FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF BACHELOR’S THESIS

Title:	Continuous integration and application deployment with the Kubernetes technology
Student:	Radek Šmíd
Supervisor:	Ing. Jan Trdlička, Ph.D.
Study Programme:	Informatics
Study Branch:	Computer Security and Information technology
Department:	Department of Computer Systems
Validity:	Until the end of summer semester 2020/21

Instructions

Learn about the Kubernetes technology and available commercial and open-source deployment tools suitable for this technology. Compare features of individual open-source deployment tools. Design a suitable solution for application deployment using some of these open-source tools and try to implement this solution. Test the functionality of your solution on a suitable containerized application and try to compare your solution with available commercial solutions (if possible under trial licenses).

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague October 22, 2019

CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F8

Faculty of Information Technology
Department of Computer Systems

Bachelor's Thesis

Continuous integration and application deployment with the Kubernetes technology

Radek Šmíd

Květen 2020
https://gitlab.fit.cvut.cz/smidrad1/ci-cd-pipeline-with-kubernetes
Supervisor: Ing. Trdlička Jan Ph.D.

Acknowledgement / Declaration

I would like to sincerely thank my supervisor Ing. Jan Trdlička, Ph.D. His guidance and support have been beyond excellent.

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague 28. 5. 2020

....

Abstrakt / Abstract

Poslední dobou by téměř každý chtěl své aplikace nasadit do Kubernetes. Jenže pro plné využití Kubernetes je třeba přijmout s otevřenou náručí postupy průběžné integrace (CI) a nasazení (CD). Je třeba CI/CD pipeline. Ale k dispozici je až zdrcující množství open-source nástrojů, kde každý pokrývá různé části celého procesu.

Následující text vysvětlí základy technologií, kterých bude pro pipeline třeba. A následně shrne některé z populárních open-source nástrojů využívaných pro CI/CD.

Z open-source nástrojů navrhneme pipeline. Závěrečné porovnání možných řešení (včetně proprietárních) poskytne čtenáři konkrétní tipy a rady ohledně vytváření vlastní pipeline.

Klíčová slova: Open-source CI/CD, Kubernetes, CI/CD pipeline, Mikroslužby, Průběžná integrace, Průběžné nasazení, Nepřetržitá integrace a nasazení.

Překlad titulu: Nepřetržitá integrace a nasazení aplikací s technologií Kubernetes

It seems nearly everyone would like to deploy to Kubernetes nowadays. To efficiently leverage the power of Kubernetes one must first fully embrace continuous integration (CI) and deployment (CD) practices. A CI/CD pipeline is needed. But there is an overwhelming amount of open-source tools that cover various parts of the whole process.

The following text explains the basics of the underlying technologies needed for a pipeline deploying to Kubernetes. And subsequently summarizes some of the popular open-source tools used for CI/CD.

Then it designs a working pipeline from the researched tools. Finally, it summarizes some of the possible pipelines (including proprietary) and provides the reader with specific bits of advice on how to implement a pipeline.

Keywords: Open-source CI/CD, Kubernetes, CI/CD pipeline, Microservices, Continuous integration, Continuous delivery, Continuous deployment.

Contents /

/ Figures

Chapter 1
Introduction

Is it possible to create a fully-featured pipeline deploying to Kubernetes only with open-source tools?

Kubernetes. The seventh most contributed project on GitHub has caused quite a buzz in the IT world. Kubernetes makes it possible to build fault-tolerant, automatically scaling, cost-efficient systems. Who wouldn't want that? Naturally, everything has a catch. An application running on Kubernetes must be designed as a set of containerized microservices. But that itself is not enough. Continuous integration and deployment (CI/CD) practices are an integral step to fully realizing the potential of microservices and Kubernetes. [1] The goal is to deploy multiple times a day to production. To deploy simultaneously with agility and confidence.

The problem with CI/CD is that there is no „one size fits all" approach. There is no „correct" way. There is only „suitable for us". [2] In the end, CI/CD boils down to figuring out and realizing the most fitting release culture for your team of developers.

The text is divided into two parts. Theoretical and practical. The theoretical part consists of the first three chapters. This bachelor thesis will explain the necessary pieces of technology used in deploying to Kubernetes. Chapter 2 will cover the basics of microservice architecture, containerization, and Kubernetes to familiarize the reader with them. The following, chapter 3 takes a closer look at Continuous integration and deployment. First in general and subsequently at the specifics with Kubernetes and microservice architecture. There is a lot of open-source projects (programs, tools) for CI/CD. The first goal of this work is **to research and compare open-source tools for CI/CD when deploying to Kubernetes**. Thus a comprehensive summary can be found in chapter 4.

Around 40 % of Kubernetes users run them on-premises (on local servers). [3] Though this is generally not a good idea it might be due to various laws, budgets, or licensing. The idea is explore how well can one build a CI/CD pipeline on local servers entirely from open-source software.

The practical part of this thesis will start with chapter 5 covering the second goal: **Designing a pipeline composed of some of the researched tools.** The designed solution will be implemented in a test environment and tested with a suitable containerized application. It shall be built based on modern principles and recommendations using only open-source tools. Chapter 6 will then list its upsides and downsides.

The third and last goal is to summarize the advantages of this „self-managed open-source tools approach" against some of the „commercial proprietary solutions" available on the market (under trial licenses). The objective is to **provide specific pieces of advice for anyone who is deciding on how to build a pipeline that is deploying applications into Kubernetes**.

The IT world has become oversaturated with tools that help you increase your release speed. A general overview will be useful for anyone tasked with picking which ones to use.

Chapter 2
Technology

Before we embark on the wonderful journey of continuous integration and deployment to Kubernetes we should familiarize ourselves with the technology surrounding this area. This chapter shall provide a brief introduction to the concepts of microservices (MSs), Docker containers, and Kubernetes orchestration. Explaining in fine detail how Docker or Kubernetes works is beyond the scope of this bachelor thesis, so the approach here will be purely pragmatic.

2.1 Microservice architecture

Microservice architecture is an approach to developing complex applications. The application is decomposed into several smaller pieces - microservices. Each microservice (or MS) has a small area of responsibility it handles. It can do nothing else, but it is exceptionally great in that one area. Each microservice has an interface. Interfaces are how microservices communicate with each other.

The trend started in 2014, but the idea has been around since the days of UNIX. With stdin and stdout as the interface, individual programs as microservices, and pipes as a way of connecting the endpoints. Microservices are widely used in Google, Amazon, and Netflix. [4], [5]

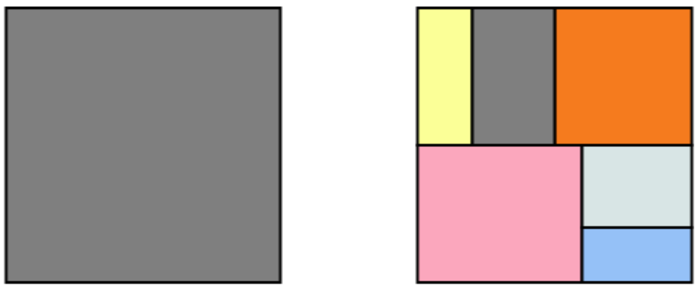


Figure 2.1. Monolithic vs microservice architecture [6]

In contrast to a monolithic architecture microservices operate independently on each other, see in figure 2.1. That means they do not have to use the same libraries, environments, and programming languages.

The left square shows a full-stack monolith app. On the right side, the big app is divided into smaller pieces. The yellow rectangle might represent a lightweight python webserver, while the pink square can be a power-hungry business logic Java MS.

Developers have the freedom to choose the tools they like to use. But it’s not only for the developer’s convenience. Each programming language has its strengths, therefore using different ones to solve different types of problems can yield better performance.

If we look at figure 2.2, we can see the difference when scaling. Monolith (left) can be scaled only vertically or in instances of each other. In contrast, microservices can be scaled individually. Scaling can be based on measured usage. Following the previous example, let's assume the company using our application hired three times more people. The pink business logic must scale three times. However, our python webserver (yellow) can handle the number of requests only scaled two times. The grey authentication MS does not have to scale at all. People sign-in only once a day.

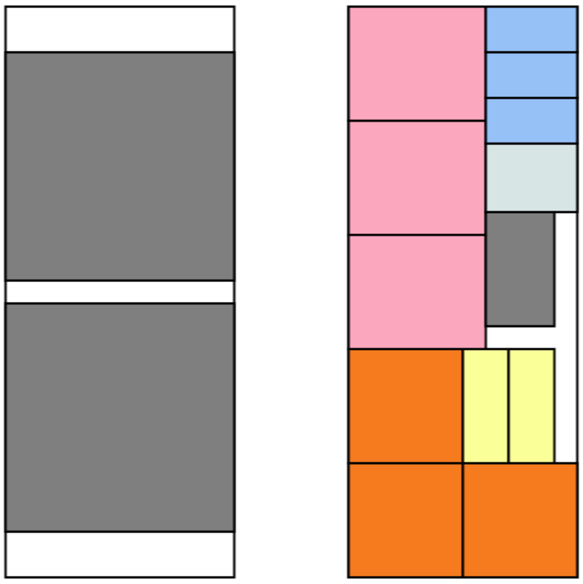


Figure 2.2. Scaling a monolithic application vs scaling a microservice application [6].

However, using this approach comes with some downsides. Each microservice has its own data storage. [1] In order to scale properly, MSs are recommended to be stateless. That means getting rid of „the one big relational database used for everything“. If we have multiple instances of the same microservice running we can encounter an inconsistency. This can be solved with several technologies. Redis, Apache Kafka, and Kappa architecture to name some. [7], [8], [9] However explaining them is not in the scope of this thesis.

Since the whole app is a one distributed system it relies heavily on the ability to send messages from service to service. If there are multiple instances of the same microservice someone needs to choose which one will get the message. This problem is solved by Kubernetes built-in DNS and load balancing support. Developers also tend to forget the 8 fallacies of distributed computing defined by Peter Deutsch and James Gosling in 1994. [10],[11] Most notably fallacy number one: `Network is reliable`.

To reach high fault tolerance the design should avoid any single point of failure. Consider a microservice e-shop, where the recommendation microservice stops responding. The customer should still be able to continue shopping. Resilience is a crucial part of designing such a system. So much so that Netflix has created a tool that randomly destroys instances of running microservices called Chaos Monkey. [12]

According to Martin Fowler, all microservice architectures share a set of common characteristics. [1] The text above mentions some of them, componentization via services, decentralized data management, decentralized governance, design for failure. In the same article, he also writes about the importance of Continuous Integration and Delivery.

„*We want as much confidence as possible that our software is working, so we run lots of automated tests. Promotion of working software 'up' the pipeline means we automate deployment to each new environment.*“

– Martin Fowler [1]

When developing a microservice application the team must trust and be able to rely on the infrastructure on which they are building. Therefore automatic deployment should be considered a prerequisite.

One microservice should handle one responsibility. However, how big can the microservice be before it stops being `micro`? The size of a team taking care of one microservice differs from company to company. The generally accepted rule comes from Jeff Bezos, founder of Amazon.

„*If you can't feed a team with two large pizzas, it's too large.*“

– Jeff Bezos [13]

2.2 Sample microservice application

To demonstrate and test all of the various CI/CD tools and practices, we need some software. A sample microservice app. The software must be open-source because of the requirement to edit the source code. Instead of reinventing the wheel, it's better to find what others have already created. Luckily researchers provide us with multiple options to choose from.

2.2.1 DeathStarBench

Deathstar Benchmark is a set of very complex MSs designed to test different types of applications developed as microservices. (Social network, Movie reviewing, Banking system, etc.) Its purpose is to benchmark the system under heavy load. [14] The development team at Cornell University has, however, not yet released all parts of its benchmark in their GitHub repository[1]. The released applications are very complicated and thus not suitable for our use.

2.2.2 Sock Shop

Sock Shop[2] is a microservice demo application maintained by Weaveworks and Container Solutions. The application is free and open-source, it has however not been updated in 2 years.

2.2.3 TeaStore

TeaStore[3] is a micro-service reference and test application developed by a research team in Würzburg, Germany. [15] It's written in Java and consists of 6 microservices. The architecture is not too complex. It's open-source and comes with documentation on how to use it, which makes it a perfect candidate for the use in this thesis.

[1] `https://github.com/delimitrou/DeathStarBench`
[2] `https://github.com/microservices-demo/microservices-demo`
[3] `https://github.com/DescartesResearch/TeaStore`

2.3 Docker & containerization

Docker is a leader and de facto industry standard for containerization. Containerization is a way of packaging and running apps independently on the infrastructure and OS. The official Docker website goes as far as calling containers „A standardized unit of software“. [16] Containerized applications are packaged only with their necessary libraries and binaries. Containers not only encapsulate the app itself but everything needed for it to run. It is a crucial step towards unleashing the full potential of microservice architecture. Each MS can be written in a different language, framework, or use a different version of a library. A unified way of distributing and running the MSs in production is needed for the automation of scaling.

