questions

As can be seen in Subsection 2.2.2 Kubernetes is the most popular and growing platform for modern applications. As these types of applications are often made up of microservices (Section 2.1.4) a continuous integration and continuous deployment (CI/CD) pipeline is an important tool. From Subsection 2.3.1 it is evident that a pipeline is important to quickly and securely release new versions of the software.

Section 2.4 shows that there is a very rich ecosystem of such CI/CD tools which can be classified as push, pull, integrated and / or standalone. As CI/CD pipelines have grown in popularity and use by large industries, pipeline security (Subsection 2.4.7) has also become increasingly crucial and different types of pipelines will have different security measures to take.

Given the numerous options and styles of CI/CD available, companies interested in integrating CI/CD with Kubernetes will find it challenging to choose which technologies and styles to utilize. From Section 2.6 some key evaluation criteria when comparing different CI/CD tools are security, fault tolerance, and the time necessary to deploy an application.

Thus, the following research questions should be answered:

- What is the best method to evaluate each pipeline for a given evaluation criteria?
- Which type of pipeline is suitable for a microservices application on Kubernetes?

### 3 Solution Design

The solution architecture solves the problems outlined in Section 2.7 by developing a system to evaluate the different pipelines against the evaluation criteria. The testing tool will commit a change to the source code, the CI/CD pipeline will pickup the change and push a new version of the application to the cluster. We use the system in Figure 2 as our system model and assume that all Kubernetes environments will be similar to this. Anything outside of this design will be beyond the scope of this study.

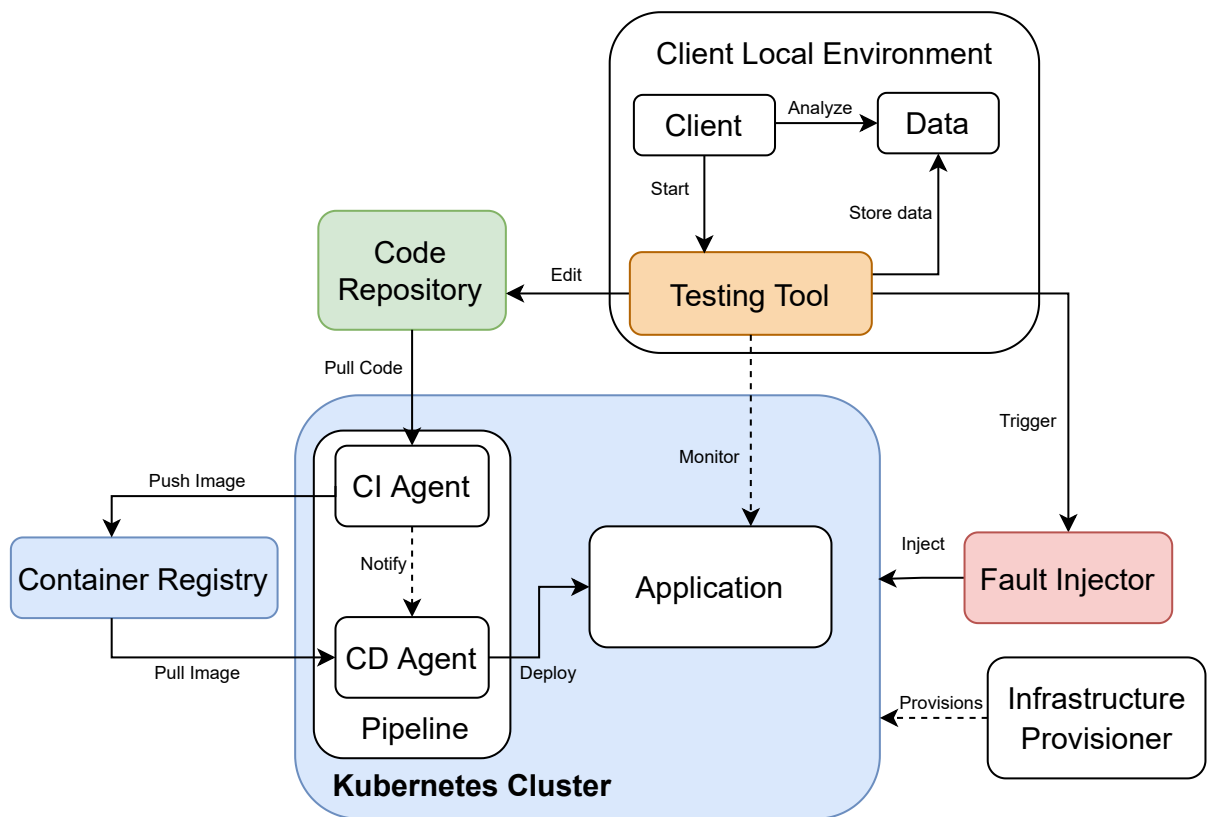


Figure 2: Solution Architecture.

#### 3.1 Components

- **Code Repository**

The code repository is a service for storing both the code and configurations for the sample application that will be deployed on the cluster. When the code repository is updated with new commits, the CI-agent will be notified and pull the latest changes.

- **Container Registry**

The container registry is a service that stores the image files of the application. The

images created in the CI-stage are pushed to the registry and can later be pulled and deployed in the CD stage. The container registry, as described in Section 2.4.2, can be self-hosted or hosted by cloud providers, and the specific container registry used in this study should not have an impact on the test results.

- **Fault Injector**

The fault injector is a tool that injects faults into the Kubernetes cluster. The types of faults that are relevant are discussed in Section 2.6.4, and any other faults will be outside the scope of this study.

- **Testing Tool**

The testing tool is a tool that will be built to run tests to measure the deployment time and fault tolerance of CI/CD pipelines. The tool is responsible for starting the tests and collecting the results. It will monitor the application for changes, commit changes to the code repository, and activate the fault injector for tests that require them.

- **Pipelines**

The pipeline consists of an CI and CD agent. These will be installed inside the Kubernetes cluster. The pipelines' CI-agents are configured in comparable ways for each type of pipeline. It will be deployed in the cluster and linked to the code repository. When changes are made to the repository, the CI-agent is alerted and the test and build phases are started. Once these processes have been performed, the created images are pushed to the container registry, where the deployment agent can obtain them.

If the tools are *push and integrated*, the CI and CD agent could be in the same tool, and the deployment can be a natural continuation of the CI process. If the tools are standalone, the CI tool might make a request to the CD agent to alert the CD tool of the latest build.

When utilizing the *pull-style*, the CD-agent will instead query another repository where the appropriate cluster configuration is defined. When modifications are made to this repository or new images are pushed, the CD-agent detects it and immediately updates the cluster to the required state.

- **Application**

The application is a simple cloud-native application consisting of micro-services. It has a version number that is reachable and can signal when a new version of the software is deployed.

- **Infrastructure Provisioner**

The infrastructure provisioner is a tool that is used to automatically provision the infrastructure and deploy tools and applications. This will increase the repeatability of the tests as all the infrastructure can be setup using simple commands. This is

commonly done through code utilizing *infrastructure as code* tools (Section 2.1.1).

### 3.2 Example Use Of The System

The following will show an example run of the testing tool to measure the deployment time and / or the fault tolerance time. Before testing starts, the correct CI/CD tools should be deployed in the cluster.

