Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Evaluation of GitOps Security in a CI/CD Environment

Alwin [redacted]

Bachelor's Thesis

Evaluation of GitOps Security in a CI/CD Environment

Bachelor's Thesis

Supervised by Prof. Dr. phil. nat. Sebastian Steinhorst
Professorship of Embedded Systems and Internet of Things
Department of Electrical and Computer Engineering
Technical University of Munich

Advisor M. Sc. Ege Korkan

Co-Advisor Robin [redacted]

Co-Advisor Christopher [redacted]

Author Alwin [redacted]

Submitted on 29 July 2019

Declaration of Authorship

I, Alwin [redacted], declare that this thesis titled "Evaluation of GitOps Security in a CI/CD Environment" and the work presented in it are my own unaided work, and that I have acknowledged all direct or indirect sources as references.
This thesis was not previously presented to another examination board and has not been published.

Signed: [redacted]

Date: 25/07/2019

Abstract

GitOps is a new solution to Continuous Delivery, which promises advances in security, usability and stability in the deployment process by storing the complete microservices deployment in a git repository. Thus, the CI agent is excluded from direct cluster access and therefore prevents its attackers to access valuable production data. However, in the existing open-source tool landscape, the connection between the two isolated pipeline parts is bridged either insecurely, where the pipeline agent has full write access to the GitOps repository or requires human interaction to finish a deployment. The software developed in this thesis limits an attacker's potential impact while still retaining fully automated deployments. It watches Docker images in a registry defined by the deployment repository and commits new image tags according to specified rules. Thus, a deployment is triggered by the pushed Docker image alone and the build agent interacts neither with the deployment repository nor with the cluster.

Contents

1

Introduction

"The greatest risk we face in software development is that of overestimating our own knowledge."

Jim Highsmith

1.1 Motivation

Most people's lives are impacted by software every day in many, often hidden, ways because the majority of operations and processes are aided or controlled by software entirely. Many software systems are connected and exchange data for various purposes, so a defect in a seemingly unrelated software can have cascading effects on systems in other areas. Thus, the reliability and security of software itself, but also tools connected to the operation and delivery of software should have the highest priority since otherwise disruptions to people's lives could occur.

Software development is nowadays performed mostly according to agile principles [1]. One important part of Agile Development is Continuous Integration/Continuous Deployment (CI/CD), which automatically tests, builds and deploys every commit made by the developers to the central code repository [2]. The deployment makes the latest version available in a test or production environment in a computing cluster. Those CI/CD tasks are performed by a pipeline agent as part of a centralised CI/CD system. Traditionally, pipelines push new deployments directly to a cluster. Thus, they need to have authorisation to modify the cluster structure and access all its resources. The pipeline agents handling the credentials to this most important system are often insecure, due to the usage of unverified third-party plugins, rare updates and because regular developers have to manage the pipeline and the pipeline agent's security without proper training. Thus, their main focus is to make the pipeline work, so they can start developing code. Therefore, the main application running in the production environment is vulnerable to attacks via the pipeline agent, which could enable an attacker to delete or

extract all stored data and modify the cluster in other ways.
To combat this issue, and for other reasons, GitOps was introduced, which proposes pull pipelines instead of push pipelines [3]. In GitOps, the whole deployment state is defined in a git repository. Then, the pipeline writes or proposes a modification of the deployment repository and does not communicate with the cluster directly. For the synchronisation of the deployment repository, an operator is placed inside the cluster, which can update all resources according to the definitions in the repository.
The final goal is to have a pipeline, which is authorised to only deploy the newly built artefacts, but not modify or access the structure or any other resources of the deployment defined in the repository to adhere to the security principle of defence in depth and limit an attacker's impact.

1.2 Problem Statement

Currently, a GitOps pipeline operator has full write access to the deployment repository to commit the change to a new version. Thus, any modification to the deployment environment can be performed via the deployment repository. The main difference to push pipelines is that every change is tracked in the git log, but the access is barely restricted. More specific rules cannot be set to limit a git user's access to specific files or even single lines in files, which would be required to restrict the pipeline effectively from modifying unrelated resources. On the other hand, no interaction between pipeline and cluster entirely would prevent the pipeline from deploying new software versions. The ideal pipeline both separates the pipeline agent from the cluster and retains the high velocity of automated deployments.

1.3 Task Definition

The goal of this thesis is to evaluate the security of current GitOps solutions and workflows and perform a comparison to push pipelines. Additionally, a tool will be developed to reduce the impact an attacker can have on the cluster when hijacking the pipeline agent in a GitOps scenario as described in the previous section. To evaluate and demonstrate the viability of the proposed solution, a case study project with a typical pipeline is set up in GitOps fashion. This will also serve as a showcase of both GitOps in general and the newly developed tool specifically so it can be included in other software development projects easily.

1.4 Structure of This Document

The first chapter gives an overview of the thesis including its relevance and embeds it into the context of software development. Following after this section, the second chapter provides an introduction into professional software development, related concepts and technologies as well as the topic of this thesis: GitOps security and related issues. In the third chapter the challenge of automatic code deployments in GitOps pipelines and the remaining connection of the pipeline agent to the cluster is described, which is followed by an explanation of the implemented solution to overcome this problem. The fourth chapter deals with the analysis and evaluation of the developed tool on a case study software project. Finally, chapter five concludes the thesis and names future tasks to advance the topic.

2

State of the Art

"The most powerful tool we have as developers is automation."
Scott Hanselman

This chapter gives an overview of the modern software development process, what GitOps itself is and which tools are available to create a GitOps pipeline. Finally, the security of a typical push and pull pipeline is analysed.

2.1 Foundations

2.1.1 Agile Development

A flexible approach in software development where a product evolves through continuous dialogue between developers and clients or customers[1] rather than a heavyweight initial specification that is implemented by a team of developers independently is called Agile Development. It is very widespread in today's software development landscape. Through intensive dialogue [4, p.17] a product or application is created to satisfy the customers needs as best as possible and future users can give feedback from an early stage before the final product is released. Tasks are defined and the progress is evaluated per "sprint", which usually lasts for a few weeks so that priorities and the focus can be adapted to changed requirements, available technologies and new challenges [5]. Regular (usually daily) meetings of the whole developer team are very common to get an overview of everyone's progress, challenges and to raise questions that came up the previous day [6]. In Agile Development, the developers are often aided by automation tools so they can focus on writing reliable and bug-free software.

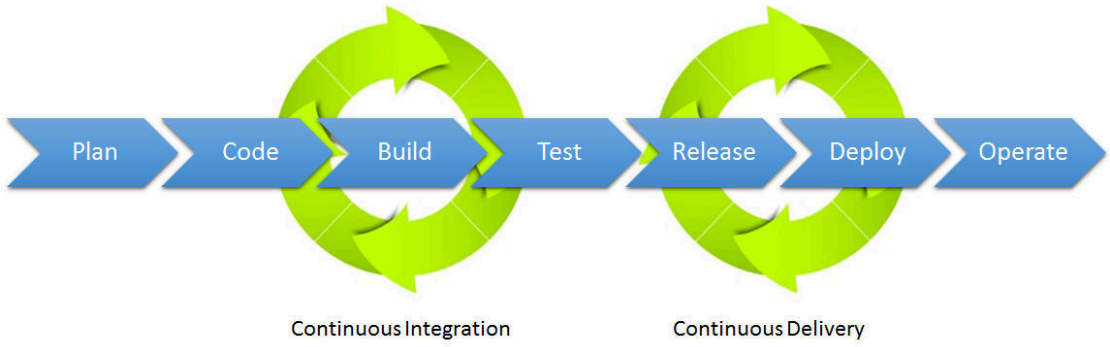

Figure 1: Generic CI/CD Pipeline [13].

2.1.2 Continuous Integration

An important method to support Agile Development, which guides the collaboration of developers and how they produce code that works together, is Continuous Integration [2]. It proposes to merge all participants contributions as a central version regularly to minimise merge conflicts [7]. As part of a continuous integration pipeline, the central version is tested for bugs, code quality and security vulnerabilities to ensure only high-quality code reaches the final product and all contributions work together in a defined environment [8].

2.1.3 Version Control

The most important tool for Continuous Integration is the decentralised version control system git. It can store and track every change made to a code repository. Therefore, it allows all developers to work on local copies of the code repository and enables easy merging once different changes come together [9]. Snapshots of the current project files (commits) can be created and prevent data loss due to merge conflicts or successive changes to the code. Every commit contains a cryptographic hash of all files and assigns an author to the changes made since the last commit. Git keeps a history of all commits to allow access to previous versions and revert the state if necessary. For developers to work in parallel without merging their code on every commit, a repository can contain branches where code is developed independently until the branches are merged again [10]. Other version control systems like Mercurial [11] and subversion provide an alternative to git but are barely used in professional software development [12].

2.1.4 Continuous Delivery

After the Continuous Integration pipeline finds a new version to be of acceptable quality, the next step is to deploy the code in an accessible system, so it can be used by other developers, and after further testing releasing it to the clients. Deploying and running

development versions regularly — ideally after every commit — is called Continuous Delivery and nowadays part of the combined Continuous Integration / Continuous Delivery (CI/CD) pipeline as depicted in Figure 1. This pipeline fully automates the process from git commit to a running version on some server or in the cloud. A deployment on a server differs from running an application on a local desktop computer in several ways, for example, the hardware and software environment is different, which can influence the stability and interactions between different parts. Other resources like web services are not available locally at all. Therefore, only regular deployments to a test environment can ensure that the final version will run in the desired production environment [14, p.117]. Usually, there are different environments (stages) to test code in a production-like setting without impacting the system that is currently serving users. In a typical development setup there are three environments:

- integration: Where all committed code is (potentially automatically) deployed to test new features and the interaction of different parts. Developers can use this stage to view their own work and observe the team's progress.
- staging: To test a full new version shortly before the release, to find final bugs and present state to a client. In this environment, a production-like data set should be available.
- production: The environment that is exposed to real users. It should only contain bug-free and stable versions.

2.1.5 Docker

A container orchestration engine to run application containers, which are separate environments that only contain a single application [15, p.6], is called Docker. Docker containers are the low-level basis of almost every cloud environment since they provide a fast and scalable way to run many applications on the same host in isolated environments [16, p.6]. They are similar to traditional virtual machines, however, containers use the host's operating system kernel and do not contain a complete operating system on their own [15, p.4]. Containers only include the bare minimum that is necessary to run a specific application. Therefore, the host's resources are used more effectively than in full-machine virtualisation. Currently, Docker is natively supported only in Linux systems and can consequently only run containers based on the Linux kernel. There are Docker versions for Mac and Windows too, but they are using full-machine virtualisation to create a Linux system to run the containers in [17, p.2]. Thus, using Docker on non-Linux systems is only advisable for development purposes.
A container is started from an immutable image, which consists of multiple layers [19, p. 72] as depicted in Figure 2. Every image consists of a base image, which is modified to create an environment for the specific application it should run. These modifications,

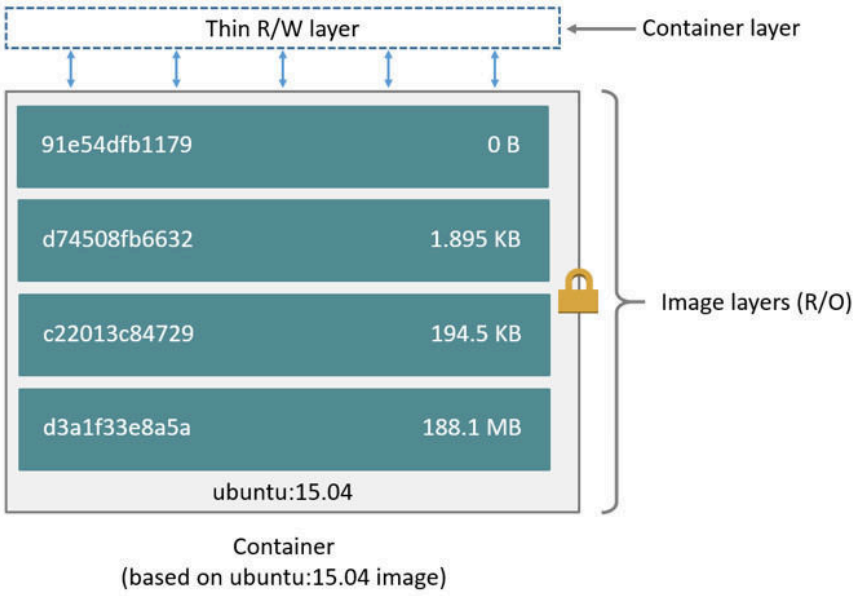

Figure 2: Docker images consist of multiple layers [18].

which can involve adding files, running shell commands or defining the environment, create each new layers. This allows reusing of previous layers for different applications that share a common base and storing all images effectively since every layer needs to be stored only once. Common Docker images for operating systems and tools are available to use directly or as a base image. For example, to build an image for a Java application, developers can use official Java images and add the `.jar` artefact in a single layer. Alternatively, they could use an Ubuntu or Debian image, which only contains the respective operating system, and perform the steps necessary to install Java on their own. In that case, they would have to update these steps regularly to install the newest Java version rather than just use the newest published Java base image, which is maintained by Oracle itself [20, p.170].

Images are stored and shared in private or public registries [19, p.76]. The official and most well-known registry is Docker Hub[1], where images for almost all (Linux) operating systems and most applications can be found, for example images for open source Java (openjdk), MySQL, Nginx (a web server and load balancer), Jenkins (a build server) and Alpine Linux (an operating system). These images are released by the application's developers so they are reliable and up to date. That is an important advantage since application maintainers can be sure to get the newest and safest versions automatically and do not have to worry about manual update procedures. They can simply rebuild their Docker images by using the updated base image. For non-free software and professional development private repositories can be used to store Docker images privately and securely [21, p.62]. Images are tagged to distinguish versions and

1 https://hub.docker.com/

environments of the same application [17, p.42]. They can be used to pin a base image to a very specific version (e.g. `openjdk:12.0.1`) or allow for smaller updates of that base image in future builds (e.g. `openjdk:12.0`) or always take newest release regardless of its version (e.g. `openjdk:latest`).
Alternative container platforms are for example Mesos, CoreOS rkt and LXC. They also provide an isolated and portable environment for applications to run, but there is not such a large ecosystem like it exists for Docker with Docker Hub. Therefore, 83% of containers were Docker containers in 2018 [22]. In this thesis the more generic term container instead of Docker container will be used if the implementation is irrelevant.