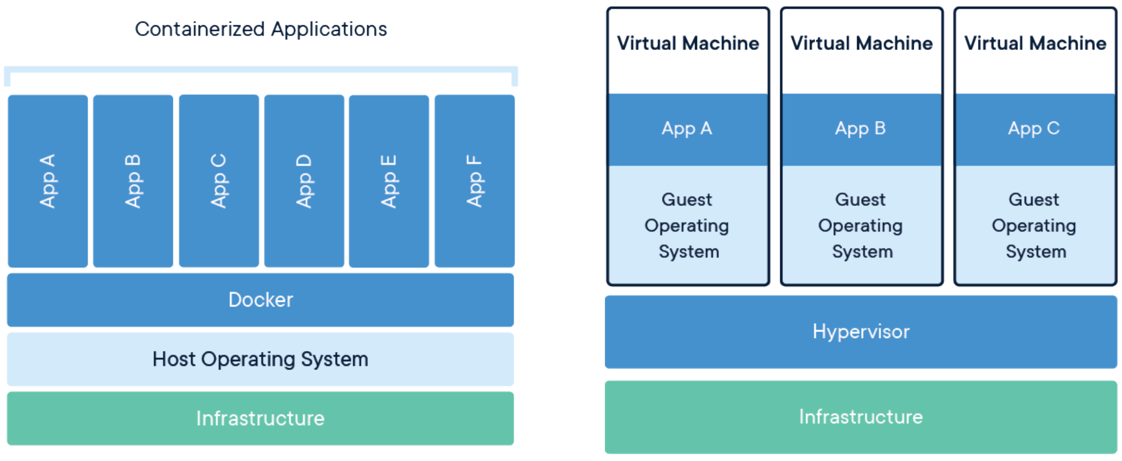


Figure 2.3. Containerization vs virtual machines [17]

Using containers saves space compared to virtual machines (VMs). Each app is packaged with, and only with its dependencies (see figure 2.3). Deploying with containers is also more efficient than with VMs because the Docker engine (runtime) has access directly to the OS's core. Multiple containers on one machine run as isolated processes inside the kernel userspace. [16]

2.3.1 Docker engine

Docker engine is a daemon managing containers. [16] For an overview of Docker engine architecture see figure 2.4. It can be controlled programmatically (by other programs, eg. K8s) or by a CLI. The binary (Docker engine) is used to[1]:

- download images stored in a (remote) docker registry
- create and remove containers from images
- execute commands in running containers
- build images described in Dockerfile

Kubernetes is orchestrating the containers. Instead of managing images and running containers ourselves (with Docker engine CLI), we instruct Kubernetes about our „desired state“ and it manages (by the Docker engine REST api) the containers instead of us.

[1] List not exhaustive. Unknown terms will be explained in following paragraphs.

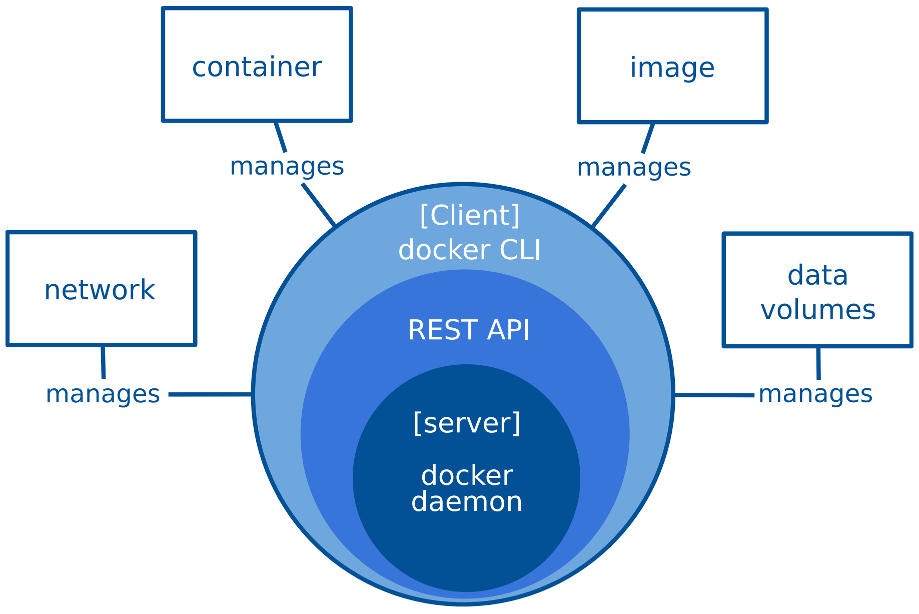


Figure 2.4. Docker engine [18]

2.3.2 Docker image

„*An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image.*“

– Docker documentation [16]

Images are stored in a repository inside some Docker registry. You can create a new image from a Dockerfile. A built image can be pushed to a repository.

2.3.3 Dockerfile

Dockerfile is a blueprint telling Docker how to create an image. It consists of instructions outlining the process of preparing the environment and building the application. Each MS should come with its Dockerfile. Dockerfile is written by the developer of each MS and should be a part of every MS's repository. Developers should have full control over how their MS is built and run.

Dockerfiles usually build upon a base image provided by the developer of each technology. For example. If a MS is written in Python we can start with the Python base image. Then we copy the source code and install all needed dependencies.

2.3.4 Docker registry

Docker registry is a program that stores, versions and manages Docker images. Just as source code is stored and versioned in a version control system (eg. Git), Docker images are stored and versioned in the Docker registry. The default Docker registry is called Docker Hub and is maintained by Docker Inc. Docker Hub (silmilary to GitHub) is free to use with only one private repository. [19]

Docker registry is available for download under the Apache license. [16] Avoid confusion with Docker Trusted Registry (DTR). DTR is a commercial product from Docker Inc.

2.3.5 Docker container

„*A container is a runnable instance of an image.*“

– Docker documentation [16]

When running a container we can map and expose its ports. Containers can be multiplied, stopped, and started at any given time (by Docker CLI or K8s). That means no data inside a container is persistent. We have briefly touched on how persistence is handled in MS architecture in chapter 2.1. Adding to that, when running a container we can mount (parts of) an outside filesystem inside it to maintain persistence even after container is rebuild.

2.4 Kubernetes & orchestration

Now that we have our application build as microservices and packaged in stateless containers, it's time for orchestration.

„*Orchestration is the automated configuration, management, and coordination of computer systems, applications, and services.*“

– RedHat [20]

The goal of orchestration is to automate the deployment of containers. The combination of MSs and orchestration solves the following problems in production:

- Horizontal scaling
- Fault tolerance
- Elasticity and const efficiency
- High availability (HA)

Kubernetes means helmsman in Greek and is abbreviated to K8s or k8s. The 8 signifies the eight missing letters. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. [21] Kubernetes simultaneously runs on multiple computers (nodes) aggregated into a cluster.

We can think about Kubernetes as an operating system for clusters with MS instances as threads and deployments as programs. After all, it has a command line interface, cron scheduler, and daemons - just like any other OS.

Another way of grasping the basic idea behind K8s is to imagine it as a football coach. [22] Every player has a certain role in the team. Someone is an attacker and someone is a goalkeeper. Just like each microservice can do something different. A coach makes sure that the number of attackers, defenders, and goalkeepers present on the field is correct. If a player gets injured (eg. microservice stops responding) coach schedules a new player instead of him and makes sure the injured player is taken out of the playing field.

2.4.1 Imperative vs declarative

Kubernetes were originally designed by Google as a remedy for the constant headache of system administrators. Years of experience with updating and maintaining servers convinced the designers that Kubernetes should be managed declaratively.

Imperative way of controlling means to issue a series of commands. Each of the commands changes the configuration of the system in some specific way. For example, as in a UNIX shell.

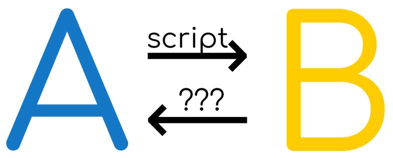


Figure 2.5. Imperative administration [23]

If we control a system declaratively it means we have a configuration file that declares a desired state. The system reads it and undertakes the steps to make its current state identical to the declared desired state. Similar to how configuration files work.

That is immensely useful for administration. Imagine we want to migrate our system from state A to state B. Imperative way is to write a long and complicated script that sets everything up. [23] Now, let's imagine that state B does not work properly and we need to migrate back to state A. We do not have an inverse script. (Fig. 2.5) The only path back is to reload the whole system from a backup or reconfigure it manually.

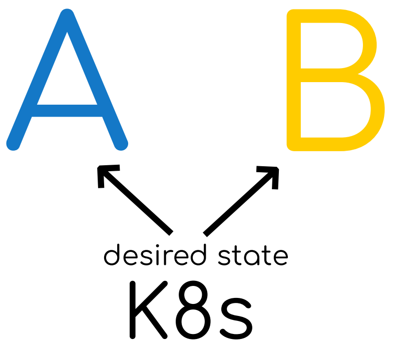


Figure 2.6. Declarative administration [23]

The declarative way solves this problem. We have the declarative configuration of state A and state B. One of them is provided to Kubernetes as the desired state. (Fig. 2.6) And that is all we have to do. Kubernetes modifies the current state of itself to match the state we provided.

When we configure K8s in practice we supply it with a YAML[1] file declaring the state we want to achieve. Which MSs we want running and how many instances. (In football that would be one goalkeeper, three attackers, etc..)

2.4.2 Kubernetes nodes

It is better to think about Kubernetes as an operating system for clusters. That way we are not surprised when we find out that Kubernetes cannot be „simply installed“, for example, on a Linux machine. Kubernetes are situated on multiple computers. Each machine is adding its computing capacity to the cluster and Kubernetes controls the cluster as a whole.

Every machine (physical or virtual) in the cluster is called a **node**. Kubernetes is a decentralized system made out of multiple nodes. There are two types of nodes – master and worker. Master node takes care of Kubernetes itself and worker node runs pods. To reach high fault tolerance it is recommended to run a Kubernetes cluster with 3-5 master nodes, each one on a different physical machine. [22]

[1] https://yaml.org

2.4.3 Pods, deployments, services

Pods are the minimal unit in K8s. Kubernetes cannot deploy anything smaller than a pod. A pod is a combination of containers and a persistent storage (volume). One pod corresponds to one microservice. Kubernetes pod is in some way similar to a thread in a common operating system. Pods are stateless and disposable. They are created and destroyed by Kubernetes at will.

Pods are seldom deployed by themselves. Kubernetes defines multiple layers of abstraction for working with pods. One of them is called **deployments**.

A deployment is similar to a program in normal OS. Deployment defines a pod (container images, volumes) and how many replicas it wants. Every deployment has a name. If we want to scale a microservice, we change the number of desired instances in its deployment. Kubernetes decides where pods are destroyed or created.

Microservices usually want to contact other microservices inside the cluster. For that Kubernetes defines internal networks for the pods and cluster. Pods could theoretically query certain IP addresses of other pods to contact them. However, pods are disposable – their IPs are not static. That is why we need a **service**. A service can aggregate and load balance queries to a deployment. To keep things simple Kubernetes internally uses DNS. Thus each pod is sending its queries to a **name** of the service it wants to contact. The request is then sent to any one of the healthy pods containing the desired microservice.

2.4.4 Kubernetes architecture

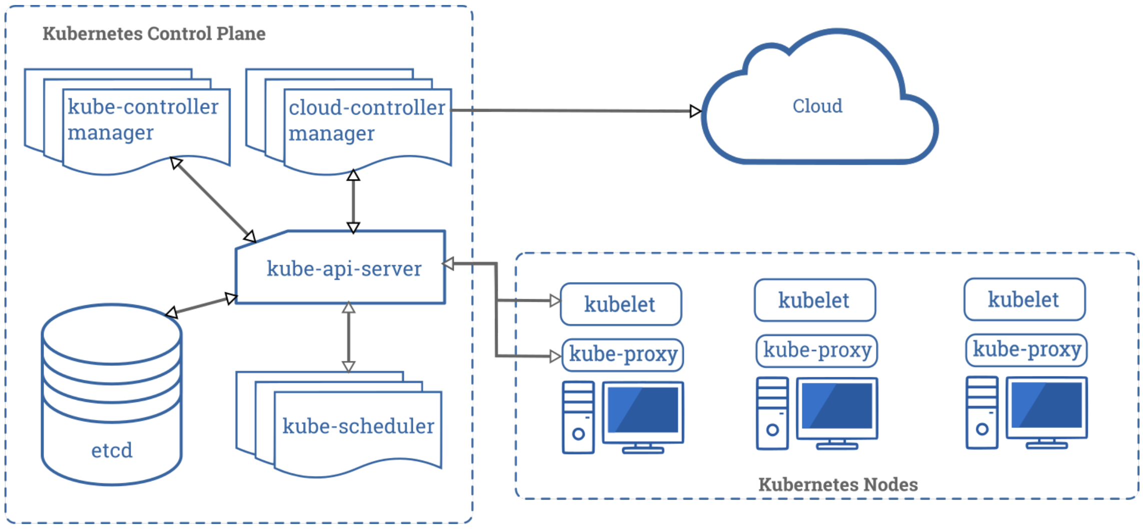


Figure 2.7. Kubernetes architecture [24]

Figure 2.7 shows the basic Kubernetes architecture. The control plane is a representation of multiple master nodes.

If we want to contact Kubernetes we need to locally install a program called **kubectl**. Kubectl is a command-line interface that sends REST requests to the Kubernetes API server and parses the response. It is possible to contact the REST API server by sending requests directly.

„**ETCD** *is a distributed, reliable key-value store for the most critical data of a distributed system.*“

– ETCD documentation [25]

The entire current state of Kubernetes along with all cluster data is saved in ETCD. In reality, ETCD is distributed along several machines to reach fault-tolerance requirements.

When K8s decides it wants to create a new pod it notifies the **kube-scheduler**. The scheduler decides on what node the new pod should be placed. It takes into account affinity (or anti-affinity) requirements, compute capacity, occupancy of the nodes and other variables. Intelligent scheduling of pods helps ensure the high availability of our application.

Simplified, the **kube-controller-manager** is a program that runs in a loop and constantly checks if the current state of K8s corresponds to the desired state. If not (for example when a node crashes) it acts to fix it.

Cloud controller is what makes Kubernetes cloud-native. A cloud controller interacts with the underlying cloud providers. [26] Imagine a typical rush-hour scenario. A large number of requests are made to our application and it is starting to slow down. To keep the app running Kubernetes needs more computing resources. If configured, Kubernetes can ask the cloud provider for additional resources. The provider (Azure, GCP, AWS, etc...) then provisions a new computer (usually a VM) and adds it as a new node to the Kubernetes cluster. This ensures elasticity, cost-efficiency, and high-availability for our application. Although the cloud-native side of Kubernetes is very interesting we can not cover it any further.

Kubelet is a program that runs on each node. It can register the node to the cluster through the API server. **API server can instruct a kubelet to run or delete a new pod on its node** - usually a Docker container. Kubelet pulls the container image from a container registry set in Kubernetes. The usual container registry for Docker images is called the Docker registry (see in section 2.3.4).

Chapter 3

Continuous integration & continuous deployment

Release. Most people in IT get nervous when they hear this word. They see all the merge errors, dependency resolutions, and downtimes due to upgrade. Continuous integration, continuous deployment (CI/CD) and DevOps are here to help.