1. The test starts with the client running the *testing-tool*. The client will specify if faults should be injected or not during the test-run.
2. The first step is to change a version number in the *code repository*. The change will be committed and pushed to the code repository.
3. A timer is started and the *application* will be polled regularly until the version change is visible in the application.
4. If faults should be injected, the *testing-tool* will after a specified amount of time send a request to the *fault injector* to introduce the faults.
5. The total time required to update the application will be the result of the tests.

### 3.3 Evaluation Criteria

The evaluation criteria considered for the study of CI / CD tools are described in Section 2.6. The specific criteria that will be used in this study are deployment time, fault tolerance, and security. The testing tool will be used to collect data for the deployment time and fault tolerance criteria, while static analysis of the system will be used to collect data for the security criteria.

#### 3.3.1 Deployment Time

It is clear from Section 2.6.3 that the deployment time of a pipeline is an essential factor to evaluate. As a result, the deployment time of the pipeline will be measured in this research by initiating a minor change in the application, committing the change to the git repository, and measuring the time until the change is visible in the cluster.

Other activities involved in the deployment process, such as acceptance testing, are considered in some studies. However, because this study is focused on CI/CD technologies, they will not be included in the tests.

### 3.3.2 Fault Tolerance

Some typical faults for Kubernetes-running applications are the restart or killing of pods, as mentioned in Section 2.6.4. These faults are common in Kubernetes clusters and other types of faults will be out of scope for this study due to time constraints. This research will replicate these types of faults by deploying a new version of the application to the cluster and then killing a pod utilized in the CI/CD pipeline after a specific time to see if the tool can recover from the fault and, if so, how long the final deployment time is.

### 3.3.3 Access Security

As detailed in Section 2.6.2, the security of CI/CD pipelines is concerned with the level of access of tools to the cluster and the image repository. As a result, this investigation will investigate the systems to which each tool has access by illustrating the connections in a diagram. Other types of security, such as pod or network security will be out of scope for this study due to time constraints.

## 3.4 Criteria For Success

The experiments are considered successful if:

- The testing tool can measure the deployment time for the application.
- A fault that kills a pod can be injected into the CI/CD tools during testing.
- A diagram can be created displaying overviews of a push and a pull pipeline.

## 4 Implementation

This chapter shows how the solution design will be implemented in order to fulfill the criteria for success described at the end of chapter 3. Figure 3 depicts an overview of the implementation that will be utilized in this study. This is an implementation of the solution architecture of Figure 2.

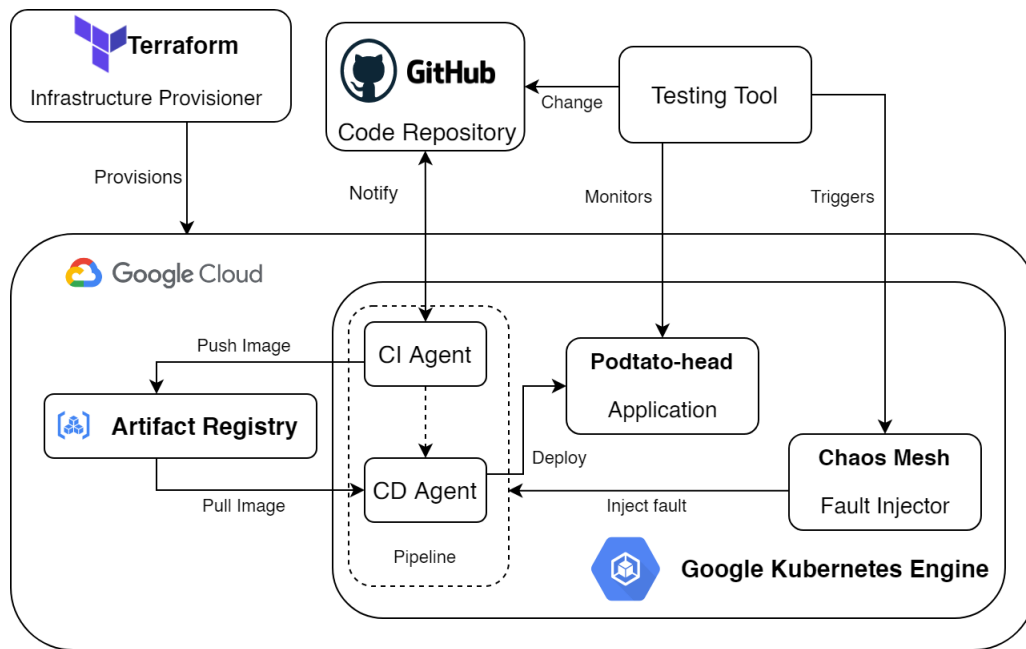


Figure 3: Implementation overview.

### 4.1 Cluster Setup

Most major cloud providers, for example, Google, Microsoft, and Amazon, provide containers as a service, with Kubernetes as the orchestrator. The experiments are not dependent on a cloud provider and can be reproducible on most installations of Kubernetes.

This project will use the Google Cloud Platform (GCP) and the Google Kubernetes Engine (GKE) for its popularity and familiarity. The cluster will be setup with three nodes as there are no big workloads that will be running. This can be increased if some tool, for some reason, requires more computing power.

### 4.2 Motivations

- **Code Repository**: The chosen code repository provider was Github, as this is the most popular alternative and it can integrate with all the different CI/CD tools.

Other version control systems, such as GitLab, would also have been sufficient.

- **Infrastructure Provisioner:** Terraform was chosen as an infrastructure provisioner as it can easily integrate with google-cloud and the Kubernetes engine. This makes it easier to deploy the infrastructure directly through CLI commands.
- **Fault Injector:** As discussed in Section 2.1.2, Chaos Mesh is a tool for injecting faults and will be installed in the cluster to be used to test fault tolerance. Chaos Mesh was chosen as it enables faults to be injected using standard YAML configuration files and can easily be automated when testing.
- **Image Registry:** The exact image registry used in this study should have no effect on the results of the tests. Therefore, as this project is deployed on the Google Cloud Platform it will use Google Artifact Registry to host the built images for the microservices.
- **Example Application:** The Application Delivery Technical Advisory Group<sup>38</sup> is a CNCF group that works on the delivery of cloud native applications which includes stages such as creating, deploying, managing, and running. As part of this work, a demo application for showcasing various sorts of delivery scenarios was created. The delivery scenarios could include a variety of technologies and services. This application, *podtato-head*<sup>39</sup>, is a typical cloud-native application made up of a set of microservices with basic functionality.  
As this application is a simple, cloud-native application created by CNCF to demonstrate different delivery scenarios, it will be used in this study to compare the different integration and delivery tools. The repository for the podtato-head application contains different delivery scenarios and a single server and microservice version, and this study will use the microservice version and remove other parts of the application that will not be needed for the study.

### 4.3 Code Repositories

The study will use two separate repositories.

- The *code repository* will contain the source code and the configuration of the CI and CD tools. This repository will be used by the CI agent when building the application images.
- The *configuration repository* will contain the configuration of the application. This is in the form of a helm chart and specifies which images to use, how they are connected, and how they are exposed. This repository will be used by the CD agent to deploy

---

<sup>38</sup>CNCF. *TAG App Delivery*. URL: <https://github.com/cncf/tag-app-delivery> (visited on 03/15/2022).