2.1.6 Microservice Architecture

A design pattern where an application consists of many small services which act predominantly independently as opposed to one monolithic service that contains the whole application [23] is the microservice software architecture [24], which is shown in Figure 3. This design enables highly parallel and agile development because teams can work on different parts of an application independently and can rapidly redeploy development versions without impacting other teams [25]. Since microservices are independent of each other, they can also be scaled individually reacting to specific resource requirements of parts of an application and satisfy high reliability requirements. The communication between microservices usually occurs via HTTP APIs or other message-based communication channels over the network[26], so a single application can be distributed over multiple servers, and even multiple instances (replicas) of the same microservice can be spread over multiple servers. Microservices are highly maintainable because they can be replaced and upgraded individually as long as they continue to support the same set of endpoints [27, p.23]. Since the communication is performed over universally understood protocols, different microservices can be written in different programming languages [28], which encourages the use of modern languages and software design paradigms. Usually, data stores are assigned to individual microservices to separate unrelated data and make future development easier [29]. From a security perspective, microservices have advantages as well. Developers do not need to have access to the whole project to develop their features; bugs and security vulnerabilities are only impacting a single microservice and in case of a successful attack, the attacker will usually be confined to a single instance of a microservice and only have access to data of this particular service [30, p.314]. Spreading out an application in microservices increases its complexity, but it is easier to understand and individual software developer teams do not have to know about the whole application and can solely focus on their microservice.
A major challenge for today's software developers is to modernise old monolithic applications and transform them into microservices. Instead of doing this in one step, often the monolithic system is converted into multiple self-contained systems with individual fron-

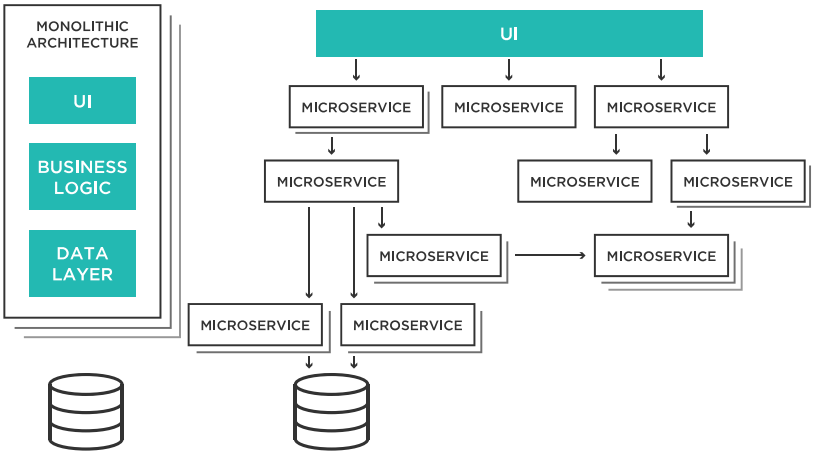

Figure 3: Architecture of microservices versus monolithic application [31].

tends first [32]. However, these self-contained systems still encapsulate everything of one particular service, so one is still structured like a monolith. From that stage, they can be further divided into microservices. When creating the structure of the microservices, usually a domain-driven design is applied, so the services are modelled according to the business logic [33] and create bounded contexts — enforce clear boundaries of code dependencies and team responsibilities [33, p.335].

2.1.7 Stateless and Stateful Services

Services that do not contain data and only process and forward it to other services are called stateless services [34]. Therefore, they can easily be replaced, updated, scaled and moved to different servers — as long as the stateless services maintain the same API endpoints. In the case that an older version has to be restored (rollback), the operation will not have an impact on depending services (as long as API endpoints do not change). Most applications need to store some kind of data, which is why stateful services are needed. A very common example of a stateful service is a database. Since containers are by nature not persistent and created to be replaced easily, special solutions for the handling of persistent data are needed [34]. The easiest way to provide persistent storage to a container is to mount a volume of the host inside the container. This way, the container can be replaced or updated without losing the data. With this approach, the container depends on the exact host and cannot be moved to another server. Scaling of stateful services is particularly hard because simply deleting a container could cause data loss. Therefore, a range of distributed database systems emerged which deal with the replication and synchronisation of data [35]. Apart from data storage, the data structure is another challenge of stateful services. Changes to the data structure need to be performed across all instances that access a data storage at the same time so that no old instance gets access to the new data structures. Even worse are cases where a database update needs to be rolled back, because many operations are not revertable (like

deleting data), so rollbacks of stateful services are avoided when possible. Often, updates to the database container are irreversible too because an older version sometimes cannot access the data structures of a newer version. During the (forward) update, the data structures in the file system are upgraded, but a downgrade is usually not implemented. A typical application contains both stateful and stateless parts since some data almost always needs to be stored, but the microservices itself are easier to maintain if they are stateless. In most cases, one or multiple databases are paired with many microservices.

2.1.8 Load Balancing

Since stateless services do not retain a state, multiple instances can run in parallel. To deal with high loads and distribute incoming requests to instances, load balancing mechanisms are utilised to distribute the load evenly among all active instances and handle the failure of single instances [36]. Specific load balancers or solutions integrated into the server cluster environment perform these job according to different load balancing strategies. Compared to client-side load balancing this approach is more flexible since the microservices themselves only have to access one specific endpoint and do not have to deal with the load balancing process.

2.1.9 Cloud

Containers enable the dynamic deployment of applications and parts of them independently from the underlying hardware. Environments that allow easy hardware-independent deployments of containers are called clouds. They abstract hardware for easy management and enable load-balancing and scaling of containers and logical groups of them [37], [38]. Although a cloud can be managed by a dedicated infrastructure team in a local data centre, most cloud environments are hosted by Amazon (Amazon Web Services), Microsoft (Azure) or Google (Google Cloud Platform). These cloud providers will ensure high availability through hardware redundancies, low latency and high data rates by placing their data centres at strategic locations all around the world and provide management options to order virtual machines and other resources in seconds for varying durations [39, p.17]. This results in cost savings and reduced risk (for example due to hardware failure) because the operation of the data centre is out-sourced.

2.1.10 Zero-Downtime Updates

One advantage of cloud environments is that updates can be performed without any downtime and very high availability is guaranteed since the load balancer can redirect traffic dynamically based on the availability of service instances [40, p.89]. In case of an update, new containers can be created without impacting the old version and according to the concept of canary testing, the new version will be rolled out to a subset

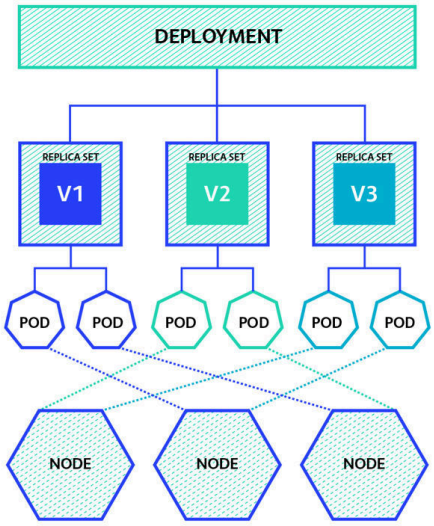

Figure 4: Relationship between Deployments, ReplicaSets, Pods and Nodes in Kubernetes [42].

of users by the load balancer and only if the service is performing as expected will all traffic forwarded to the new instances. If problems arise, the old version is available for requests and can be switched live in seconds again. Another deployment strategy is blue/green deployments where two versions run in parallel and only after thorough testing in the production environment is the new version shown to real users. Once the old version is not serving users any more, the containers can be removed.
Although cloud providers can provide the environment to run containers, in many cases virtual machines are ordered from the cloud providers and infrastructure engineers to build a cloud for specific software projects or companies. With the need to provision many servers with the same software to host the cloud operating environment multiple infrastructure as code tools emerged to allow automated and often declarative provisioning and setup of machines. These tools — for example Terraform, Ansible or Chef — read configurations from a set of files that might contain instructions to create users, file structures and install software [41]. Therefore, the configuration of potentially hundreds of computers can be adjusted through a text file and applied easily.

2.1.11 Kubernetes

The predominantly used container orchestration system is Kubernetes, which was initially developed by Google [43]. It schedules and deploys Docker containers on machines that are part of the Kubernetes cluster based on their resource requirements. All Kubernetes resources are declared in configuration files using the data-serialisation language YAML, which can be transferred and updated using the command line client `kubectl`. To separate multiple projects on the same Kubernetes cluster, they can be organised in

namespaces. Fine-grained access control based on external authentication providers can be set up so multiple developer teams can share the same cluster. The most important resources are Deployments. A single Deployment creates a controller to supervise a number of Pods of the same kind — multiple replicas of a microservice. This static controller created by a Deployment is called ReplicaSet. When a Deployment is updated, a new ReplicaSet is created that runs a new version of Pods, or the existing ReplicaSet is updated to increase or decrease the number of Pods. The Deployment also handles operations like blue/green or canary updates by providing another abstraction layer over ReplicaSets and is the recommended way to interact with those [44, p.77]. A Pod usually contains a single Docker container, but if two containers always work as one unit, they can also be combined in one Pod. The relations between Deployments, ReplicaSets, Pods and Hardware Nodes are depicted in Figure 4. Resource requirements (CPU, memory) for Pods can be defined in Deployments so the scheduler can place Pods more effectively on the available Kubernetes nodes [45, p.70]. The connection between Pods needs to be set up specifically through the creation of Services. Services provide a unique endpoint (hostname) that other Pods can resolve and also automatically implement load balancing to all Pods behind a Service. However, a Service does not just assign a hostname to a set of IP addresses, it only forwards traffic on specific ports that are necessary to access a microservice as defined in the Service definition. Services are particularly important because they provide a static endpoint to dynamically changing Pods with different IP addresses on potentially different Kubernetes cluster nodes [3, p.180]. While Services provide connectivity between Pods within the cluster, an Ingress manages the communication to the outside of the cluster [46, p.156]. Typically an Ingress implements TLS termination to encrypt outgoing connections and forwards HTTP and HTTPS traffic on different (sub-)domains to services within the cluster. Different implementations for an Ingress are available, for example, based on Nginx or HAProxy, and Kubernetes provides an independent configuration interface. Kubernetes supports ConfigMap resources, which can be mapped as environment variables or files into Pods so adjustable parameters can be submitted to microservices based on the cluster or environment they are in. Secret resources are very similar to ConfigMaps but supposed to be used for important and confidential information like authentication tokens and passwords [43, p.118]. For stateful services PersistentVolumes (PV) and PersistentVolumeClaims (PVC) can be created. When a PVC is assigned to a Pod, files in specific folders will persist during Pod restarts and modifications. Depending on the storage provider a volume could be created automatically or manually, for example on the Pod's host or on a distributed file system, for the Pod to use.

2.1.12 Alternatives to Kubernetes

Apart from plain Kubernetes, a few other container orchestration platforms are common. OpenShift by RedHat is based on Kubernetes but adds better graphical cluster management, an internal Docker registry and native support for certain kinds of CI/CD pipelines [47, p.2]. Despite the large modifications, OpenShift is still compatible with the native Kubernetes management tool and most resource definitions. Specifically for stateful services Mesosphere DC/OS was created which is using the Apache Mesos scheduler and kernel [48, p.379]. Therefore, DC/OS does not only provide a runtime for Docker containers but also a simpler environment for applications that can be started from a binary alone, the Mesos Containerizer. Regardless, Kubernetes is the most commonly used container orchestration platform and is therefore also supported by the most extensions and tools to simplify and enhance the deployment and operation of microservices applications [45].

2.1.13 Monitoring

Most production-level applications interact with an application performance monitoring (APM), which supervises the application and verifies that it works according to set specifications regarding availability, average or peak processing times and the number of served users. Specifically in microservices APM plays an important role because of the higher complexity compared to monolithic applications. They contain more independent services that can fail or fail to communicate with other services effectively [49, p.292]. Since many microservices continue to run when one of them is not responsive, service problems are not always obvious and more importantly for developers, their origins are hard to pinpoint. Identifying the cause of delays is particularly hard when many microservices are interacting to fulfil a request and application performance monitoring can support these efforts [50]. Usually, APM consists of a component within the services that collect and publish internal metrics and an external software which collects and processes these metrics [51]. It provides a dashboard and potentially forwards resulting information to an alerting system to notify operators about inconsistencies, for example via text message or email.

2.2 GitOps

The traditional and still most commonly used kind of pipeline, where the final build artefact — usually a Docker image — gets deployed in a cluster as the final pipeline step, is the push pipeline. Obviously, the CI/CD tool where the pipeline runs needs to have write access to the cluster in order to deploy anything. The deployment will start immediately once the build process is finished and instructions about cluster con-

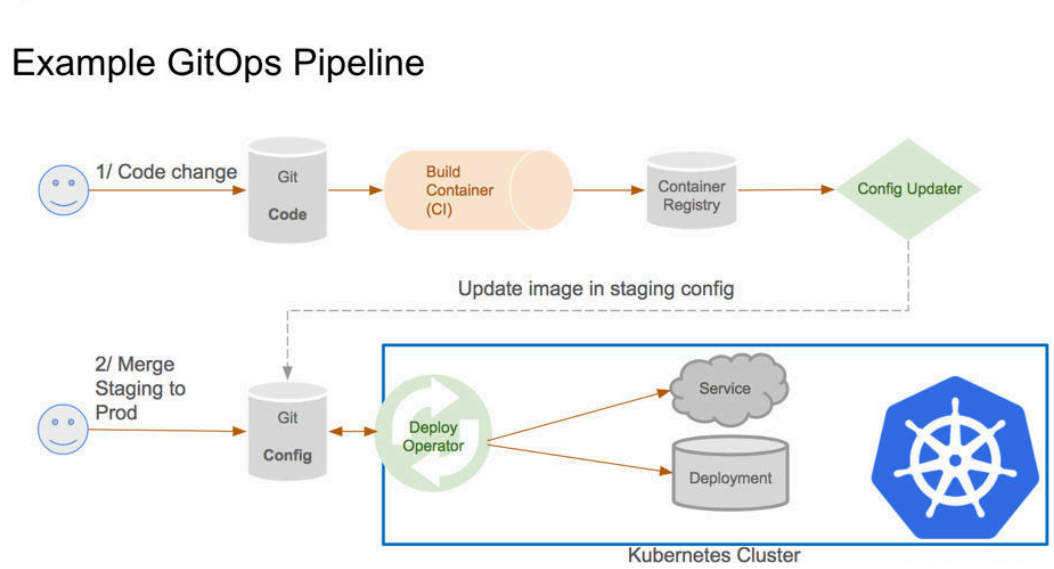

Figure 5: Example GitOps Pipeline [54].

figuration changes (mostly updated Docker image tags) are submitted to the cluster [52, p.25]. Through direct contact, the pipeline receives feedback on the validity of the submitted configuration, which can be incorporated in the final result of the pipeline. Plugins for the CI/CD system are required for the communication with the cluster, but no independent tools are necessary. The pipeline steps, which are defined in the source code repository are specific to a cluster environment [53] — the cluster location can be set through external environment variables but OpenShift resource definitions will not work in a Kubernetes cluster.

