JOHANNES KEPLER
UNIVERSITY LINZ

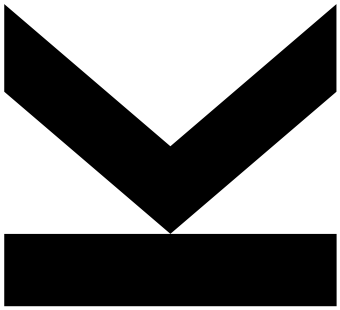
Author
Manuel Fragner

Submission
Institute of System Software

Thesis Supervisor
DI Lukas Makor, Bsc

September 2023

Applying GitOps principles to a cloud-native application



Bachelor’s Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor’s Program

Informatik

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Bachelor's Thesis

Applying GitOps principles to a cloud-native application

Dipl.-Ing. Lukas Makor, BSc
Institute for System Software
T +43-732-2468-3435
lukas.makor@jku.at

Student: Manuel Fragner
Advisor: Dipl.-Ing. Lukas Makor, BSc
Start date: March 2023

Dynatrace developed an intentionally insecure internal microservices application that is used to research Kubernetes security and demonstrate the security scanning capability of their platform. Currently, this application is deployed manually using the Skaffold tool to create and deploy multiple microservices in Kubernetes.

GitOps is an alternative method of implementing continuous deployment for cloud-native applications.

In a GitOps approach, each software or infrastructure component is treated as one or more files in a version control system (VCS). In addition, an automated process is in place for state synchronization between the VCS and the runtime environment.

With GitOps, changes to infrastructure code are now traceable and require review and approval by other developers before being merged into the main branch, resulting in less error-prone code. Another benefit is that a continuous integration (CI) pipeline can now automatically test this modified code, further preventing defects. After merging, a continuous delivery (CD) tool can now pull the new configuration and automatically apply it to the cluster.

Goals of this thesis:

- Research a realistic and up-to-date DevOps workflow that makes life easier for developers who simply want to push code and see automated deployments.
- Apply these findings to Dynatrace's internal microservices application and set up a DevOps workflow.
- Research and develop tests for the new CI/CD pipeline.
- Since GitOps operators need many (elevated) privileges to do their jobs, explore and document potential new security risks.

Modalities:
The progress of the project should be discussed at least every four weeks with the advisor. A time schedule and a milestone plan must be set up within the first four weeks and discussed with the advisor. It should be continuously refined and monitored to make sure that the thesis will be completed in time. The final version of the thesis must be submitted not later than 31.08.2023.

Abstract

This thesis documents the application of GitOps principles to a cloud-native application. A cloud-native application is an application that is specifically designed for the cloud - an infrastructure provided by third parties via the internet. According to GitOps principles, a system's infrastructure is defined as code and stored in Git. Additionally, GitOps employs automation that synchronizes the infrastructure with its intended state. The problem at the center of this thesis is a cloud-native application called *Unguard*, which previously did not adhere to GitOps principles. This issue is particularly relevant in the contemporary context because cloud applications and the technologies fueling them are state-of-the-art. In the context of this work, Unguard was analyzed for violations of GitOps principles, and these violations were then resolved by developing a new architecture and workflow that adhered to GitOps principles. This is accomplished using state-of-the-art technologies such as *Kubernetes*, *Helm* and *Argo CD*. The application of GitOps principles results in a streamlined workflow for engineers with less potential for human error. Furthermore, the implemented steps offer a quicker and simpler process for installing and upgrading Unguard. Lastly, security implications of the newly adopted architecture and workflow were assessed.

Kurzzusammenfassung

Diese Arbeit dokumentiert die Umsetzung von GitOps-Prinzipien in einer Cloud-Native-Anwendung. Eine Cloud-Native-Anwendung ist eine Anwendung, die speziell für die Cloud entwickelt wurde - eine Infrastruktur, die von Dritten über das Internet bereitgestellt wird. Nach den GitOps-Prinzipien wird die Infrastruktur eines Systems als Code definiert und in Git gespeichert. Darüber hinaus setzt GitOps auf Automatisierung, um die Infrastruktur mit dem gewünschten Zustand zu synchronisieren. Das Problem, das im Mittelpunkt dieser Arbeit steht, ist eine Cloud-Native-Anwendung namens *Unguard*, die sich bisher nicht an die GitOps-Prinzipien gehalten hat. Dieses Problem ist besonders relevant, da Cloud-Anwendungen und die sie unterstützenden Technologien dem neuesten Stand der Technik entsprechen. Im Kontext dieser Arbeit wurde Unguard auf Verstöße gegen die GitOps-Prinzipien untersucht und diese Verstöße wurden dann durch die Entwicklung einer neuen Architektur und eines neuen Arbeitsablaufs behoben. Dies wurde mithilfe von aktuellen Technologien wie *Kubernetes*, *Helm* und *Argo CD* erreicht. Die Anwendung der GitOps-Prinzipien führt zu einem optimierten Arbeitsablauf für die Ingenieure mit weniger Potenzial für menschliche Fehler. Außerdem bieten die implementierten Schritte ein schnelleres und einfacheres Verfahren für die Installation und Aktualisierung von Unguard. Schließlich wurden die Auswirkungen der neuen Architektur und des neuen Arbeitsablaufs auf ihre Sicherheit untersucht.

Contents

1 Introduction

This thesis centers around applying the GitOps principles to a Cloud-native application, two rather broad topics. GitOps is a set of procedures that harnesses the power of the version control system Git to provide both revision and change control within cloud environments. Cloud and cloud-native applications are vague terms and are frequently used as buzzwords without a clear understanding of their meaning. Roughly speaking, the term cloud refers to infrastructures and services provided by third parties over the internet. Cloud-native applications are specifically designed to meet the requirements of cloud services and infrastructures and to take advantage of them. One such cloud-native application is Unguard, a research and demonstration tool developed by Dynatrace. It is a deliberately insecure microblogging application for conducting research on cloud security. Previously, Unguard did not adhere to GitOps principles, which impeded an efficient and convenient workflow. In the course of the work done for this thesis, GitOps principles were applied to Unguard to resolve the issues inherent in the previous approach.

To adopt GitOps principles in Unguard, the following steps were taken. First, it was necessary to acquire extensive background knowledge, given the topic's vast scope. The subsequent steps included familiarizing oneself with Unguard and its development and deployment workflow up until then. Next, GitOps principles were thoroughly researched to understand to what extent Unguard did not adhere to these guidelines. The deficiencies were addressed by developing and implementing a practical, up-to-date GitOps architecture for Unguard. This led to a streamlined workflow for engineers, with less potential for human error, as well as to a faster and easier process for installing and upgrading Unguard. Finally, the security implications of the newly implemented solution were analyzed. The thesis concludes with a list of limitations and opportunities for future work.

2 Background

This thesis aims to introduce GitOps to a cloud-native application. In order to provide the necessary background, this chapter introduces a classic software architecture, describes its drawbacks, and explains what distinguishes a cloud-native approach. Such cloud-native architectures introduce numerous abstraction layers and require familiarity with multiple concepts and technologies. To illustrate the function of these concepts and technologies, this chapter also provides small examples.

2.1 Monolithic Architectures and Their Problems

Historically, applications have been designed and developed using a monolithic architecture, resulting in a unified software program [1, 2]. These applications are self-contained and comprise multiple tightly coupled functions and components. To compile or execute an application developed using such an architecture, all its components must be present because they often run as a single process [3]. This means that even if an engineer is only working on one component of the application, such as the frontend, all other components must still be available on the engineer's machine. As a result, these components are typically part of the same code repository, leading to a substantial codebase. Managing the codebase can become more cumbersome as the application grows in complexity. The subsequent examples highlight some of the potential drawbacks of this architecture.

Firstly, as the complexity and size of the application grows, the build time rises significantly, slowing down the software development lifecycle. The collaboration of engineers presents an additional problem. If everyone develops in the same repository, the likelihood of two engineers making conflicting changes to the same file from different branches is significantly higher. Attempts to merge these changes back into the main branch lead to what is called a *merge conflict*. These conflicts will likely occur more frequently with more collaborative work in a single repository.

As the software is compiled as a whole, altering the language or framework for an existing large codebase is difficult. If, for instance, another language or framework turns out to be more suitable for future requirements, changing the language or framework may be impractical or even impossible because of the application's strong dependency on them. Such changes would necessitate significant effort or even result in a rewrite of the whole application. Even updating the programming language or framework to a newer version can pose a challenge since the entire project depends on the version that is currently being used. Minor changes in the updated version can result in unforeseen issues in other modules, thus making the updating process laborious. Finally, altering a single component requires rebuilding and re-deploying the entire application, leading to an unnecessary expenditure of resources.

2.2 Microservice Architecture

One potential solution to the aforementioned issues is splitting complex monolithic applications into smaller, independent components called microservices, leading to a microservice architecture. Each service is responsible for a specific part of the application, such as user authentication or serving the frontend, and runs as an independent process [3]. Together, these distinct services provide the entire application functionality, fully transparent to the user, behaving like a single program. These services can be deployed separately and are loosely coupled. The inter-service communication is established and maintained through well-defined APIs.

The microservice architecture allows for polyglot programming where each service can utilize the most appropriate technology, language, and framework to achieve a specific business objective. All services can be managed through separate code repositories. This allows for independent

compilation of the services, resulting in faster build times and the ability to re-deploy a single component. Moreover, each service can be owned by a separate small development team responsible for its maintenance and development, allowing them to focus their efforts on a specific module of the overall application.

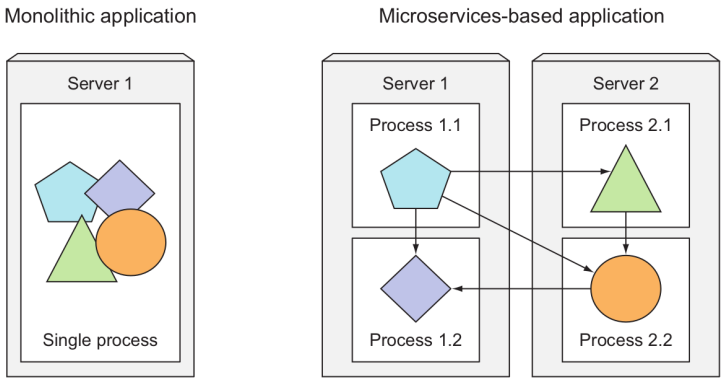


Figure 1: Monolithic Application vs. Microservices-based [3].

Figure 1 visualizes the difference between the monolithic and the microservice architectures. While a monolithic approach unifies all the components of an application on a single server in a single process, a microservice approach splits the components into separate processes capable of running on different servers and communicating through APIs.

2.3 Cloud-Native Application Architecture

Cloud-native refers to software designed for running in a cloud computing environment on cloud-native platforms like *Kubernetes*[1] [4]. Such applications are built to be scalable, highly available, and easy to manage. To achieve this, cloud-native applications use cloud services. Cloud services are infrastructure, platforms, or software delivered and hosted by third-party providers via the internet, for example, the services provided by *Amazon Web Services (AWS)*[2]. It is also possible to run this type of application in on-premise data centers, provided that an appropriate cloud platform such as Kubernetes is available [5]. The microservice architecture is the most widespread architectural pattern for cloud-native applications. It divides the application into multiple small services that collectively provide its functionality. Every microservice is packaged and run in an isolated software environment called a container. Cloud platforms like Kubernetes are, among other things, utilized to manage these containers. This managing is called container orchestration. A cloud-native application utilizes various tools and frameworks and introduces several abstraction layers. These are presented in the following sections.