<sup>39</sup>CNCF. *Project pod tato Head*. URL: <https://github.com/podtato-head/podtato-head> (visited on 03/15/2022).

the application to the cluster.

When the CI-agent has built the images with a new version, this version will be updated in the configuration repository, and the CD-agent can then update the cluster. This is done to enable the CI and CD agent to monitor different repositories and thereby run at different times. For example, when a new change is made to the application, only the CI agent should run. If the CI-agent successfully integrates the changes, it will update the configuration repository and then the CD-agent can update the cluster with the new application version.

#### 4.4 Application Setup

The Makefile at the root of the project contains instructions to build an image of the service and push the image to a container registry. Another command will also be added to test the service. As the contents of the tests are not relevant to the thesis they will be empty and simply be a stage in the pipeline.

The application is deployed to Kubernetes using a helm-chart that contains the configuration of the application, including the versions of the images to use. By changing these versions, the application can be updated with new images.

#### 4.5 CI/CD Tools Setup

The various CI/CD tools have somewhat different configurations, and this section will highlight the key aspects for configuring the pipeline.

##### 4.5.1 Drone

The Drone tool handles the CI and CD stage and consists of a runner and a server. These may be installed in a variety of methods, but the preferred method when using Kubernetes is to use helm charts to install and configure them. The helm charts are applied using Terraform, which is connected to the cluster.

After the Drone server and runner have been installed through the helm-charts an ingress is created to reach the Drone server. The ingress will expose the Drone GUI, but also enable webhooks to reach the Drone server.

Drone will be notified of new commits in the Github code repository through webhooks. These webhooks are set up by creating a Github OAuth application<sup>40</sup> and then connecting

---

<sup>40</sup>Github. *Creating an OAuth App*. URL: <https://docs.github.com/en/developers/apps/building-oauth-apps/creating-an-oauth-app> (visited on 03/28/2022).

them to the Drone server using the Drone GUI, which can be accessed from the ingress to the Drone server. The connection is established by logging in to the Github account and choosing the project on the user that should be activated.

The drone integration and delivery steps are then configured in a file typically named *.drone.yaml*. This configuration file will include the steps and commands to perform when building, testing, and deploying the application.

When building Docker images from within a Docker container, the Docker commands need to access the Docker daemon running the container. To achieve this, a service which connects to the outside docker daemon through a Kubernetes volume is used. This service will use the `docker/dind` (Docker IN Docker) image. Other steps in the Drone pipeline can then use this volume to run docker commands. An example of this service can be seen in Listing 1. The name is specified as `docker`, the image to use is specified, and a volume is created to connect to the docker daemon running outside the container.

Listing 1: Docker In Docker

```
services :
  - name: docker
    image: docker:dind
    privileged: true
    volumes :
      - name: dockersock
        path: /var/run

volumes :
  - name: dockersock
    temp: {}
```

An example of the configuration of a step in the Drone pipeline can be seen in Listing 2. First, the name of the step is specified and then the image to run the commands, the `cloud-sdk` image is chosen to be able to authenticate to the Google Cloud using the `gcloud` tool. Then the volume of the docker daemon is specified to be able to run the docker commands. The command section then specifies the commands to run in this step, which will authenticate to the Artifact registry and then build and push the microservice.

Listing 2: Drone Pipeline Step

```
- name: build-entry
  image: google/cloud-sdk
  volumes:
    - name: dockersock
      path: /var/run
  commands:
    - sleep 5
    - gcloud auth configure-docker europe-west1-docker.pkg.dev
    - cd podtato-head
    - chmod +x ./podtato-head-microservices/build/build-entry.sh
    - make push-entry
```

#### 4.5.2 Argo Workflows

Argo Workflows is setup to handle the continuous integration for the application. The Argo Workflows tool is installed through a helm-chart that automatically installs the server, that handles incoming requests, and the workflow-controller which orchestrates the workflows.

The integration job is defined as a Kubernetes Custom Resource called *WorkflowTemplate*. This defines the containers to use, commands to execute, and in what order. This template is submitted to the Argo server along with a *WorkflowEventBinding*, which binds the template to an external event. The event to watch for is a http request, in this way, the workflow server can be notified of changes in the Github repository through webhooks fired when a new commit enters the repository.

To authenticate the webhooks from the Github repository, a shared secret is created and will be sent with the webhooks from the repository, and a custom service account with the same secret will be created in the cluster and used to validate the incoming webhooks. The argo server also needs to be set with Kubernetes roles to access service accounts to be able to see this new service account.

Similarly to Drone, the Argo Workflow pods also need to run docker commands from within a docker container. Argo Workflows handles this by defining a sidecar that runs the docker/dind image and through volumes exposes the outside docker daemon. An example of the sidecar configuration can be seen in Listing 3. The image tag specifies the image to use, the command specifies which initial command to run and the securityContext sets the container as privileged to be able to mount the docker volume.

Listing 3: Argo Workflows Sidecar

```

sidecars:
  - name: dind
    image: docker:19.03.13-dind
    command: [dockerd-entrypoint.sh]
    env:
      - name: DOCKER_TLS_CERTDIR
        value: ""
    securityContext:
      privileged: true
    mirrorVolumeMounts: true

```

The configuration for the build step can be seen in Listing 4. This template takes an input parameter that specifies the part to build and then issues the commands to build the microservice. The env section defines the docker host to use the outside docker daemon, and the volumeMounts is used to get the git repository containing the source code.

Listing 4: Argo Workflows Build

```

- name: build-part
  inputs:
    parameters:
      - name: part
  container:
    image: google/cloud-sdk
    command: ["/bin/sh", "-c"]
    args: ["
      cd /workdir/src/podtato-head/;
      until docker ps; do sleep 3; done;
      gcloud auth configure-docker europe-west1-docker.pkg.dev;
      chmod +x ./podtato-head-microservices/build/build-part.sh;
      make push-{{inputs.parameters.part}};
    "]
    env:
      - name: DOCKER_HOST
        value: 127.0.0.1
    volumeMounts:
      - name: workdir
        mountPath: /workdir

```

As can be seen in Listing 4 the git repository must be manually passed between the different stages of the pipeline using volumes.

The last step of the pipeline is to fetch the configuration repository, update the version of the image to use in the application, and commit the change to git. This will enable the CD-tool to notice the change and update the application.

### 4.5.3 ArgoCD

Argo CD will be used to deploy the application to the cluster after the CI stage has built the required images. The tool is deployed to the cluster using a helm-chart that installs the ArgoCD server. The ArgoCD GUI is exposed through a Kubernetes ingress to be able to visualize the applications and their statuses.

The application to be deployed is configured using a custom resource called "Application". It specifies the git repository to watch, the path to the helm-chart that deploys the application, and the destination where the application should be deployed (in this case, the current cluster). It also defines the policies that Argo will use for the application, such as removing old containers and automatically healing the application in case of drifts. The configuration file can be seen in Listing 5.

The source section defines the git repository and the folder to monitor for changes. The destination section defines in which cluster and namespace to deploy the application, since this will be deployed in the same cluster as ArgoCD is running, the server will be *https://kubernetes.default.svc*. SyncPolicy has options to specify whether the ArgoCD agent should monitor the running application and correct possible deviations and prune containers that are no longer needed.