DevOps is a culture. [27] The world is a combination of the words „developers“ and „operations“. It signifies the importance of the two groups communicating with each other. Simple sharing of information, motivations, and approaches can go a long way. Developers should not toss their code to the operations team with minimal testing and an alibi „it worked on my machine“. Just as operations should not force unnecessary restrictions on the developers because of bad experiences with them.

If operations give some access to the servers (eg. gives them their own VM) there are no records of what the developers did to make their applications work. (open ports, default installations of programs, imported certificates, etc.)

Generally, no one is confident about the release process because they fear that the blame will be assigned to them.

The idea behind DevOps is to find out how the build and release process is currently done. For example, sharing automation scripts/playbooks across the departments. Then, decide where and how automation could help. And finally, implement a CI/CD solution that reflects the company's needs.

„Job of a CI/CD pipeline is to prove that the code is not good enough.“
– Ken Mugrage [28]

The goal is to release multiple times a day with confidence. Releasing large changes means large risk something goes wrong. On the other hand, releasing smaller changes means lowering the risk. The problems are easier to identify. [29]

Before we dive into the CI/CD practices concerning Kubernetes we should clear up the terminology. What does CI/CD mean? [1]

3.1 Continuous integration

Continuous integration (CI) means to frequently integrate developer's work with the existing codebase. [30] The integration should happen at least a few times a day. When the new/changed code is pushed it is automatically built and rigorously tested. Part of the process can, for example, be a code review from fellow developers.

The CI process ends with a prepared version of the application (eg. a binary, container, jar...) sometimes called artifact. CI can also automatically generate documentation or build an installer.

[1] Please bear in mind that this thesis does not describe how to set up a production Kubernetes cluster. That would require setting up an ingress, load balancer, external volumes, secrets storage and many other things which are beyond the scope of this thesis.

By integrating often and thoroughly testing every commit both developers and operations become more confident with the releases. Bugs can be simpler to find due to the iterative nature of this approach. Proper continuous integration means that:

- Everybody commits to mainline at least once a day.
- Reliable automated tests keep us confident that we can find any bug.
- If the automated tests fail, no one has a more important job, than fixing it. [29]

3.2 Continuous delivery

Continuous delivery is almost the automation of deployment. Every commit is built into an artifact with CI. **Continuous delivery describes the ability to automatically deploy said artifacts to production.**

Every company has a specific need when it comes to deployment. There might be a testing environment for quality assurance (QA) or perhaps multiple staging environments before production. While doing continuous delivery releasing to any one of those environments should be automated or only a matter of a single mouse click.

„*The decision which commit to deploy to production is based on business or marketing needs, and it has nothing to do with engineers.*“

– Viktor Farcic [31]

Releasing to production does not have to be automatic. Some might get scared, some might have other reasons. That's not important. Continuous delivery only means that **every commit can be deployed to production.** [31], [32], [29]

3.3 Continuous deployment

Continuous deployment takes continuous delivery one step further. Every change is deployed to production. Automatically. No human intervention needed. As mentioned above, this might not be a good idea based on marketing needs. However, many big names in the world of technology are following this path.

Werner Vogels (the CTO of Amazon.com) writes that amazon deploys on average less than every second. [4] A recent research conducted by Google shows that the number of companies deploying multiple times per day tripled in the last year. [33]

3.4 CI/CD & Kubernetes

How does it all fit together? What are the parts, tools, programs that need to click the create the pipeline? Individual tools are going to be highlighted with semibold-font and a unique color to assist the reader in the chapter 4.

3.4.1 Git repository manager

Just like any other development environment, everything starts with version control. Git is the industry standard. Git itself is open-source but git itself is not enough. We need a **Git repository manager**. It provides us with GUI for the management of multiple repositories and users. Git managers often integrate other useful features such as **issue tracking** or wiki management. Having a well-organized git repository manager helps to integrate code as well as new colleagues. [34]

Most importantly, Git managers contain integrations to other tools. For example. If our continuous integration tool fails a build it can reach back to the git manager to display a red mark next to the commit signifying it wasn't able to build it. Integrations can range from very simple (on every merge send HTTP POST) to very complex (a deploy to production button next to every commit).

An important note here. As we mentioned in the continuous integration section 3.1 everybody pushes to mainline. That does not mean one repository. It's still a microservice architecture. Each developer integrates into the repository of their microservice. **That microservice is built and deployed independently on other microservices.**

3.4.2 Continuous integration & Kubernetes

Once code is pushed to the repository it is good practice to let other developers take a look. When it reaches some level of approval (thumbs-ups) it is then moved along the continuous integration pipeline.

What happens and in what order is determined by the company/team culture. The code might go through static code analysis to reveal some bugs or insight. Or it can go through steps automatically generating documentation, installers, etc. This can and should be tailored to every company's needs.

The crucial parts of continuous integration are building the MS and sunbsequent automatic testing.

Every team is responsible for its own MS, including how it is built. In practice, this normally means writing a Dockerfile in the MS's repository. The Dockerfile explains how an image is built from the repository. The developers themselves define the dependencies in the Dockerfile. The whole process is transparent, reproducible and versioned in Git.

The pipeline builds the Docker image and pushes it inside a Docker registry. From there, automatic testing downloads it and runs tests on the built MS. The result should then be reported back in some way.

3.4.3 Registry or repository?

Let us step back for a moment now and explain the chaotic terminology surrounding registries.

- **Docker repository** stores, provides, and versions one specific Docker image. Similarly to how Git repository stores, versions, and provides source-code.
- Docker registry is an open-source program, that stores and lets you distribute Docker images. [16] Docker registry usually contains multiple Docker repositories.
- Container registry is a term used for any program that can store and serve container images. Docker registry is a specific container registry. Kubernetes needs a container registry to pull container images from.
- Artifact repository is a program that can store, version and provide packages for multiple technologies. Artifact repository can, for example, work with RPM packages, .war files, binaries or... Docker images. Artifact repository can thus function as, but not only as, a container registry.

3.4.4 GitOps

GitOps is a term coined a few years back by Weaweworks.[35] It redefines the approach to continuous deployment. The files describing the desired state of Kubernetes are saved in a Git repository. A GitOps operator is watching this repository. When it

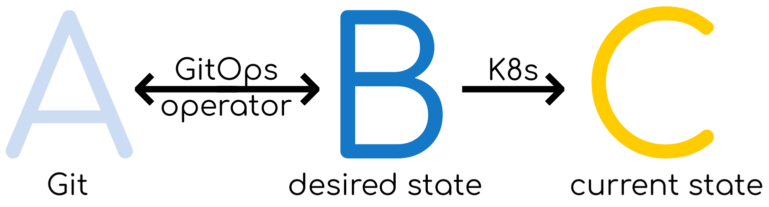


Figure 3.1. GitOps diagram

detects a change it passes the new desired state to Kubernetes. Kubernetes at its core always tries to make the current state identical to the desired one. See figure 3.1

The GitOps operator does not just transfer the new configuration. It also cleans up resources previously created and now no longer desired. The goal is to synchronize the desired state of Kubernetes with the state declared in Git.

3.4.5 Continuous deployment & Kubernetes

Continuous deployment is in theory fairly straight forward. Kubernetes desired state is configured in such a way it deploys the desired image from our registry. In practice, there are many different approaches. It greatly depends on the tool used and chosen tactics. Some of the most popular advanced deployment strategies are.

- Canary testing – New code is deployed to small number of unaware users. Actual usage is analyzed.
- A/B Testing – Data-driven experimenting on users. Traffic is routed to users selected through criteria. The behavior of users with new code and old-code is measured and statistically evaluated.
- Blue/Green – New code is deployed alongside the old code. Once verified to work, the traffic is sent to the new code. Then the old code is destroyed.
- Rolling deployment – Pod by pod deploy the new code and reroute parts of traffic to it.

3.4.6 Monitoring & Kubernetes

An equally important part plays **monitoring and logging**. If we don't know what is going on our resilience in case of emergency crumbles. For that reason, Kubernetes itself should be monitored. We want some kind of overview of the current state vs the desired state.

But not only that. We also hope to know how the individual nodes are doing. Knowledge of the CPU or RAM usage can be useful in debugging scenarios. This problem should be (at least partly) solved by properly configuring Kubernetes not to allow pods to use beyond their limits.

Last but not least, we need to monitor individual MSs. Are they running? Do they have traffic? Are they overloaded? How much are they using? Health checks should be configured in Kubernetes for every pod. Kube-controller periodically checks the health of the pod and if it fails (eg. HTTP not 200) it restarts it or scales it. This all sounds great, but as with any technology, it's probably not a great idea to rely on it 100 % as a black box without monitoring.

„Luck favors the prepared.“

– Proverb

Chapter 4
Open-source tools

One of the goals of this thesis is to research available open-source tools for continuous integration and deployment with Kubernetes. The preceding chapters explained the basics of all the underlying technology. We are building a modern application, designed as microservices. Microservices help to divide the functionality into manageable chunks that are independent of each other.

A developer makes a change into the code. It does not matter if we are talking about a small new feature or about a bug fix. The CI/CD pipeline should start. The CI part means that the code is then automatically built, tested, and evaluated by all sorts of methods. If it passes we are confident that the commit/push will not break the application, thus it is included, integrated, into the codebase.

The CD part is all about the road to production. The built MS, in the form of a container is deployed to various environments depending on business processes or customer needs. Everything is automated and deploying to production is only a matter of a single mouse click.

There is a lot of open-source projects that cover various parts of this process. This chapter summarizes the most popular ones. It includes but is not limited to tools found in the Cloud Native Computing Foundation (CNCF) interactive landscape. [36] Tools affiliated with CNCF are marked with a star symbol (★).

4.1 Methodology

Due to the sheer amount of available tools we had to decide which ones to include and which ones to leave out. The following were the criteria used when deciding:

- The tool must be open-source.
- No discontinued projects. There should be recent commits to the tool repository.
- Usage. The more people are actively using it, the better.
- Integrations. The more a tool can integrate with other tools, the better.
- Relevance to usage with Kubernetes.

Sorting the tools can be very problematic. Some of them cover multiple parts of the pipeline at one time. The tools are sorted into groups based on the way they are presenting themselves. Inside the groups similar tools tend to be closer together. The groups are ordered by the general flow of the pipeline. Every tool is displayed with:

- A motto and a logo – cited from the tool site
- A URL of the tool site and repository
- The license of the source-code
- Endorsed by – notable companies using the tool
- Used for – summary of color-coded usages (What parts of the pipeline it covers)
- Pricing model – open-source does not have to mean free of charge for everything
- Notes – additional information about the tool (lessons, quirks, certifications, etc.) presents a personal opinion.

4.2 All the parts of a pipeline

All the different components of a pipeline can leave the reader confused. Don't worry. For this reason, there is a color-coded summary of everything needed to build a proper pipeline. Every piece has a short list of its purposes inside the pipeline. A modern CI/CD pipeline usually contains most of the following:

- Git manager
  - Provides access to Git repository
  - Shows the repository in a GUI, usually together with issues.
  - Shows/contains code of/links to wiki
  - Shows status of the pipeline for each commit/push/merge
- Issues manager
  - Tracks issues, bugs, requests, etc...
  - Can also track time, comments, and upvotes
- Continuous integration tool
  - Performs static code analysis
  - Builds code into artifacts (eg. compilation) and pushes them to artifact repository
  - Builds container images and pushes them to container registry
  - Generates other useful files such as documentation, installer, or wiki
  - Evaluates the code quality
  - Performs extensive testing of built application
  - Performs security and vulnerability analysis
  - Gives feedback on what its doing
- Artifact repository
  - Stores, versions and provides artifacts (anything from binary to container image)
  - Container registry stores, versions and provides container images. That usually means Docker images.
- Continuous delivery tool
  - Changes or shows changes to the desired/current state of Kubernetes
  - Deploys app to predefiend environments (QA, staging, production, etc.)
  - Handles deployment strategies (Canary, Blue/Green, etc.)
- Kubernetes
  - Manages a cluster of nodes (computers)
  - Schedules changes to match the desired state („configuration“)
  - Pulls and deploys images from container registry as pods onto nodes
  - Manages traffic to pods
- Monitoring & logging
  - Collects logs and metrics from pods
  - Visualizes metrics about cluster and pods
  - Alerts in case of irregularities

4.3 Git managers & Issues

4.3.1 Gitlab

GitLab is a complete DevOps platform.

Site: `about.gitlab.com` **Endorsed by:** Nasa, Goldman Sachs
Repo: `gitlab.com/gitlab-org/gitlab` **License:** MIT License
Used for: Git manager, Issues, CI, CD, Artifact repository, Testing, Container registry, Monitoring, Code quality, Wiki, Code review

Pricing model: GitLab uses the open-core model. GitLab Community Edition is open-source and free to use. GitLab Enterprise Edition contains additional features and is paid. GitLab offers both self-managed and SaaS solutions for its Enterprise Edition.

Notes: The Community Edition contains a lot of features and is very usable. Great for getting started fast. GitLab is widely used and offers numerous integrations with other tools. ★ CNCF silver member.

4.3.2 Gitea

Git with a cup of tea.

Site: `gitea.io` **Endorsed by:** DiDi Cloud, DigitalOcean
Repo: `github.com/go-gitea/gitea` **License:** MIT
Used for: Git manager, Issues, Wiki

Pricing model: Gitea is open-source and free of charge.

Notes: Gitea is fast and lightweight, but at the same time very featureful. For example, it provides OAuth2 and FIDO U2F (2FA) authentication. In addition, the Gitea community curates a list of its „awesome“ integrations with other CI/CD tools[1]. Gitea is a fork of Gogs.

4.3.3 Gogs

A painless self-hosted Git service.

Site: `gogs.io` **Endorsed by:** igt, University of Mississippi
Repo: `github.com/gogs/gogs` **License:** MIT
Used for: Git manager, Issues, Wiki

Pricing model: Gogs is open-source and free of charge.

Notes: Gogs is a fast, lightweight, and intuitive Git manager. Its built-in issues are very simple, but it can work with external issue tracking systems. Gogs supports webhooks but does not boast with many comprehensive 3rd party integrations with other tools.