2.4 Container

In a microservice architecture and, thus, a cloud-native architecture, each service is typically deployed and executed within a containerized environment [5]. A container is a standardized software unit that packages application code and its dependencies to enable reliable and fast execution in any computing environment. The services within containers should be designed to be stateless to

[1] `https://kubernetes.io/`
[2] `https://aws.amazon.com/`

make the container as ephemeral as possible, meaning that the container can be stopped, destroyed, rebuilt, and replaced with minimal setup and configuration. Containers provide isolated environments for running applications without affecting the rest of the system [3]. This isolation allows multiple instances of a service to run, thereby improving performance and reliability. On Linux, containers run natively and share the host's kernel, eliminating the need for guest operating systems for each container. This, coupled with the absence of a hypervisor, results in a more lightweight solution compared to virtualization. Figure 2 illustrates this distinction. Although there are many container platforms, this thesis focuses on *Docker*[3].

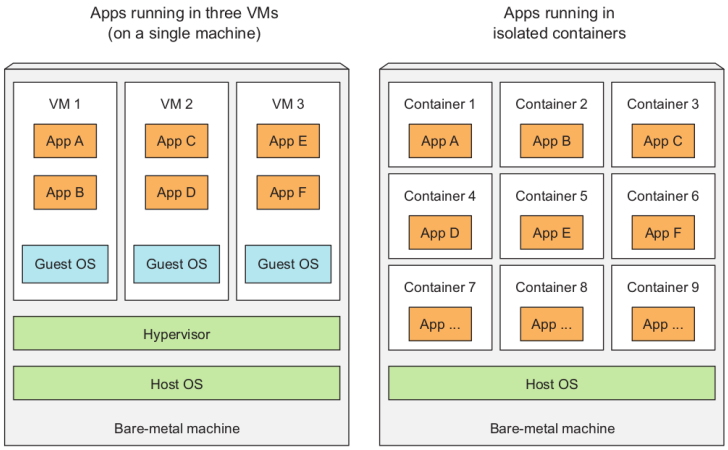


Figure 2: Virtualization versus Containerization [3].

2.4.1 Container Image

A *container image* packs everything necessary to run a service, including application code, runtime, system tools, system libraries, and settings [5]. Such images are static – once built, they are immutable. In other words, container images are unmodifiable templates that contain instructions for creating a container. Docker container images are defined in a Dockerfile [6]. A Dockerfile is a plain-text file containing the commands to create a docker container image. When Docker processes such a Dockerfile, which is also referred to as building, the outcome is a new Docker container image. The final container images can be uploaded and shared on industry-standard platforms like *Docker Hub*[4], which is an official platform for hosting and sharing Docker container images. Listing 1 illustrates a basic example of a Dockerfile.

Listing 1: Dockerfile

```
1 FROM curlimages/curl
2 RUN curl http://example.com
```

The `FROM` statement defines the parent container image used as the starting point. This example uses an image with the tool *curl*[5] preinstalled. Curl is a command-line tool for transferring data using various network protocols. The second line specifies the command to be executed if the container runs. In this case, curl fetches data from `http://example.com` and shows the output on the console.

[3] `https://www.docker.com/`
[4] `https://hub.docker.com/`
[5] `https://curl.se/`

2.4.2 Container Runtime

Running a container image requires a container runtime. A container runtime is a software package that utilizes certain features on a supported operating system to create an environment for running a specified container image [4, 7]. The container runtime is responsible for managing the entire container lifecycle, from launch to reconciliation and termination. Multiple providers are available, with *Docker Engine*[6] being the most prominent one. *Containerd*[7] is also worth mentioning, as it is the most common runtime included with Kubernetes.

2.5 Container Orchestration

Another essential part of the cloud-native application architecture is the use of container orchestration tools. Running containers in production can quickly become a massive effort due to their ephemeral and lightweight nature [8–10]. Especially with a containerized microservice architecture, running a significant number of containers can be necessary for any large-scale system. This can lead to substantial complexity when managed manually. Container orchestration reduces the operational complexity of running containerized workloads by automating container deployment, management, scaling, and networking. Most container orchestration tools support a declarative configuration model. Declarative programming describes the process of defining the desired output instead of describing the steps needed to make it happen. With declarative container orchestration, developers write a configuration file that defines which container images to use, how many resources a container can allocate, how containers are connected, and how container storage is provisioned. A container orchestration tool can then use this file to achieve the desired end state automatically. There are many container orchestration tools available. Some popular options are *Apache Mesos*[8], *Docker Swarm*[9], and *Kubernetes*. This thesis focuses on the latter.

2.6 Kubernetes

The de facto industry standard for a container orchestration tool is *Kubernetes*[10] [11]. It was initially developed by Google but was open-sourced and donated to the newly formed Cloud Native Computing Foundation in 2015. Kubernetes empowers developers to quickly build containerized applications by providing mechanisms for deploying, scheduling, scaling, and monitoring containers. It also enables load balancing, self-healing, storage orchestration, managing secrets, automated rollouts or rollbacks, and more [12, 13]. To accomplish these tasks, Kubernetes introduces many abstractions and concepts, the most important of which will be discussed below. Kubernetes utilizes the concept of a cluster. A cluster consists of one or multiple nodes or workers. A node refers to a virtual or physical machine that executes and manages containerized workloads. Each cluster also needs at least one primary node that handles the Kubernetes control plane.

2.6.1 Control Plane

The role of the control plane is to manage the cluster and ensure that each component is consistently maintained in a desired state [3, 11, 14, 15]. It receives information about cluster activity, as well as internal and external requests. The control plane then processes this data and makes the appropriate

[6] `https://docs.docker.com/engine/`
[7] `https://containerd.io/`
[8] `https://mesos.apache.org/`
[9] `https://docs.docker.com/get-started/swarm-deploy/Docke`
[10] `https://kubernetes.io/`

decisions. To enhance availability and fault tolerance, it can be distributed across multiple primary nodes. The control plane is made up of several core components, which are depicted on the left side of Figure 3:

- **kube-apiserver**
  The API server exposes the Kubernetes API using JSON over HTTP. It is responsible for data transfer within the cluster as well as with external services.

- **etcd**
  This is a distributed, persistent, key-value data store that serves as the backing store for all cluster data. It holds the cluster's current and desired state at any given time.

- **kube-scheduler**
  Unscheduled workloads are assigned to nodes by the scheduler based on resource requirements, availability, and other constraints. The scheduler monitors the resource allocation of each node and distributes the load equally across the entire cluster.

- **kube-controller-manager**
  This is a single process responsible for executing Kubernetes controllers. A controller is a software program that directs the current cluster state to match the intended state by communicating with the API server. There are several pre-defined controllers, including the node controller, which is responsible for detecting and responding when nodes go offline. It is also possible to add custom controllers.

- **cloud-controller-manager**
  This component embeds cloud-specific logic into the cluster and is responsible for linking the cluster to the API of the chosen cloud provider.

2.6.2 Nodes

A node refers to a virtual or physical machine on which Kubernetes processes workloads [11, 14]. A typical cluster consists of multiple nodes. Each node is managed by the control plane and must contain a container runtime environment in addition to the following components. Figure 3 shows two exemplary nodes with their components.

- **kubelet**
  This agent is responsible for monitoring and controlling the state of the containers running on its node, keeping them in the desired state as directed by the control plane.

- **kube-proxy**
  The kube-proxy is a network proxy and load balancer that manages the network rules on the nodes [11]. It routes traffic to the appropriate containers and enables network communication with workloads from outside or inside the cluster.

2.7 Kubernetes Resources

All Kubernetes objects are represented as YAML-formatted records within the internal database [17]. Although it is possible to create a resource with a command-line interface, defining Kubernetes resources declaratively in a resource manifest is recommended. All resource manifests include an `apiVersion`, `kind`, `metadata`, `spec`, and additionally resource-specific entries, but detailing them is

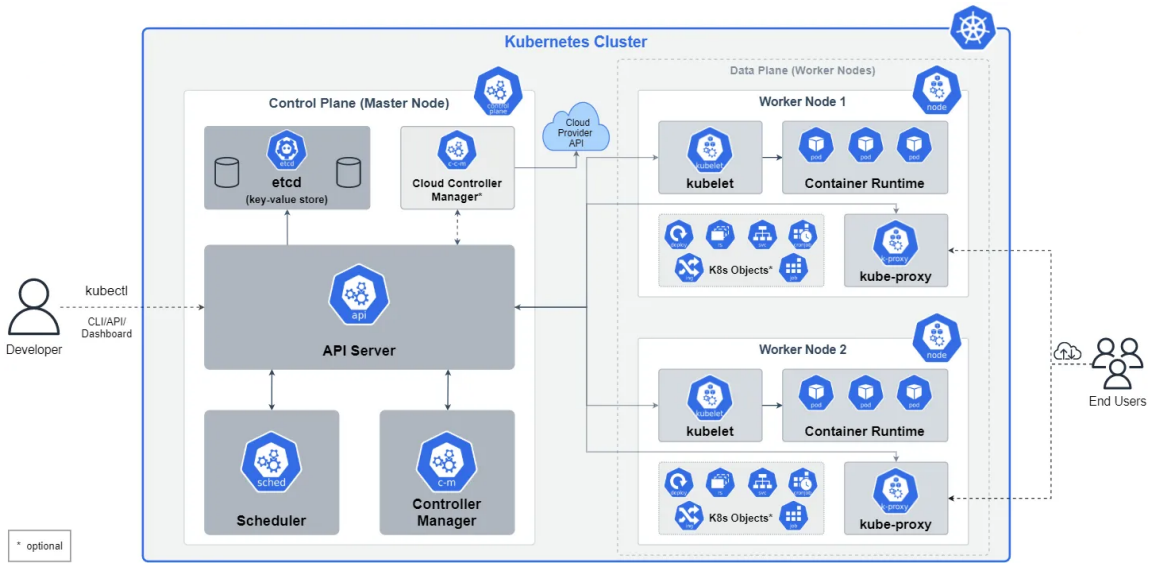


Figure 3: Kubernetes Architecture and Components [16].

beyond the scope of this thesis. The following sections provide more information about Kubernetes resources covered by this thesis.

2.7.1 Workloads

A Kubernetes workload is an executable that runs on the cluster. A workload can consist of a single or multiple components working together. All workloads run within Kubernetes *Pods*.

Pods The smallest scheduling unit in Kubernetes is a *Pod*, which functions as an abstraction layer for containers [3, 11, 14]. A Pod may include one or more containers, although it is common for each container to run in its own Pod. The container and, thus, its file system is ephemeral by default, meaning all data is deleted at the end of the Pod's life. If data needs to be persisted, special considerations are required. Each Pod is assigned a unique internal IP address within the cluster and receives a new one upon restarting. Pods have defined lifecycles and can be declaratively managed. Nevertheless, it is not advisable to manage workloads manually through Pods; instead, one should use workload controllers that automatically manage a set of Pods. Listing 2 shows a basic example of a Pod resource consisting of a container running the image `nginx:latest`. A Pod resource, like all Kubernetes resources, can specify numerous elements. Refer to the *Kubernetes API Reference*[11] for a comprehensive list.