Listing 5: ArgoCD Configuration

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: podtato-head
  namespace: argocd
spec:
  project: default

  source:
    repoURL: https://github.com/wluhl/k8s-ci-cd.git
    targetRevision: HEAD
    path: podtato-head/chart
  destination:
    server: https://kubernetes.default.svc
    namespace: default

  syncPolicy:
    automated:
      selfHeal: true
      prune: true
```

When deploying the application, ArgoCD will automatically watch for any changes in the specified git repository and ensure that the environment matches the configuration of the repository.

#### 4.5.4 GoCD

The GoCD tool is configured to manage the deployment of built images. The GoCD server is installed via a helm chart using Terraform. This will install the GoCD server that handles incoming requests and the GoCD agent that manages the agents used in the pipelines.

The first step in setting up GoCD is to configure the agent that will run the workloads. GoCD agents cannot use any image and must use one of the images provided by GoCD. These images include the ability to connect to the GoCD server when new pods are created. This study used *gocd-agent-ubuntu-20.04* as a base image and extended it, by creating a new Docker image, with the tools needed for the deployment. These tools included gcloud, kubectl, and helm. The agent can then be configured to use this image through the GoCD UI.

The next step is to configure the pipeline that runs the workloads. This is achieved by a configuration file, as can be seen in Listing 6.

Listing 6: GoCD Configuration