[1] `https://gitea.com/gitea/awesome-gitea/src/branch/master/README.md`

4.3.4 Phabricator

Discuss. Plan. Code. Review. Test.

Site: `phacility.com/phabricator` **Endorsed by:** Wikimedia, KDE
Repo: `github.com/phacility/phabricator` **License:** Apache v2.0
Used for: Git manager, Issues, CI, Project management, Wiki, Code review

Pricing model: Phabricator is open-source and free of charge. Phacility, the company maintaining Phabricator, offers hosted and Enterprise support plans.

Notes: Phabricator is a very mature platform that integrates project management features, issues, wiki, and version control. It is highly customizable and featureful. However, not very modern.

4.4 CI tools

4.4.1 Buildbot

The Continuous Integration Framework

Site: `buildbot.net` **Endorsed by:** Chromium, Blender
Repo: `github.com/buildbot/buildbot` **License:** GNU v2.0
Used for: CI, Automation

Pricing model: Buildbot is open-source and free of charge.

Notes: Buildbot is a comprehensive automation framework. It provides an abstraction of general automation for CI. Its versatility makes it a great fit for gigantic complex projects.

4.4.2 Travis CI

Test and Deploy with Confidence

Site: `travis-ci.org` **Endorsed by:** Heroku, BitTorrent
Repo: `github.com/travis-ci/travis-ci` **License:** MIT
Used for: CI, CD

Pricing model: Travis CI is technically open-source but the lack of even minimal support makes it very hard to deploy on your own. Travis offers a plethora of plans for small and big customers. Using Travis CI as a service is free for open-source projects. Travis CI Enterprise offers an on-premise solution.

Notes: Travis is a modern, widely used platform for frictionless CI. It is not intended to be used with Kubernetes, although it can be accomplished.

4.4.3 Concourse

Concourse is an open-source continuous thing-doer

Site: `concourse-ci.org` **Endorsed by:** Nasdaq, Home Depot
Repo: `github.com/concourse/concourse` **License:** Apache v2.0
Used for: CI, Atutomation, Pipeline visualization

Pricing model: Concourse is open-source and free of charge.

Notes: Concourse is an excellent enterprise CI tool for any automation and pipelines. Concourse is very flexible, but it has a steeper learning curve. Also, it's UI is very elegant. ★ CNCF platinum member.

4.4.4 Jenkins

Build great things at any scale

Site: `jenkins.io` **Endorsed by:** Dell, eBay
Repo: `github.com/jenkinsci` **License:** MIT
Used for: CI, CD, Automation

Pricing model: Jenkins is open-source and free of charge. A contributing company called CloudBees offers its flavor of Jenkins called CloudBees Jenkins with a paid support.

Notes: Jenkins is an open-source automation tool. It has been released in 2011 and its UI changed very little since. The sheer amount of community plugins make Jenkins the „king of 3rd party integrations". Jenkins is the most used CI/CD tool. [3] However, Jenkins can prove to be difficult to maintain at a big scale. [37] ★ CD foundation graduated project.

4.4.5 Jenkins Blue Ocean

Continuous Delivery for every team

Site: `jenkins.io/projects/blueocean` **Endorsed by:** Intuit
Repo: `github.com/jenkinsci/blueocean-plugin` **License:** MIT
Used for: CI, CD, Automation

Pricing model: Jenkins Blue Ocean is open-source and free of charge.

Notes: Jenkins Blue Ocean is a very pleasant user interface for Jenkins pipelines. It receives constant updates, but so far, it cannot fully replace the classic UI. All the other downsides of regular Jenkins still apply.

4.5 CI & CD tools

4.5.1 Drone

Automate Software Testing and Delivery

Site: `drone.io` **Endorsed by:** Cisco, eBay
Repo: `github.com/drone/drone` **License:** Apache v2.0
Used for: CI, CD, Testing

Pricing model: Drone comes in two versions. A community edition and an enterprise edition. The community edition is free and open-source for everyone. It lacks support for advanced databases, non–docker runners, and advanced secret management. The enterprise edition is source–available, without limitations, and is free of charge for smaller companies (under 1 mil USD).

Notes: Drone is an easy to use tool with generous pricing policy. It integrates very well with other tools and has a no–nonsense approach. Drone is lightweight, fast, but incredibly powerful. A Drone pipeline is a declarative YAML, and it can even be run locally by a developer. That eliminates the common case of „it worked on my machine“.

4.5.2 GoCD

Free & Open Source CI/CD Server

Site: `gocd.org` **Endorsed by:** ThoughtWorks
Repo: `github.com/gocd/gocd` **License:** Apache v2.0
Used for: CI, CD, Artifact repository, Pipeline visualization

Pricing model: GoCD is open-source and free of charge. GoCD includes third-party software with various licenses. ThoughtWorks offers paid enterprise support.

Notes: GoCD is a less-known CI/CD tool. It has been open-sourced by ThoughtWorks in 2014. Declarative pipelines, unlimited „runners“, and built-in Kubernetes support make it a great fit for any modern open-source CI/CD pipeline.

4.5.3 Agola

CI/CD redefined

Site: `agola.io` **Endorsed by:** –
Repo: `github.com/agola-io/agola` **License:** Apache v2.0
Used for: CI, CD

Pricing model: Agola is open-source and free of charge.

Notes: Agola originated at sorint.oss in spring 2019. [38] That means the project is only a year old at the time of writing with only 5 contributors. The project holds great promise, but only time will show if it develops a community. It promises and delivers highly available and scalable CI/CD driven by Git workflow. Agola has built-in support for Kubernetes as an execution platform.

4.5.4 Tekton

Kubernetes-native CI/CD

Site: `tekton.dev`
Endorsed by: Google
Repo: `github.com/tektoncd`
License: Apache v2.0
Used for: CI, CD

Pricing model: Tekton is open-source and free of charge.

Notes: Tekton is very different from other tools listed here. Tekton is Kubernetes-native. It defines custom resources (CRDs) inside Kubernetes for running pipelines. The pipelines are then available in the Kubernetes API. This way a pipeline execution can be unified throughout various tools. ★ CD foundation graduated project.

4.5.5 Jenkins X

Accelerate Your Continuous Delivery on Kubernetes

Site: `jenkins-x.io`
Endorsed by: CloudBees
Repo: `github.com/jenkins-x/jx`
License: Apache v2.0
Used for: CI, CD, Container registry, Kubernetes

Pricing model: Jenkins X is open-source and free of charge.

Notes: Jenkins X is very different from other tools listed here. Jenkins X connects to (or creates) a Kubernetes cluster. There, it creates a DevOps platform with a complete CI/CD pipeline ready for usage. It integrates with various Git providers. ★ CD foundation graduated project.

4.6 CD tools

4.6.1 Argo CD

Argo CD is a declarative, GitOps continuous delivery tool for Kubernetes.

Site: `argoproj.github.io/argo-cd`
Endorsed by: Intuit
Repo: `github.com/argoproj/argo-cd`
License: Apache v2.0
Used for: CD, Advanced deployment strategies, GitOps

Pricing model: Argo CD is open-source and free of charge.

Notes: Argo ensures that the current state of a Kubernetes cluster is equal to a state defined by files in a connected Git repository. Such practice is called GitOps. Argo is a featureful tool with excellent security and UI. ★ CNCF incubating project.

4.6.2 Flux

The GitOps operator for Kubernetes

Site: `fluxcd.io` **Endorsed by:** Weave cloud, Starbucks
Repo: `github.com/fluxcd/flux` **License:** Apache v2.0
Used for: CD, GitOps

Pricing model: Flux is open-source and free of charge.

Notes: Flux ensures that the current desired state of a Kubernetes cluster is equal to a state defined by YAML files in a connected Git repository. Such practice is called GitOps. It can effectively eliminate the need for a separate CD tool. Flux does not have a UI. ★ CNCF sandbox project.

4.6.3 Flagger

Progressive Delivery Operator for Kubernetes

Site: `flagger.app` **Endorsed by:** Weave cloud
Repo: `github.com/weaveworks/flagger` **License:** Apache v2.0
Used for: CD, Advanced deployment strategies

Pricing model: Flagger is open-source and free of charge.

Notes: Flagger is a Kubernetes operator – software extension to Kubernetes. It implements advanced deployment strategies like Canary, A/B Testing, and Blue/Green. It can also send notifications to Slack or Microsoft Teams.

4.6.4 Keptn

Keptn is a control-plane for DevOps automation of cloud-native applications.

Site: `keptn.sh` **Endorsed by:** Dynatrace, Citrix
Repo: `github.com/keptn/keptn` **License:** Apache v2.0
Used for: CD, Advanced deployment strategies, (GitOps)

Pricing model: Keptn is open-source and free of charge.

Notes: Keptn is a tool specifically designed for continuous delivery. It implements numerous deployment strategies. The application delivery flow is defined in a (shipyard – Keptn proposed standard) YAML file and stored in Git. ★ CNCF silver member.

4.6.5 Spinnaker

Fast, safe, repeatable deployments for every Enterprise

Site: `spinnaker.io` **Endorsed by:** Adobe, SAP, [and many more]
Repo: `github.com/spinnaker` **License:** Apache v2.0
Used for: CD, Advanced deployment strategies

Pricing model: Spinnaker is open-source and free of charge. Unrelated companies (Armory, OpsMx) provide paid support and Spinnaker as a service.

Notes: Spinnaker is a cloud-native enterprise-ready tool for CI/CD. Originally developed by Netflix, it is currently embraced and supported by some of the biggest names in current IT. Spinnaker enables zero-downtime deployment strategies for mission-critical systems. ★ CD foundation graduated project.

4.7 Container registry manager

4.7.1 Harbor

Our mission is to be the trusted cloud native repository for Kubernetes.

Site: `goharbor.io` **Endorsed by:** Axa, China Mobile
Repo: `github.com/goharbor/harbor` **License:** Apache v2.0
Used for: Container registry, Security and vulnerability analysis (with Clair)

Pricing model: Harbor is open-source and free of charge.

Notes: Harbor is a container registry designed to be secure and robust. It provides support for LDAP and OpenID authentication. Harbor also offers a sophisticated project structure with user roles, quotas, and automatic container vulnerability scanning. ★ CNCF incubating project.

4.7.2 Portus

Claim control of your Docker images.

Site: `port.us.org` **Endorsed by:** SUSE team
Repo: `github.com/SUSE/Portus` **License:** Apache v2.0
Used for: Container registry (UI), Security and vulnerability analysis (with Clair)

Pricing model: Portus is open-source and free of charge.

Notes: Portus is not a container registry. It is a UI and an authorization service for a container registry. Currently, it connects to a single container registry. Nonetheless, supported features such as LDAP, OAuth, auditing, and vulnerability scanning make it a worthy competitor. ★ CNCF gold member.

4.7.3 Project Quay

Quay [builds, analyzes, distributes] your container images.

Site: `projectquay.io` **Endorsed by:** Nasa, eBay, RedHat
Repo: `github.com/quay/quay` **License:** Apache v2.0
Used for: Container registry, Security and vulnerability analysis (with Clair)

Pricing model: Project Quay is an open-source project behind the paid hosted service quay.io and RedHat Quay. Quay Enterprise was renamed to RedHat Quay after RedHat bought it's creator CoreOS. RedHat Quay is currently sold by RedHat and included in OpenShift.

Notes: Quay is a true enterprise tool with support for the majority of use-cases. Even though Quay's functionality may be considered superior to other tools there is a problem with it. The whole project is entangled with RedHat technology which makes it harder to utilize without it. ★ CNCF platinum member.

4.8 Code quality tools

4.8.1 Clair

Automatic container vulnerability and security scanning for appc and Docker

Site: – **Endorsed by:** RedHat, Quay.io
Repo: `github.com/quay/clair` **License:** Apache v2.0
Used for: Analysis of container vulnerabilities

Pricing model: Clair is open-source and free of charge. Clair was developed by CoreOS and included in Quay Enterprise which all now belong to RedHat. Clair is currently integrated into RedHat Quay Security Scanner but can be downloaded and run on its own.

Notes: There is no official Clair site apart from its GitHub repository README. Clair is a mature tool used in the enterprise. Clair does not have a UI, instead, it is controlled by its API. Numerous 3rd party tools integrate with Clair. ★ CNCF platinum member.

4.8.2 Anchore Engine

An Open-source Tool For Deep Image Inspection And Vulnerability Scanning

Site: `anchore.com/opensource` **Endorsed by:** US Air Force
Repo: `github.com/anchore/anchore-engine` **License:** Apache v2.0
Used for: Analysis of container vulnerabilities

Pricing model: Anchore Engine is open-source and free of charge. The company Anchore also provides paid versions called Achore Enterprise and Anchore Federal for more demanding customers. Anchore Federal is used in the most critical air-gapped environments, for example by the US Air Force.

Notes: Anchore Engine provides a truly able tool for Docker container security scanning. As mentioned before, it is used even in some of the most critical systems. The open-source version does not have a UI, although it comes with a very handy CLI and REST API.

4.8.3 SonarQube

SonarQube empowers all developers to write cleaner and safer code

Site: `sonarqube.org` **Endorsed by:** Erste, DB
Repo: `github.com/SonarSource/sonarqube` **License:** LGPL 3.0
Used for: Code review, Bug detection, Vulnerability detection

Pricing model: SonarQube comes in four versions Community, Developer, Enterprise, and Data Center. The Community Edition is open-source and free of charge, however, it is missing support for some programming languages, branch analysis, or Git integrated pull request decorations.

Notes: SonarQube is a great and thorough tool for static code analysis. It can detect, measure, and visualize the number of bugs, vulnerabilities, or code coverage. The UI is clean and very pleasant. SonarQube is used by 58 of the fortune 100 companies. [39] Open-source projects can use the hosted SonarCloud for free.

4.9 Monitoring

4.9.1 Kubernetes dashboard

Dashboard is a web-based Kubernetes user interface.

Site: `kubernetes.io`
Repo: `github.com/kubernetes/dashboard`
Used for: Monitoring
Endorsed by: –
License: Apache v2.0

Pricing model: Kubernetes dashboard is open-source and free of charge.