Listing 2: Kuberntes Pod Resource

```
1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: nginx
5 spec:
6   containers:
7   - name: nginx
8     image: nginx:latest
```

[11] `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.27/#pod-v1-core`

2.7.2 Workload Controllers

Several workload controllers are available in Kubernetes. In this thesis, we will focus only on two of them, namely, Deployments and StatefulSets.

Deployments Kubernetes *Deployments* offer developers an abstraction layer over Pods, which provides the ability to define a common blueprint for a set of Pods [14, 17]. A Deployment is relevant for managing a stateless application workload, where any Pod is interchangeable and can be replaced if needed.

StatefulSets This type of workload resource is similar to Deployments but is used for stateful workloads, such as databases [14, 17]. Scaling a stateful application is harder as each Pod's state must be accounted for and preserved. In a *StatefulSet*, Pods have a distinct identity and are unique, prohibiting their random creation or deletion. Only one Pod is typically allowed to write data to ensure data consistency. Manual configuration is required for data synchronization with other Pods. Additional instances of Pods can then have read access to speed up the application.

2.7.3 Services

A Kubernetes *Service* exposes a network application running as one or more pods in a cluster [3, 14, 17]. The Service has a stable IP address assigned to it, providing consumers of the Service with stable access to the Pods it bundles. When the Service contains multiple Pods, it load-balances traffic on a round-robin basis.

2.7.4 Ingress

To access Services from outside the cluster, an *Ingress* resource is utilized. This resource exposes HTTP/S routes from outside the cluster to Services within the cluster. The routing is defined by the rules in the Ingress resource [3, 14].

2.7.5 Storage

Kubernetes includes various storage concepts. However, two are particularly important for this paper and will be discussed further.

Volumes As a container's file system is ephemeral, any data stored or modified during the container's lifetime will be lost if the container crashes or if the Pod containing it crashes or stops [3, 14]. Also, it can sometimes be necessary to enable two containers running on the same Pod to access a shared filesystem. The Kubernetes *Volume* abstraction resolves both of these problems. Kubernetes supports several types of Volumes. Volumes can be divided into two categories: ephemeral and persistent. Each category has multiple sub-types which utilize different storage technologies. Ephemeral volumes provide persistence only during the Pod's lifetime. In contrast, persistent volumes outlast the lifetime of a Pod. The various sub-types have distinct storage options, from using local storage on the Pod to employing cloud storage providers like AWS.

ConfigMaps and Secrets Applications typically require configuration, such as a URL and password, to access a database [3, 14]. One way to decouple application code from configuration is to provide configuration via environment variables. Using *ConfigMaps* and *Secrets*, engineers can avoid embedding such configurations and sensitive data in Pod specifications or container images.

A ConfigMap is a dictionary of key-value pairs. Containers running in Pods can be configured to access the values in ConfigMaps. The primary distinction between a ConfigMap and a Secret is that the latter is intended to store sensitive information like passwords, tokens, or keys. However, Secrets are not encrypted at rest by default and require additional configuration to ensure their complete secrecy.

2.7.6 Namespaces

Kubernetes *Namespaces* enable the isolation of a group of resources within a cluster [3, 14]. This is intended for environments with many users spread across multiple teams or projects or to separate development, test, and production environments. They are also helpful for providing separation between applications deployed in the cluster.

Kubernetes is a platform that is both comprehensive and highly intricate. The aforementioned components are just a fraction of the available elements.

2.8 Kubectl

Kubernetes offers a command-line utility to interact with the control plane of a Kubernetes cluster [18]. The tool called *kubectl*[12] enables the management of all components of a Kubernetes cluster. It is available for most operating systems and architectures. Listing 3 displays the most fundamental command: *get*. It displays a list of the available resources of a specific type in the cluster, in this case, Pods.

Listing 3: Displaying all Pods using Kubectl.

```
1 kubectl get pods
```

To develop applications for Kubernetes, engineers need access to a cluster to deploy and test their code changes. Multiple projects aim to provide a simple installation of a local cluster for learning and developing applications.

2.9 Minikube

Minikube[13] is a tool used to quickly set up a local single-node Kubernetes cluster on a developer's machine, ideal for developing and testing Kubernetes applications [19]. After installation, the cluster can be started with the with the *start* command depicted in Listing 4.

Listing 4: Creating a Kuberenetes Cluster Using Minikube.

```
1 minikube start
```

2.10 Configuration Management

Using the Kubernetes resources depicted above, Kubernetes applications can be easily created. Engineers can always manually edit the manifest files when they need to change a configuration. However, more often than not, there is a need for an automated way of handling such configuration

[12] https://kubernetes.io/docs/reference/kubectl/
[13] https://minikube.sigs.k8s.io/docs/

modifications [11]. One of the most common use cases is a need to tailor the configuration for different environments, such as the development or production stages. There are various configuration management tools available. However, in this thesis only two of them will be presented.

2.10.1 Kustomize

Kustomize[14] is an open source configuration management tool included in the kubectl CLI tool since Kubernetes 1.14 [11, 17, 20, 21]. It uses the untouched YAML Kubernetes manifest files and, with overlays, allows for substitution and reuse. Kustomize introduces the concept of base and overlay YAML manifests. Engineers develop base manifests and use overlays to patch the manifests for different environments. The specification for the customization is defined in a `kustomization.yaml` file. Kustomize reads these files and then merges the base files with the changes specified in the overlay files to generate the final manifest. Kustomize has several inbuilt methods for modifying the base manifests, but this thesis focuses on patches. Patches add or override fields on resources.

Example Figure 4 displays a typical directory structure for using Kustomize. The `base` directory comprises the base manifests and a kustomization file that specifies them. The `dev` folder contains the overlays and a corresponding kustomization file.

```
.
├── base
│   ├── base-kustomization.yaml
│   └── base-pod.yaml
└── dev
    ├── dev-kustomization.yaml
    └── dev-pod.yaml
```

Figure 4: Kustomize Directory Structure.

The `base-pod.yaml` file describes a Pod resource and its content is identical to that presented in Listing 2. The `base-kustomization.yaml` file is depicted in Listing 5. The `resources` list specifies all the Kubernetes resources that kustomize should manage.

Listing 5: `base-kustomization.yaml`

```
1 apiVersion: kustomize.config.k8s.io/v1beta1
2 kind: Kustomization
3 resources:
4 - base-pod.yaml
```

Listing 6 depicts the kustomization file in the `dev` directory. Here, the resources list specifies the base directory, and it includes the `base-kustomization.yaml` file as a resource to be used. In addition, it includes a patches definition. The `path` specifies where to take the input for the patch operation from. In this example this is the `dev-pod.yaml` file. Since the `target` defines the kind `Pod`, the patch operation applies to all resources of the type `Pod`. There are also other target selectors to identify the resource the patch must apply to.

[14] `https://kustomize.io/`

Listing 6: `dev-kustomization.yaml`

```
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
resources:
 - ../base
patches:
  - path: dev-pod.yaml
    target:
      kind: Pod
```

The `dev-pod.yaml` file only includes the content shown in Listing 7, defining a fragment of a Kubernetes Pod resource, precisely the container definition, but with a different name. This kind of patch is called a *strategic-merge-style* patch. The default behavior is to replace values. This patch strategy needs as input just enough context to identify the elements to change. When kustomize renders the final manifest, the name of the container will be replaced with `nginx-renamed`.

Listing 7: `dev-pod.yaml`

```
spec:
  containers:
  - name: nginx-renamed
```

Kustomize makes it easy to leverage existing Kubernetes manifests and make them configurable. However, the framework has limitations. In some cases, complex nested configurations may be necessary, making it challenging to comprehend how the final rendering will be produced. Furthermore, kustomize does not have parameters and templates, which may be necessary sometimes [11].

2.10.2 Helm - The Package Manager for Kubernetes

An alternative approach to managing configurations is to utilize *Helm*[15]. Although it functions primarily as a package manager for Kubernetes, Helm leverages a template engine to provide configurable manifests [3, 17, 22]. Helm provides a fast and convenient method for installing applications on a Kubernetes cluster. The Helm command line tool enables engineers to install and configure applications. The *Artifact Hub*[16] is the primary source for browsing applications. However, users can add other repositories to Helm. As of version 3, Helm is now only installed on the machines of the engineers, eliminating the need for installation on a cluster. It also allows for the straightforward upgrading of applications and rollback to a previous state in case of problems. Helm leverages the concept of templates. This involves developers creating a template for a Kubernetes resource, defining values that should be configurable, and then Helm uses the Go template engine to generate the final resource. For example, the command in Listing 8 installs a *MySQL*[17] database to a cluster.

Listing 8: Intalling MariaDB Using the Helm Install Command.

```
helm install my-database stable/mysql
```

Helm Charts Helm application packages are called *Charts*, and they fully specify the resources needed to run the application, its dependencies, and its configurable settings. A chart is a directory

[15] `https://helm.sh/`
[16] `https://artifacthub.io//`
[17] `https://www.mysql.com/`

that only needs to contain a few files. It has a structure as shown in Figure 5. Helm charts do not include the container image itself. It has metadata that specifies the location of the image, similar to a Kubernetes Pod resource. Kubernetes will then download the container image from the specified location.

```
.
├── Chart.yaml
├── templates
│   ├── deployment.yaml
│   └── _helpers.tpl
└── values.yaml
```

Figure 5: Structure of a Simple Helm Chart.

Chart.yaml The `Chart.yaml` is a YAML file containing information about the chart. In its simplest form, it contains only the `apiVersion`, a `name`, and a `version` of the chart as shown in Listing 9.

Listing 9: `Chart.yaml`

```yaml
apiVersion: v2
name: demo
version: 0.1.0
```

Templates Directory The templates directory contains the templates for Kubernetes resource manifests. The template engine will process these files and replace the placeholders with actual values. Listing 10 shows the example Pod manifest from above but with a placeholder for some entries. The Go template language utilizes double braces to denote the start and end of a replaceable value in a template.