```

format_version: 3
pipelines:
  deploy_app:
    group: sample
    label_template: ${COUNT}
    lock_behavior: unlockWhenFinished
    display_order: 1
    environment_variables:
    materials:
      git:
        git: https://github.com/wluhl/k8s-ci-cd-conf.git
        shallow_clone: true
        auto_update: true
        branch: main
    stages:
      - deploy_app:
          fetch_materials: true
          keep_artifacts: false
          clean_workspace: false
          approval:
            type: success
          jobs:
            build_image:
              timeout: 0
              elastic_profile_id: demo-app
              tasks:
                - exec:
                    arguments:
                      - -c
                      - gcloud container clusters get-credentials \
                        k8s-ci-cd-1-gke --zone europe-west1-b
                      - helm upgrade --install podtato-head ./ \
                        --namespace default
                    command: /bin/bash
                    run_if: passed

```

The configuration file for the CD pipeline can be seen in listing Listing 6. The configuration is written in YAML and the *pipelines* tag specifies the pipelines to run, in this case, only one pipeline called *deploy\_app*. The pipeline then contains some basic configuration followed by the *materials* tag that specifies the trigger for the pipeline. In this case the git-configuration repository will be monitored for changes. If a change is detected, the pipeline will run as specified by the *stages* tag. Here the jobs that the pipeline performs

are specified, this pipeline only contains one job, to connect to the cluster and deploy the application using helm.

The configuration file can be applied in different ways to the GoCD server. In this study the configuration is applied by setting up the GoCD server to monitor the Github repository containing the configuration and automatically updating the pipeline when new changes are committed.

## 4.6 Testing Tool

The testing tool is built as a Python script. The script has functionality to increase a version and commit the change to Github. This is done using the *GitPython* package.

It then starts a timer and repeatedly sends an http request (every second) using the *requests* package to the application in the cluster and examines the returned version. If the versions match the expected version after the change, then the test is complete.

The script will also include functionality to inject a fault into the CI/CD components of the cluster.

### 4.6.1 Fault Injection

Chaos-mesh will be used to insert faults in tests that assess fault tolerance of the CI/CD tools. The Chaos-mesh experiments are configured via YAML files, and after a certain length of time, a fault will be injected into the CI/CD tools. The injection is performed by applying a YAML file to the Kubernetes cluster that kills one of the pods operating the pipeline. The pods of the different tools that are in charge of executing the pipeline have different characteristics (such as labels or annotations) that are used to distinguish them when selecting which pods to kill.

## 5 Testing/Results

### 5.1 Experiment Design

The pipelines will be subjected to two different types of experiments:

- *Deployment-time experiments* are used to determine how long it takes from the time a change is made in the git repository to the time the change is visible in the running application. This experiment will satisfy the first point of the success criteria defined in section 3.4 as the testing tool will measure the deployment time.
- *Fault-tolerance tests* are conducted to assess the pipelines resilience to pod-faults. This will satisfy the second point of the criteria for success as faults will be injected into the CI/CD tools during the tests.

To get a sufficient sample size, the experiments will be repeated five times for each pipeline and experiment.

*Access security* will be examined by developing an overview of the system and the various access privileges that the various subsystems have.

#### 5.1.1 Deployment-time Experiment

1. The testing-tool changes a version number in the code repository and commits the change to the code repository.
2. A timer is started and the application version number will be polled every second.
3. When the version update is visible, the timer will stop and the time is the result of the test.

#### 5.1.2 Fault-tolerance Experiment

1. The testing-tool changes a version number in the code repository and commits the change to the code repository.
2. A timer is started and the application version number will be polled every second.
3. After 1 minute of polling, a fault will be injected into one of the CI/CD tools that execute the pipeline. The 1 minute delay gives the pods time to start-up and start building the images.
4. When the version update is visible, the timer will stop and the total time is the result of the test.
5. If the pipeline does not recover from the fault, the testing tool will run for 15 minutes.

## 5.2 Results

### 5.2.1 Deployment Time

Figure 4 show the results of the deployment time tests. The vertical axis displays the time from a change is committed in the source code until it is visible in the cluster. The horizontal axis shows the tools that make up the pipeline.

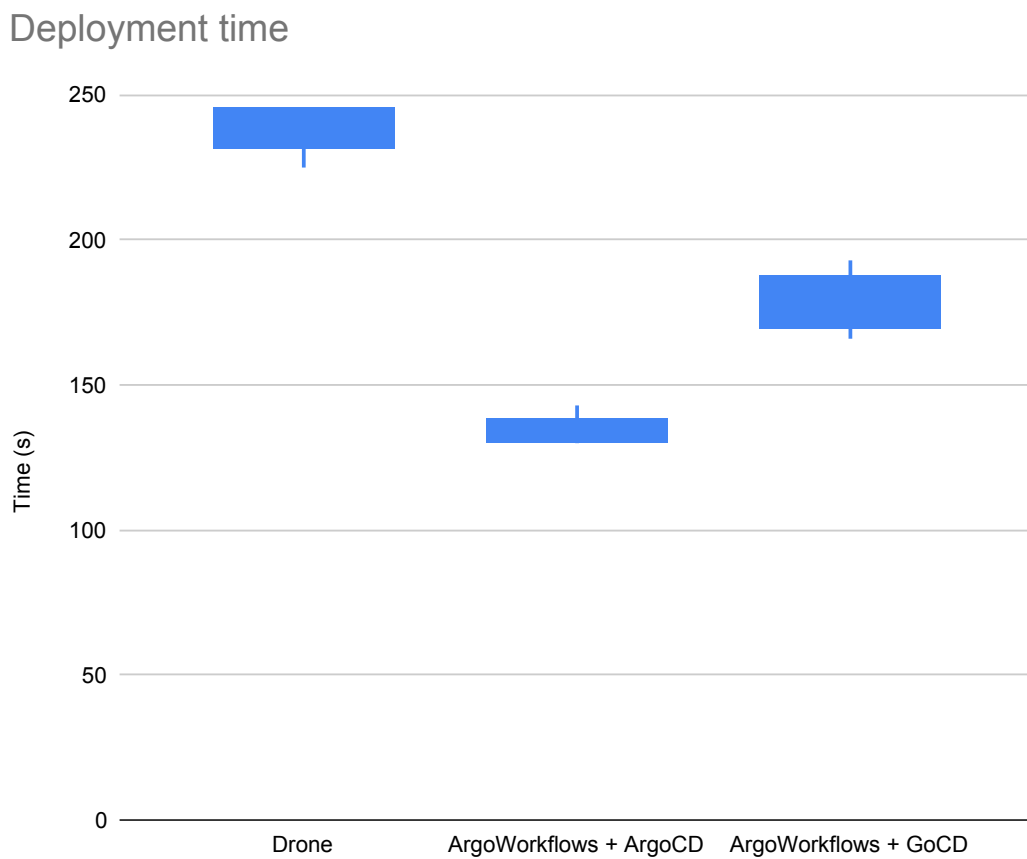


Figure 4: Deployment time results.

Figure 5 show the results of the deployment time tests with fault injection. The vertical axis again shows the time from a change being made in the source code until it is visible in the cluster. The horizontal axis shows the tools that make up the pipeline.

The pipelines with time 0 timed out after 15 minutes and could not recover from the fault.

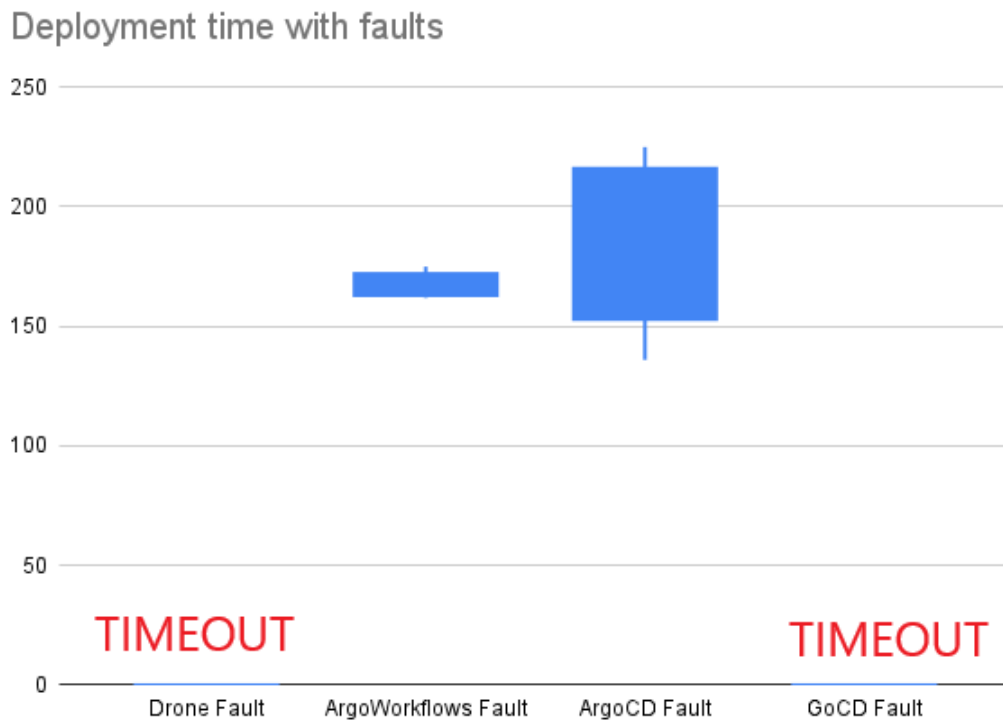


Figure 5: Deployment time results with faults.

### 5.2.2 Access Security

Figure 6 shows an overview of a push-pipeline. The artifacts built by the CI tool will be pushed to the image registry and the CD tool will then deploy the images to the cluster using access credentials.

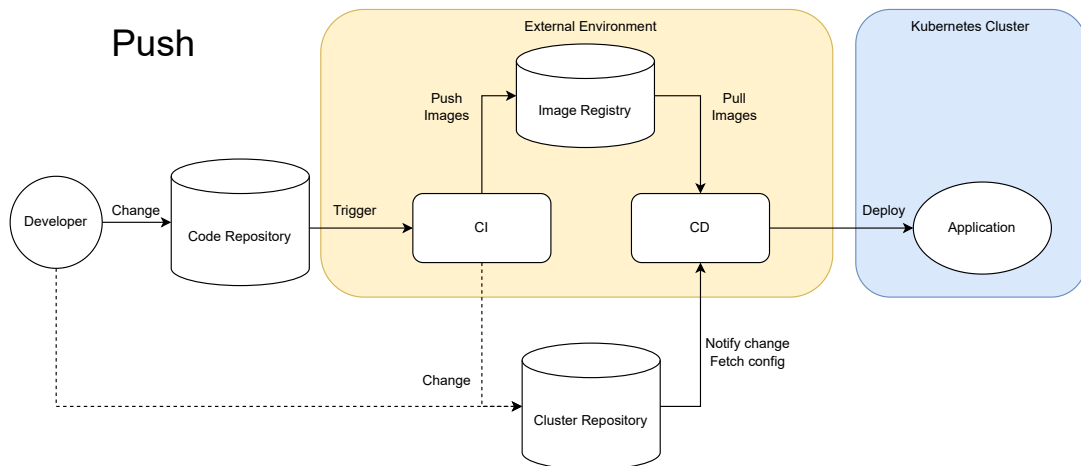


Figure 6: Overview of a push-pipeline.

Figure 7 shows an overview of a pull-pipeline. The artifacts will be built by the CI tool and pushed to the image registry. The CD tool will be running inside the cluster and will pull the images from the registry and can using an assigned service account update the cluster.

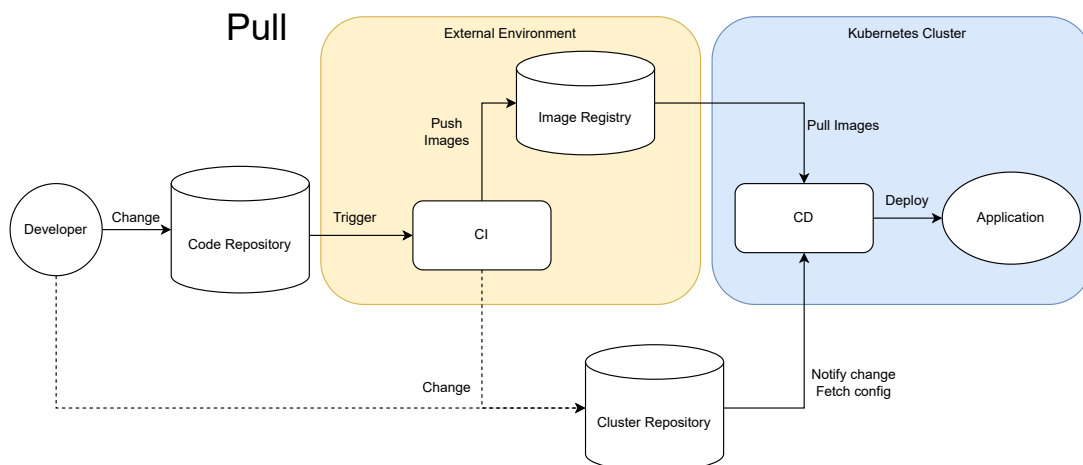


Figure 7: Overview of a pull-pipeline.

## 6 Evaluation

In order to answer the research questions, this part will examine and evaluate the data from Section 5.2.

### 6.1 Test Results

The tests are considered successful as the criteria for success described in section 3.4 was met. The testing tool is able to measure the deployment time of the applications. Faults can be injected into the various pipelines during execution, and diagrams of push and pull pipelines can be visualized.

#### 6.1.1 Deployment Time Test

The results of the deployment time test can be seen in the Results section in Figure 4. The pipeline using Drones was the slowest, followed by the pipeline using ArgoWorkflows and GoCD. The combination of ArgoWorkflows and ArgoCD was the fastest. Time discrepancies are not significant for such a modest project, but may become more pronounced in larger pipelines with more steps and phases.

The Drone pipeline and the ArgoCD and GoCD pipelines are similar in that they consist of images that run a sequence of commands to produce and publish images. The timing discrepancy between them is probably related to how workloads are managed in the cluster. ArgoWorkflows will generate a pod for each step in the pipeline, and stages that can run concurrently will run in separate containers in separate pods and perhaps on different nodes. The Drone tool, on the other hand, will execute each step in the pipeline in a separate container but will run all of the containers in a single pod. This implies that pods cannot be divided across several nodes in the Kubernetes cluster and hence cannot use more resources than a single node.