Notes: Kubernetes Dashboard (or Kubernetes UI) is an application from the makers of Kubernetes. It deploys directly on Kubernetes and provides a simple material dashboard. The Dashboard shows an overview of Kubernetes resources (pods, deployments, services). It also shows an overview of resources such as CPU or RAM usage. The Kubernetes Dashboard can control some fundamental Kubernetes tasks such as scaling. It is a great tool for getting a grip on what is going on in the cluster.

4.9.2 Prometheus

From metrics to insight

Site: `prometheus.io`
Repo: `github.com/prometheus`
Used for: Monitoring, Alerting
Endorsed by: DigitalOcean, SoundCloud
License: Apache v2.0

Pricing model: Prometheus is open-source and free of charge.

Notes: Prometheus is a monitoring service that is designed to fit well in the dynamic world of microservices. Prometheus actively collects metrics from each part of the application and saves them in a time-series database. It then allows querying on collected data with its language PromQL. For visualization, it integrates very well with Grafana. ★ CNCF graduated project.

4.9.3 Graphana

The open observability platform

Site: `grafana.com`
Repo: `github.com/grafana/grafana`
Used for: Monitoring, Alerting, Data visualization
Endorsed by: PayPal, Intel, Booking.com
License: Apache v2.0

Pricing model: Grafana is open-source and free of charge. Grafana Labs offer Grafana Enterprise with authentication plugins and commercial support. Grafana Cloud is a paid SaaS solution including (but not limited to) Grafana and Prometheus.

Notes: Grafana connects to various data sources (eg. Prometheus) and displays the data. Users can create beautiful dashboards with custom graphs and visualizations, or choose from hundreds of community prepared ones. ★ CNCF silver member.

4.9.4 cAdvisor

Analyzes resource usage and performance characteristics of running containers

Site: –
Repo: `github.com/google/cadvisor`
Used for: Monitoring
Endorsed by: Google
License: Apache v2.0

Pricing model: cAdvisor is open-source and free of charge.

Notes: cAdvisor is an open-source tool from Google used for monitoring running containers. It used to be accessible with UI on a Kubelet – that feature was removed. As of early 2020 cAdvisor endpoints are disabled in Kubernetes by default. cAdvisor is becoming an implementation-specific detail. In newer versions, this will be provided by the CRI layer (Container Runtime Interface) – Kubernetes being compatible with a number of „container providers“, not just Docker. Monitoring containers in Kubernetes is possible through **kubectl top**.

4.9.5 Fluentd

Build Your Unified Logging Layer

Site: `fluentd.org`
Repo: `github.com/fluent/fluentd`
Used for: Logging
Endorsed by: Microsoft, Nintendo
License: Apache v2.0

Pricing model: Fluentd is open-source and free of charge.

Notes: Fluentd centralizes data log collection from various sources. Because it was written in C and Ruby, it is extremely lightweight and fast. Fluentd can be integrated with Kubernetes and extended with many additional plugins. Fluentd is a tool used by many companies in the enterprise. ★ CNCF graduated project.

4.10 Other useful tools

4.10.1 Helm

The package manager for Kubernetes

Site: `helm.sh`
Repo: `github.com/helm`
Used for: –
Endorsed by: Google, IBM, Microsoft
License: Apache v2.0

Pricing model: Helm is open-source and free of charge.

Notes: Every common Unix OS has repositories with applications prepared for it. Helm makes the same thing possible with Kubernetes. Helm parametrizes the Kubernetes YAML files defining our application (services, deployments, etc) and packages then into so-called Helm charts. Helm charts of popular applications created by the community can be downloaded from Helm „repositories“ and deployed to any Kubernetes cluster. ★ CNCF graduated project.

4.10.2 Falco

Cloud-Native Runtime Security

Site: falco.org
Repo: github.com/falcosecurity/falco
Used for: –
Endorsed by: –
License: Apache v2.0

Pricing model: Falco is open-source and free of charge. Sysdig offers a customized enterprise solution.

Notes: Falco monitors Kubernetes security at runtime. It is the first cloud-native runtime security project to join the CNCF. ★ CNCF incubating project.

4.10.3 Utilities & other helpful projects

Artifactory OSS	`jfrog.com/open-source` JFrog Artifactory Open Source For Artifact Management
n8n.io	`n8n.io/` Free and Open Workflow Automation Tool.
Jaeger	`www.jaegertracing.io/` Open source, end-to-end distributed tracing
kube-shell	`github.com/cloudnativelabs/kube-shell` An integrated shell for working with the Kubernetes CLI
kubebox	`github.com/cloudnativelabs/kube-shell` Terminal and Web console for Kubernetes
Searchlight	`github.com/searchlight/searchlight` Alerts for Kubernetes
Kube-monkey	`github.com/asobti/kube-monkey` An implementation of Netflix's Chaos Monkey for K8s clusters

Chapter 5
Open-source CI/CD pipeline design

Let us start with building pipelines. A CI/CD pipeline is an integral part of every team's technology stack. It helps to automate routine tasks and eliminate errors in the whole process. Thus speeding up the time it takes new code to get to the customer. A brief summary of the parts present in a CI/CD pipeline to Kubernetes is available in the section 4.2

First, we will take a look at our developer's environment and analyze what pipeline we want to create. In reality, this would include talking to our developers and administrators to determine, how should the pipeline behave. We, sadly cannot do that. Nonetheless, an analysis of the deployed MSs is in order. We are deploying an open-source microservice application called TeaStore written in Java. It's a simple e-shop selling teas.

Subsequently, we will design two pipelines from some of the open-source tools described in the previous chapter. **The designed pipelines will cover the entire process starting with a developer's commit to a Git repository and finishing with the whole MS application deployed in a Kubernetes cluster.** Both of the pipelines shall be constructed exclusively from open-source tools.

Trying to deploy an app with only open-source tools is not very realistic, nevertheless, there are reasons to do this. Every pipeline should be tailored specifically to the needs of its users. Showing off what the open-source tools are able to do can provide a better overview of what parts of the pipeline to buy.

5.1 Test enviroment

The first part of building a pipeline is to take a step back and make an overview of what technologies are used by our developers (eg. programming languages) and what the current build process is. And that is exactly what this chapter is going to be about. The only difference is, that the developers of our MS app (TeaStore) are in Würzburg, Germany and not next doors.

5.1.1 Teastore breakdown

TeaStore is a mockup MS application written in Java – a simple e-shop selling all sorts of teas. It consists of five microservices and a registry (shown in figure 5.1).

- WebUI – The MS serving the e-shop website.
- Image – A service that provides generated images of teas when queried.
- Auth – Users can log-in to the e-shop to buy teas. This MS is responsible for the verification of both login and the session data.
- Recommender – The Recommender recommends other items for purchase based on various factors.
- Persistence – Provides access to the store's relational database.
- Registry – Keeps track of what instances of services are running.

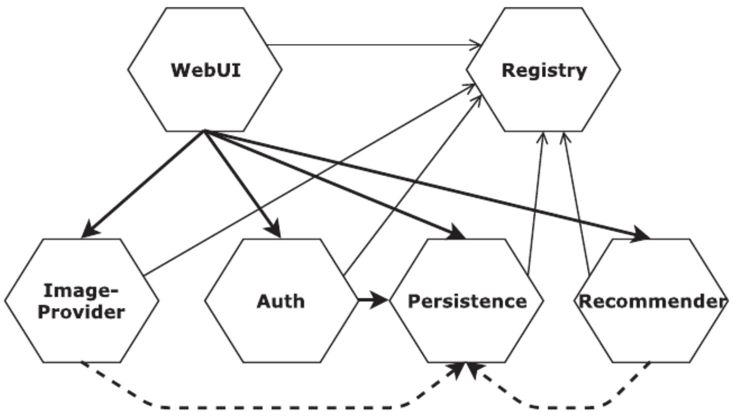


Figure 5.1. TeaStore architecture [15]

The Java source code is compiled by Maven into a .war file. That .war file is then packaged into a Docker container and the container is pushed into a registry. Authors of TeaStore provide the Maven pom.xml and a Dockerfile for each MS. They also provide example scripts for building the Docker images and Kubernetes .yaml files for deploying.

For greater agility when trying out new CI/CD tools, the WebUI MS was separated from the central repository and reconfigured to be able to build on its own. It better reflects how each MS should be built independently.

The Maven build script, Dockerfile, and Kubernetes manifest have been specifically modified to suit the purposes of the pipeline. The changes were needed to reflect the separation of one service from the central repository.

5.1.2 Kind

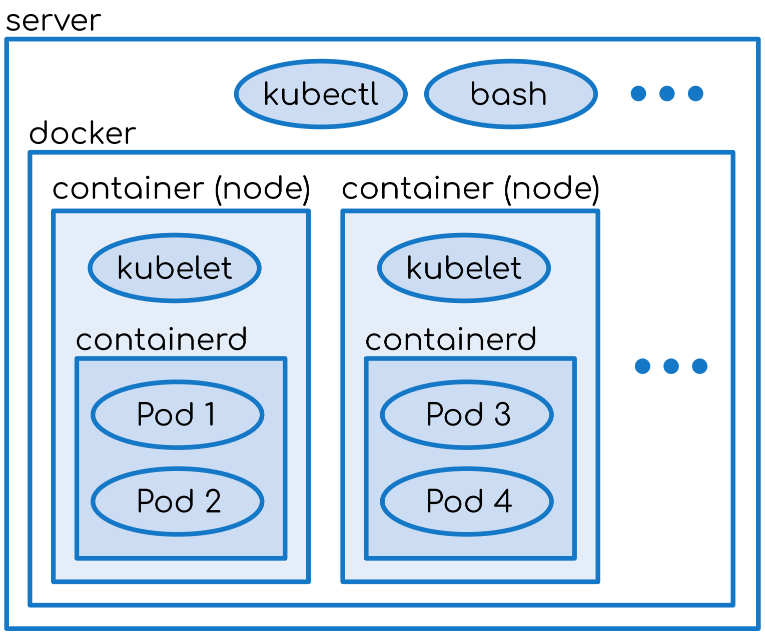


Figure 5.2. Kubernetes in Docker architecture

There are many ways to obtain a Kubernetes cluster. The way chosen here is to create a K8s cluster inside docker containers with a tool called Kind.

Kind[1] is an open-source tool developed for testing Kubernetes. Kind is short for Kubernetes in Docker. Its primary benefit is the fact that it can start a whole Kubernetes cluster in a few minutes. Instead of using virtual (or physical) machines for the nodes, it uses Docker containers with predefined „node images" based on the latest Ubuntu image. The nodes themselves deploy pods as containers inside the node container. As shown in figure 5.2. The containers inside containers are managed with containerd. Containerd is a container runtime based on Docker engine (section 2.3.1).

There are several downsides to this method. It's fast and simple to create a cluster but the cluster itself is slow. Multiple layers of virtualization add additional ballast. Moreover, self-signed SSL certificates and other additional configuration has to be either injected inside the container whilst running or pre-built in a custom Kind container image.

In spite of such downsides – in this thesis, we will be trying out lots of different CI/CD tools. The ability to spin up and destroy a Kubernetes cluster without waiting is particularly useful for us. And that is the reason why we will be using kind as a way to create a Kubernetes cluster to deploy into.

5.2 Only GitLab pipeline

The first pipeline consists solely of tools included with GitLab. But why GitLab?

GitLab is an open-source application that covers the whole pipeline. For structured detail see section 4.3.1. According to a recent CNCF survey, GitLab is the second most popular CI/CD tool (at 34 %). [3] Preceded only by Jenkins (section 4.4.4). It is also considered to be the leading provider of self-hosted Git and a DevOps platform. [40], [41]

It's open-source, widely used, stable - covered by a company and it is an „all in one solution". Despite having a product for every part of the pipeline GitLab does not force the user inside its ecosystem. Each user can choose which parts to use and which parts to substitute with 3rd party SW they would like to use. That, combined with their very generous pricing model makes GitLab a great fit for anyone creating a CI/CD pipeline.

GitLab CI/CD solution consists of two independently installed programs. GitLab and GitLab Runners.

5.2.1 GitLab Runners

A GitLab Runner is the „worker" of the system. There are usually multiple Runners connected to a single GitLab. GitLab instructs the runner on jobs as shown in figure 5.3. A job can be anything from compiling a binary to running vulnerability tests. Runner, as the name implies, runs the jobs and sends the results back to GitLab. Separating the Runners from the rest of GitLab is useful for scaling purposes. Runners can be run on and connected from different machines. Hence distributing the computing load.

In this case, the instructions GitLab sends to the GitLab Runner are shell commands for Maven and Docker. The Runner prepares a special environment for the execution. What environment it prepares depends on the Runner's predefined executor. Executors can be for example Docker, Shell, VirtualBox or Kubernetes. A common choice is to execute the build inside a Docker container. It provides a consistent and clean environment. However, it's important to understand the layers it creates. We are building a Docker container inside a Docker container. That could lead to some unexpected behavior. [42], [43] Another note on building inside Docker is about SSL certificates.

[1] `https://kind.sigs.k8s.io`

The Docker container must have access to the certificates it needs for pushing. There are ways to configure this in the Runner itself or we can use a custom Docker image. A custom Docker image is also useful for preparing cached data (such as Maven plugins). Any customized images should be stored securely and privately in a Docker Registry. They should also be subject to a CI pipeline

Inside the prepared environment, the Runner executes the job and shares back the results. The fail or success status of a job is determined by the exit-code of run commands. Runners can also share back artifacts (files) if GitLab specifies it in the job definition.