Listing 10: Pod Template Manifest

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: {{ include "demo.name" . }}
spec:
  containers:
    - name: {{ .Values.pod.name }}
      image: {{ .Values.pod.repository }}
```

Values The `values.yaml` file contains the default values for a chart, which can be accessed in a template using the `Values` object. The template instruction `{{ .Values.pod.name }}` looks in the Values object for an element named `pod` with a sub-element `name` and then injects the found value into the template.

Helm utilizes these values to substitute the placeholders in the templates and render the final chart. The Helm CLI can be passed alternative value files as parameters for commands. This allows engineers to override the default values to tailor the configuration of a Helm application for different environments. Listing 11 shows a `Values.yaml` file with the values for the above example.

Listing 11: `Values.yaml`

```
1 pod:
2   name: nginx
3   repository: nginx:latest
```

Functions The Go template language is Turing-complete, allowing developers to create functions that can be reused in all templates. This language makes extensive use of the pipe operator. The function presented in Listing 12 obtains the name of a Pod and transfers it to the `trunc` function, which truncates all strings after after a defined number of characters, in this example 63. To call such a function in a template, the *include* keyword is used. This is demonstrated in Listing 10 in line 4, where the function presented in this section is utilized. Functions for Helm charts are typically stored in the `_helpers.tpl` file.

Listing 12: Go Template Function

```
1 {{/*
2 Get the name of the pod.
3 */}}
4 {{- define "demo.name" -}}
5 {{- .Values.pod.name | trunc 63  }}
6 {{- end }}
```

Chart Tests Helm implements the functionality to test charts. These tests could validate that the chart works as expected when installed. A Helm chart test is located in the `templates/test` directory and consists of a Kubernetes *Job* resource that defines a container with a specific command to execute. This command could, for example, try to connect to a service and thus validate that it is reachable as expected. The container must successfully exit in order for the test to be considered a success.

With Helm, engineers can conveniently install and share Kubernetes applications. The template functionality offers an advanced approach to managing configurations, but it has a steeper learning curve due to the language used and the additional concepts introduced [20].

2.11 Developing Cloud-Native Applications

A microservice architecture-based application can complicate the development process because each distinct service has to be independently built and run to provide the application's overall functionality [23]. If done manually, the engineer is required to build individual container images and subsequently deploy them to the Kubernetes cluster. Every code change would imply repeating that cycle.

2.11.1 Skaffold

To help developers, *Skaffold*[18] aims to automate this process and provide a fast local development workflow [24]. Skaffold enables developers to define the container images and Kubernetes resources that the application consists of declaratively in a YAML configuration file. Upon receiving the corresponding command, Skaffold uses this file to build the container images automatically, augment

[18] `https://skaffold.dev/`

the Kubernetes resources with the new container image information, and then deploy the application to a Kubernetes cluster. The process of augmenting the Kubernetes resources is called hydration. To enable fast development and local testing, Skaffold can continuously watch a code directory, detect changes, and trigger a rebuild and redeployment of the changed code. Skaffold supports several tools to render Kubernetes manifests, such as Kustomize and Helm. It also supports environment management through profiles. Profiles enable modification of the defined workflow and can be enabled as needed. Skaffold profiles work similarly to kustomization patches and can be enabled by appending the `-p` flag followed by the desired profile name.

Listing 13 shows a typical skaffold configuration. The `build` section defines the artifacts to be built. Here, a name, `demo-image`, and the path to the Dockerfile are specified. The `manifests` section configures the renderer and the path to the manifests. This example uses Kustomize and specifies the appropriate base manifests. In the `deploy` section, developers specify how the application will be deployed. In this example, kubectl is used. The last section specifies the profiles. This example defines a `dev` profile, and when enabled, the path of the manifest files for kustomize will be replaced by the `dev` overlays, as described in Section 2.10.1.

Listing 13: Skaffold Configuration

```
1  apiVersion: skaffold/v3
2  kind: Config
3  metadata:
4    name: demo
5  build:
6    artifacts:
7      - image: demo-image
8        context: src/demo
9  manifests:
10   kustomize:
11       paths:
12         - ./k8s-manifests/base
13 deploy:
14   kubectl:
15     {}
16 profiles:
17   - name: dev
18     patches:
19       - op: replace
20         path: /manifests/kustomize/paths
21         value: ./k8s-manifests/dev
```

Skaffold speeds up the Kubernetes application development process for engineers by handling the repetitive task of rebuilding and re-deploying modified code. This also provides the engineers with immediate feedback throughout the development process.

2.12 GitOps for Kubnernetes

To manage deployed applications and control a cluster, engineers can use tools like Kubectl, Kustomize, Helm, or Skaffold directly. However, under this approach, every engineer must have cluster access and the appropriate permissions [25]. Additionally, the state of the cluster and deployed applications might lack transparency. Changes to the cluster or an application are invisible, including who made the change and why. GitOps tools can significantly improve observability and provide other benefits.

2.12.1 Argo CD

Argo CD[19] is a GitOps tool built for Kubernetes. With Argo CD, engineers can declaratively define the Kubernetes applications they want to deploy in the cluster [17, 26]. The definition can be stored in a Git repository, and Argo CD synchronizes the cluster with the defined state. This approach of defining the desired state makes deployments and other lifecycle events automatable, traceable, and auditable because they are documented in a version-controlled file. Instead of *pushing* changes via Kubectl or Helm, Argo CD continuously *pulls* in changes from the Git repository and applies them to the cluster. This technique can also enhance the security of a cluster since fewer highly privileged users are required to interact with it directly. Argo CD includes a user-friendly graphical interface, illustrated in Figure 6. This interface provides engineers with complete visibility of the applications managed by Argo CD. To define an application, Argo CD adds custom resources to Kubernetes. These resources, like all other Kubernetes resources, are formatted in YAML. A basic example of an Argo CD Application resource is depicted in Listing 14. The crucial part is the `source` section. The link specified here leads to the repository of Kubernetes resources required to install an application.

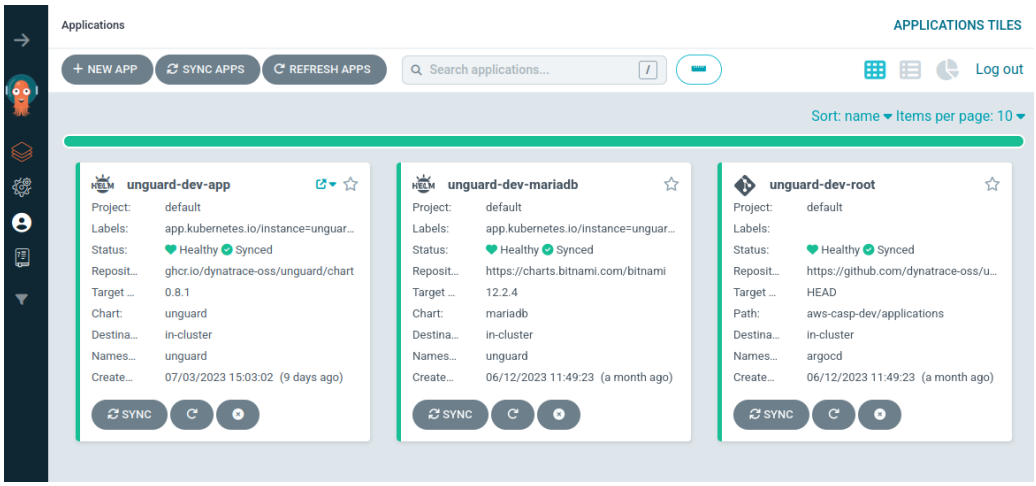


Figure 6: Argo CD Graphical User-Interface.

Listing 14: Argo CD Application Manifest

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: guestbook
  namespace: argocd
spec:
  project: default
  source:
    repoURL: https://github.com/argoproj/argocd-example-apps.git
    targetRevision: HEAD
    path: guestbook
  destination:
    server: https://kubernetes.default.svc
    namespace: guestbook
```

[19] `https://argo-cd.readthedocs.io/en/stable/`

2.13 GitHub Actions

GitHub Actions[20] is a continuous integration and continuous delivery (CI/CD) platform [27]. It enables developers to automate various tasks in their workflow, including building, testing, and deploying code changes. Developers can create customized workflows by utilizing existing Actions from the *GitHub Marketplace*[21] or writing their own. These workflows can be triggered by various events, such as new commits or the creation of a pull request. Overall, GitHub Actions provide an efficient and flexible way to automate development processes within the GitHub platform. Listing 15 depicts a simple demo GitHub Action. The action is triggered on each push, runs on an Ubuntu environment, and outputs a message on the console.

Listing 15: GitHub Action

```
1 name: GitHub Actions Example
2 on: [push]
3 jobs:
4   GitHub-Actions-Example:
5     runs-on: ubuntu-latest
6     steps:
7       - name: Print Message
8         run: echo 'This action is triggerd on a push!'
```

[20] https://github.com/features/actions
[21] https://github.com/marketplace?type=actions

3 Overview

After detailing the necessary concepts in the previous chapters, the following section introduces the application at the core of this thesis. It contains an overview of the application, its functionality, and its architecture. This thesis was written in collaboration with Dynatrace, which offers a sophisticated Kubernetes security and observability solution [28]. To improve and validate the solution internally and to demonstrate it to customers, Dynatrace needed a worst practice example – a testing ground. This led to the development of *Unguard*[22], an application that disregards all Kubernetes configuration and application-security best practices.

3.1 Unguard, an Insecure Cloud-Native Microservice Demo Application

Unguard is a rudimentary microblogging application that allows users to post text and images, provides basic user management capabilities, and can place advertisements [29]. Unguard utilizes a cloud-native architecture, where containerized microservices provide all functionality. Its services are implemented using various programming languages, including Java, C#, Go, and JavaScript. Unguard uses Kubernetes to orchestrate its services in a cloud computing environment, which means the entire infrastructure is defined as Kubernetes resources. The microservices are all packaged as Docker container images. Unguard is intentionally insecure, incorporating several security vulnerabilities. As this thesis does not focus on these vulnerabilities per se, see the official GitHub page [30] for further information. Figure 7 displays a screenshot of a logged-in user's timeline on Unguard.

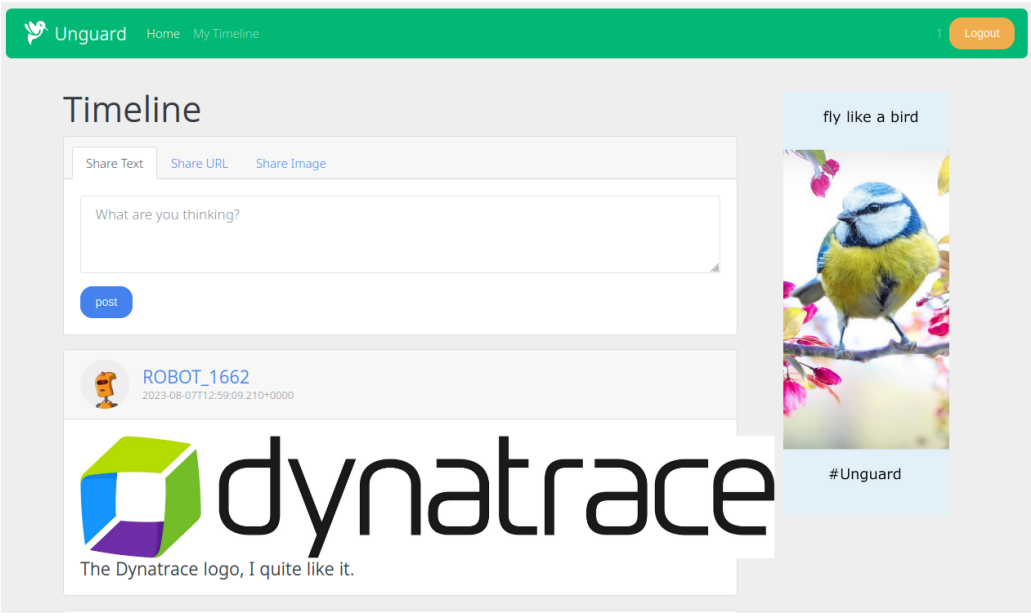


Figure 7: Unguard's Homepage.