In pull pipelines, deployment artefacts are pulled into the cluster. Therefore, the last pipeline step of the push pipeline is omitted after the image has been pushed into a Docker registry. Thus, there is no direct contact between the CI/CD pipeline and the cluster [55] as shown in Figure 5. With this approach, the cluster state is declared in a separate git repository for all microservices of an application, which is why it is also called GitOps [3]. So, in the pipeline, the final step is to clone this deployment repository and commit a new image tag to be used for the microservice. Alternatively, a pull request can be created by the pipeline if every deployment should be signed off by a human operator [56, p.278]. On the other end, in the cluster, a GitOps operator runs and regularly compares the cluster state with the deployment repository. After the CI/CD pipeline has finished, the GitOps operator detects the new image tag in the deployment repository and deploys a new version. Often, this operator provides a dashboard and monitoring endpoints to detect and log deviations caused by cluster degradation or deployment repository updates and connects to existing notification solutions to reach responsible people quickly in case of errors or unexpected results. The whole system is described declaratively in the deployment repository, so every change made to the cluster will be

recorded with a timestamp and reason, which allows easy auditing and rollbacks [54]. Different GitOps operators can theoretically work with the same deployment repository and deploy into different clusters without a change to the pipeline. In fact, a new deployment in a new cluster can be created from the deployment repository alone if the old cluster gets damaged or needs to be replaced for another reason. The only exception is the stateful services' data, which needs to be manually imported into the new location.

2.2.1 Tools that enable GitOps

There are multiple tools which automatically update Kubernetes Deployments when a new image with the same or different tag is available. They, therefore, form the pull part of a pipeline. However, these tools do not follow the declarative principle that the exact cluster state is defined in a git repository. Thus, they only realise some of the GitOps advantages (and disadvantages) — the CI/CD system does not need access to the cluster, but there is no clear deployment history, especially in which combination multiple images were active in the cluster.

Watchtower

The most basic tool for automatically updating Docker images is Watchtower [57]. It works directly with the Docker daemon of a regular Docker installation, that means it does not support clusters. Watchtower runs as a Docker container itself, so it needs the Docker socket of the host mounted within the container to access other containers and images, which is a security risk since it allows Watchtower to access all containers' data as well even though that is not necessary for its operation. By default, Watchtower watches all containers (including its own) and checks the image registry for a new image version or tag, but it can also be restricted to a subset of the running containers. If there is a new image, the tool will pull it automatically and recreate the container with the new version. Basic monitoring via email, Slack or Microsoft Teams can be enabled to alert developers of failed updates.

Keel.sh

Keel.sh[45, p.251] provides similar functionality as Watchtower to Kubernetes clusters, so it can also automatically update Deployments with new images. In addition to completely automatic updates, Keel.sh can ask for approval in Slack before performing the update. It can not only regularly poll a Docker registry for image update but also provides an endpoint for webhooks so the CI/CD pipeline or the Docker registry can trigger the update directly. Unfortunately, it does not support changes to anything other than images and does not interact with a git repository.

OpenShift ImageStreams

OpenShift clusters natively support automatic image updates through ImageStreams in connection with the internal or an external Docker registry [58]. They can be configured in the same way as Kubernetes resources through YAML files to trigger a new Deployment or OpenShift Build that utilises the new image. ImageStreams contain all metadata of an image and can add additional tags to existing images to be referenced by Deployments and OpenShift Builds [59].

Gitkube

Gitkube[45, p.250][2] aims to replace the whole CI/CD pipeline from within the cluster by watching for changes of a git source code repository. When deploying the Gitkube operator to the cluster, it creates a new git repository that can be used to trigger new builds and deployments. Therefore, not every commit to the main code repository has to be deployed and build, but every build in the Gitkube repository (probably) ends up deployed in the cluster. The build itself is restricted to a Docker build, so Gitkube reads all build instructions from a Dockerfile in the repository, which is not a real pipeline definition since the Docker build is usually the last step of a build; and building and testing the commit is done outside of the Docker build. Utilising multi-stage Docker builds [60, p.80], a full build pipeline can be emulated from a Dockerfile alone by basically chaining multiple Docker builds and copying results between them [61, p.5]. Since normal Docker builds add a new layer with every instruction, the resulting Docker image would contain all the source code and operations performed in addition to the final build artefact. That is why a multi-stage Docker build, where the first layers are skipped, is required to create a full build using Gitkube. An additional downside of multi-stage Docker builds is that build dependencies need to be downloaded on every build (because volumes cannot be mounted into Dockerfile builds), which can be worked around through custom build base images adding more complexity to a system that is made to be simple [61]. To conclude, Gitkube allows quick deployments for simple services that do not require complex test and build stages, like for example python services, but for professional and demanding continuous integration and continuous deployment Gitkube needs many workarounds to add required functionality. Additionally, the git repository does not define or contain the exact state that is to be deployed in the cluster, but only the source code.

Weave Flux

The company that basically invented GitOps, Weaveworks, released an open-source GitOps operator called Flux [45, p.251]. Weaveworks recommends a GitOps workflow as depicted in Figure 6. The Flux operator runs within a Kubernetes or OpenShift Cluster

2 https://gitkube.sh/

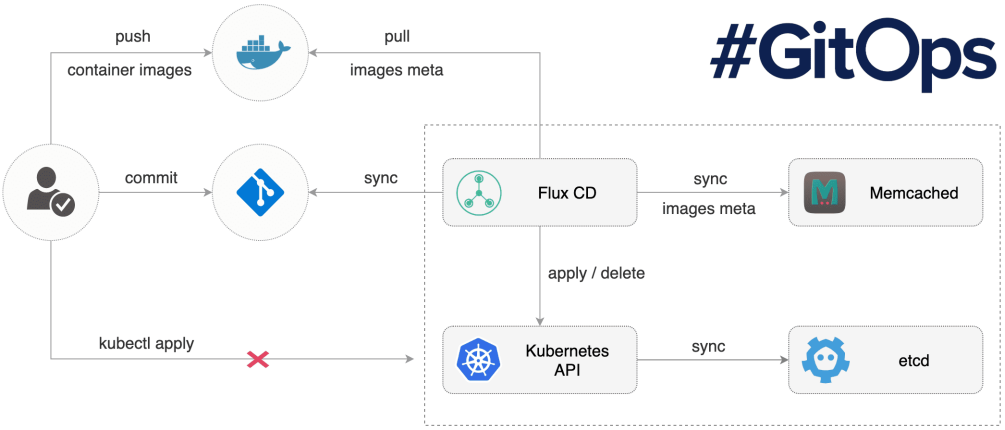

Figure 6: Weave Flux as GitOps operator with separate data store [62].

and uses a separately configured git repository to read the desired cluster state from. It requires write permission to the git repository to update image tags it found in a Docker registry if the automatic sync feature is enabled [63]. Thus, the deployment repository is modified through an additional channel from inside the cluster instead of just through the CI/CD pipeline or by selected people with write access to the repository. This automatic sync feature allows to automatically update deployments based on updated image tags according to regular expressions or semantic versioning[3] filters. The Flux operator can be controlled and configured by a command-line client fluxctl [64], which utilises the Kubernetes `kubectl` client to contact Flux within the cluster. Therefore, users who want to make changes to the Flux operator need to have cluster access credentials. Alternatively, the API port of Flux could be exposed, but since it does not require or allow authentication, it would be publicly accessible and pose a significant security threat. The Flux operator can read plain Kubernetes YAML files or interpret files created by the Kubernetes templating tool Helm[4]. However, the cluster operator to process Helm templates called Tiller needs to be installed into the cluster manually [65]. Customisation and fine-grained control over which deployments should be automatically updated, and according to which filters, can be defined through fluxctl (as described previously) or by adding annotations to the Kubernetes YAML declarations of the respective resources [66]. A single Flux operator can only work with one deployment repository, and therefore one project. Thus, to run multiple projects in the same cluster, multiple Flux operators need to be deployed. However, a Flux operator will track image changes and available new images for the whole cluster, even if resources are not specified in the deployment repository. Flux focuses mainly on images, as image changes can be rolled back, but incorporates the core GitOps philosophy that the whole deployment should be committed to git.

3 https://semver.org/
4 https://helm.sh/

Weave Cloud

Apart from user management, Flux also lacks a frontend and monitoring, which is all integrated into the commercial Weave Cloud[5] cloud service. It interacts with a Flux operator within a cluster via an additional microservice gateway, the Weave Cloud Agent. In the browser, users can see all resources within the cluster, but those not defined in git are marked as read-only. Additionally, users can review Pods' logs, get their Kubernetes resource description and open a shell into the Pod [67]. User management in Weave Cloud is relatively basic since it is based on separate accounts associated with email addresses and only three available roles to assign to the users. Interaction with external authentication providers, for example via OpenID Connect, is not supported. The powerful monitoring solution Prometheus is already integrated into Weave Cloud and notifications to Slack, email and others can be easily configured [68]. Unfortunately, Weave Cloud is a cloud service provided by Weaveworks, so it cannot be self-hosted and the security of this critical infrastructure data and access to the cluster depends on the US-based company and their security considerations.

ArgoCD

The final GitOps tool presented in this section is ArgoCD[6], which has a similar feature set as the whole Weave Cloud ecosystem, but as an open-source application. It reads the whole cluster configuration from a git repository and deploys changes automatically or manually to the cluster. Currently, only Kubernetes and since recently released v1.0.0 OpenShift is supported. ArgoCD strongly focuses on security by providing support for many authentication providers and roles. Its frontend, which is also deployed in the cluster with the GitOps operator, allows to manage resources manually, perform synchronisations and rollback the cluster to previous commits. The current cluster state can be easily observed and differences to the desired state are highlighted. One ArgoCD instance supports multiple independent and separated projects in different Kubernetes namespaces [69]. Monitoring and alerting is not included within ArgoCD, but it provides a metrics endpoint to be consumed by an external Prometheus instance [3, p.98].
In conclusion, only ArgoCD and Weave Cloud allow fully declarative cluster deployments that can be easily observed and monitored. Because Weave Flux (the basis of Weave Cloud) has been in development for some years, Weave's solution is most stable and also supports more different cluster environments. However, since Weave Cloud is a commercial non-free service its use might not be allowed for privacy or security reasons in many professional software development projects in Germany. ArgoCD, on the other hand, is rapidly evolving and due to its transparent structure, public code base and support for third-party authentication providers appears to be a suitable solution to

5 https://cloud.weave.works
6 https://argoproj.github.io/argo-cd/

introduce proper GitOps into software development projects. Therefore, it will be used in the following chapters of this thesis.

2.2.2 ArgoCD

Looking at ArgoCD more closely, it offers almost all features of the commercial Weave Cloud while adding many additional endpoints for customisations and extensions. Its installation is performed by a single `kubectl` command which installs all modules into a custom namespace, that allows the removal with a single command as well [3, p.99]. As mentioned before, ArgoCD's source code is publicly hosted on GitHub and new commits are made and pull requests opened on a daily basis, which resulted in the release of version 1.0.0 in mid-May 2019 and more releases following at a rapid pace. ArgoCD provides a web management platform and a command-line utility in addition to the GitOps operator, which controls the cluster state itself. Through the use of "projects", multiple teams can use the same ArgoCD instance without sharing access to their development setups [69]. One project can contain multiple "applications", which are defined each by one deployment repository and deployed into a single Kubernetes namespace. The authentication at the deployment repository can be performed via username and password (in GitLab: deployment token) or via SSH key [70]. Although most configuration can be done via command line or web interface, an SSH key can only be added on the command line. As mentioned, the web interface allows the setup of projects and applications but the main and most important feature is the application dashboard, which shows all resources created by ArgoCD as well as resulting resources created by Kubernetes on that basis. On the dashboard, ArgoCD shows the ReplicaSet created by the Deployment which is defined in the deployment repository as well as the resulting Pods and the status and health of all of them [3, p.102]. Users can view the Pods' logs and delete individual resources directly in ArgoCD if they have the necessary permissions. But, deleted Pods will be immediately recreated by Kubernetes based on the ReplicaSet definition and deleted ReplicaSets will be recreated from the Deployment. If the whole Deployment gets deleted, ArgoCD will create a new Deployment if the synchronisation mode is set to automatic. An application's synchronisation mode can be chosen from "automatic with pruning", "automatic without pruning" and "manual". The available rollback functionality to restore a previously deployed git commit is only available in manual mode since the latest version would be applied immediately after the rollback if an automatic mode was active.