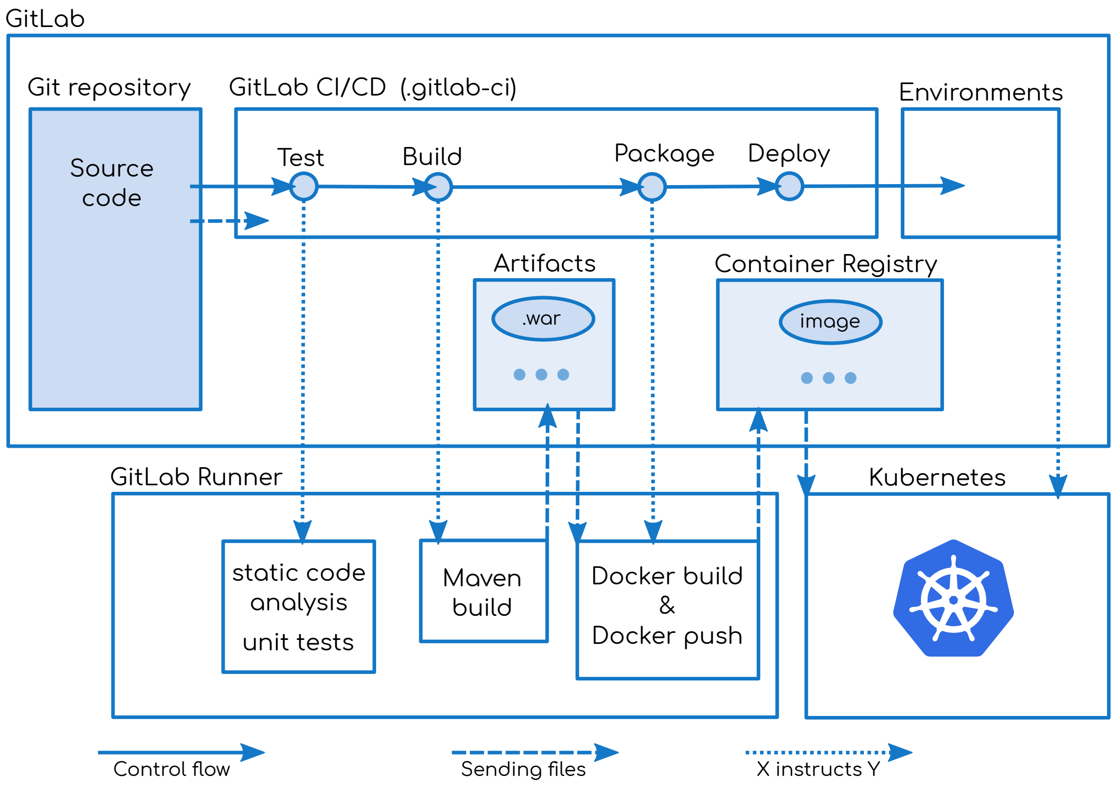


Figure 5.3. Gitlab CI/CD workflow for Teastore

5.2.2 GitLab CI

GitLab manages the Git repository of our MS. If a file called **.gitlab-ci.yml** is present in the root of the repository it starts a pipeline defined by that file. The file needs to be a valid YAML. GitLab recognizes many directives inside it[1].

Firstly we identify the stages of our pipeline. The stage container-test is absent from figure 5.3 to produce a cleaner diagram. There can be multiple stages and multiple jobs for each stage. All the jobs in a stage are run in parallel. Intricate multithreaded pipelines can be drawn using the „**needs**“ keyword in a job definition. An example of the **.gitlab-ci.yml** syntax can be seen in figure 5.4. Once the **.gitlab-ci.yml** is pushed, the UI shows the pipeline running in real-time. A running pipeline can be seen in figure 5.5.

[1] `https://docs.gitlab.com/ce/ci/yaml/`

```
stages:
  - test            # Static code analysis, Create documentation, etc...
  - build           # Maven build and test
  - package         # Building image from Dockerfile
  - container-test  # API testing, Vulnerability scanning, etc...
  - deploy          # Deploying to various environments

# --- test stage ---
documentation:                      # Name of the job
  image: "nxpleuvenjenkins/doxygen" # Docker image of enviroment
  stage: static                     # Stage of the job
  script:
    - doxgen services/webui/Doxyfile
# ...
```

Figure 5.4. A simple example of the **.gitlab-ci.yml** syntax

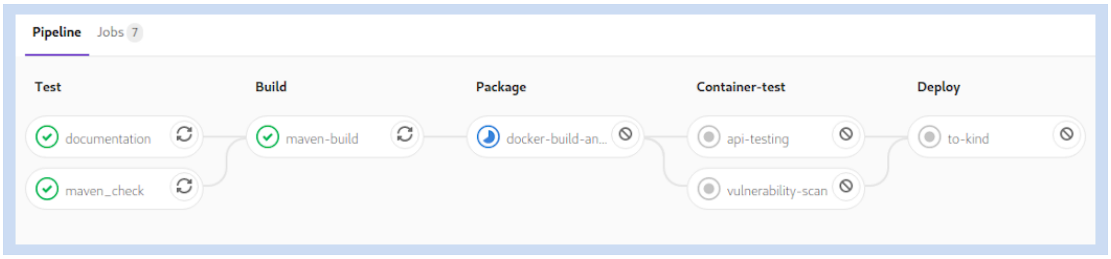


Figure 5.5. GitLab UI of a pipeline

GitLab then instructs runners to execute jobs. It can even show the progress of the commands in a terminal-like output window inside a browser. GitLab already comes with a simple artifact repository. Files that should be considered artifacts and saved alongside a repository must be properly tagged in the job definition. When „Maven build“ finishes it produces a *.war* file. If a Runner would finish the job now, the container execution environment would be destroyed and the hard built *.war* file with it. If we tag the *.war* file as an artifact it is instead copied to the **Job Artifacts** section of GitLab. Every repository has its own Job Artifacts.

Other jobs in the pipeline have access to artifacts. In our pipeline, the **docker-build-and-push** job pulls the artifact and packages it into a Docker image. Specifically, the Docker image builds on top of a Tomcat image with predefined configurations. Docker build copies the *.war* file to the container. The packaging stage continues with Docker pushing an image to the GitLab Container Registry. The configuration YAML file for simple Gitlab artifact management and container registry in the **.gitlab-ci.yml** can be seen in figure 5.6.

The Container Registry in embedded in Gitlab but can be accessed by external applications. When Kubernetes is instructed to pull a new image it pulls it from the GitLab Container Registry. We can define a number of deploy environments for the repository. They are simply connections to servers on which we wish to deploy. Which environment to deploy to is specified in the pipeline definition. The process can be fully automatic (continuous deployment) or it can require a mouse-click input (continuous delivery).

It is possible to let GitLab manage a Kubernetes cluster for us. This provides access to some predefined deployment strategies. However, in the Comunity Edition,

```
# --- Build .war with Maven ---
maven-build:
  image: "maven:3.6-jdk-8"
  stage: build
  script:
    - mvn clean install
  artifacts:
    paths:
    - ./services/tools.descartes.teastore.webui/target/*.war

# --- Build docker image from .war artifact ---
docker-build-and-push:
  stage: package
  script:
    - ls -lh ./services/tools.descartes.teastore.webui/target
    - docker login -u $CI_REGISTRY_USER \
                   -p $CI_REGISTRY_PASSWORD $CI_REGISTRY
    - docker build -t "$CI_REGISTRY_IMAGE" \
                      services/tools.descartes.teastore.webui/
    - docker push "$CI_REGISTRY_IMAGE"
```

Figure 5.6. A simple artifact and container registry configuration in **.gitlab-ci.yml**

we can only add one cluster. Some other features like canary deployment and cluster monitoring are also, missing in this edition.

5.2.3 Pros & cons

GitLab is an excellent tool providing enterprise-ready features. Not everything can be for free, nonetheless, there is a lot of it.

Pros:

- Simple setup
- Stability of „corporate open-source“
- Amazing UI
- Does not force its ecosystem
- Integrates well with 3rd party software

Cons:

- Downsides of „corporate open-source“ (possible fears over future licensing)
- Community edition by itself does not scale to big enterprise
- **One Kubernetes cluster per project** (a cluster for production, staging, development will eventually be needed)
- Integrated tools are not as versatile as individual ones
- Possible single point of failure

5.3 Mixed tools pipeline

In contrast to GitLab's all-in-one approach, the second pipeline is built from five separate tools. The tools used are lightweight yet powerful. The tools were picked with an emphasis on integration and usability. The more the tools can mutually cooperate, the better. It provides Developers with necessary feedback on their code. As much as possible of the pipeline should be versioned. It is good practice not to have configuration lying around.

From the tools researched in chapter 4, we have decided to implement the pipeline with the following tools: Gitea, Drone, Harbor, Clair, Flux. With the sole exception of Drone, the tools are 100 % open-source, maintained by the community, and with no pricing model in place. Drone is free of charge until the company using it has an annual revenue of 1 million US dollars and even after that it has a community version.

The overall diagram of the finished pipeline can be seen in figure 5.7.

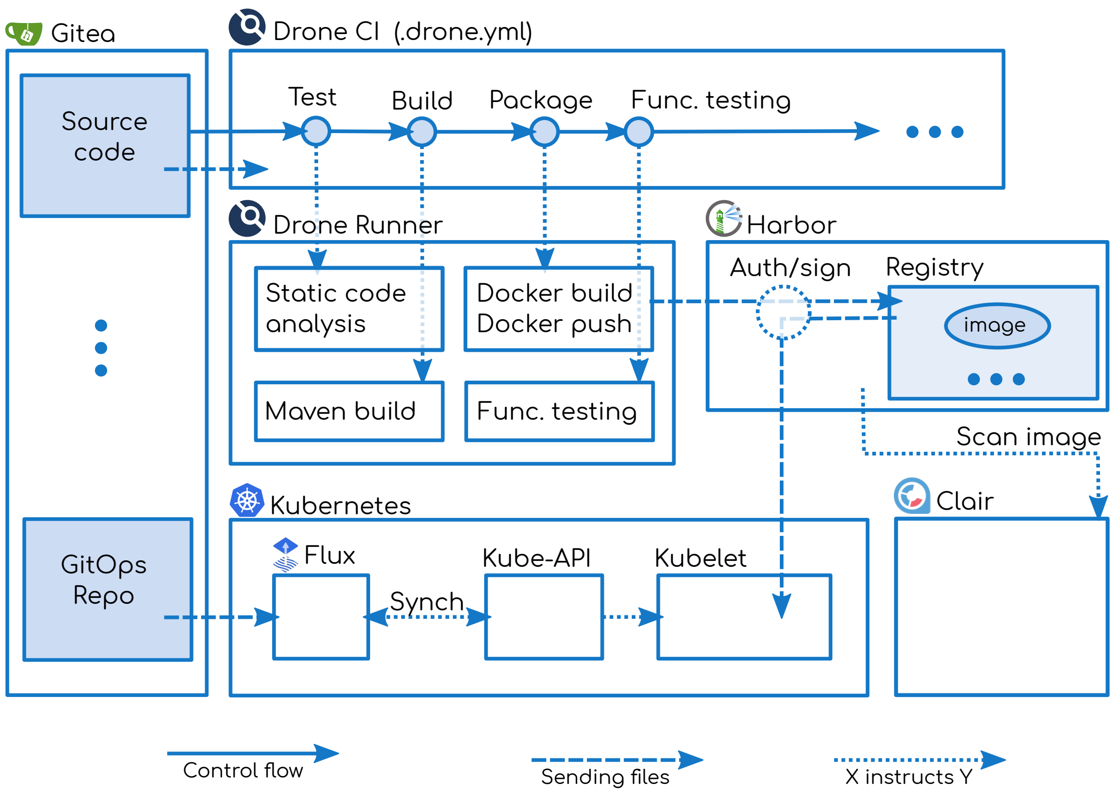


Figure 5.7. Drone UI of a pipeline

5.3.1 Gitea

Gitea is a featureful fork of Gogs written in Go. It provides a Git repository manager with enterprise features like OAuth, LDAP, two-factor authentication, and attention toward issues and merge/pull request workflow. All of that while being light on system resources usage. Gitea is also quite popular and maintained which means there are many integrations with other tools.

Just as in the GitLab pipeline, TeaStore has been separated into individual microservices. Each MS has its repository and inside every repository, there is a **.drone.yml** file. The file describes the pipeline for the next tool: Drone.

Drone itself has no authentication. Instead, it uses OAuth from Gitea. Gitea authenticates a user and authorizes her to her repositories. It then passes this information to Drone.

Apart from the source code itself, Gitea houses another important repository. The GitOps repository. It contains all the Kubernetes YAML files – manifests specifying the desired state of Kubernetes. Flux, the GitOps operator, will be looking for them later.

For now, let's say our developer merged a small feature she developed into main. Gitea is set to send a webhook in case this happens. The webhook is immediately sent to Drone and our pipeline begins.

5.3.2 Drone

Drone is a CI/CD tool written in GO. It is very fast and simple although simultaneously featureful. Drone is not 100 % open-source and community-driven. As previously mentioned it has an enterprise version. However, the benevolent licensing policy does not make this a big issue. If it did, its solid alternative would be Jenkins Blue Ocean which is fully developed by the community. However, Drone (written in GO) is much more lightweight than Jenkins (written in Java). It is true, that Jenkins has more plugins. Nevertheless, Drone integrates very well with Gitea and provides a more modern experience.

We have originally built a pipeline with both. Due to the mentioned reasons we ultimately decided to favor Drone over Jenkins.

Drone is, similarly to GitLab CI, divided into two parts. Drone CI and Runners. The general idea is identical. Multiple runners connect to one CI. The master schedules jobs for them to execute in a specified environment (eg. Shell, Docker, or even Kubernetes). An example of the pipeline definition in the **.drone.yml** file can be seen in figure 5.8

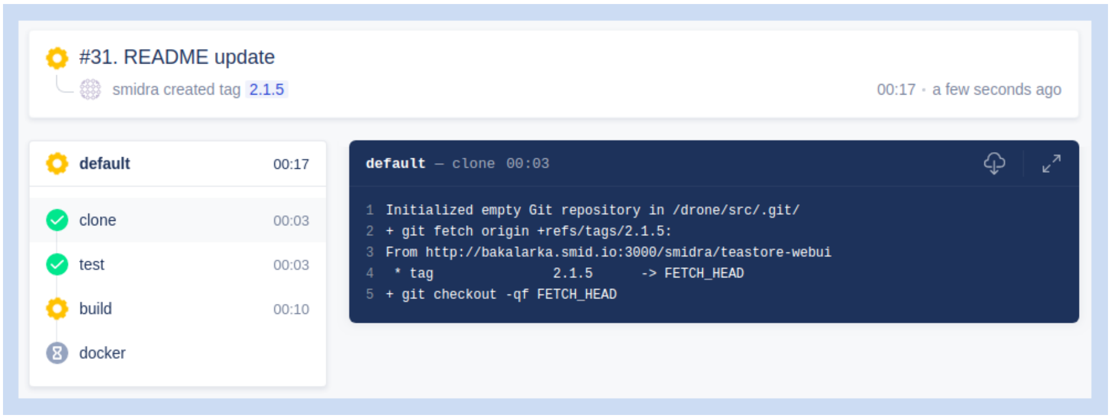
```
# --- Pipeline information ---
kind: pipeline
type: docker
name: webui-pipeline

# --- All the steps ---
steps:
- name:  test                       # Name of the step
  image: nxpleuvenjenkins/doxygen   # Docker image
  commands:
  - doxgen services/webui/Doxyfile

- name: build
  image: ...
# ...
```

Figure 5.8. A sample of .drone-ci.yml file syntax.