[22] https://github.com/dynatrace-oss/unguard

3.2 Service Architecture of Unguard

Unguard comprises nine microservices, two databases, and two load generators. Figure 8 shows the architecture diagram from a service level. The *envoy-proxy* handles all inbound traffic to the application. Depending on the URL, the traffic is then forwarded to the *ad-service* or the *frontend*. The *status-service* is responsible for providing a user with information about the status of Unguard's Kubernetes deployment. The ad-service allows admins to upload images that serve as advertisements to users. The frontend is the primary interaction point for users. It utilizes all the other services to provide the functionality of a microblogging application.

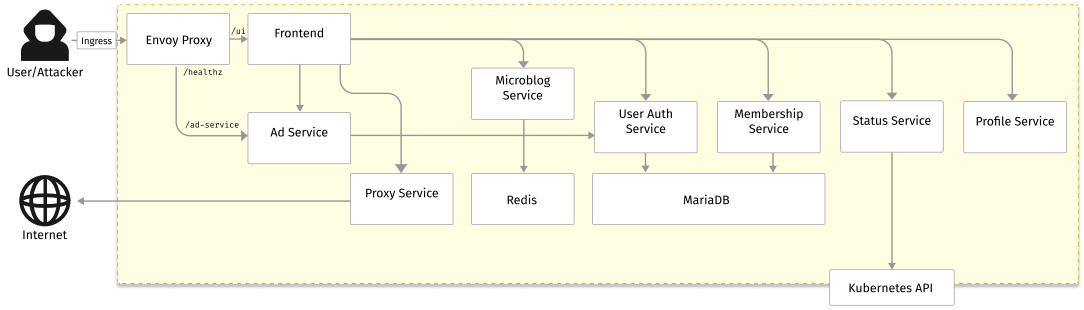


Figure 8: Unguard's Service Architecture [29].

A typical user session might look like this: After visiting the Unguard website, the user needs to log in. To authenticate or register a user, the frontend sends a request to the *user-auth-service*. This service maintains a MariaDB database of all registered users. If the user authentication succeeds, the user-auth-service returns an authentication token. The user is then redirected to the home page. The frontend loads the timeline by requesting data from the *microblog-service*. This service holds all the users' posts in a Redis database. When the user posts something to the timeline, the frontend sends a request to the microblog-service. In Unguard, users can share URLs and images alongside text. However, unlike Twitter, directly uploading and publishing pictures is not supported. Instead, users need to provide a link to the image. The frontend utilizes the *proxy-service* to fetch the image from the internet and then passes the obtained data to the microblog-service, which saves it to the Redis database. URLs are handled similarly. However, rather than an image, the proxy-service retrieves the metadata of the link. Unguard features a simulated membership model with two tiers: free and pro. The free membership contains advertisements. Users can upgrade to the pro membership to consume Unguard without advertisements. To manage the membership status of users, the frontend sends requests to the *membership-service*, which persists this data in MariaDB. A user can visit the profiles of other users, and the *profile-service* handles this.

Unguard offers a *user-simulator* to generate synthetic user traffic. This program emulates a genuine user's behavior by generating traffic through a real browser. It shares texts, images, and URLs and visits other user profiles. These bots are run periodically, filling the platform with content. The post in Figure 7 is created by such a bot, as indicated by the username with the prefix ROBOT. Unguard also includes a program to simulate an attacker by periodically exploiting the vulnerabilities of Unguard's services.

4 Previous Workflow for Unguard

With the necessary technological background presented in Section 2 and an understanding of the overall structure and function of Unguard provided in Section 3, this chapter now focuses on the current development and implementation process and associated obstacles.

4.1 Source Code Architecture

Unguard utilizes a microservice architecture, but all the code of its services is part of the same code repository. The services are containerized and use Docker as container technology, and thus, each service maintains a Dockerfile specifying its container image. The Kubernetes resources defining Unguard are also part of the repository. Previously, Unguard utilized Kustomize as a configuration management tool to tailor its configuration for different environments. Since Unguard is open source, the Kubernetes configuration for the Dynatrace internal clusters cannot be included in the public repository and is instead stored in a separate internal repository.

4.2 Development and Local Deployment

Unguard uses Skaffold to manage its building, pushing, and deployment workflows, providing engineers with a more convenient development experience. It is configured to create all container images and to deploy Unguard to a cluster. Furthermore, Skaffold was previously utilizing Kustomize to render the Kubernetes resources. Unguard's Skaffold configuration included profiles to support various environments and configurations. These profiles defined the source location for the Kustomize overlays. Engineers only needed to run Skaffold with the desired profile, and the related Kustomize configuration would be applied upon deployment. With a local cluster, such as Minikube installed, engineers merely needed to execute the Skaffold command, as shown in Listing 16, to deploy their version of Unguard locally.

Listing 16: Previous Development Command for Unguard

```
1 skaffold run -p localdev-minikube
```

4.3 Production Cluster Deployment

Unguard runs on various dedicated Kubernetes clusters, each responsible for different stages of development within the internal Dynatrace infrastructure. Deploying or upgrading Unguard on these clusters was similar to the workflow mentioned above. However, in addition to the publicly available Unguard source code, the engineer had to also download another internal repository onto their machine. It contained a second Skaffold configuration file with extra Skaffold profiles and associated Kustomize overlays for the internal environments. For example, these overlays modified the Kubernetes Ingress resource by adding internal IP addresses and configuring routes to access Unguard.

After downloading the public and the private repository and connecting the engineer's machine to the desired cluster, the engineers could run the Skaffold file with the appropriate profile from the private repository. Skaffold would then build the container images, render the Kubernetes resources, and deploy the application to the remote cluster. Figure 9 depicts this deployment workflow.

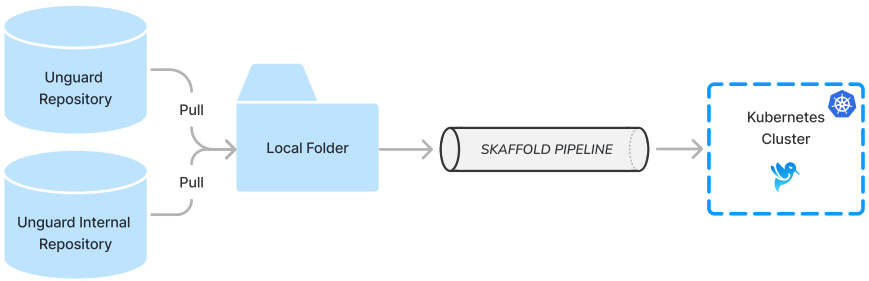


Figure 9: Previous Deployment Workflow.

4.4 Benefits and Drawbacks

The approaches presented in this chapter have advantages and disadvantages. The advantages include:

- Since the application utilized Kustomize, adding or altering overlays to tailor Unguard to the environments was relatively simple.
- Using Skaffold significantly simplifies the development process.
- The remote deployment and local development processes were similar and utilized Skaffold.

However, the previous approach also had notable disadvantages, such as:

- The nested Kustomize structure that spans multiple repositories made it challenging to understand the final Kubernetes manifest's rendering.
- The nested Kustomize structure introduced a dependency on the order of the applied patches.
- The previous process of deploying or upgrading a cluster was manual and, hence, error-prone. Due to being a manual process, whenever a new version of Unguard was released, an engineer needed to follow the aforementioned steps to upgrade Unguard.
- The remote deployment process necessitated extra AWS-specific tools, including those for connecting to the cluster.
- No prebuilt container images for Unguard's services were available to download; they had to be rebuilt from scratch for every newly created cluster. However, for subsequential deployments or upgrades of a cluster, the unchanged images could be reused.
- The fact that there are no prebuild container images introduces several additional problems, such as:
  - The builds of the same version of Unguard were not guaranteed to be reproducible because of potentially changed base images or other differences in the building process of the container images.
  - It took a significant amount of time for Unguard to be built and deployed on a cluster for the first time. Even on a powerful machine, it took about 30 minutes.
  - To build the container images, all base images and other dependencies had to be downloaded from artifact registries, which resulted in unnecessary resource consumption and often exceed the rate limits of online services, e.g., Docker Hub.

5 Introducing GitOps

The core aspect of this thesis is to research and implement an up-to-date GitOps workflow for Unguard. This section presents the GitOps model and its origins. It assesses whether Unguard currently adheres to these practices and determines the necessary steps to implement a complete GitOps workflow.

5.1 Traditional Ops

In a traditional IT organizational model, an organization comprises distinct development, quality assurance (QA), and operations teams, wherein each focuses on a particular aspect of the application development process [11]. Development teams provide new application versions to a QA team, which tests the application increment before passing it on to the operations team for deployment. With three entities involved, the likelihood increases that critical details are not communicated, potentially leading to gaps in testing or incorrect deployment.

5.2 DevOps

DevOps is an organizational structure and set of software development practices that combine software development (Dev) and IT operations (Ops). In a DevOps model, the development teams are also responsible for testing, deploying, and operating a software application. Figure 10 shows the difference between traditional Ops, where an organization is divided based on functional boundaries, and DevOps, where teams are structured according to distinct products or components. In a DevOps team, each member is ideally responsible for programming, testing, deploying, and operating the respective product or component.

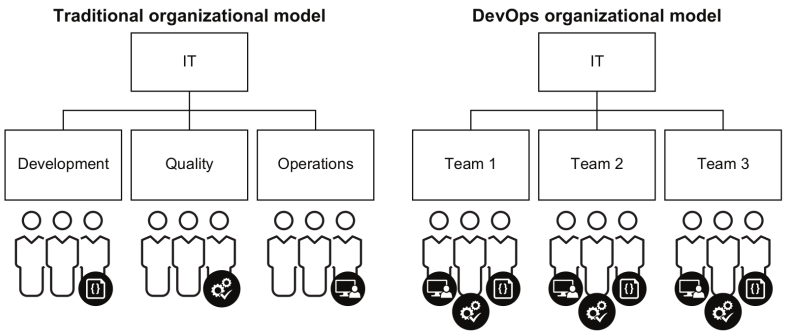


Figure 10: Traditional Ops versus DevOps [11].

A core DevOps principle entails automating the software development lifecycle. This involves the automation of tests, builds, releases, development environment provision, and other manual tasks that could hinder the software development process or lead to human error.

Two software development practices address automation in the development process:

- *Continuous integration* (CI) describes the practice of regularly integrating all code changes into the main branch, automatically testing each change, and automatically kicking off a build.
- *Continuous delivery* (CD) describes the practice of automating the infrastructure provisioning and application release process.

5.3 GitOps

GitOps is an operational framework that takes DevOps best practices and extends them to IT infrastructure management. It enables developers to define and automate IT infrastructure and manage it similarly to their codebase through Git-based repositories. In other words, GitOps embraces Git repositories as the single source of truth for Infrastructure as Code (IaC). IaC refers to the management of IT infrastructure using configuration files.

In a GitOps model, the infrastructure code is kept in a version control system, just like the application source code. Instead of manually making changes to the infrastructure through a UI or CLI, an engineer makes changes to the configuration files that declare the desired state. These changes then undergo the standard version control processes. First, the engineer opens a pull request for others to review. Once approved and merged with the main branch, an operator software process is responsible for converting the system's current state to the desired state based on the stored configuration in the Git repository.