The pipeline that uses ArgoWorkflows and ArgoCD is marginally faster than the pipeline that uses ArgoWorkflows and GoCD. Because both pipelines use ArgoWorkflows in the CI stage, the time difference will be in the CD stage. Both CD tools (ArgoCD and GoCD) will be activated by webhooks from the configuration repository, therefore, the trigger time will be the same. Instead, the difference will be seen in the amount of time it takes to deploy the new images to the cluster. The GoCD tool must create a new pod in the cluster that will download the cluster repository and then deploy the application to the cluster using the helm chart. When a webhook arrives at the ArgoCD server, the ArgoCD tool does not need to create a new pod, but instead has a workflow controller statefulset that is always present in the cluster. Because the controller already exists in the cluster and does not need to be built, the application will be delivered faster.

The pipeline with ArgoWorkflows and GoCD also has slightly higher variance than the one with ArgoWorkflows and ArgoCD. This is likely due to the GoCD tool needing to create a new pod and wait for it to register to the GoCD server each time. Whereas the ArgoCD tool has a controller that always is running and no new pods needs to be created.

From these results, it shows that the pipeline using Kubernetes-specific tools is faster than the others that utilize generic tools. The fully generic pipeline using Drone is the slowest, the pipeline using ArgoWorkflows, a Kubernetes-specific tool, as well as the generic GoCD tool is a bit faster, and the fully Kubernetes-specific pipeline using Argo is the fastest.

From these results, it shows that the integrated pipeline using Drone is slower than the other pipelines that utilize two different tools for the CI and CD steps. This discrepancy is probably more dependent on how well the tool runs on Kubernetes.

### 6.1.2 Deployment Time With Faults Test

Figure 5 depicts the results of the deployment time tests with fault injection. Only the pipelines that used ArgoWorkflows and ArgoCD were able to recover from the faults, and the time it took to do so can be observed. Both the Drone and GoCD pipelines were stopped after 15 minutes and were unable to recover from the fault. All of the pipelines were set to retry on errors if the option was available, however the Drone and GoCD pipelines did not have this option. It is possible that the configuration to retry exists, but in that case, it would be difficult to locate.

When a fault is injected to kill a pod in the Drone pipeline, all containers running workloads are deleted because they all operate within the same pod. The Drone server will never notice that the pod has disappeared and will continue to wait for it to finish indefinitely. Within the Drone UI, the job must be manually canceled.

When a fault is introduced into the ArgoWorkflows tool, one of the pods in charge of creating application images is terminated. The ArgoWorkflows workflow controller will detect the absence of the pod and the tool can be configured to retry a specified number of times. As a result, the ArgoWorkflows tool will simply generate a new pod that will then complete the terminated job. This will result in a slight time increase since a new pod must be created and the image rebuilt, but the time increase will be only approximately 20-30 s.

When a fault is introduced into the ArgoCD tool, the pod in charge of the deployment is terminated. Because this pod is part of a statefulset, the Kubernetes cluster will replicate it automatically. The time increase from an ArgoCD fault is significantly higher and has a larger fluctuation; this might be due to the statefulset taking a little longer to rebuild the container at times. Another possibility is that when the ArgoCD server gets the webhook to update the application, the workflow controller is unavailable, thus the webhook event is lost, and the application will instead update the next time ArgoCD checks the application

against the configuration git repository.

From the results, it is evident that Kubernetes-specific tools can withstand pod faults as the pipelines with Argo were still able to update the application. Both the pipelines using generic tools were not able to recover from the faults and timed out.

Faults can affect both integrated and standalone pipelines, as both the Drone pipeline and the pipeline using Argo and GoCD timed out. As a result of these findings, fault tolerance appears to be concerned with whether the tools are generic or Kubernetes-specific.

## 6.2 Access Security

When using a push pipeline, as can be seen in fig. 6, the external environment where the CI and CD tool is running must have access to the Kubernetes cluster where the application is running as the CD tool needs to update the cluster with new versions of the application.

If the CD-tool is instead deployed inside the cluster with the application, the cluster credentials do not have to leave the cluster. The configurations are updated through a Git repository, and the CD tools which resides inside the application cluster can read the configuration from there and fetch the images from the image repository. All this setup needs to expose is some way to configure the CD tool inside the cluster. In this way the credentials to the Kubernetes api will never have to leave the cluster.

A CI/CD pipeline may be configured in push or pull mode, employing both Kubernetes-specific and generic tools. Therefore the benefit of having cluster credentials inside the cluster can be achieved with both Kubernetes-specific and generic tools. If the pipeline employs integrated CI/CD tools, the CI and CD tools must be located in the same environment. This has no effect if a push pipeline is utilized, as both tools are outside the application cluster. However, if a pull pipeline is utilized, the CI tool must also be located within the cluster. When the CI stage is constructing containers, it may require cloud resources from the application cluster, and it may pose security issues if malicious code is submitted to the code repository or insecure plugins are used for the CI tool.

## 6.3 Aims & Objectives

Aim 1 of creating infrastructure to test the different solutions was completed, as the implementation overview can be seen in fig. 3 and the results could be gathered from the tests as can be seen in section 5.2.

The second aim, to establish evaluation criteria and to evaluate the pipelines was also implemented. The criteria were established in section 2.6 and then evaluated using both the testing tool with the results described in section 5.2 and also by creating overviews of different types of pipelines, as can be seen in section 5.2.2.

The third aim, to compare Kubernetes-specific tools with generic tools and integrated compared to standalone tools, is also completed and is discussed in the evaluation of the test results in Section 6.1.

According to the findings:

- *Kubernetes-specific* tools have a faster deployment time and can also withstand pod faults with slight increases in deployment time.
- *integrated* tools need more time to deploy, albeit this is most likely owing to the tool's generic nature. Integrated tools are more difficult to set up in a pull manner since both the CI and CD tools must be deployed in the application cluster.

From the metrics used in this study, the standalone and Kubernetes-specific tools got the best results. However, integrated and generic tools could have other positives, such as being easier to setup or having plugins to make the integration step easier.