The moment a pipeline starts Drone notifies Gitea it received the webhook and started working. To signify that Gitea displays a blinking orange dot next to the commit name. Clicking it redirects the developer to the pipeline overview in Drone. The Drone UI of a running pipeline is captured in figure 5.9


#31. README update

smidra created tag 2.1.5

00:17 · a few seconds ago

default	00:17
clone	00:03
test	00:03
build	00:10
docker	

```
default — clone 00:03

Initialized empty Git repository in /drone/src/.git/
+ git fetch origin +refs/tags/2.1.5:
From http://bakalarka.smid.io:3000/smidra/teastore-webui
 * tag               2.1.5      -> FETCH_HEAD
+ git checkout -qf FETCH_HEAD
```

Figure 5.9. An open-source pipeline.

Drone CI schedules a job on a runner. The runner executes the pipeline stages in succession while streaming console output back to the UI. Additional software could be triggered by the pipeline stages. For example SonarQube for static testing.

Once the **.war** artifact is built it could also be pushed to an external artifact repository. For example Artifactory. In this case, preserving the **.war** artifact is not necessary. Saving the finished Docker image will be enough.

The „docker stage“ builds the Docker container. If the original push was tagged with a version number, Drone recognizes and tags the Docker image too. The finished Docker image is then pushed to a container registry – Harbor. The finished image is also used for various testing (functional, stress, etc).

5.3.3 Harbor & Clair

Harbor has been a clear choice for a container repository. Portus is only an authentication frontend to an external registry and Quay is too dependent on RedHat-based technology. Harbor, on the other hand, is easy to set-up and fully driven by the community.

In Harbor, container images are stored inside repositories and repositories are collected to projects. Users can be assigned to projects or repositories in different roles (Developer, Admin, Guest, etc). Harbor manages all the registry endpoints and both authentication and authorization of users.

Every time a new image is recieved, Harbor instructs Clair to scan it. Clair conducts a vulnerability scan and provides Harbor with results. Vulnerability testing could also be triggered by Drone as a pipeline stage.

5.3.4 Deployment

There are three main ways to continue with deployment.

1. Drone applies Kubernetes manifests itself.
2. An independent tool applies Kubernetes manifests after a tool triggers it.
3. An independent tool applies Kubernetes manifests when a user requests it.

The first method is not a fail-safe solution. The manifests would be scattered around different repositories. Lack of centralized versioning would prevent us from restoring any complete previous version of the application (consisting of multiple MSs) with ease.

The second solution could be fitting when deploying to proprietary managed Kubernetes. Drone itself has a lot of plugins that allow such triggering. Other deployments tools can be watching and waiting for new versions of containers. In theory, a container registry could also send webhooks to other deployment tools. Drone could also trigger a tool like Keptn (see 4.6.4) to automate deployment.

The third and last option is the most flexible one. It mimics a workflow in which a developer requests a deployment. Consider a feature branch. A developer creates a branch and fills it with commits. Once he is confident the feature is ready he can request a deployed preview of his changes in a production-like environment. If all tests passed and he is satisfied with the result, he can submit a merge request back to main.

We have decided to used the third. It strikes a balance between too bare-bones and too streamlined. A disadvantage is that advanced deployment strategies must be implemented by hand or by another tool. Which is to be expected when avoiding proprietary managed Kubernetes.

5.3.5 Flux

Flux is a GitOps operator. That means it synchronizes the Kubernetes desired state with a state defined in a Git repository. For reference see 3.4.4. The state is defined by YAML files in a separate git repository. That way all changes are versioned. **Restoring the state of the deployed production cluster to any earlier point in time is only a matter of one „git revert".**

Flux is running deployed on Kubernetes. It watches the repository at regular intervals. If it detects a change it pulls the latest manifests (git clone via SSH) and deploys them to Kubernetes. But not only that. If some manifests are removed from the repository it instructs Kubernetes to destroy appropriate resources. To implement advanced deployment strategies like „canary" or „blue/green" flux can be combined with a tool called Flagger.

Kubernetes sees the change in the desired state. It schedules new pods to Kubelets. **Kubelets pull the desired images from Harbor and deploy it.**

5.3.6 Pros & cons

This solution has many flaws and is sometimes rough around the edges. Nonetheless, it is very capable, affordable, and lightweight.

Pros:

- Free of charge
- Open-source
- Highly customizable configuration
- Full control over the entire process
- Any part can be switched independently

Cons:

- A lot of work to securely manage all the tools
- Does not implement any HA deployment strategies
- Cannot be itself implemented in a HA way
- The developer's experience is not particularly smooth and pleasant

Chapter 6
Proprietary vs. open-source CI/CD pipeline solution

This chapter is split into three distinct parts. First, it explores all the different proprietary solutions. Then it compares them with some of the open-source approaches and finally, it discusses lessons learned. The discussion section provides specific bits of advice for anyone tasked with building a CI/CD pipeline.

6.1 Proprietary solutions

We are leaving the open-source free of charge territory. We want to buy some software to make our job a bit easier. Unfortunately, there are many ways to spend money on CI/CD with Kubernetes. This section outlines some of the possible approaches.

Firstly, the surrounding terminology is very confusing and often interchangeable. Kubernetes can be deployed in two main ways.

- Managed Kubernetes – Cloud provider creates and manages Kubernetes cluster for us. We decide when to provision new nodes (computers) through an interface.
- Self-managed Kubernetes – We install Kubernetes on servers ourselves. The servers can be both on-premises and hosted (purchased from cloud providers).

When considering proprietary solutions, there are several diverse parts available for purchase. We can buy:

- (Computing) resources – CPU time, RAM, HDD with RAID. Resources can be accessed in many ways. For example, VM, deployed container, Kubelet, etc.
- Managed Kubernetes – Cloud provider gives us interface through which we define our Kubernetes cluster.
- Pipeline tools – Tools from categories described in section 4.2 can be bought individually or as a service.

The following is a summary of 4 interesting distinct ways a commercial Kubernetes CI/CD pipeline can be implemented.

6.1.1 Managed Kubernetes & external tools

Azure, Google, Digital Ocean, and Amazon all offer managed Kubernetes clusters. A customer generally pays mostly for resources she actually uses. The added price for a managed Kubernetes cluster is small[1] or none[2].

Of course, having a cluster is not enough. Other, external tools for CI/CD must be in place. There are a plethora of commercial tools, some of them even listed in the open-source summary. They can be deployed to the same cloud provider on the provisioned server, on-premises, or anywhere else.

[1] Amazon EKS and Google GKE – 0.10 USD per hour, Digital Ocean starting at 10 USD per month
[2] Azure AKS

This approach is very flexible. And, most paid tools integrate very well with all the major cloud providers. To what extent the pipeline shall be self-managed is entirely up to the customer.

6.1.2 Cloud provider's way

The big cloud providers[1] offer an entire „**DevOps suite**" for their customers. Kubernetes cluster deployment aside. The suite usually contains all the tools expected. A **Git repository manager**, Pipelines-maker (**CI**/**CD** automation tool), and an **artifact repository**. All the tools are tightly integrated with each other effectively solving the biggest problem of open-source tools.

Bear in mind that the presented complete „DevOps suites" can significantly differ. Careful consideration is advised when picking which one to use. The „suite" tools generally should be replaceable by whatever other tools from chapter 4 but there still is the risk of getting too „locked in" with the cloud provider.

6.1.3 Openshift

A specific way to buy a solution is via RedHat OpenShift. OpenShift is an enterprise modified version of Kubernetes budled with many other tools to provide a platform. It provides everything required to develop and deploy containerized applications. Meaning, the CI/CD tools are integrated together and with Kubernetes providing a truly frictionless experience.

OpenShift can be deployed on RedHat cloud, Azure, IBM, or on-premise. This is not specific to OpenShift. There are several container platforms with Kubernetes, although OpenShift is one of the most notable ones.

6.1.4 GitHub Actions

GitHub's Actions are another specific approach to CI/CD. GitHub provides an integrated, fully-featured CI solution which can be meticulously integrated with everything that happens in a Git repository. GitHub also provides a community „marketplace" where people can contribute prepared integrations and extensions.

This approach is quite unique but also very familiar. It combines the „git at the center" approach we have seen with GitLab, the Drone's method of „building your own external plugins", and the generic „own picked tools" mindset adopted in this whole thesis.

[1] Google GCP, Azure DevOps, Amazon AWS tools, IBM Urban Code

6.2 Pipeline comparison

Any comparison listed here is bound to be wrong in some way. There are many methods a CI/CD pipeline deploying to Kubernetes can be built. A pipeline can be absolutely anywhere on the scale from „bought as a proprietary service“ to „built from open-source tools on-premise“. Still there some general findings learned about using previously described open-source tools.

Before deploying a pipeline one should decide how much of the process he wants to outsource. Building the pipeline from tools with strictly defined interfaces and workflows means a lot of flexibility to customize. It also means a lot of work in getting the details just right.

If we decide to give away a lot of freedom the tools become more interconnected. The whole workflow is more fluent and painless. However, the tools become more proprietary and expensive. In addition, being more interconnected means it is harder to decouple any one tool. It means bigger lock-in to one tool/provider.

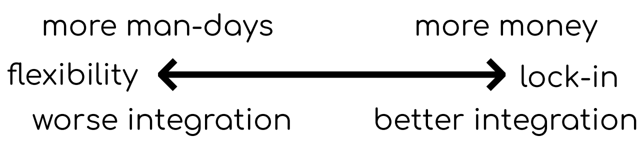


Figure 6.1. Idividual tools vs all-in-one solution

The open-source tools frequently have some of the following advantages:

- Free of charge
- Lightweight on system resources
- Enterprise quality tools
- Industry-standard, developers are used to them
- Less lock-in to one company/provider
- Comprehensive security is entirely in your hands

Before deciding to use just open-source tools, one should also consider possible disadvantages:

- Locked features from paid „enterprise solutions“
- Developers might be used and more comfortable with proprietary tools they have used before (eg. GitHub).
- Integrations with other tools are limited. Even integrations between open-source and proprietary are better than open-source to open-source.
- Self-managing multiple tools requires (expensive) time and expertise
- Quirks of individual open-source projects are hard to know in advance and can create a lot of work
- Comprehensive security is entirely in your hands

6.3 Discussion

The goal of this section is o share some of the learned lessons. **To provide specific pieces of advice for anyone who is deciding on how to build a pipeline that is deploying applications into Kubernetes**.

6.3.1 In practice

It's not an either-or question. Even though it may seem that you have to choose between managed proprietary tools or open-source tools on-premises it is not the case. It is common practice to combine all the tools that best describe the company's workflow and use them.

It's crucial to have a clear idea of what processes are to be built before building them. One must pick tools to suit the company's pipeline, not the other way around.

Migrating and untangling a finished pipeline implies a lot of work. It's critical to estimate what functions are going to be needed in the future (scaling) and if the picked tools can provide them.

6.3.2 Developer at the center

The pipeline is built for developers, not administrators. A good idea is to examine what tools and workflows are the developers comfortable with. In the end, developers are the ones that are going to be using the pipeline.

A developer's comfort can be raised by integrations. A little green tick mark or automated message with CI build results may seem insignificant but can mean a lot for a developer checking the status multiple times a day.

6.3.3 Cloud controller

Installing and deploying enterprise-ready Kubernetes on your own servers (hosted/VMs) requires great expertise. Paying for managed Kubernetes in the cloud saves valuable time and headaches when maintaining and debugging. But not only that. Kubernetes is built on the beliefs in high availability. Self-installed K8s on-premises can never[1] leverage automated scaling through a cloud controller. **You cannot script provisioning new computing capacity to K8s cluster when it means you have to buy and set up the physical hardware yourself.** This would be particularly strange considering some cloud providers offer managed Kubernetes free of charge.

6.3.4 Caching CI & testing

We would advise thinking about caching. CI pipelines especially can take a long time. Any time saved enhances the developer's experience. Faster pipeline means more runs and better agility. Specific build tools must not be downloaded again for every build. This can, for example, be achieved by custom-made build containers.

Never shorten the pipeline by deleting tests. Rigorous testing is at the heart of proper continuous delivery. What can be done is carefully defining what test should be run and when. Merging might have different consequences than pushing.

[1] If it could it would technically make the company a cloud provider.

6.3.5 Docker in Docker

Build jobs can be executed in two main ways. Running them directly on the server or use a Docker container. A Docker container holds all the advantages. It creates a clean repeatable environment every time. Dockerfiles can be versioned and integrated into a pipeline. However, if we are building an MS app to be deployed in Kubernetes it has to be packaged inside a container. Thus our clean Docker build environment wishes to build with Docker. Docker in Docker (DinD) images are available and used, but can yield unexpected results as the creator of DinD himself points out. [42]

This does not have to be a big issue, nonetheless, it is essential to be aware of it.

6.3.6 Helm Charts

There is a helm chart available for most of the tools covered in this thesis. Helm charts make it easy to deploy applications to Kubernetes. Just like a package manager installs and configures programs in a regular OS.

There is a lot of prepared Helm charts. Just because there is a Helm chart of a tool does not mean it is a good idea to use it. Helm deployed applications run on Kubernetes. What benefits does it actually bring if the tool is itself a monolith with a relational database? Moreover, if a tool for CI/CD would be deployed on the same cluster as it is deploying to there might be some unforeseen deadlocks.

6.3.7 Consider GitOps

GitOps might not be the best fit for everyone but at least consider some of its advantages.