The GitOps Working Group defines GitOps with these principles [31]:

- "Declarative - A system managed by GitOps must have its desired state expressed declaratively.
- Versioned and immutable - The desired state is stored in a way that enforces immutability and versioning and retains a complete version history.
- Pulled automatically - Software agents automatically pull the desired state declarations from the source.
- Continuously reconciled - Software agents continuously observe the actual system state and attempt to apply the desired state."

The GitOps model emphasizes equal treatment of application and infrastructure code, using Git as the single source of truth [11]. As a result, utilizing a version control system benefits both domains, improving code quality, traceability, auditability, and collaboration. Furthermore, GitOps relies on automation that synchronizes the infrastructure with the intended state. In its most extreme manifestation, the GitOps model prohibits any manual changes to the infrastructure.

5.4 How to Apply GitOps to Unguard

After discussing these DevOps and GitOps principles and examining Unguard's previous development and deployment processes, we can identify where they diverge. This section lists GitOps principles that have been violated and suggests solutions for addressing them, in Table 1.

Violation	Solution
Unguard does not employ any CI or CD delivery practices.	A CI/CD pipeline is implemented to improve the development process and reduce human error.
The cluster's desired state is not declaratively defined and stored in a version control system.	The cluster's desired state is declaratively defined in a Git repository.
Deploying or upgrading Unguard is done by making manual changes to the Kubernetes infrastructure.	A GitOps tool automatically keeps the cluster in the desired state and eliminates the need for manual changes to Kubernetes resources.
No software agents automatically pull and reconcile the desired state of the system.	

Table 1: Violations of GitOps Principles and Proposed Solutions.

6 Applying GitOps Principles to Unguard

This chapter describes the steps taken to implement the aforementioned solutions for applying GitOps principles to Unguard. It describes the new architecture that was developed. The solutions shown are the result of research and discussions with GitOps experts employed at Dynatrace.

6.1 Helm Chart for Unguard

The initial task was to create a Helm chart for Unguard, as detailed in Section 2.10.2. It would have been possible to apply GitOps principles to Unguard without developing a Helm chart. However, a Helm chart significantly simplifies Unguard's deployment and installation process. The chart lets engineers quickly install Unguard to a local or remote Kubernetes cluster with just a few commands without using Skaffold and without having to build it themselves. This makes Unguard accessible to a much broader audience, also enabling non-developers to install it easily. The chart also simplifies the implementation of the other required steps for applying GitOps principles.

6.1.1 Creating the Chart

Helm offers a command to generate a skeleton chart containing the standard files and directories needed. This skeleton was used as a starting point. The generated Chart.yaml was adapted for Unguard. The starting point was to create Helm chart templates for Unguard's Kubernetes resources. As Kubernetes manifests had previously been rendered using Kustomize, the next step was to generate a template for each base manifest. This was achieved by copying the contents of the Kustomize base directory into the new chart's template folder. Then, these templates were made configurable, meaning that all the default settings from the templates were extracted and relocated to the values file. The next step was to manually analyze all the overlay directories and recreate the effects these overlays previously would have had. These effects were added to the templates. The process is illustrated in the following example.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: unguard-envoy-proxy
  labels:
    app.kubernetes.io/name: envoy-proxy
    app.kubernetes.io/part-of: unguard
spec:
  type: ClusterIP
  selector:
    app.kubernetes.io/name: envoy-proxy
    app.kubernetes.io/part-of: unguard
  ports:
    - name: http
      targetPort: 8080
      port: 8080
    - name: health
      targetPort: 8081
      port: 8081
```

```yaml
apiVersion: kustomize.config.k8s.io/v1beta1
kind: Kustomization
namespace: unguard
resources:
  - ../base
  - ingress.yaml
patchesStrategicMerge:
  - |-
    apiVersion: v1
    kind: Service
    metadata:
      name: unguard-envoy-proxy
    spec:
      type: NodePort
```

Figure 11: Utilizing Kustomize to Adjust Service Type.

Figure 11 shows the `AWS kustomization` file on the right, which, when applied, changes the value of `spec.type` in the `unguard-envoy-proxy` Service from `ClusterIP` to `NodePort`. For a comprehensive introduction to Kustomize, please refer to Section 2.10.1.

To achieve the same effect with the Helm chart, new values were added, as shown on the right side in Figure 12. Using the template language, a conditional block was created in the `unguard-envoy-proxy` template file. The output is determined by the value of `aws.enabled`. If `aws.enabled` is `true`, `NodePort` is used. Otherwise, the default value defined in the values file is used.

Figure 12: Utilizing Helm to Adjust Service Type.

After representing all overlays in the chart, the subsequent step was to modify Skaffold to utilize the recently incorporated Helm chart instead of Kustomize.

6.1.2 Adopting Skaffold

First, the existing configuration for Kustomize was removed. Instead, the new Helm chart was added. The subsequent step involved instructing Skaffold to substitute the information of the container image in the Helm chart with the ones created during a Skaffold run.

```
name: unguard
chartPath: ./chart
namespace: unguard
createNamespace: true
valuesFiles: ["./chart/values.yaml"]
setValueTemplates:
  # ad-service
  adService.deployment.container.image.repository: "{{.IMAGE_REPO_unguard_ad_service}}"
  adService.deployment.container.image.tag: "{{.IMAGE_TAG_unguard_ad_service}}@{{.IMAGE_DIGEST_unguard_ad_service}}"
```

Figure 13: Adopting Skaffold.

Figure 13 displays an excerpt from the added section in the Skaffold file. The example only exhibits the adjustments made for the *ad-service*. However, the adjustments for the other services are similar. Skaffold stores information about container images as environment variables during the building process. The `setValueTemplates` field allows engineers to assign the content of environment variables to Helm values. This feature was utilized to replace the metadata of the images in the chart with the metadata of the images that Skaffold built. This is shown in the red section on Figure 13. Consequently, when Skaffold installs the chart, the appropriate images are selected.

To transition from Kustomize to Helm, the Skaffold profiles needed to be migrated as well. This process was straightforward; now, each profile uses a distinct values file to adopt the configuration of the Helm chart. Therefore, new value files were created for all required skaffold profiles.

6.1.3 Local Development

Since Skaffold has been adopted, the workflow for developing Unguard remains unchanged. The only change is that Unguard is now configured for Minikube by default. Consequently, the Minikube Skaffold profile has been removed.

6.2 GitHub Action CI Pipeline

The initial step in the proposed solution was to implement a CI pipeline for Unguard. The Unguard repository is hosted on GitHub, so the pipeline has been implemented as a GitHub Action. An example of a GitHub Action is depicted in Section 2.13. The CI pipeline consists of a GitHub Action for testing and a GitHub Action for building and pushing artifacts.

6.2.1 Testing

This GitHub Action tests the entire infrastructure of Unguard. Firstly, the well-formedness of the Helm chart is verified. Then, the artifacts are built using Skaffold and deployed to a freshly created Minikube cluster within the GitHub Action environment. It performs an entire Skaffold run, and by that, it ensures that all container images of the services can be built. Since Skaffold uses the Helm chart, it also verifies that the chart can be installed. A Skaffold run only exits successfully if the installation stabilizes, meaning that all containers start and do not immediately crash, thus ensuring no fatal errors occurred in one of the services. Additionally, a basic Helm chart test is performed. Helm chart tests, as outlined in Section 2.10.2, can be utilized to confirm the correct installation of a chart and to validate specific features of the installed application. The test developed for this thesis is rudimentary and merely tests the connectivity to the frontend. This test can be extended in future work to test all the functionalities of Unguard.

The GitHub Action is executed only if modifications to a service's source code, Helm chart, or Skaffold configuration are included in a pull request. The process is repeated as new commits are added to this pull request. Completing this GitHub Action is obligatory before the pull request can be merged to give engineers feedback on possible errors.

6.2.2 Building and Pushing Artifacts

This GitHub Action rebuilds the container images and the Helm chart and then pushes them to the GitHub Container Registry (GHCR). A commit to either the main or release branch triggers the GitHub Action. The pushed artifacts are tagged with either a release or intermediate release tag.

6.3 Argo CD

So far, implementing a CI pipeline has been the first step in applying GitOps principles to Unguard. Further steps, namely the need for a declaratively defined state and an automatically synced cluster infrastructure, were accomplished with Argo CD.

Argo CD, as described in Section 2.12.1, provides a custom Kubernetes resource, the Application resource. With this resource, engineers can declaratively define Kubernetes applications that should

be installed and managed by Argo CD. However, Argo CD must be made aware of this newly defined Application resource. This could be done through the UI or the CLI but would contradict the GitOps principle of not performing manual changes. In addition, GitOps requires that all configurations, including the Argo CD configuration, reside within a Git repository.

App of Apps Pattern This is where the *App of Apps* pattern comes into play. This pattern enables the definition of a *top-level* Argo CD Application that does not reference Kubernetes resource files but instead refers to a directory containing Argo CD Application manifests for all Kubernetes applications to be installed. Each *sub-application* manifest in this directory references the Kubernetes resources necessary to install the desired applications. In this approach, the entire configuration, including Argo CD's configuration, is defined declaratively and can be stored in a Git repository.

6.3.1 Infrastructure Repository

A second private GitHub repository has been created to host the Argo CD configuration for Unguard. This also includes the values for configuring Dynatrace's internal Unguard clusters. In order to present the work conducted in this thesis publicly, a second repository[23] has been created without any confidential information. The public repository has the same structure as the private repository. The structure of this repository is illustrated on the left in Figure 14.

Figure 14: Root Argo CD Application.

The `unguard-root.yaml` is the aforementioned Argo CD *top-level* Application resource for the *App of Apps* pattern. As highlighted in red in Figure 14, it points to the `unguard/application` subfolder located within this repository. This folder contains the Argo CD Application manifests for the Kubernetes applications to be installed. In this case, there are two Applications: *Unguard* and *MariaDB*.

MariaDB Figure 15 shows, on the left, the Argo CD Application resource defining the MariaDB database instance for Unguard. The *Bitnami MariaDB Helm chart*[24] is defined as the source in the blue highlighted area. The `targetRevision` field defines the chart version to be installed. It is possible to define Helm values within the Argo CD Application resource to override the default settings. In the case of MariaDB, the `primary.persistence.enabled` parameter was set to false, as

[23] `https://github.com/MfCrizz/unguard-infra/tree/main`
[24] `https://github.com/bitnami/charts/tree/main/bitnami/mariadb`

shown in the red highlighted section. This leads to the database operating in *in-memory* mode, i.e., without storing the data in a file system. The data is not retained since Unguard is a demonstration application that stores no crucial data.

Unguard's Dependency on MariaDB Unguard utilizes MariaDB for both the user-auth-service and membership-service, with both services expecting a running instance before starting. The dependency on the database posed a problem, as Kubernetes lacks the ability to specify the sequence in which pods are started. This was not an issue with the previous deployment approach, as Skaffold handled the installation and was configured to deploy the database before Unguard. A solution had to be found as the new approach utilizes Argo CD, and relying on Skaffold was not an option anymore.

When deploying applications in distributed environments using Kubernetes, multiple components are launched simultaneously, making it impossible to guarantee a particular launch order [32]. There are several methods for managing dependencies in Kubernetes. One option is implementing `initContainers`, which are specialized containers that run before the actual containers in a Pod. These containers can execute commands to ensure that, for example, a database is accessible and ready before the app container starts. This approach was initially chosen for Unguard but was later replaced by another solution utilizing Argo CD.

While Kubernetes does not offer a means to manage deployment sequences, Argo CD does [26]. In Argo CD, there is a concept called *Sync Waves*. These enable the Application resources of Argo CD to be assigned an integer that specifies the *wave*, or order, in which an application should be installed. All the operations specified in one wave have to be completed before Argo CD starts the next one. To utilize sync waves for Unguard, the database has been annotated with *1* and Unguard with *2*, as illustrated in Figure 15 in green. This instructs Argo CD first to install Maria DB and then Unguard, thus resolving the dependency issue.

MariaDB Argo CD Application

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: unguard-mariadb
  namespace: argocd
  finalizers:
    - resources-finalizer.argocd.argoproj.io
  annotations:
    argocd.argoproj.io/sync-wave: "1"
spec:
  project: default
  destination:
    server: https://kubernetes.default.svc
    namespace: unguard
  source:
    chart: mariadb
    repoURL: https://charts.bitnami.com/bitnami
    targetRevision: 12.2.4
    helm:
      releaseName: unguard-mariadb
      parameters:
        - name: "primary.persistence.enabled"
          value: "false"
```