The definitions in the deployment repository are supported in four different languages [71]. Firstly, the repository can contain plain Kubernetes YAML files as they could be applied manually by `kubectl apply`. Secondly, the definitions can be written in the Helm templating language[7], which allows storing all important values in one place to

7 https://helm.sh/

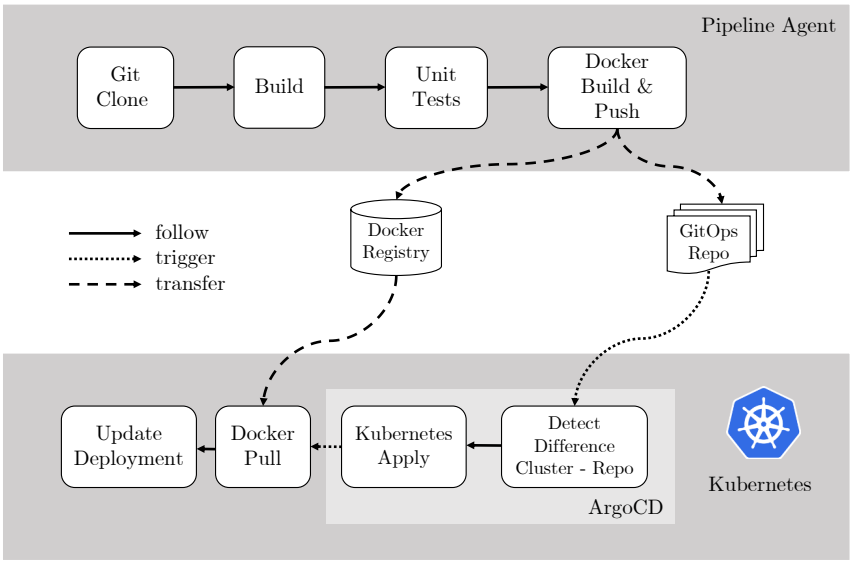

Figure 7: ArgoCD pipeline with write access to deployment repository.

be used to replace templating variables in all other files. This is particularly helpful for different staging environments because most of the deployment will be the same and only some tags, secrets or endpoints would need to be changed between stages. The next Kubernetes configuration option is Ksonnet [46, p.504], which enables developers to create Kubernetes resources as JSON files. These files structured in three layers: "parts" form the basis, which can be grouped into "prototypes". "prototypes" with added parameters are called "components" and multiple "components" form a deployable application. Ksonnet has been discontinued in February 2019 [72], but is still supported by ArgoCD. Another way to specify a deployment is via Kustomize [73][8], which is a declarative language, and explicitly not a templating language. Instead, configurations are created as overlays, which can be stacked infinitely to add or overwrite parameters to Kubernetes YAML files. Additionally, it supports the generation of Kubernetes ConfigMaps and Secrets. Since March 2019 kustomize is included in `kubectl` and therefore accessible without separate installation. ArgoCD also allows creating custom plugins to parse even more configuration formats.

Concerning authorisation and authentication, ArgoCD supports Single Sign On (SSO) via OpenID Connect and the more basic Oauth2 protocol as well as LDAP, GitLab, Microsoft and many more via the OpenID Connect Provider dex [3, p.98]. For these external users, complex roles and permissions can be configured to limit users' permissions to the exact subset they need to perform the tasks on their specific projects. In addition to the git log, ArgoCD stores all events with their origin to allow comprehensive audits [3, p.98]. Access tokens can be created to interact with ArgoCD in an automated fashion and internal as well as external webhooks can be called or created to trigger refreshing

8 https://kustomize.io/

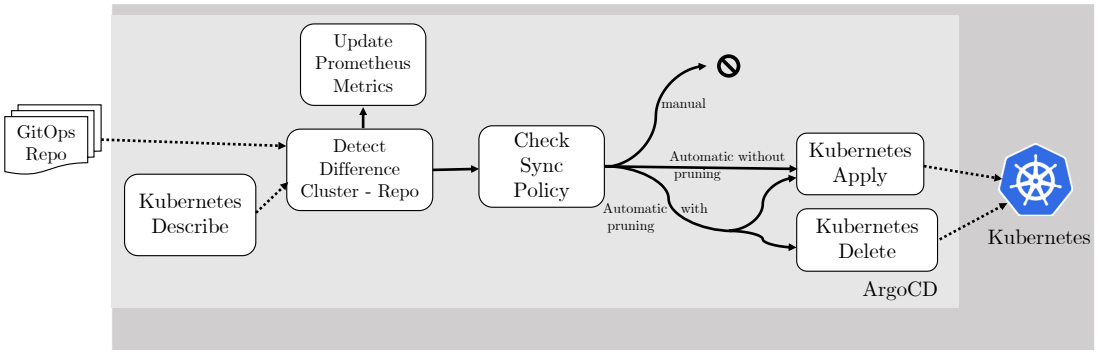

Figure 8: ArgoCD Update and Deployment Process.

of the deployment repository or trigger external services before or after synchronisation occurs. Since Prometheus metrics are exposed automatically, comprehensive monitoring can be facilitated with external tools [3, p.98]. One important feature that does not exist in ArgoCD so far is to watch a Docker image and automatically deploy new versions once they are pushed to a Docker registry. The Flux operator supports automatic image updates and updates the GitOps repository on its own.

The detailed process of how ArgoCD processes changes in a git repository is depicted in Figure 8. The ArgoCD daemon reads the deployment repository periodically (by default: every three minutes) and creates a diff with the generated cluster state (including Deployments, Secrets, StatefulSets and ConfigMaps) as returned by an API from Kubernetes. If the automatic mode is enabled, resource definitions of diverging resources will be updated from the repository immediately [74]. When automatic synchronisation with pruning is active, resources not declared in git but existing in the cluster will be removed as well. In manual mode, differences will not be applied to the cluster but displayed in the dashboard. Then, a user can manually apply the new version or rollback to a previous commit. When updating to the newest version, a partial synchronisation where only some resources are updated, can be performed, too. Thus, ArgoCD monitors the cluster continuously, but in addition to objects defined in the repository, it also monitors the general health of resulting resources like Pods and ReplicaSets created by a defined Deployment. As mentioned, the git repository is polled and cached only every three minutes, but changes made to the cluster manually (for example by someone running `kubectl apply`) will be detected much quicker and processed in the same way as a new commit.

2.3 Security

2.3.1 Pipeline Security

This thesis' focus is on the security of the DevOps process. Thus, the CI/CD system is the central operator with the most access to other systems and data stores. For example, in a traditional push pipeline, the CI/CD agent needs to have full write access to the cluster, the Docker registry and read access to the source code repository. If an attacker can successfully control or even access the credentials to the cluster, they could

- delete every deployment and data store without leaving a trace in the cluster [75, p.28]
- deploy additional Pods to access or expose critical data and perform Man-In-The-Middle attacks against communication between microservices or with users
- create a backdoor for easy access at a later time when the vulnerability in the pipeline agent is already fixed
- download all databases' content
- modify existing deployments and services [75, p.30].

These modifications are difficult to detect because the agents' credentials would be used and because detailed logging of successful logins or modifications to the cluster is not enabled by default [76]. Usually, monitoring checks the availability of Services or Pods, but if they are modified such that they still work (or appear to be working), the monitoring system will not detect the difference. Also, additional deployments or simply data leakage is not detected by monitoring since its main goal is to check if services are ready to serve customers' requests.

2.3.2 Pipeline Agent Security

According to the previous section, the security of the whole cluster depends on the CI/CD system with access to the cluster when deploying from a push pipeline, but partly also in a GitOps setup. The Jenkins build server is the most commonly used CI/CD system [77], so analysing Jenkins' security will cover most professional development processes.
The first infiltration point is the pipeline agent's operating system [78, p.95]. It usually runs in a virtual machine on its own without sharing the VM with other services because it irregularly needs many resources which would unpredictably interfere with other services. An advantage is therefore that not many users need to have access to the virtual machine. But on the other hand, someone has to manually and regularly update the operating system, which is often neglected.

Another attack vector is the Jenkins web UI. New versions are released weekly containing new features and fixes for security vulnerabilities, which need to be applied regularly to reduce the attack surface since security fixes often reveal the underlying vulnerability. At time of writing 139 CVE listings have been published, which means over Jenkins' lifetime 139 vulnerabilities have been found in the Jenkins core that have been released to the public.[9] The National Vulnerability Database by the National Institute of Technology in the US also lists vulnerabilities for Jenkins plugins and contains 477 listings[10], which shows one major issue with Jenkins: Plugins are often written by third parties without strict security guidelines and released in the Jenkins marketplace [79]. Through these plugins, access to a Jenkins instance can be gained or privileges escalated. Even if developers release updates for their plugins, these updates have to be applied manually or the respective Jenkins instance stays vulnerable.
In conclusion, keeping a Jenkins instance — and with it the cluster credentials — secure requires much and regular effort to install released updates alone but some vulnerabilities of less used plugins might still stay undetected (by the public). Despite these issues, Jenkins is still the most used CI/CD tool, mainly because of the flexible plugin system containing over 1000 plugins but also because of its long history as one of the first CI/CD tools [77].

2.3.3 Man-In-The-Middle Attacks

A MITM attack is often performed because it does not require access to the system itself, but only the network facilitating the communication between various systems. Its aim is to intercept, read and potentially modify the communication between different actors in a network. In a pipeline, the CI/CD system communicates with git, the Docker registry and the cluster (when using a push pipeline). Usually, all communication is encrypted using TLS, so the attack potential is limited [80].

2.3.4 GitOps

In a GitOps scenario, Jenkins does not have access to the cluster, which reduces the attack surface massively. However, instead of applying new versions to the Kubernetes cluster directly, the pipeline accesses the deployment repository and commits the new version there. This action modifies the cluster state not directly, but indirectly. Because the cluster is independent from the build agent, there does not have to be a route between the two systems and the cluster does not have to expose a management interface at all. Additionally, every change would be documented in the git history (as long as the pipeline user cannot force-push) [54]. But most operations described in the first

9 https://www.cvedetails.com/product/34004/Jenkins-Jenkins.html?vendor_id=15865
10 https://nvd.nist.gov/vuln/search/results?form_type=Basic&results_type=overview&query=jenkins&search_type=all

section of this chapter are still possible, just very easy to detect. Multiple solutions are proposed to mitigate the issue of modifying the cluster through the GitOps repository in an unauthorised way.

The first option is to prevent direct write access to the deployment repository. Instead, the pipeline creates a merge request, which has to be confirmed by a human before any changes are made and the cluster gets updated. This approach is recommended by many GitOps guides [81]. However, such a pipeline is not fully automatic and always requires someone's time and attention before a rollout is performed. Once the change is merged, a GitOps tool like ArgoCD can apply the defined state. This approach provides good protection against unintended changes by developers, scripts and attackers, but slows down the deployment process substantially, considering that ideally many builds happen per day.

Another option is to set the ArgoCD synchronisation setting to manual. Then, someone has to trigger the cluster update manually in ArgoCD [74]. Again, this pipeline is not fully automatic and all changes are supervised, but this time in ArgoCD. This means, the cluster state will not always be as defined in git and could further diverge if multiple commits to the deployment repository are not applied to the cluster. Therefore, the full observability is not always given and the deployment repository cannot be considered the definition of the cluster state at all times. Thus, this option diverges from some core GitOps principles but reliably prevents unauthorised modifications to the cluster through the pipeline.

A slight modification of the previous approach is to enable automatic synchronisation, but without pruning [74]. This setting prevents the deletion of previously defined resources but otherwise updates resources automatically. Then, the pipeline runs without human interaction, but the protection against unintended modifications is very weak since new resources can be deployed and existing modified without consequences.

The final approach presents a solution which limits the attacker's impact considerably while still allowing automatic deployments. The main idea is that the pipeline agent does not interact with the deployment repository any more, and newly tagged images are detected by the GitOps operator or a separate agent. Weaveworks Flux supports such an operation where different rules or filters can be defined to specify which images should be deployed automatically based on the assigned tag [63]. At the same time as Flux updates the cluster, it also commits the new tag to the deployment repository. Thus, an attacker could only change the image to be deployed, but nothing in the deployment structure or access databases. With this approach, the write access originates not in the pipeline, but in a separate operator from within the cluster which does not interact with users. Therefore, the attack surface is minimised at the price of additional complexity. Modifications to the deployment structure can still be performed by selected people with write access to the deployment repository. The cluster state will always correspond to the GitOps repository because automatic synchronisation can be enabled safely. Unfor-

Push Pipeline	Pull Pipeline
▶ Pipeline operator runs entire pipeline	▶ Second part of pipeline runs within cluster
▶ Developers can access system which contains credentials	▶ GitOps operator runs independently
▶ Credentials to cluster stored with pipeline operator	▶ Cluster credentials only stored within cluster
▶ No easily accessible history of previous deployments	▶ Full deployment history available in git log
▶ Attacker taking over pipeline can modify cluster unconditionally	▶ Attacker taking over pipeline operator can only deploy new images of selected microservices
▶ Cluster needs to be reachable via network from build agent	▶ Cluster can be physically air-gapped from build agent
▶ Cluster has to expose management port	▶ Management ports of cluster can be closed entirely

Table 2.1: Comparison of Push and Pull Pipeline Security.

tunately, ArgoCD does not support watching image registries and committing changes to git. However, the demand for such a feature has been voiced by the community in ArgoCD's issue tracker [82].

Summary

In comparison to push pipelines, the GitOps approach is more secure by default through the git repository. Because of the 'pull' part of the pipeline, a separation of the vulnerable and important infrastructure is achieved and cluster credentials can never leak. Table 2.1 gives an overview of security-relevant aspects of push and pull pipelines, which have been described in the previous sections. Through the use of a deployment repository all modifications are tracked (as long as force-pushing is not allowed) so even temporary alterations are retained for analysis and a single unauthorised modification is directly visible [54]. Additionally, direct modifications to the cluster will be detected by the GitOps operator and can alert the people responsible, which is very hard to implement in a deployment setup via push pipeline without a clear deployment definition. Thus, GitOps pipelines are more resilient against attacks from the deployment operator and multiple intuitive ways exist to prevent unauthorised access effectively. Unfortunately, these methods involve a human confirming every deployment, which reduces the deployment speed and depends on selected team members to be available to confirm changes.

The different GitOps solutions to reduce the attack surface can also be applied to different stages. Therefore, the automatic — less secure — option could be used for integration and staging environments, while for production the confirmation of one or multiple people is required. Then, the high velocity is retained for most builds while the important environment stays separated and secure. Traditionally, builds would not start to production automatically, but security-wise there is no separation between stages and credentials to the production cluster are still stored on the CI/CD agent.
Although this section assumes a self-hosted CI/CD agent and CI/CD systems in the cloud are growing in popularity, the results are easily transferable. In this case software updates and security policies are controlled by a central external authority, which might spend more or less time on this topic than individual teams of developers. However, a shared cloud service for multiple teams and even companies provides a higher-value target than a single Jenkins instance, which might only be accessible from an internal network. Another important aspect is that the company (and its employees) hosting the CI/CD system will have cluster access too, if the implemented push pipeline connects to a cluster [83]. Thus, even in that case, a GitOps pipeline provides security benefits.
In this chapter, the security of the source code itself is not analysed because this attack scenario is independent of the pipeline and includes many other areas like social engineering. Nevertheless, malicious code can pose a serious threat to any software development project.