## 7 Conclusion

The first chapter of the study offers an introduction to the topic, as well as the aims and objectives of the study. The chapter also has a risk register showing the dangers that can prevent the objectives of the study from being achieved. The chapter concludes with a time schedule that defines when each section of the report will be worked on and when each section should be completed.

The following chapter, **chapter 2**, presents a literary basis on the topic of cloud computing. It explains how the microservices design has increased the importance of CI/CD pipelines and various features of the pipeline. It demonstrates the vast ecosystem of tools available in the CI/CD arena, picks a few for comparison, and covers certain assessment criteria for CI/CD pipelines. Finally, it identifies the research issues that this thesis should address.

The solution architecture that will be used to address the research questions specified in the previous chapter is discussed in **chapter 3**. It presents an overview of the architecture and describes the individual components. Then a solution test run is displayed, the assessment criteria utilized to acquire the results are explained, and finally, the success criteria are specified.

The previous chapter's solution concept is executed and shown in a diagram in **chapter 4**. The configuration of the cluster in which the tests will take place is described and the specific provider for each component of the implementation is justified. Each of the tools used in the pipeline and to be tested is discussed in further detail, with samples of how each tool is configured. The testing tool is also discussed in depth, such as how the testing tool will insert faults into pipelines.

The experiments and how they are measured are subsequently described in **chapter 5**. The deployment time and fault tolerance studies are measured by triggering a change and then measuring how long it takes for the change to become observable. By developing an overview of the various push and pull pipelines, access security criteria are investigated. The results of these studies are then represented graphically for each of the pipelines investigated.

The results are discussed in **chapter 6**, and each of the tests is examined individually.

### 7.1 Technical Issues Encountered

The process of setting up the tools mainly includes reading their documentation to configure the tool and link it to the Github repository. When installing Jenkinsx in the Kubernetes cluster and configuring the Github webhooks, a few errors occurred. The Kubernetes installation encountered an issue because some secrets were not initialized correctly. To fix this, the secret had to be manually entered, and the boot job would then finish correctly. Jenkins would also automatically configure the webhooks to listen to the

Github repository; but, because the cluster only had an IP address and not a full domain name, the subdomain that Jenkinsx would configure was unable to route to the cluster, and the webhooks would not trigger.

The pipelines' deployment stage is fairly rapid because there is only a helm-chart that will be deployed to the cluster. This made automating fault injection for the GoCD tool problematic since the time-window where faults could be injected was too short to specify a delay for. To resolve this, the fault was manually injected after the deployment phase had just begun by manually checking the GoCD dashboard to see when the step had started.

When configuring the pipeline's CI stages, the containers must run docker commands. However, no docker daemon is running within the container to carry out the tasks. To enable this, a volume must be mounted to the parent VM's Docker daemon. This will allow the container to interact with the Docker daemon and execute Docker commands on the external Docker daemon from within the container.

## 7.2 Personal Reflections

The study gave me a very valuable personal experience in the development cycle of cloud native projects. It gave me insight in all from setting up pipelines to continuous integration, to image storage, to continuous delivery.

Communication with both supervisors helped and guided me throughout the thesis. My internal supervisor, Paul Townend, initially assisted me in organizing the project and structuring my report appropriately. Mulugeta Tamiru, my external supervisor, advised me on which areas to focus on and how to evaluate the CI / CD pipelines. Both sections were critical in helping me to complete the report.

In hindsight, I could have redistributed my time early on and put more time into the design of the different parts of the solution and put some more effort into how the different parts of the solution should be designed initially, this would have made the time plan more accurate if I initially thought out the design of the testing tool and other parts.

Initial planning could also have used more time before choosing the evaluation criteria and starting the implementation, as there were some parts of a CI/CD solution that I had understood slightly incorrectly at the beginning of the study.

When I started working on the solution and implementation, I did not have any tool to track my progress or structure the work I had to do. I would simply start working on the different tools and environment I had to implement rather randomly. To use a tool to track tasks that are done and to-do would make it easier to see what I had done and needed to do. This would have been helpful, as sometimes I had to go back and fix systems that I had implemented incorrectly or not entirely.

The feedback from making a change to being able to test the change was pretty large

throughout the development phase. This was due to several reasons, one of them was that the tools to be tested took a few minutes to set up and some of the pipelines were set up using two tools. Some of the tools also required configuration through a GUI which also contributed to the total time needed. This could have been slightly mitigated by fully configuring one tool and then moving on to the next one. However this would also have required more initial planning of the different tools and the testing tool.

Another aspect that also contributed to the long waiting times was that I created the Kubernetes cluster at the beginning of the day and destroyed it at the end of the day using Terraform. This would take 10-15 minutes. I did this because I had a limited amount of credits in Google Cloud and tried to limit the amount I would spend. However, this could have been easier by instead using a local Kubernetes environment using minikube or to simply leave the cluster running and try to compress the time implementing by not mixing the time with writing on the report. I could also simply have left the cluster running as I had plenty of credits remaining at the end of the project, I could probably have calculated this at the beginning if I put some time calculating the cost.

In the planning phase, I created a risk register where I assessed the risks that might hinder me from finishing the study. This was helpful in making a decision on what to do if any of them happened. However, one of the risks that some tools took a bit longer to implement occurred, and the reaction would be to narrow the scope of the thesis by excluding some tool from the comparison. I was a bit slow to react to the risk and I should have checked the time table more frequently and realized that I should focus more on completing a draft of the report and then, if I had time, go back and finish the tools.

### 7.3 Future Work

There are no defined comparison criteria to evaluate different CI / CD solutions, as research on CI / CD tools and pipelines is sparse and there are few large studies that compare different tools. The deployment time used in this study is helpful, but there may be other factors that are more important in determining which CI/CD systems to use. Research is needed on which criteria and traits are the most important to consider when choosing tools to use.