Unguard Argo CD Application

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: unguard
  namespace: argocd
  finalizers:
    - resources-finalizer.argocd.argoproj.io
  annotations:
    argocd.argoproj.io/sync-wave: "2"
spec:
  project: default
  destination:
    server: https://kubernetes.default.svc
    namespace: unguard
  sources:
    - chart: unguard
      repoURL: ghcr.io/dynatrace-oss/unguard/chart
      targetRevision: 0.8.0
      helm:
        releaseName: unguard
        valueFiles:
          - $values/unguard/values.yaml
    - repoURL: https://github.com/MfCrizz/unguard-infra.git
      targetRevision: HEAD
      ref: values
```

Figure 15: MariaDB and Unguard Argo CD Application.

Lesson Learned While researching a solution to the dependency problem, it became clear that the implementation of Unguard's services was not always optimal for a cloud-native architecture. Such architectures require special considerations for dealing with the dependencies between their services. These services should be designed to be stateless and able to handle unresolved dependencies without crashing. The issue was resolved using Argo CD's sync waves as a workaround. However, it is essential to keep in mind that Kubernetes does not offer a means to specify the launch order of pods for any future extensions to Unguard.

Unguard The `unguard-app.yaml` is the Argo CD Application resource that defines the installation of Unguard. The section references the Unguard Helm chart, which can be found in the GHCR. The `targetRevision` parameter specifies the version of Unguard to install, as shown in the blue highlighted section of Figure 15 on the right side.

Sourcing the Values File from the Infrastructure Repository To configure Unguard for different environments, engineers can provide additional value files to override the default values. Since the infrastructure repository also contains the additional value files, Argo CD must be configured to source them from this repository. Before version 2.6, Argo CD solely supported obtaining value files contained within the chart. Fortunately, this has changed, and now it is possible to obtain values from other sources. This is accomplished defining a secondary source, as illustrated on the right side in Figure 15 in red. The infrastructure repository is listed as a second source, and a reference to a variable `values` is established. This variable can serve as the origin of the values file for Helm operations by Argo CD.

6.4 Argo CD Setup

Installing Argo CD on a Kubernetes cluster is straightforward, requiring only one command [26]. Next, the software must be made aware of the root Argo CD Application resource. This can be accomplished using kubectl as depicted in Listing 17. Once applied, Argo CD immediately shows the applications in the dashboard. As the intended state of having Unguard installed differs from the current state without Unguard, Argo CD initiates the setup process for Unguard in the cluster. First, MariaDB is installed, then Unguard. After a few seconds, the process stabilizes, and everything is installed. The dashboard, as depicted in Figure 6, shows that all applications are synchronized and healthy. The entire infrastructure is now declaratively defined, and Argo CD monitors the infrastructure repository for changes. For example, if the target version of Unguard is updated, Argo CD will immediately apply the changes and install the corresponding version of Unguard.

Listing 17: Initiate the Setup of Unguard using ArgoCD

```
1 kubectl apply -f unguard-root.yaml -n argocd
```

6.5 Unguard's new GitOps Architecture

With all described components in place, Unguard now predominately adheres to GitOps principles. The addition of a CI pipeline has automated the testing, building, and deploying of Unguard's artifacts. The infrastructure's state is now declaratively defined in a Git repository. With Argo CD, a GitOps tool automatically synchronizes the target and actual state of the cluster. Figure 16 depicts the new GitOps architecture for Unguard with all its components.

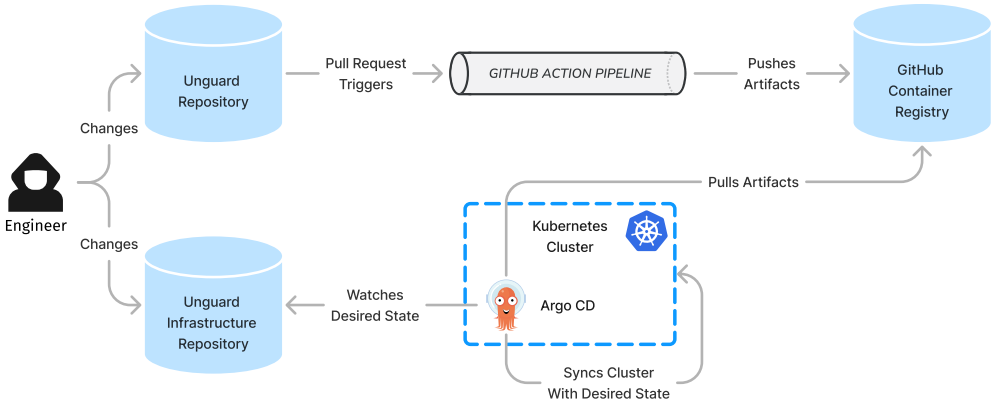


Figure 16: Unguard's new GitOps Architecture.

6.6 Quantification of the Speed Improvement

The following tests were conducted to quantify the installation speed gained with the newly developed solution using Helm compared to the previous approach using Skaffold. Testing was performed on a Dell Precision 7670 laptop with an Intel i9-12950HX and 64GB of DDR5 RAM at 4800MT/s. The Minikube cluster was configured with virtual 4 CPUs and 12 GB of RAM. For both approaches, the Minikue cluster was newly created to eliminate any remnants of a previous installation. The execution time of the commands was measured with `time`, a command line tool that summarizes the use of system resources.

6.6.1 Testing Procedure

First, the Minikube cluster was deleted and recreated using the command provided in Listing 18 to ensure a controlled and clean test environment.

Listing 18: Delete and Recreate Minikube Cluster

```
1 minikube delete && \
2 minikube start --driver=kvm2 --addons=ingress --cpus=4 --memory=12GB --disk-size=30GB
```

Unguard was then installed with Skaffold using the command shown in Listing 19. Using Skaffold, the container images must be created as they are not present on the cluster. With `time`, the elapsed time needed by this procedure was captured.

Listing 19: Measure Time for Instalation of Unguard using Skaffold

```
1 time skaffold run -p localdev-minikube
```

To measure the time it takes to reinstall Unguard without making any changes to the container images, in other words, allowing Skaffold to reuse the images already created, Unguard was deleted with the command in Listing 20 and then reinstalled with Skaffold using the command from the previous step, as shown in Listing 19.

Listing 20: Remove Unguard using Skaffld

```
skaffold delete
```

Next, the new approach with Helm was measured. The Minikube cluster was reinstalled with the command from Listing 18 to ensure the same starting conditions as in the previous measurement. Unguard was installed using the command presented in Listing 21. The duration for Unguard's complete setup was once again measured with `time`.

Listing 21: Measure Time for Instalation of Unguard using Helm

```
time ( \
helm repo add bitnami https://charts.bitnami.com/bitnami && \
helm install unguard-mariadb bitnami/mariadb \
    --set primary.persistence.enabled=false --wait && \
helm install unguard  oci://ghcr.io/dynatrace-oss/unguard/chart/unguard --wait )
```

Similarly, to measure the time required to reinstall Unguard using the new Helm approach, first, Unguard was uninstalled using the command shown in Listing 22 and then reinstalled using the command shown in Listing 21. In this scenario, the images are already on the cluster, and Helm simply must recreate the Kubernetes resources.

Listing 22: Remove Unguard using Helm