2.4 Rollback

Another important requirement for dealing with attacks and mistakes during the deployment process is the ability to revert every change made to the environment. Ideally, not only single deployments should be revertable, but also multiple updates to various microservices over longer time periods. Since most microservices are stateless, they are fully described by their Docker image and the Kubernetes Deployment. The easiest approach to rollbacks is to simply deploy an older image tag, which only works in static environments where only the image changes. However, two issues arise with this rollback strategy. If during the deployment process endpoints, routes, services or the number of instances are changed too, reverting the image tag alone will not necessarily result in a working deployment and definitely not in the previously deployed state. Additionally, if multiple microservices run together and not all of their versions are compatible with each other, further dependent downgrades have to be performed to return to a working state. In a GitOps deployment strategy, on the other hand, the whole deployment structure can be determined from the deployment repository and a single or multiple commits can be reverted to achieve the exact cluster state as it was previously including all microservices, their images and dependent resources [54]. Because of the git history, attacks to the cluster via the deployment repository can be analysed thoroughly and of

course reverted too, to identify the exact steps taken by the attackers, their objectives and the damage they caused.
Of course, stateful services like databases still have to be considered separately since both their contents and their structure are not reflected in the deployment repository. Thus, for database rollbacks and upgrades, separate strategies and methods are needed. Therefore, operations like regular backups of the stored data retain their importance even with GitOps, which only tracks the deployment structure [84, p.99].

2.5 Monitoring

The earlier an attack to a software system is detected, the better are the chances to identify the origin and prevent damage. Thus, comprehensive monitoring is a crucial part of information security in addition to its general purpose of supervising the stability and operation of applications (see Chapter 2.1.13). GitOps does not transform monitoring and logging best practises, but adds another perspective to existing service availability and security monitoring. Since the desired state is available in the deployment repository at all times, a GitOps monitoring agent can easily detect modifications, additional deployed resources or deleted deployments in the cluster if they are performed outside the verified channel of the GitOps repository [54]. This approach does not require much separate configuration as normal monitoring solutions would since deployment changes directly influence the monitoring rules derived from the deployment repository.

3

Approach

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."
Edsger Dijkstra

This chapter presents the software developed to enhance the security of open-source GitOps pipelines.

3.1 Problem

At present, ArgoCD supports deploying to a cluster from a deployment repository with static definitions [85]. However, as described in the previous chapter, the pipeline needs to either have direct write access to the deployment repository or a human operator needs to approve every change. The security risks of direct write access have been discussed resulting in the recommendation to avoid this setup. During most builds, only the image tag changes to a new version, so ideally the pipeline should only be able to influence this parameter in the Kubernetes Deployments. The Flux operator by Weaveworks can watch a Docker registry and automatically deploy and commit a new version [63]. However, this is not possible with ArgoCD, which currently does not allow any write operations to the deployment repository.
A new tool could bridge this gap by regularly polling the Docker registry and updating referenced image tags in Deployments in a GitOps repository when a new is image pushed as shown in Figure 9. Then, only this new tool or operator needs to have write access to the deployment repository, which does not directly interact with developers, so the attack surface is quite low. To eliminate the delay arising from polling the registry, the tool could expose a webhook to be called by the pipeline once the image is pushed. Ideally, the operator can be configured to update the Deployments according to specific filters, so that not every new image gets deployed instantly. For example, the production environment should not receive every build immediately.

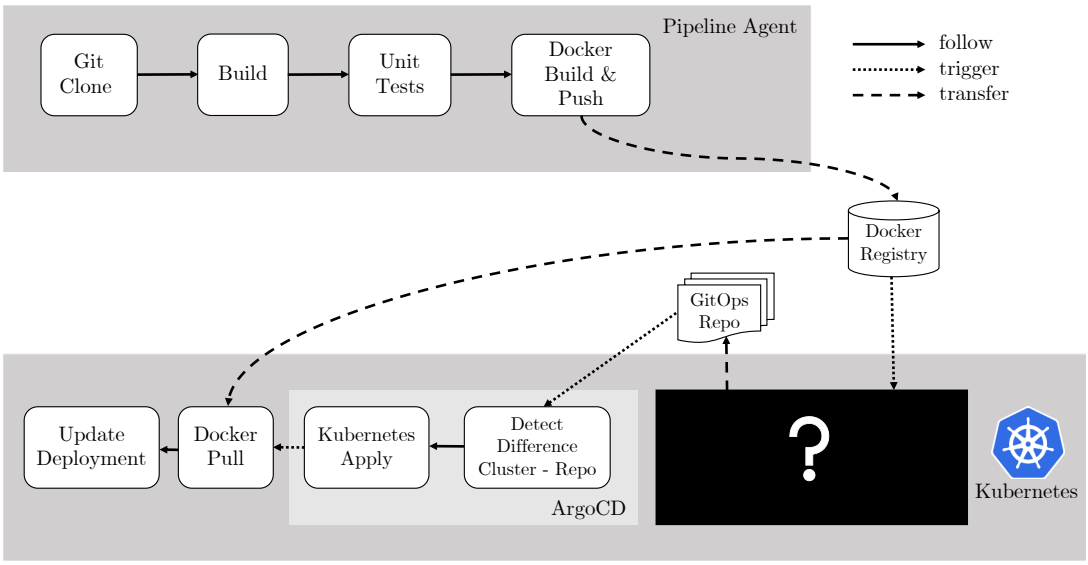

Figure 9: Principal setup with new tool.

3.2 Implementation

The previously presented software Keel (see 2.2.1) already shares features with the tool described in the previous section. It watches a Docker registry to detect new images pushed to it, but does not interact with a git repository [45, p.251]. Instead, it exclusively communicates with the Kubernetes cluster to read configured images and update the Deployments. Thus, it does not comply with retaining a deployment history and the concept of declarative deployments. If the communication with the cluster is replaced by a new submodule for the deployment repository, much of Keel's core, like version processing, secrets handling and even deployment notifications can be reused. This submodule would have to handle cloning, pulling, updating, committing and pushing programmatically and manage the replacement of the image tag in the right place, while Keel only instructs the Kubernetes cluster to replace the image without complex operations on the deployment YAML files or templates.

Since many deployment repositories use Helm templates (see 2.2.2) for easier management of Deployments, Helm templates should be supported by the new tool in addition to plain YAML files. Because Helm interacts with a Kubernetes cluster and an agent within the cluster, it has to be modified to process templates locally and extract images without the need for a cluster since the processed Deployments should not be deployed by this tool anyway. Dependencies from Helm's source could conflict with Keel's dependencies too because both rely on Kubernetes modules in potentially different versions.

Good arguments speak for and against forking Keel versus creating a completely new tool to satisfy the requirements. Keel already has a large codebase which developed over more than two years and 1100 commits that is hard to understand and extend without

explanations [86]. On the other hand, many features that developed over that time are helpful for the new use-case too, like for example notifications to many providers, approvals via endpoints or a Slack bot and comprehensive filter rules support. Additionally, Keel's core perfectly aligns with the requirement of processing an image and retrieving newer versions from a registry. Although two years in software development is not much time in general, in the Kubernetes and DevOps domain many changes and updates have happened, so using two-year-old Kubernetes code and dependencies can limit or hinder development and integration of external and new code. The overarching structure of Keel simplifies the organisation of new source code and the huge code base provides many examples how to use data structures or internal functions efficiently and already provides data structures for most resources and information to be processed. But on the other hand, the structure was created for a specific purpose and with a new purpose, the structure might not fit or be ideal anymore. Also, some features of Keel create a huge overhead, both in terms of source code and dependencies, which are unnecessary but deeply ingrained in many functions, like the Kubernetes integration and communication. The interaction between the tool and Kubernetes compared to a git repository is also quite different. The cluster is always available and provides endpoints and event triggers for both simple and complex actions at any time, but a git repository has to be polled regularly to detect changes but also only provides a folder and files, not native Kubernetes objects. Even bigger are the differences during the update process. Instead of calling an update endpoint, a file has to be modified in the right place and then pushed to the git remote.

After reviewing all arguments, the decision has been made to fork Keel. It is available on GitHub under the new name Bow[1]. The main reason is that a modified Keel would have many more features by default (like notifications, approvals, webhooks) as soon as it has been adapted to use git compared to a new tool built from scratch. Thus, even if forking requires slightly more work, the outcome will greatly exceed a fully self-built tool.

Keel is written in Golang (Go), which is a fairly new programming language developed at Google that provides an alternative to C for high-performance compiled software with a strong focus on concurrency via message passing [39, p.9]. Most DevOps tools are written in Go, including Docker and Kubernetes itself [87, p.49]. Therefore, code snippets and packages can be imported and used directly from the Kubernetes source code or other official management tools. For example, Helm source code can be imported or modified to access the `helm template` command, which processes templates locally without accessing a Kubernetes cluster.

The main challenge transforming Keel was to untangle it from Kubernetes because it needed a running Kubernetes cluster for many operations like checking images, updating deployments and triggering webhooks. Although some of these operations do not

1 https://github.com/Alwinius/bow

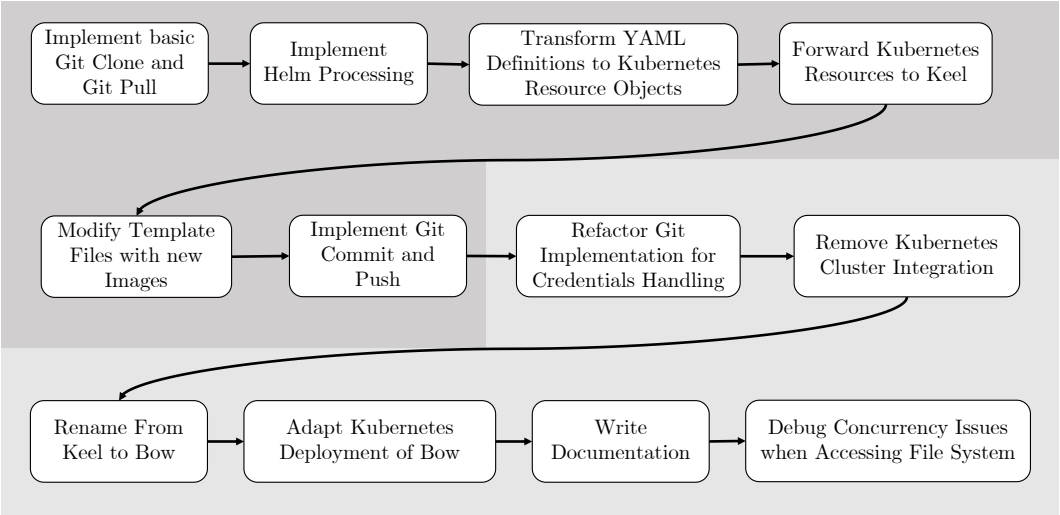

Figure 10: Development process of Bow.

need the cluster explicitly, they were written in connection with other instructions that communicate with Kubernetes and thus had to be reimplemented. That everything communicating with Kubernetes had to be removed or replaced was clear from the beginning, but doing so at the start hinders development since many working examples would be removed and a completely broken software remains. On the other hand, leaving too much unneeded code increases (or keeps) the overall complexity and thus impedes navigating the code while developing. Another difficulty was that internal Kubernetes resource definitions require Kubernetes package dependencies in a very old version, which is incompatible with source code from Helm needed to process templates.

3.3 Process

The development of Bow can be divided in two phases as depicted in Figure 10. The goal of the first phase was to modify Keel to achieve a working update of the deployment repository. Thus, the cloning of the git repository was the first step to implement, followed by the Helm processing to acquire full Kubernetes YAML definitions from the templates stored in the deployment repository. Then, the Kubernetes definitions had to be transformed into Kubernetes objects within Golang for further processing. The development up to this point could have been performed independently from Keel. To prevent dependency conflicts at a later stage and ensure that the implemented steps fit into the main code base, the new code has been added to Keel from the beginning. Next, the Kubernetes objects were passed to Keel to automatically extract images and tags and query the registry for new tags. In the next step, the new image tag had to be written into the git repository. The challenge in this step was that the original image had been retrieved from a processed template and the origin of the image tag

had to be found first in the template files in order to update it. Next was the relatively simple step of creating a commit from the modified files and push it to the remote repository. This concludes the first phase, because at that point the first successful deployment (update of the repository followed by ArgoCD automatically detecting it) was achieved. However, major changes had to be made in the second phase to support effective handling of configuration parameters and repository credentials. Next, the Kubernetes Cluster integration was removed. Until that point, Keel still required a working `kubectl` command-line client to start, even though the cluster was not supplying images to watch any more. Then, the project was renamed from Keel to Bow, which required code changes in almost all code files due to the absolute imports recommended in Golang [88, p.137]. To deploy Bow in a Kubernetes cluster, rather than running the compiled binary locally, the original Keel Kubernetes definitions had to modified to support additional environment variables to define necessary parameters. Since Bow was developed in a public GitHub repository and its usage and further development is encouraged, a basic documentation was created.[2] Finally, after some tests, concurrency issues were resolved when multiple threads accessed the local git repository at the same time.

2 https://github.com/Alwinius/bow

4

Evaluation

"Go is not meant to innovate programming theory. It's meant to innovate programming practice."

Samuel Tesla

This chapter will evaluate the developed tool both in general and on a specific case study to assess its effectiveness in regards to various aspects.

4.1 Features of the Developed Tool Bow

The forked Keel, which is named Bow, performs the required task of automatically updating Deployment definitions in a git repository if new image tags are detected as depicted in Figure 11. Those image tags can be accessed from public or private Docker registries, for which credentials can be provided via an environment variable or Kubernetes secret if running in a cluster. But Bow cannot only run in a Kubernetes cluster — since it does not depend on any interaction with a cluster, it can also run directly from the compiled binary or in a plain Docker container. The authentication to the git repository occurs with a provided username and password or via SSH public/private key. Notifications about performed updates or detected image changes can be sent to HipChat, Slack or Mattermost channels. Alternatively, a webhook can be called on these events to connect even more messaging platforms or automate tasks. Annotations in the deployment definitions can control Bow's behaviour for the respective Deployment to specify which tags should be applied and which should be skipped with regular expressions[1], semantic versioning[2] or by allowing every update. Through annotations, the number of required approvals can be defined, too. Users can give approval via a Slack bot or custom solutions can be implemented via webhooks. All approval votes will be logged for later auditing and an optional web frontend allows easy approval manage-

1 https://www.regular-expressions.info/
2 https://semver.org/

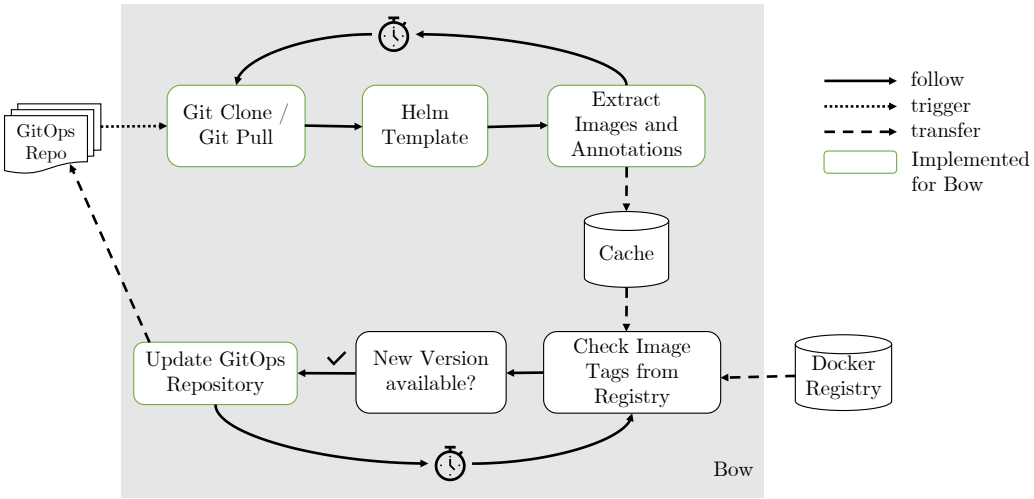

Figure 11: Workflow of Bow.

ment. Docker registries are polled every five minutes by default, but a custom interval can be set or a manual refresh triggered by a webhook for instantaneous updates. The deployment definitions can be stored as plain YAML or Helm templates. Since a subfolder can be specified, one deployment repository could contain multiple applications or stages of which only one is considered by Bow.