More research would also be required to determine the security of pipeline design and how businesses should choose between alternative CI / CD approaches, such as push or pull. Organizations who advocate for the pull-style and Gitops pipeline methods argue that there are security benefits, but this has to be proven and discussed more.

Other tools may have additional properties that make them more suitable for specific locations or tasks. More study might be undertaken into comparable tools and their strengths and limitations. This might include functionality provided by tools that is not fully integrated with the CI/CD workflow. As an example, to automatically prune outdated application deployments and monitor for changes in the ongoing application.

Another area that may benefit from greater investigation is how the various tools respond when something goes wrong in the pipelines. For example, if a new deployment of a program involves flaws, there may be a simple mechanism to revert to previous versions of the application so that high availability is ensured.

---

## References

- [1] Isam Mashhour Al Jawarneh et al. “Container orchestration engines: A thorough functional and performance comparison”. In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE. 2019, pp. 1–6.
- [2] Algirdas Avizienis et al. “Basic concepts and taxonomy of dependable and secure computing”. In: *IEEE transactions on dependable and secure computing* 1.1 (2004), pp. 11–33.
- [3] Florian Beetz, Anja Kammer, and Dr. Simon Harrer. *GitOps*. URL: <https://www.gitops.tech/> (visited on 02/15/2022).
- [4] David Bernstein. “Containers and cloud: From lxc to docker to kubernetes”. In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.
- [5] Adam Björnberg. *Cloud native chaos engineering for IoT systems*. 2021.
- [6] Szilárd Bozóki et al. “Application of extreme value analysis for characterizing the execution time of resilience supporting mechanisms in Kubernetes”. In: *European Dependable Computing Conference*. Springer. 2020, pp. 185–199.
- [7] Gary Chen. “The rise of the enterprise container platform”. In: *IDC White Paper* (2018).
- [8] Suman De. “A Study on Chaos Engineering for improving Cloud Software Quality and Reliability”. In: *2021 International Conference on Disruptive Technologies for Multi-Disciplinary Research and Applications (CENTCON)*. Vol. 1. 2021, pp. 289–294. DOI: [10.1109/CENTCON52345.2021.9688292](https://doi.org/10.1109/CENTCON52345.2021.9688292).
- [9] Nicola Dragoni et al. “Microservices: yesterday, today, and tomorrow”. In: *Present and ulterior software engineering* (2017), pp. 195–216.
- [10] Alwin Ebermann. *Evaluation of GitOps Security in a CI/CD Environment*.
- [11] Christof Ebert et al. “DevOps”. In: *IEEE Software* 33.3 (2016), pp. 94–100. DOI: [10.1109/MS.2016.68](https://doi.org/10.1109/MS.2016.68).
- [12] “Exploring the Uncharted Space of Container Registry Typosquatting”. In: *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022. URL: <https://www.usenix.org/conference/usenixsecurity22/presentation/liu-guannan>.
- [13] Cloud Native Computing Foundation. *CNCF Cloud Native Landscape – Continuous Integration & Delivery*. URL: <https://landscape.cncf.io/card-mode?category=continuous-integration-delivery> (visited on 01/19/2022).
- [14] Open Source Security Foundation. *FLOSS Best Practices Criteria*. URL: <https://bestpractices.coreinfrastructure.org/en/criteria/0> (visited on 01/19/2022).
- [15] Dennis Gannon, Roger Barga, and Neel Sundaresan. “Cloud-native applications”. In: *IEEE Cloud Computing* 4.5 (2017), pp. 16–21.
- [16] Jack Henschel. *A comparison study of managed CI/CD solutions*.

- 
- [17] Hui Kang, Michael Le, and Shu Tao. “Container and microservice driven design for cloud infrastructure devops”. In: *2016 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE. 2016, pp. 202–211.
- [18] Timo Lehtonen et al. “Defining metrics for continuous delivery and deployment pipeline.” In: *SPLST*. 2015, pp. 16–30.
- [19] Leonardo Leite et al. “A survey of DevOps concepts and challenges”. In: *ACM Computing Surveys (CSUR)* 52.6 (2019), pp. 1–35.
- [20] Jamal Mahboob and Joel Coffman. “A Kubernetes CI/CD Pipeline with Asylo as a Trusted Execution Environment Abstraction Framework”. In: *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*. 2021, pp. 0529–0535. DOI: [10.1109/CCWC51732.2021.9376148](https://doi.org/10.1109/CCWC51732.2021.9376148).
- [21] Lakshay Malhotra, Devyani Agarwal, Arunima Jaiswal, et al. “Virtualization in cloud computing”. In: *J. Inform. Tech. Softw. Eng* 4.2 (2014), pp. 1–3.
- [22] Mathias Meyer. “Continuous integration and its tools”. In: *IEEE software* 31.3 (2014), pp. 14–16.
- [23] Vaishnavi Mohan and Lotfi Ben Othmane. “Secdevops: Is it a marketing buzzword?-mapping research on security in devops”. In: *2016 11th international conference on availability, reliability and security (ARES)*. IEEE. 2016, pp. 542–547.
- [24] Marek Moravcik and Martin Kontsek. “Overview of Docker container orchestration tools”. In: *2020 18th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE. 2020, pp. 475–480.
- [25] Kief Morris. *Infrastructure as code: managing servers in the cloud*. " O'Reilly Media, Inc.", 2016.
- [26] Fotis Nikolaidis et al. “Frisbee: A suite for benchmarking systems recovery”. In: *Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems*. 2021, pp. 18–24.
- [27] Fotis Nikolaidis et al. “Frisbee: automated testing of Cloud-native applications in Kubernetes”. In: *CoRR* abs/2109.10727 (2021). arXiv: [2109.10727](https://arxiv.org/abs/2109.10727). URL: <https://arxiv.org/abs/2109.10727>.
- [28] Amit M Potdar et al. “Performance evaluation of docker container and virtual machine”. In: *Procedia Computer Science* 171 (2020), pp. 1419–1428.
- [29] Njegoš Raičić and Mihajlo Savić. “Architecting Continuous Integration and Continuous Deployment for Microservice Architecture”. In: *2021 20th International Symposium INFOTEH-JAHORINA (INFOTEH)*. 2021, pp. 1–5. DOI: [10.1109/INFOTEH51037.2021.9400696](https://doi.org/10.1109/INFOTEH51037.2021.9400696).
- [30] Aaqib Rashid and Amit Chaturvedi. “Cloud computing characteristics and services: a brief review”. In: *International Journal of Computer Sciences and Engineering* 7.2 (2019), pp. 421–426.

- 
- [31] Tony Savor et al. “Continuous Deployment at Facebook and OANDA”. In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 2016, pp. 21–30.
- [32] Charanjot Singh et al. “Comparison of different CI/CD tools integrated with cloud platform”. In: *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*. IEEE. 2019, pp. 7–12.
- [33] Diomidis Spinellis. “Choosing and Using Open Source Components”. In: *IEEE Software* 28.3 (2011), pp. 96–96. DOI: [10.1109/MS.2011.54](https://doi.org/10.1109/MS.2011.54).
- [34] Priyanshu Srivastava and Rizwan Khan. “A review paper on cloud computing”. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 8.6 (2018), pp. 17–20.
- [35] Junzo Watada et al. “Emerging trends, techniques and open issues of containerization: a review”. In: *IEEE Access* 7 (2019), pp. 152443–152472.
- [36] WeaveWorks. *How GitOps Improves the Security of Your Development Pipelines*. 2020-12-17. URL: <https://www.weave.works/blog/how-gitops-improves-security-development-pipelines> (visited on 03/01/2022).
- [37] Alex Williams. *CI/CD With Kubernetes*. Linux Foundation. 2018. URL: [https://resources.linuxfoundation.org/LF+Projects/CNCF/TheNewStack\\_Book3\\_CICDwithKubernetes\\_20180615.pdf](https://resources.linuxfoundation.org/LF+Projects/CNCF/TheNewStack_Book3_CICDwithKubernetes_20180615.pdf) (visited on 02/15/2022).
- [38] Yang Zhang et al. “One size does not fit all: an empirical study of containerized continuous deployment workflows”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 295–306.
- [39] Liming Zhu, Len Bass, and George Champlin-Scharff. “DevOps and Its Practices”. In: *IEEE Software* 33.3 (2016), pp. 32–34. DOI: [10.1109/MS.2016.81](https://doi.org/10.1109/MS.2016.81).