GitOps increases security. Instead of giving CI access to Kubernetes, we supply Kubernetes with a way to extract the information directly from Git. GitOps pulls from a repository, while classic CI pushes to production.

Infrastructure should be versioned as code. GitOps takes this a step further and says that the cluster state can and should be versioned. This allows reverting to any point in history which can be very useful. Unversioned changes to the cluster may cause unpredictable behavior in the future.

Chapter 7
Conclusion

In the beginning, we set out three main goals. First, learn about tools suitable for creating an open-source pipeline. Then design a solution with some of the researched tools. And finally, compare the designed solution with proprietary ones and share gained knowledge.

The first goal required to research and compare open-source tools for CI/CD when deploying to Kubernetes. A structured comparison of some of the most popular open-source tools has been assembled in the fourth chapter. The tools were categorized and systematically described. Each category was appropriately explained and presented. Furthermore, underlying technologies such as Docker or Kubernetes were introduced for the reader's comfort.

The second goal was to design a pipeline composed of some of the researched tools. Not one, but two distinct pipelines were built and tested with the selected sample application. The first used an all-in-one approach while the second connected numerous different tools. Both were fully functional demonstrating it is possible to create an open-source pipeline deploying to Kubernetes.

The third goal challenged to provide specific pieces of advice for anyone deciding on how to build a K8s pipeline. It is possible to create fully open-source pipelines, yet we would argue it is generally not a good idea. The sheer amount of work done maintaining all the individual tools and exploring various license limitations is scarcely worth the trouble. Not to mention the senselessness of on-premise Kubernetes. Such views are discussed in more detail in the preceding chapter.

The cloud-native space is evolving at unprecedented speed. It would be interesting to see how many of the mentioned tools get abandoned or discontinued. There are also big gaps in this text concerning advanced deployment strategies. What are the enterprise benefits of each one? Can they all be implemented only with open-source tools?

Appendix A
Dictionary

CD	Continuous delivery or continuous deployment (Distinguished where difference matters)
CI	Continuous integration
CLI	Command line interface
CNCF	Cloud Native Computing Foundation
ČVUT	České vysoké učení technické v Praze
DinD	Docker in Docker
DNS	Domain Name System
GUI	Graphical user interface
HA	High availability
K8s	Kubernetes
MS	Microservice
MSs	Microservices
MS’s	Microservice’s
on-prem	Or „on premises“. To run software on local servers rather than outsourcing the infrastructure to a cloud provider.
QA	Quality assurance
REST	Representational State Transfer
SaaS	Software as a service
VM	Virtual machine
VMs	Virtual machines
YAML	YAML Ain’t Markup Language

Appendix B
Tools honorable mentions

Project management

Kallithea	`https://kallithea-scm.org/` Kallithea is a fast and powerful management tool for Mercurial and Git with a built-in push/pull server, full text search and code-review.
Bugzilla	`https://www.bugzilla.org/` Bugzilla is server software designed to help you manage software development.
Redmine	`https://www.redmine.org/` Redmine is a flexible project management web application.
Trac	`https://trac.edgewall.org/` Trac is an enhanced wiki and issue tracking system for software development projects.
Launchpad	`https://launchpad.net/` Launchpad is a set of Web services to help software developers collaborate.

CI & CD tools

Screwdriver	`https://screwdriver.cd/` Screwdriver is an open source build platform designed for Continuous Delivery.
Talkcluster	`https://docs.taskcluster.net/` Taskcluster is the task execution framework that supports Mozilla's continuous integration and release processes.
Strider	`https://strider-cd.github.io/` Strider is an Open Source Continuous Deployment / Continuous Integration platform.

Container registry manager

Dragonfly	`https://d7y.io/` An Open-source P2P-based Image and File Distribution System
Huawei dockyard	`https://github.com/Huawei/dockyard` Container & Artifact Repository

Code quality tools

OpenSCAP	`https://www.open-scap.org/` The OpenSCAP ecosystem provides multiple tools to assist administrators and auditors with assessment, measurement, and enforcement of security baselines.
PMD	`https://pmd.github.io/` An extensible cross-language static code analyzer.

Monitoring

Kubernetes	`https://kubernetes.io/` K8s is very good for monitoring the MS it is running. Liveness and Readiness Probes are a powerful way to prevent problems.
Zabbix	`https://www.zabbix.com/` Zabbix is the ultimate enterprise-level software designed for real-time monitoring of millions of metrics collected from tens of thousands of servers, virtual machines and network devices.
Grafana Loki	`https://grafana.com/oss/loki/` Loki is a horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus.
Logstash	`https://www.elastic.co/logstash` Logstash is a free and open server-side data processing pipeline that ingests data from a multitude of sources, transforms it, and then sends it to your favorite „stash“.
Kieker	`http://kieker-monitoring.net/` Kieker provides complementary dynamic analysis capabilities, i.e., monitoring and analyzing a software system's runtime behavior — enabling Application Performance Monitoring and Architecture Discovery.

References

[1] Fowler, Martin, and James Lewis. Microservices. *martinFowler.com*. [online], 2014 [cit. December 8, 2019]. Available from `https://martinfowler.com/articles/microservices.html`.

[2] Fowler, Susan J. *Microservices in Production: Standard Principles and Requirements*. USA: O'Reilly Media, 2016. ISBN 9781492042846.

[3] Cloud Native Computing Foundation. *CNCF SURVEY 2019*. [online], March 3, 2020 [cit. March 15, 2020]. Available from `https://www.cncf.io/wp-content/uploads/2020/03/CNCF_Survey_Report.pdf`.

[4] Vogels, Werner. The Story of Apollo - Amazon's Deployment Engine. *All Things Distributed*. [online], November 12, 2014 [cit. April 2, 2020]. Available from `https://www.allthingsdistributed.com/2014/11/apollo-amazon-deployment-engine.html`.

[5] Harris., Derrick. Talking microservices with the man who made Netflix's cloud famous. *Medium.com*. [online], May 29, 2015 [cit. May 23, 2020]. Available from `https://medium.com/s-c-a-l-e/talking-microservices-with-the-man-who-made-netflix-s-cloud-famous-1032689afed3`.

[6] Tišnovský, Pavel. Mikroslužby: moderní aplikace využívající známých konceptů *Root.cz*. [online], 2019 [cit. December 8, 2019]. Available from `https://www.root.cz/clanky/mikrosluzby-moderni-aplikace-vyuzivajici-znamych-konceptu/`.

[7] Sponsors, Redis. home. *redis.io*. [online]. [cit. May 14, 2020]. Available from `https://redis.io/`.

[8] Apache Software Foundation. home. *kafka.apache.org*. [online], 2017 [cit. May 14, 2020]. Available from `https://kafka.apache.org/`.

[9] Pathirage, Milinda. home. *kappa-architecture.com*. [online], 2019 [cit. May 14, 2020]. Available from `https://milinda.pathirage.org/kappa-architecture.com/`.

[10] Deutsch, Peter. The Eight Fallacies of Distributed Computing . [online], 1994. [cit. December 8, 2019]. Available from `http://nighthacks.com/jag/res/Fallacies.html` .

[11] Rotem-Gal-Oz, Arnon. *Fallacies of Distributed Computing Explained*. [online], 2007 [cit. December 8, 2019]. Available from `http://www.rgoarchitects.com/Files/fallacies.pdf`.

[12] Chang, Michael Alan, Brendan Tschaen, Theophilus Benson, and Laurent Vanbever. Chaos Monkey: Increasing SDN reliability through systematic network destruction. In: *SIGCOMM 2015 - Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 2015. ISBN 9781450335423. Available from DOI 10.1145/2785956.2790038.

[13] Brandt, Richard L. Jeff Bezos of Amazon: Birth of a Salesman - WSJ. The Wall Street Journal. *The Wall Street Journal*. [online], October 15, 2011 [cit. April 1, 2020]. Available from `https://www.wsj.com/articles/SB10001424052970203914304576627102996831200`.

[14] Gan, Yu, Yanqi Zhang, and Dailun Cheng et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In: *International Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS*. 2019. ISBN 9781450362405. Available from DOI 10.1145/3297858.3304013.

[15] Kistowski, Jóakim von, Simon Eismann, Norbert Schmitt, André Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In: *Proceedings of the 26th IEEE International Symposium on the Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. 2018. MASCOTS '18.

[16] Docker Community. *Docker Documentation*. [online], 2019 [cit. February 15, 2020]. Available from `https://docs.docker.com`.

[17] Docker Community. Comparing Containers and Virtual Machines *Docker.com*. [online], 2020 [cit. February 15, 2020]. Available from `https://www.docker.com/resources/what-container`.

[18] Docker Community. Docker Engine *Docker Documentation*. [online], 2019 [cit. February 16, 2020]. Available from `https://docs.docker.com/v17.09/engine/docker-overview/`.

[19] Docker Inc. Home. *huhb.docker.com*. [online], 2020 [cit. May 23, 2020]. Available from `https://hub.docker.com/`.

[20] Documentation., Red Hat. What is orchestration?. *redhat.com*. [online]. [cit. May 23, 2020]. Available from `https://www.redhat.com/en/topics/automation/what-is-orchestration`.

[21] The Linux Fundation. Production-Grade Container Orchestration. *Kubernetes.io*. [online], 2020 [cit. March 6, 2020].
Available from `https://kubernetes.io`.

[22] Poulton, N., and P. Joglekar. *The Kubernetes Book*. Independently Published, 2017. ISBN 9781521823637.

[23] Hightower, Kelsey, Brendan Burns, and Joe Beda. *Kubernetes: up and running: dive into the future of infrastructure*. 1 ed. USA: O'Reilly Media, Inc, 2017. ISBN 9781491935675.

[24] The Linux Fundation. Kubernetes Components *Kubernetes documentation*. [online], 2020 [cit. March 7, 2020]. Available from `https://kubernetes.io/docs/concepts/overview/components/`.

[25] The etcd authors. *etcd.io*. [online], 2020 [cit. March 7, 2020]. Available from `https://etcd.io`.

[26] The Linux Fundation. Kubernetes Components. *Kubernetes documentation*. [online], January 16, 2020 [cit. March 6, 2020]. Available from `https://kubernetes.io/docs/concepts/overview/components/`.

[27] Reed, Paul J. *DevOps in practice*. 3 ed. USA: O'Reilly Media, Inc, 2015. ISBN 9781491913062.

[28] Mugrage, Ken. *Continuous Delivery with Docker and Kubernetes* . In: *Youtube* [online], 2018 [cit. April 2, 2020]. Available from `https://www.youtube.com/watch?v=xAziflV3ah4`. Channel ThoughtWorks.

[29] Fowler, Martin. Software Development in the 21st century. *ThoughtWorks.* [online], 2014 [cit. April 2, 2020]. Available from `https://www.thoughtworks.com/talks/software-development-21st-century-xconf-europe-2014`.

[30] Fowler, Martin. Continuous Integration. *martinFowler.com.* [online], May 1, 2006 [cit. April 2, 2020]. Available from `https://martinfowler.com/articles/continuousIntegration.html`.

[31] Farcic, Viktor. *The DevOps 2.4 Toolkit: Continuous Deployment to Kubernetes: Continuously deploying applications with Jenkins to a Kubernetes cluster*. Packt Publishing, Limited, 2019. ISBN 9781838648787.

[32] Humble, Jez. Continuous Delivery - Priciples. *continuousdelivery.com.* [online], 2017 [cit. April 2, 2020]. Available from `https://continuousdelivery.com/principles/`.

[33] Forsgren, Nicole, Dustin Smith, Jez Humble, and Jessie Frazelle. Accelerate: State of DevOps 2019. *Google.com.* [online], July 20, 2019, [cit. April 2, 2020]. Available from `https://services.google.com/fh/files/misc/state-of-devops-2019.pdf`.

[34] Poirier, Greg. *Not Everyone Can Be Kelsey Hightower*. In: *Youtube* [online], 2015 [cit. April 4, 2020]. Available from `https://www.youtube.com/watch?v=8NFyrEVSBT8`. Channel DevOpsDays Silicon Valley.

[35] Alexis Richardson, William Denniss . *GitOps - Operations by Pull Request [B]* . In: *Youtube* [online], 2017 [cit. March 12, 2020]. Available from `https://www.youtube.com/watch?v=BSqE2RqctNs`. Channel CNCF [Cloud Native Computing Foundation].

[36] Cloud Native Computing Foundation. CNCF Cloud Native Interactive Landscape. *cncf.io.* [online], May 23, 2020 [cit. may 23, 2020]. Available from `https://landscape.cncf.io/`.

[37] Kumar, Shray. The Case Against Jenkins In 2020. *medium.com.* [cit. April 28, 2020]. Available from `https://medium.com/@shrayk/the-case-against-jenkins-in-2020-310276e39280`.

[38] Gotti, Simone. Introducing Agola: CI/CD redefined. *Sorintoss.io.* [online], July 14, 2019 [cit. April 28, 2020]. Available from `https://sorintoss.io/blog/agola-introduction/`.

[39] sonarsource. List of Customers. *sonarsource.com.* [online], 2020 [cit. April 30, 2020]. Available from `https://www.sonarsource.com/customers/`.

[40] Pilarinos, Dennis. GitHub? Bitbucket? Cloud or Self-hosted?. *buddybuild.com.* [online], Octomber 12, 2016 [cit. March 21, 2020]. Available from `https://www.buddybuild.com/blog/source-code-hosting`.

[41] Shimel, Alan. 5th Annual DevOps Dozen Awards: And the Winner Is .... *devops.com.* [online], January 21, 2020 [cit. March 21,2020]. Available from `https://devops.com/5th-annual-devops-dozen-winners-announced/`.

[42] Petazzoni, Jérôme. Using Docker-in-Docker for your CI or testing environment? Think twice.. *jpetazzo.github.io.* [online]. [cit. March 21, 2020]. Available from `https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/`.

[43] LITVINENKO, Alexander. https://medium.com/faun/docker-in-docker-the-real-one-e54133639c55. *Medium.com*. [online], December 1, 2019 [cit. March 21, 2020]. Available from `https://medium.com/faun/docker-in-docker-the-real-one-e54133639c55`.