4.2 Limitations

Every repository is considered a Helm chart, so it always has to have the right structure for a Helm chart. If plain Kubernetes definitions without templating should be used, they still have to be placed in the folder structure like a Helm template. An image needs to occur always directly with a tag, the replacement process does not succeed if only the image or the tag is defined in a `values.yaml` file. In the rare case that the same image and tag is referenced twice but with different annotations, the replacement of the tag will not honour the filters specified in the annotations.

4.3 Case Study

4.3.1 Software Project

As a case study to show a typical software development project and workflow, the Springboot Petclinic will be deployed in a Kubernetes cluster both via a push and a pull pipeline [89]. The Petclinic is a traditional example for some capabilities of the Java Springboot framework that allows users to view a list of veterinarians and manage their customers and pets. Thus, customers' personal details including pets and a visitation history can be retrieved and updated. The microservices fork of the Petclinic splits the

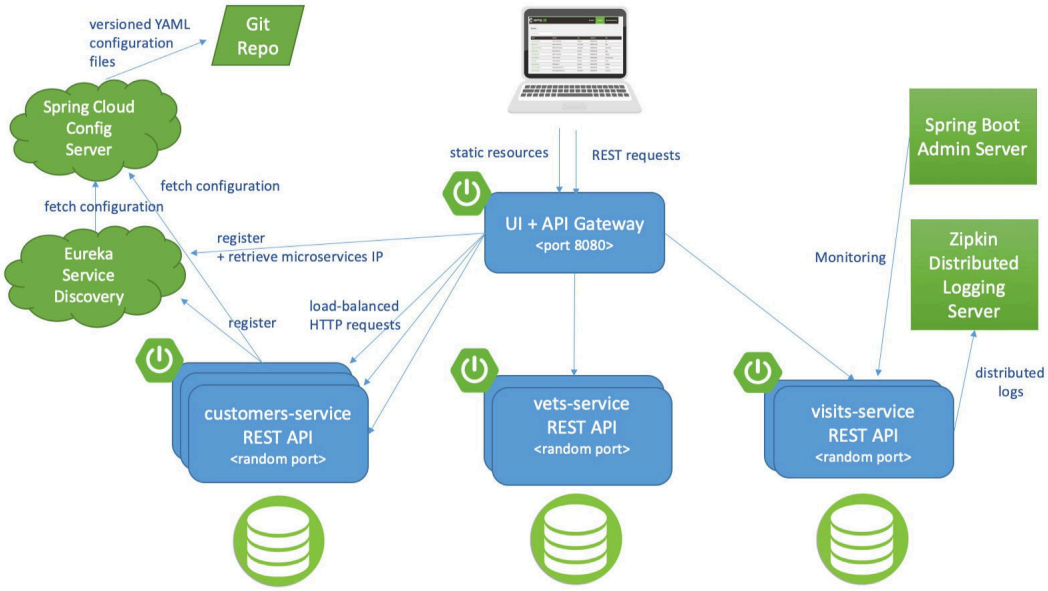

Figure 12: Structure of the petclinic microservices application [89].

application into six microservices and optionally an external database as depicted in Figure 12.

- The configuration-server provides all microservices with their specific configuration and will be queried on startup of each microservice.
- The discovery-service exposes an endpoint for service discovery, where core services can signal their availability to depending services.
- The api-gateway serves the frontend and REST API, which is available to users. It does not process or store data, but forwards requests to the following services.
- The vets-service handles all requests concerning veterinarians, the current implementation returns all veterinarians found in the database.
- The customers-service reads and writes customers' personal data to/from a database and mainly communicates with the api-gateway.
- The pets-service allows the management of customers' pets and their visits.

The API-Gateway uses client-side load balancing, so it retrieves all instances' hostnames from the discovery-service and chooses one of potentially multiple that provide the same service. That is the alternative to server-side load balancing, where the client always contacts the same defined endpoint and does not need to handle load balancing and node discovery. In client-side load balancing, additional rules can be applied when selecting an instance to process a request, which allows for example easy session persistence. Instances can be scaled and modified independently and the other microservices will pick up changes through the discovery-service automatically.

The data store is provided by a single MySQL instance with persistent storage so that it can be updated and replaced without data loss. The access to the database is managed via a Kubernetes Secret, so a secret which contains a username and a randomly generated password is created before the database initialisation and used for both database creation and access by other services. There, it is injected into each Pod as an environment variable and not contained in the Docker image or any persistent way [43, p.118]. Thus, the whole application or individual microservices could connect to a different database without any code changes.
All services collect internal metrics through a Springboot plugin and expose them at a defined endpoint, which is registered at a Prometheus instance deployed alongside the Petclinic. Prometheus will regularly query the metrics endpoints and store the collected data [90]. Those are processed and displayed in a Grafana dashboard hosted in the Kubernetes cluster as well.

4.3.2 CI/CD Pipeline

A Visualisation of both pipelines to be compared here is shown in Figure 13 and Figure 14. The project's source code is managed in a company-internal GitLab version control system and the deployment pipeline runs on a Jenkins build server, which is also used by other project teams. The source code is in separate repositories for each microservice together with pipeline definitions for both push and pull pipelines. Additional repositories contain instructions to build the monitoring stack and the deployment configuration needed for the ArgoCD GitOps operator. Once a git commit is pushed to GitLab, it triggers a new build through a webhook. Jenkins pulls the latest version on a defined branch and reads the pipeline definition either from `pipeline/Jenkinsfile-push.groovy` or `pipeline/Jenkinsfile-pull.groovy` (see Listing 1). According to those instructions, Jenkins builds the source code and runs unit tests. Next, it builds and tags a new Docker image from the build artefact. The tag is generated from the service version and build time. That new image is then pushed to a Docker registry, where it is publicly available. The following steps differ in the push and pull pipelines. In the push pipeline, Jenkins reads the deployment configuration for the particular service from the source code repository and applies that configuration with an updated image tag directly in Kubernetes. On the other hand, the pull pipeline finishes with calling a webhook provided by Bow, which will update the deployment repository with the new image found in the Docker registry. At that point, the Jenkins pipeline stops and remaining actions are performed independently by ArgoCD, which detects the new commit through regular polling or via webhook called from GitLab. It finds the new image tag and updates the Deployment accordingly, which will cause Kubernetes to create a new ReplicaSet and replace all old Pods. All webhooks trigger an asynchronous refresh which would otherwise happen periodically every few minutes to check for changes in Docker registries or git repositories,

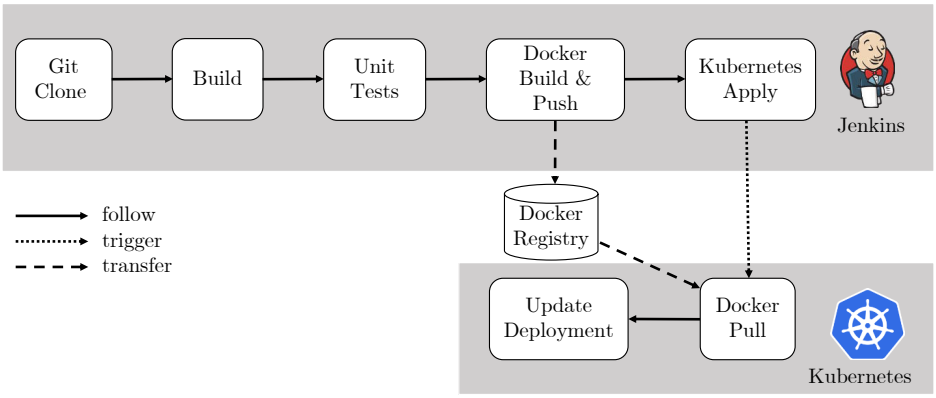

Figure 13: Structure of case study push pipeline.

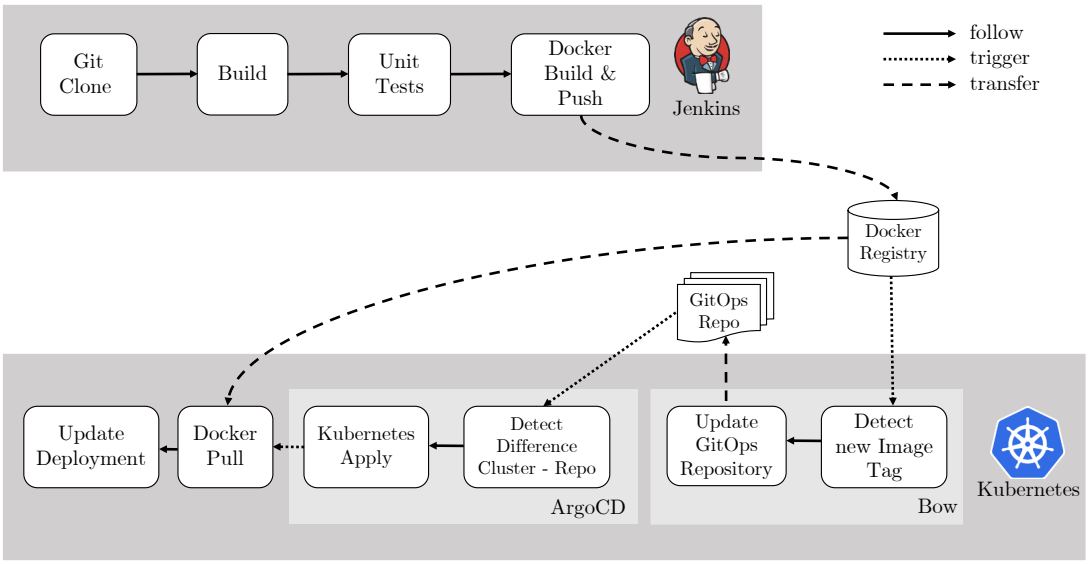

Figure 14: Structure of case study pull pipeline.

so there is no security risk or attack potential.

4.3.3 Jenkins

The Jenkins runs outside of a cloud environment directly on the company's hardware and is shared by several teams who all have access to it, but only their respective build pipelines. Thus, this thesis' results can be easily transferred to cloud CI environments which are also shared and exist outside of the application cluster. Advantages compared to a dedicated Jenkins for a project are that build resources are shared and therefore used more evenly, and a few administrators are tasked with keeping the build server up to date. If a Jenkins instance is reserved for a single project, developers would have to ensure all updates are installed and the configuration is secure. Secrets, for example to

access the source repository, the registry, the deployment repository and the Kubernetes cluster (when using push pipelines), are stored separately for every project and are usually not accessible to other teams.

4.3.4 Evaluation of the Push and Pull Pipeline

Large parts of both the push and pull pipeline are the same. They only differ in the last section, where the push pipeline directly applies the new deployment in the Kubernetes cluster. Therefore, the cluster credentials have to be stored on the Jenkins server and can potentially be stolen. For a detailed comparison of security aspects of push and pull pipelines, see Table 2.1 and Section 2.3.4. The pull pipeline, on the other hand, triggers the new deployment through the push of the image to the Docker registry. As mentioned above, the webhook trigger is only included for better deployment velocity and for a better comparison of both pipelines. For the push pipeline to work, the cluster needs to be reachable via network from the build agent, otherwise the deployment will not occur at all and the pipeline will fail, even if the disconnect is only temporary. The pull pipeline will retry applying new images and is therefore more resistant to temporary network outages. The cluster does not have to expose any management ports and pulling achieves better portability. If the whole application needs to move to a new cluster, the pipeline does not have to change at all, images would just be pulled into the other cluster. In the push pipeline, the endpoint and the cluster credentials would have to be updated to deploy to a new cluster. Sometimes, new versions of microservices should be deployed to multiple clusters at the same time, for example in different geographical regions. In this case, the pull pipeline would be exactly the same, but a push pipeline needs to have access to all clusters and specific pipeline steps defined to deploy to all clusters. Thus, the pull pipeline provides many advantages and solves important security issues arising from the direct communication between pipeline agent and Kubernetes cluster.

4.4 Evaluation of Bow in the Example Project

In the example project, two versions of the same pipeline can run simultaneously, which allows an exact comparison of deployment velocity. The same webhook from GitLab triggers both pipeline jobs. Of course, every commit contains an exact timestamp when the commit was made, which is used as the starting point for the following velocity measurements. When these experiments were performed, there was no other activity on the Jenkins build server, so both pipelines could run at the same time. However, due to non-deterministic scheduling, the build and test steps potentially did not finish exactly at the same time, so the build itself does have a small influence on the measurements

Commit	Pod creation time push	Pod creation time pull	Difference
13:28:31	13:29:40 (+1:09)	13:32:13 (+3:42)	0:02:33
13:56:02	13:57:21 (+1:19)	13:58:37 (+2:35)	0:01:16
14:01:06	14:02:20 (+1:14)	14:03:31 (+2:25)	0:01:11
14:08:42	14:10:01 (+1:19)	14:10:09 (+1:27)	0:00:08
14:12:24	14:13:46 (+1:22)	14:13:59 (+1:35)	0:00:13
14:15:34	14:16:52 (+1:18)	14:17:02 (+1:28)	0:00:10

Table 4.1: Comparison of deploy times of push and pull pipeline, in hh:mm:ss

acquired. The final timestamp is retrieved from the Pod description of the newly created Pods either via push or pull pipeline. Both pipelines deploy into the same Kubernetes cluster, but into different namespaces. Thus, the same full application runs twice in the same cluster. This method of measuring does not require instantaneous and noisy manual timekeeping, but all values can be obtained later either from the git log or via the `kubectl get pods` command.

The timing results shown in 4.1 confirm that the pull pipeline as described does take longer in all cases. However, the difference between push and pull pipeline is quite low, especially considering that professional development projects usually have a longer compile time than the petclinic example project. In the first three commits, the pull pipeline took roughly twice as long as the push pipeline, but a difference of one minute is still acceptable for most projects since the build itself often takes more than five minutes. On the other hand, for the last three builds the difference is almost unnoticeable at around ten seconds. This huge discrepancy might arise from network connectivity errors which interrupted webhook calls in the first three tests. In this case, the pipeline would idle until the polling interval is expired which can take by default up to five minutes for Bow or three minutes for ArgoCD. If the connection of the push pipeline to the cluster would be interrupted, on the other hand, the pipeline fails and not recover on its own.

These measurements show that the pull pipeline does not significantly influence the deployment velocity, but actually ensures delivery of the new version even during unreliable connections and thus increases overall resilience and stability. The initial setup for an instantaneous deployment with three webhooks and two additional tools (ArgoCD and Bow) is more complex than a simple push of the new Deployment by the pipeline, but apart from the achieved goal of stronger cluster security the additional complexity is hidden in the usual deployment flow and ArgoCD provides additional monitoring from the GitOps perspective.

4.5 Summary

Bow implements the single missing feature of the open-source GitOps platform ArgoCD compared to the commercial Weave Cloud platform and provides therefore an important contribution to the open-source landscape. Bow fills the gap between the pipeline operator and the ArgoCD GitOps operator by securely updating the deployment repository when triggered by the pipeline. Its reliability has not been solidly proven, but preliminary tests (see Figure 4.1) show high velocity and consistent deployments in all test cases. Since Bow is a fork of Keel, the majority of code has been peer reviewed and tested, particularly code sections dealing with polling Docker registries, credentials handling, notifications, approval management, webhook support and update rule parsing.

5

Conclusion and Outlook

"The gap between theory and practice is not as wide in theory as it is in practice."

anonymous

5.1 Conclusion

This thesis has found GitOps pull pipelines to be slightly more secure than push pipelines by default with considerable potential to facilitate a secure deployment process. This is because push pipelines always require credentials to the cluster environment to perform a deployment. Pull pipelines, on the other hand, inherently have a structure which supports the separation of cluster and pipeline operations. When utilising pull pipelines, a trade-off has to be made between full pipeline automation and requiring manual approval before deployment. Only the latter ensures the best protection against unintended or malicious deployments, but decreases overall velocity and requires developers' time for the approval. Of course, the decision for or against manual approval can different on different stages, to take their data security requirements into account. Apart from security, the core components of GitOps — Infrastructure as Code and declarative deployments — allow easier service monitoring and disaster recovery as well as rollback. Thus, the thesis recommends the implementation of GitOps in new or existing pipelines since the only disadvantage is additional complexity because of the GitOps operator and the overall performance drop is barely noticeable.
The tool developed for this thesis called Bow implements the single missing feature of the open-source GitOps platform ArgoCD compared to the commercial Weave Cloud platform and provides therefore an important contribution to the open-source landscape. Bow fills the gap between the pipeline operator and the ArgoCD GitOps operator by securely updating the deployment repository when triggered by the pipeline. It supports comprehensive monitoring and alerting as well as many customisations via deployment

annotations. Thus, even though it is not part of the deployment definition, it can be configured through the GitOps repository as well. Bow can run platform-independently from a binary, in a Docker container or in every cloud environment that supports Docker containers like Kubernetes.

5.2 Outlook

Although Bow's mentioned features have been tested with the petclinic project, its overall stability remains largely uncertain. Therefore, it should be tested and potential bugs fixed before it is used in production pipelines. The remaining limitations concerning the handling of plain Kubernetes files, the processing of multiple references of the same image and the handling of separately defined images and tags either have a workaround or are a special edge-case which will usually not occur. However, these issues could be resolved in future development to create a solid and mature tool. Additionally, the integration of Bow into ArgoCD is another option for continuing Bow to create a single product with the same features as the commercial Weave Cloud. This would decrease the (visible) complexity of the deployed DevOps tools but might open up new vulnerabilities. Currently, Bow runs independently without user interaction and thus does not open up intrusion entries. ArgoCD, on the other hand, exposes the web interface to the internet (or an internal network) and directly interacts with users and processes user input, which are typical attack vectors to gain access to a software system.
In another area of GitOps the question of secure, declarative secrets storage still remains an open area of research. At the moment, the deployment repository contains most resources to replicate the application in a new cluster. Only the secrets connecting internal and external services cannot be stored in the GitOps repository easily without exposing them in plain text. Sealed Secrets, the only existing solution is not under active development at the moment but gives valuable impulses what the answer to this challenge could be. Although there is considerable interest from companies and individual community members to continue the development of Sealed Secrets, due to the acquisition of the parent company the development is currently stalled.
Due to its long history, the most commonly used CI/CD pipeline agent Jenkins is not particularly secure. The development of capable CI/CD systems would also make a valuable contribution to the DevOps space. Some alternatives like GitLab CI have been created and continue to gain in popularity for its simplicity Jenkins lacks. However, they do not have as many capabilities and plugins as Jenkins which complicates many typical pipelines.

A

Appendix

```
import java.text.SimpleDateFormat
/**
 * Project Parameters
 */
def appName = 'customers-service'
def branchName = 'master'
def gitUrl =
↪  'git@git.iteratec.de:thesis-aeb/petclinic-customers-service-push.git'
def deploymentRepo =
↪  'git@git.iteratec.de:thesis-aeb/petclinic-deployment.git'
def registryCredentials = 'docker-hub-alwin2'
def deploymentBranchName = 'master'
node('master') {
    try {
        def buildVersion
        def version
        def gitCommit
        properties([
                buildDiscarder(logRotator(numToKeepStr: '5')),
                disableConcurrentBuilds(),
                pipelineTriggers([
                        [$class: 'GenericTrigger',
                         causeString: 'Triggered on $ref',
                         token: "${appName}",
                         printContributedVariables: true,
                         printPostContent: true,
```

```
                    silentResponse: false,
                ]
            ])
        ])
        stage("Git Clone") {
            deleteDir()
            git branch: "${branchName}", credentialsId:
            ↪ 'petclinic-git-secret', url: "${gitUrl}"
            gitCommit = sh(returnStdout: true, script: "git log -n 1
            ↪ --pretty=format:'%h'").trim()
        }
        stage('Set Build Version') {
            def date = new Date()
            def sdf = new SimpleDateFormat("yyyyMMdd.HHmmss")
            def buildTime = sdf.format(date)
            version = readMavenPom().getVersion()
            buildVersion = "${buildTime}-${branchName}-v${version}"
            sh "echo 'Build version ${buildVersion}'"
        }

        stage('Build') {
            sh """
                chmod +x ./mvnw
                ./mvnw -Dmaven.test.skip=true
↪ -Ddependency-check.skip=true clean install
                cp pipeline/Dockerfile target
            """
        }
        stage('Unit Tests') {
            sh """
                ./mvnw test
            """
        }
        stage('Docker Build & Push') {
            withCredentials([usernamePassword(credentialsId:
            ↪ "${registryCredentials}",
                    usernameVariable: 'REGISTRY_USER', passwordVariable:
                    ↪ 'REGISTRY_PASSWORD')]) {
                sh """
```

```
                    echo '${REGISTRY_PASSWORD}' | docker login -u
↪   '${REGISTRY_USER}' --password-stdin
                    cd target
                    docker build -t alwin2/petclinic-${appName} -t
↪   alwin2/petclinic-${appName}:${buildVersion} --build-arg
↪   DOCKERIZE_VERSION=v0.6.1 --build-arg \
                    ARTIFACT_NAME=spring-petclinic-${appName}-${version}
↪   --build-arg EXPOSED_PORT=8081 --build-arg APP_NAME=${appName} .
                    docker push alwin2/petclinic-${appName}
                    docker push
↪   alwin2/petclinic-${appName}:${buildVersion}
            """
            }
        }
        stage('Trigger bow to update from registry') {
            sh """
                curl -H "Content-Type: application/json" -d \
                "{  \\"name\\": \\"alwin2/petclinic-${appName}\\",
↪   \\"tag\\": \\"${buildVersion}\\" }" \
                http://192.168.0.30:9300/v1/webhooks/native
            """
        }
    } catch (e) {
        currentBuild.result = 'FAILED'
        throw e
    } finally {
        currentBuild.result = currentBuild.result ?: 'SUCCESS'
    }
}
```

Listing 1: `Jenkinsfile-pull.groovy`: Definition of pull pipeline with call to Bow.

Bibliography

[1] K. Beck, M. Beedle, A. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland and D. Thomas, *Manifesto for Agile Software Development*, http://agilemanifesto.org/, 2001. [Online]. Available: http://agilemanifesto.org/.

[2] J. Shore and S. Warden, *The art of agile development*. O'Reilly Media, Inc, 2008, p. 409, ISBN: 0596527675.

[3] G. Sayfan, *Hands-On Microservices with Kubernetes*. Packt Publishing Ltd, 2019.

[4] L. Apke, *Understanding the Agile Manifesto*. LULU COM, 2015, p. 6, ISBN: 1312863919.

[5] K. Beck, M. Beedle, A. V. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland and D. Thomas, *Principles behind the Agile Manifesto*, http://agilemanifesto.org/principles.html, 2014. [Online]. Available: http://agilemanifesto.org/principles.html (visited on 04/06/2019).

[6] V. G. Stray, N. B. Moe and A. Aurum, 'Investigating daily team meetings in agile software projects', in *Proceedings - 38th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2012*, IEEE, Sep. 2012, pp. 274–281, ISBN: 9780769547909. DOI: 10.1109/SEAA.2012.16.

[7] S. Rossel, *Continuous integration, delivery, and deployment : reliable and faster software releases with automating builds, tests, and deployment*. Packt Publishing Ltd, 2017, ISBN: 1787284182.

[8] J. Mukherjee, *Continuous delivery pipeline - where does it choke?*. LULU COM, 2012, ISBN: 1329964411.

[9] J. D. Blischak, E. R. Davenport and G. Wilson, 'A Quick Introduction to Version Control with Git and GitHub', *PLOS Computational Biology*, vol. 12, no. 1, F. Ouellette, Ed., e1004668, Jan. 2016, ISSN: 1553-7358. DOI: 10.1371/journal.pcbi.1004668.

[10] D. Spinellis, 'Git', *IEEE Software*, vol. 29, no. 3, pp. 100–101, May 2012, ISSN: 0740-7459. DOI: 10.1109/MS.2012.61.

[11] M. Lanza, Y. Institute of Electrical and Electronics Engineers, P. IEEE Working Conference on Mining Software Repositories 9 2012.06.02-03 Zurich and MSR 9 2012.06.02-03 Zurich, '9th IEEE Working Conference on Mining Software Repositories (MSR), 2012 2-3 June 2012, Zurich, Switzerland ; proceedings', in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, IEEE, 2012, pp. 112–115, ISBN: 9781467317610.

[12] B. De Alwis and J. Sillito, 'Why are software projects moving from centralized to decentralized version control systems?', in *Proceedings of the 2009 ICSE Workshop on Cooperative and Human Aspects on Software Engineering, CHASE 2009*, IEEE, 2009, pp. 36–39, ISBN: 9781424437122. DOI: 10.1109/CHASE.2009.5071408.

[13] N. Mazur, *What is CI/CD in DevOps? - Quora*. [Online]. Available: https://www.quora.com/What-is-CI-CD-in-DevOps (visited on 16/07/2019).

[14] J. Humble and D. Farley, *Continuous delivery*. Pearson Education, 2010, p. 512, ISBN: 0321670221.

[15] A. Mouat, *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, Inc., 2016, p. 346, ISBN: 9781491915769.

[16] J. Turnbull, *The Docker Book: Containerization is the new virtualization*. James Turnbull, 2014, ISBN: 9788578110796. DOI: 10.1017/CBO9781107415324.004. arXiv: arXiv:1011.1669v3.

[17] R. McKendrick, P. Raj, J. S. Chelladhurai and V. Singh, *Docker bootcamp : less is more with Docker*. Packt Publishing Ltd, 2017, ISBN: 978-1-78728-698-6.

[18] Docker Inc., *About images, containers, and storage drivers — Docker Documentation*. [Online]. Available: https://docs.docker.com/v17.09/engine/userguide/storagedriver/imagesandcontainers/%7B%5C#%7Dimages-and-layers (visited on 15/07/2019).

[19] J. S. Chelladhurai, P. Raj and V. Singh, *Learning Docker : faster app development and deployment with Docker containers*. Packt Publishing Ltd, 2017, pp. 50–72, ISBN: 178646201X.

[20] S. Grubor, *Deployment with Docker Apply continuous integration models, deploy applications quicker, and scale at large by putting Docker to work*. Packt Publishing Ltd, 2017, ISBN: 9781786469007.

[21] K. Matthias and S. P. Kane, *Docker: Up & Running: Shipping Reliable Containers in Production*. O'Reilly Media, Inc., 2015, p. 232, ISBN: 978-1-4919-1757-2.

[22] Eric Carter, *2018 Docker Usage Report. — Sysdig*, 2018. [Online]. Available: https://sysdig.com/blog/2018-docker-usage-report/ (visited on 16/07/2019).

[23] M. Villamizar, O. Garces, H. Castro, M. Verano, L. Salamanca, R. Casallas and S. Gil, 'Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud', in *2015 10th Colombian Computing Conference, 10CCC 2015*, IEEE, Sep. 2015, pp. 583–590, ISBN: 9781467394642. DOI: 10.1109/ColumbianCC.2015.7333476.

[24] A. Balalaie, A. Heydarnoori and P. Jamshidi, 'Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture', *IEEE Software*, vol. 33, no. 3, pp. 42–52, May 2016, ISSN: 07407459. DOI: 10.1109/MS.2016.64.

[25] S. Fowler, *Production-Ready Microservices Building Standardized Systems Across an Engineering Organization*. O'Reilly Media, Inc., 2016, p. 172, ISBN: 9781491965979.

[26] M. Fazio, A. Celesti, R. Ranjan, C. Liu, L. Chen and M. Villari, 'Open Issues in Scheduling Microservices in the Cloud', *IEEE Cloud Computing*, vol. 3, no. 5, pp. 81–88, Sep. 2016, ISSN: 23256095. DOI: 10.1109/MCC.2016.112.

[27] C. Pahl, M. Vukovic, J. Yin and Q. Yu, *Service-oriented computing : 16th International Conference, ICSOC 2018, Hangzhou, China, November 12-15, 2018, Proceedings*. Springer, 2018, p. 891, ISBN: 3030035964.

[28] O. Zimmermann, 'Microservices tenets: Agile approach to service development and deployment', *Computer Science - Research and Development*, vol. 32, no. 3-4, pp. 301–310, Jul. 2017, ISSN: 18652042. DOI: 10.1007/s00450-016-0337-0.

[29] W. Hasselbring, 'Microservices for Scalability', in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering - ICPE '16*, New York, New York, USA: ACM Press, 2016, pp. 133–134, ISBN: 9781450340809. DOI: 10.1145/2851553.2858659.

[30] K. Indrasiri and P. Siriwardena, *Microservices for the Enterprise: Designing, Developing, and Deploying*. Apress, 2018, ISBN: 1484238575.

[31] TIBCO, *What is a Microservices Architecture? — TIBCO Software*. [Online]. Available: https://www.tibco.com/reference-center/what-is-microservices-architecture (visited on 16/07/2019).

[32] G. K. Aroraa, L. Kale and K. Manish, *Building microservices with .NET Core : transitioning monolithic architecture using microservices with .NET Core*. Packt Publishing Ltd, 2017, ISBN: 1785884964.

[33] E. Evans, *Domain-driven design : tackling complexity in the heart of software*. Addison-Wesley, 2004, p. 529, ISBN: 0321125215.

[34] D. Richter, M. Konrad, K. Utecht and A. Polze, 'Highly-Available Applications on Unreliable Infrastructure: Microservice Architectures in Practice', in *Proceedings - 2017 IEEE International Conference on Software Quality, Reliability and Security Companion, QRS-C 2017*, IEEE, Jul. 2017, pp. 130–137, ISBN: 9781538620724. DOI: 10.1109/QRS-C.2017.28.

[35] H. Kang, M. Le and S. Tao, 'Container and microservice driven design for cloud infrastructure DevOps', in *Proceedings - 2016 IEEE International Conference on Cloud Engineering, IC2E 2016: Co-located with the 1st IEEE International Conference on Internet-of-Things Design and Implementation, IoTDI 2016*, IEEE, Apr. 2016, pp. 202–211, ISBN: 9781509019618. DOI: 10.1109/IC2E.2016.26.

[36] N. Dragoni, I. Lanese, S. T. Larsen, M. Mazzara, R. Mustafin and L. Safina, 'Microservices: How to make your application scale', in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10742 LNCS, Springer, Cham, 2018, pp. 95–104, ISBN: 9783319743127. DOI: 10.1007/978-3-319-74313-4_8.

[37] J. A. Medina Iglesias, *Hands-on microservices with Kotlin : build reactive and cloud-native microservices with Kotlin using Spring 5 and Spring Boot 2.0*. Packt Publishing Ltd, 2018, ISBN: 1788473493.

[38] R. Sturm, C. Pollard and J. Craig, 'The NIST Definition of Cloud Computing', in *Application Performance Management (APM) in the Digital Enterprise*, Elsevier Inc, 2017, pp. 267–269. DOI: 10.1016/B978-0-12-804018-8.15003-X.

[39] M. Andrawos and M. Helmich, *Cloud native programming with Golang : develop microservice-based high performance web apps for the cloud with Go*. Packt Publishing Ltd, 2017, ISBN: 1787127966.

[40] V. Farcic, *The DevOps 2.3 toolkit : Kubernetes : deploying and managing highly-available and fault-tolerant applications at scale*. Packt Publishing Ltd, 2018, p. 418, ISBN: 1789133297.

[41] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero and D. A. Tamburri, 'DevOps: Introducing infrastructure-as-code', in *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*, IEEE, May 2017, pp. 497–498, ISBN: 9781538615898. DOI: 10.1109/ICSE-C.2017.162.

[42] M. Hoogendoorn, *How Kubernetes Deployments Work - The New Stack*. [Online]. Available: https://thenewstack.io/kubernetes-deployments-work/ (visited on 16/07/2019).

[43] H. Saito, H.-C. C. Lee and C.-Y. Wu, *DevOps with Kubernetes : accelerating software delivery with container orchestrators*. Packt Publishing Ltd, 2017, ISBN: 9781788396646.

[44] K. Hightower, B. Burns and J. Beda, *Kubernetes Up and Running; Dive into the Future of Infrastructure*. O'Reilly Media, Inc., 2017, p. 202, ISBN: 9781491935675.

[45] J. Arundel and J. Domingus, *Cloud Native DevOps with Kubernetes : building, deploying, and scaling modern applications in the cloud*. O'Reilly Media, Inc., 2019, ISBN: 1492040711.

[46] J. Baier, G. Sayfan and J. White, *The the Complete Kubernetes Guide Become an Expert in Container Management with the Power of Kubernetes.* Packt Publishing, Limited, 2019, ISBN: 1838647708.

[47] A. Lossent, A. Rodriguez Peon and A. Wagner, 'PaaS for web applications with OpenShift Origin', in *Journal of Physics: Conference Series*, vol. 898, Oct. 2017. DOI: 10.1088/1742-6596/898/8/082037.

[48] F. Klaffenbach, *Deployment of Microsoft Azure Cloud Solutions : a complete guide to cloud development using Microsoft Azure.* Packt Publishing Ltd, 2018, ISBN: 1789953855.

[49] R. RV, *Spring 5.0 Microservices - Second Edition: Scalable systems with Reactive Streams and Spring Boot.* Packt Publishing Ltd, 2016, ISBN: 9781786464682.

[50] D. Jones, *The Five Essential Elements of Application Performance Monitoring.* Realtime Publishers, 2015.

[51] G. Katsaros, R. Kübert and G. Gallizo, 'Building a Service-Oriented Monitoring Framework with REST and Nagios', in *2011 IEEE International Conference on Services Computing*, IEEE, Jul. 2011, pp. 426–431, ISBN: 978-1-4577-0863-3. DOI: 10.1109/SCC.2011.53.

[52] O. Mironov, 'DevOps Pipeline with Docker', PhD thesis, Helsinki Metropolia University of Applied Sciences, 2017. [Online]. Available: https://www.theseus.fi/bitstream/handle/10024/143357/DevOps%7B%5C_%7DThesis%7B%5C_%7DFinal.pdf?sequence=1.

[53] M. Laverse, Schakko and F. Diaz, *jenkinsci/kubernetes-cli-plugin: Allows you to setup kubectl to access Kubernetes clusters from your Jenkins jobs.* [Online]. Available: https://github.com/jenkinsci/kubernetes-cli-plugin (visited on 02/07/2019).

[54] Weaveworks, *GitOps.* [Online]. Available: https://www.weave.works/technologies/gitops/ (visited on 02/07/2019).

[55] A. Richardson, *The GitOps Pipeline - Part 2.* [Online]. Available: https://www.weave.works/blog/the-gitops-pipeline (visited on 02/07/2019).

[56] D. Gonzalez, *Implementing Modern DevOps: Enabling IT organizations to deliver faster and smarter.* Packt Publishing Ltd, 2017, ISBN: 1786464004.

[57] C. Redaktion, 'c't wissen Docker: Komplexe Software einfach einrichten', *c't wissen*, 2019.

[58] S. Picozzi, M. Hepburn and N. O'Connor, *DevOps with OpenShift: Cloud Deployments Made Easy.* O'Reilly Media, Inc, 2017, ISBN: 1491975962.

[59] D. Zuev, A. Kropachev and A. Usov, *Learn OpenShift : deploy, build, manage, and migrate applications with OpenShift Origin 3.9*. Packt Publishing Ltd, 2018, ISBN: 1788999649.

[60] S. Bhat, *Practical Docker with Python : build, release and distribute your Python app with Docker*. Apress, 2018, ISBN: 9781484237847.

[61] K. Dolui and C. Kiraly, 'Towards Multi-Container Deployment on IoT Gateways', in *2018 IEEE Global Communications Conference, GLOBECOM 2018 - Proceedings*, IEEE, Dec. 2019, pp. 1–7, ISBN: 9781538647271. DOI: 10.1109/GLOCOM.2018.8647688.

[62] Weaveworks, *weaveworks/flux: The GitOps Kubernetes operator*. [Online]. Available: https://github.com/weaveworks/flux (visited on 08/07/2019).

[63] ——, *Flux/get-started.md at master · weaveworks/flux*. [Online]. Available: https://github.com/weaveworks/flux/blob/master/site/get-started.md%7B%5C#%7Dgiving-write-access (visited on 04/07/2019).

[64] ——, *Flux/fluxctl.md at master · weaveworks/flux*. [Online]. Available: https://github.com/weaveworks/flux/blob/master/site/fluxctl.md (visited on 04/07/2019).

[65] ——, *Flux/helm-operator.md at master · weaveworks/flux*. [Online]. Available: https://github.com/weaveworks/flux/blob/master/site/helm-operator.md (visited on 04/07/2019).

[66] ——, *Flux/annotations-tutorial.md at master · weaveworks/flux*. [Online]. Available: https://github.com/weaveworks/flux/blob/master/site/annotations-tutorial.md (visited on 04/07/2019).

[67] ——, *Controlling Containers*. [Online]. Available: https://www.weave.works/docs/cloud/latest/tasks/explore/control-containers/ (visited on 04/07/2019).

[68] ——, *Weave Cloud Features*. [Online]. Available: https://www.weave.works/docs/cloud/latest/overview/weave-cloud-features/ (visited on 04/07/2019).

[69] ArgoCD, *Projects - Argo CD - Declarative GitOps CD for Kubernetes*. [Online]. Available: https://argoproj.github.io/argo-cd/user-guide/projects/ (visited on 04/07/2019).

[70] ——, *Private Repositories - Argo CD - Declarative GitOps CD for Kubernetes*. [Online]. Available: https://argoproj.github.io/argo-cd/user-guide/private-repositories/ (visited on 04/07/2019).

[71] ——, *Tools - Argo CD - Declarative GitOps CD for Kubernetes*. [Online]. Available: https://argoproj.github.io/argo-cd/user-guide/application%7B%5C_%7Dsources/ (visited on 04/07/2019).

[72] R. Kukulinski, *Welcoming Heptio Open Source Projects to VMware - Cloud Native Apps Blog*. [Online]. Available: https://blogs.vmware.com/cloudnative/2019/02/05/welcoming-heptio-open-source-projects-to-vmware/ (visited on 04/07/2019).

[73] Kustomize, *Kustomize - Kubernetes native configuration management*. [Online]. Available: https://kustomize.io/ (visited on 04/07/2019).

[74] ArgoCD, *Automated Sync Policy - Argo CD - Declarative GitOps CD for Kubernetes*. [Online]. Available: https://argoproj.github.io/argo-cd/user-guide/auto%7B%5C_%7Dsync/ (visited on 04/07/2019).

[75] S. Goasguen and M. Hausenblas, *Kubernetes Cookbook: Building Cloud Native Applications*. O'Reilly Media, Inc, 2018, p. 215, ISBN: 2013436106. DOI: 10.1360/zd-2013-43-6-1064.

[76] B. Burns and C. Tracey, *Managing Kubernetes : operating Kubernetes clusters in the real world*. O'Reilly Media, Inc, ISBN: 1492033863.

[77] D. Bryant and A. Marin-Perez, *Continuous Delivery in Java: Essential Tools and Best Practices for Deploying Code to Production*. O'Reilly Media, Inc., 2018, pp. 2, 20, 70–73, ISBN: 978-1-491-98602-8.

[78] D. J. Barrett, R. E. Silverman and R. G. Byrnes, *Linux security cookbook*. O'Reilly, 2003, p. 311, ISBN: 0596003919.

[79] A. M. Berg and H. Van der Heijden, *Jenkins continuous integration cookbook : over 90 recipes to produce great results from Jenkins using pro-level practices, techniques, and solutions*. Packt Publishing Ltd, 2015, p. 408, ISBN: 9781784399245.

[80] J. Amann, O. Gasser, Q. Scheitle, L. Brent, G. Carle and R. Holz, 'Mission accomplished?', in *Proceedings of the 2017 Internet Measurement Conference on - IMC '17*, New York, New York, USA: ACM Press, 2017, pp. 325–340, ISBN: 9781450351188. DOI: 10.1145/3131365.3131401.

[81] Alexis Richardson, *GitOps - Operations by Pull Request*. [Online]. Available: https://www.weave.works/blog/gitops-operations-by-pull-request (visited on 19/07/2019).

[82] ArgoCD, *Image tag updates · Issue #1648 · argoproj/argo-cd*. [Online]. Available: https://github.com/argoproj/argo-cd/issues/1648 (visited on 19/07/2019).

[83] N. S. Patel and B. S. Rekha, 'Software as a Service (SaaS): security issues and solutions', *International Journal of Computational Engineering Research (IJCER)*, vol. 4, no. 6, 2014.

[84] S. Newman, *Building Microservices*. O'Reilly Media, Inc, 2015, p. 280, ISBN: 978-1-491-95035-7.

[85] ArgoCD, *argoproj/argo-cd: Declarative continuous deployment for Kubernetes.* [Online]. Available: https://github.com/argoproj/argo-cd (visited on 04/07/2019).

[86] Keel, *Commits · keel-hq/keel.* [Online]. Available: https://github.com/keel-hq/keel/commits/master (visited on 19/07/2019).

[87] G. Sayfan, *Mastering Kubernetes Master the art of container management by using the power of Kubernetes, 2nd Edition.* 2018, ISBN: 9781788999786.

[88] V. Vivien, M. Castro Contreras and M. Ryer, *Go : design patterns for real-world projects : building production-ready solutions in Go using cutting-edge technology and techniques.* Packt Publishing Ltd, 2017, ISBN: 1788392876.

[89] Petclinic, *spring-petclinic/spring-petclinic-microservices: Distributed version of Spring Petclinic built with Spring Cloud.* [Online]. Available: https://github.com/spring-petclinic/spring-petclinic-microservices (visited on 08/07/2019).

[90] J. Turnbull, *Monitoring with Prometheus.* Turnbull Press, 2018, ISBN: 0988820285.

List of Figures

List of Tables