```
helm uninstall unguard  && \
helm uninstall unguard-mariadb
```

6.6.2 Results

The results for the measurements are presented in two tables. Table 2 displays the time elapsed for the initial installation, whereas Table 3 shows the time elapsed for a reinstallation of Unguard.

Approach	Elapsed Time
Previous: Skaffold	710.33 s
New: Helm	187.56 s
Difference	522.77 s

Table 2: Elapsed Time for an Initial Installation.

Approach	Elapsed Time
Previous: Skaffold	52,256 s
New: Helm	39,580 s
Difference	12.676 s

Table 3: Elapsed Time for a Reinstallation.

The percentage improvement in the time required to install Unguard was computed using Equation 1.

$$\frac{t_{prev} - t_{new}}{t_{prev}} \tag{1}$$

First, the difference between the time needed by the previous approach t_{prev} and the time needed for the new approach t_{new} was calculated. The result is then divided by the time taken for the previous approach t_{prev}.

Initial Installation

$$\frac{t_{prev} - t_{new}}{t_{prev}} = \frac{710.33s - 187.56s}{710.33s} = 0.73595 \tag{2}$$

For an initial installation, as presented in Table 2, the value for t_{prev} is 710.33 seconds, and the value for t_{new} is 187.56 seconds. The newly implemented solution is *522.77* seconds faster, corresponding to a *73.595* percent reduction in the time required to install Unguard. The speed improvement can be explained by the fact that in the new solution with Helm, the container images do not have to be built and can be downloaded instead.

Reinstallation

$$\frac{t_{prev} - t_{new}}{t_{prev}} = \frac{52.256s - 39.58s}{52.256s} = 0.24257 \tag{3}$$

In the case of a reinstallation, as presented in Table 3, the value for t_{prev} is 52.256 seconds, and the value for t_{new} is 39.58 seconds. The newly implemented solution is *12.676* seconds faster, corresponding to a *24.257* percent reduction in the time required to install Unguard. This result cannot be explained so easily. The container images are already available for both methods. Skaffold is likely slower because it must first check if the container images need to be rebuilt or can be reused, resulting in a few seconds lost compared to Helm.

6.7 Benefits of the Applied GitOps Principles

The new implementation has several advantages over the previous one as the following list shows:

- Using a Helm chart instead of Kustomize makes it much easier to understand how the final rendering of the Kubernetes resources is created. All the resources are in one repository and have no nested structures. Unguard is configured by obtaining various value files rather than applying multiple patches in a specific order from different repositories.
- Unguard's Helm chart simplifies the installation process significantly. Engineers and non-engineers alike can easily install Unguard without Skaffold and without having to build it themselves.
- The deployment or upgrade process is automated, reducing the risk of human error.
- Argo CD enhances the observability of the cluster's and Unguard's status by providing a dashboard.

- The availability of prebuilt container images for Unguard offers numerous benefits:
  - The images do not need to be rebuilt every time a new cluster is installed, or an existing cluster is upgraded. Instead, the images can be downloaded, significantly speeding up Unguard's installation process and reducing unnecessary resource consumption. The speedup is quantified in Section 6.6
  - The installation time now depends on the internet connection's download speed, which is advantageous for clusters in the cloud or on development machines in an enterprise environment, where internet speeds are typically adequate.
  - If images are already downloaded and cached, reinstalling Unguard takes only seconds.
  - Builds of the same version of Unguard are reproducible.
  - Since the images contain all dependencies and are hosted on the GHCR, the Docker Hub rate-limiting issue is resolved.
- The desired cluster state is declaratively defined and stored in a Git repository, providing various benefits:
  - It is immediately apparent which version of Unguard is installed, with what configuration, on which cluster.
  - The engineers need not worry about the steps required to configure and set up Unguard. They merely have to specify what the final result should look like, and Argo CD will handle its realization.
  - Code stored in a Git repository offers the benefits of a version control system, such as enhanced code quality through code reviews and pull requests, providing traceability of code changes, and fostering collaboration between engineers.

6.8 Drawbacks of the Applied GitOps Principles

By contrast, there are also a few disadvantages to this approach:

- Additional tools like Argo CD and Helm must be installed and learned by the engineers.
- Helm chart templates introduce a new syntax. For complex applications and configurations, this syntax can be hard to read.
- The GitOps architecture has increased the complexity of the system.
- Three commands as depicted in Listing 21 are necessary to install Unguard using the Helm chart.

7 Security Implications

Since Unguard intentionally includes vulnerabilities that provide access to the Kubernetes cluster, and ArgoCD is now installed with elevated privileges, new attack vectors may be present. To analyze potential threats, a threat modeling session was conducted. This chapter explains the concept of threat modeling, provides a description of the methodology used, and outlines the session's results.

7.1 What is Threat Modeling?

Threat modeling is the process of analyzing a system for weaknesses [33]. Ideally, this is an integral part of the development process and is conducted regularly to identify risks as early as possible. Threat modeling should help to understand how a system design should be changed to reduce the risk of weaknesses.

According to Adam Shostack, an expert in threat modeling, the process should address four critical questions [34]:

1. *What are we working on?*
   Understand the current state of the system and its development goals.

2. *What can go wrong?*
   With an understanding of the system, identify possible security threats.

3. *What are we going to do about it?*
   Identify steps to minimize the liability resulting from the issue identified in the previous step.

4. *Did we do a good enough job?*
   Reflect on the process and identify if the threat modeling mitigated the threat effectively. This allows the process to be refined for future iterations.

Answering these questions should help evaluate whether the threat modeling effort was successful. If these questions cannot be sufficiently answered, it may be advisable to consider an alternative methodology.

7.2 Threat Modelling Methodologies

Many threat modeling methodologies are available, all with distinct features and specific focuses [33]. They all have advantages and disadvantages, so it's wise to experiment with various approaches to determine the most appropriate one. This paper presents the STRIDE model as it was utilized in the conducted session.

7.2.1 STRIDE

STRIDE was formalized at Microsoft in 1999 by two engineers, Koren Kohnfelder and Praerit Garg, in their letter called *The Threats To Our Products* [35]. STRIDE is a mnemonic for the six categories of security threats in Table 4. The table also provides a brief explanation and example for each type.

A threat modeling session with STRIDE begins with creating a representation of the system that can help examine its characteristics. The most common way is to create a Data-Flow Diagram.

Name	Explanation	Examples
Spoofing	Pretending to be something or someone.	Replacing a dll by a malicious dll.
Tampering	Modifying something.	Injecting malicious dll into memory.
Repudiation	Denying responsibility for a action or event.	Performing action and then deleting logs.
Information Disclosure	Viewing unauthorized information.	Data breaches.
Denial of Service	Absorbing all resources and stalling operation.	Starveing a process by absorbing all resources.
Elevation of Privilege	Performing an unauthorized action.	Buffer overflow to gain higher privileges.

Table 4: The STRIDE Methodology.

7.3 Data-Flow Diagram

A Data-Flow Diagram (DFD) illustrates the system's components (elements) and their communication (data flows). It also includes trust boundaries that distinguish areas of the system with varying trust levels. Circles or rectangles typically denote *elements*, while arrows denote *communication*. Red dotted lines or boxes indicate *trust boundaries*. Figure 17 depicts the diagram, which was the result of the modeling session. It displays the Kubernetes cluster with Unguard and Argo CD. On the right-hand side are the additional components introduced through the implemented GitOps approach.

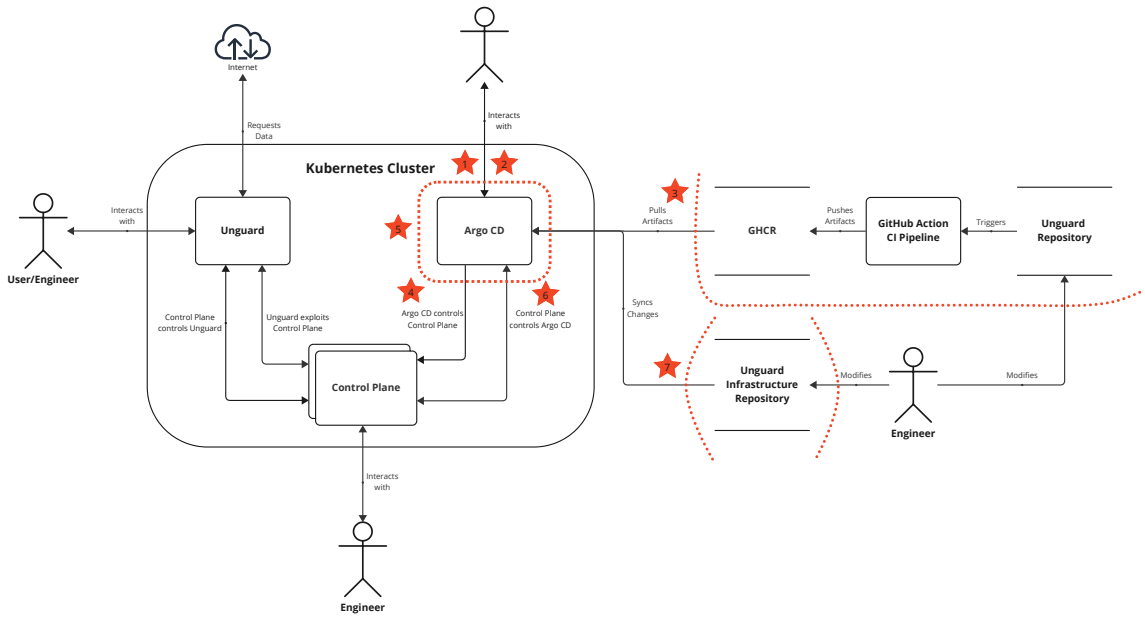


Figure 17: Data-Flow Diagram of Unguard.

7.4 Applying STRIDE

The next stage involved reviewing the diagram and, for every trust boundary crossing, developing possible attack scenarios for each STRIDE component. The diagram's numbered stars indicate the identified threats and their corresponding trust boundaries. Table 5 lists the threats and potential countermeasures identified during this process. Since the emphasis was on analyzing the security implications introduced by GitOps, the identified threats focused on the new components.

	Name	Description	Possible Countermeasures
1	Default Argo CD credentials	The default credentails of Argo CD can be easily accesed.	Delete default credentials.
2	Impersonate other Argo User/admin	JWT tokens are used, which can potentially be forged.	Log actions from users.
3	Compromise image registry	An attacker can push a compromised image to the registry and these changes are deployed in cluster.	Restrict write access to registry.
4	Expose secrets	The secrets are not encrypted, only base64 encoded.	Encrypt the secrets.
5	Lateral movement from other app	If another app in the same cluster is compromised, the Argo CD pods are directly reachable.	Create network policy to prohibit ingress from other Namespaces.
6	Compromised Service-Account/ Role-Binding (of other apps) can modify Argo CD	ArgoCD can be misconfigured arbitrarily.	Audit Kubernetes API requests.
7	Compromised infrastructure repository can modify Argo CD	ArgoCD can be misconfigured arbitrarily.	Require/audit pull requests.

Table 5: Identified Threats.

7.5 Learnings

This session, in combination with the research for this thesis, was educational as it allowed us to gain a fundamental understanding of the concepts and procedures of threat modeling. It is beneficial for future work because it highlights the importance of security considerations from the very beginning of the whole software development lifecycle. However, since it was the first session for most participants, the findings for Unguard were not particularly sound but rather an opportunity to play around and to get familiar with the concepts.

The STRIDE methodology chosen is not ideally suited for security novices, as it is required to understand what can be a threat and how that threat, if exploited, becomes a vulnerability [33]. For example, one needs to know what spoofing is and what and how something can be spoofed. Other methodologies might be better suited for beginners.

Looking back to the four questions in Section 7.1, we succeeded in answering the first one: *What are we working on?* After the session, we gained a better understanding of Unguard and its development goals and, hence, a better idea of what we are working with. The second and third questions could not be answered sufficiently, as most participants were security novices and lacked the necessary knowledge to use the STRIDE methodology. Finally, answering the fourth question (*Did we do a good enough job?*) clarified that more extensive security-related knowledge is needed to use the STRIDE methodology effectively. Alternatively, more beginner-friendly methods could be used in future work. Only with these alterations it would be possible to do a good enough job.

8 Limitations and Future Work

Section 6 outlines the process of implementing the GitOps principles in Unguard. However, the implemented solution has limitations that require future work.

8.1 Automatically Update Chart Version

The GitOps principles state that any changes made to the source code should be automatically applied to the application. However, for Unguard, one step has not been implemented to meet this requirement. After the release of a new version of Unguard, the infrastructure repository must still be adapted manually to the updated version. To implement this principle in Unguard, this work suggests two potential solutions:

- First, this could be implemented in the CI pipeline. After pushing the artifacts, an automated pull request could be opened in the infrastructure repository to update the version. However, accessing and creating a PR in the private infrastructure repository would require significant additional implementation effort.
- The second solution involves using Renovate[25], a tool designed to identify and suggest updated artifacts automatically [36]. This option would be a perfect fit, as it can be installed as a GitHub app for the private infrastructure repository and requires minimal setup. The drawback of this method is that it would necessitate an additional tool.

8.2 Implement Sophisticated Helm Test

The Helm test implemented in this work is rudimentary since it only checks whether the frontend is reachable. This does not ensure that this or the other services work as expected. A better solution would be to use the already existing *user-simulator* as a testing framework. This service performs all the tasks that are available in Unguard and is ideal to ensure Unguard's functionality. This approach was implemented in the Helm test's first iteration, but problems were encountered. For a helm test to fail, the container must exit with a non-zero exit code. The user simulator consistently terminated successfully despite failed tasks. Further work is required to resolve this issue. In the meantime, the solution presented in this paper has been developed.

8.3 Dynamic Kubernetes Resource Names

Unguard's Kubernetes resource should match the Helm release name chosen during the installation of Unguard. Currently, this is not the case. All resources have their names hard-coded. It is possible to include this functionality using the templating language.

The work presented in this thesis provides the benefits of GitOps to Unguard.

[25] https://docs.renovatebot.com/

9 Conclusions

This thesis focused on applying GitOps to the cloud-native application Unguard, which is an intentionally insecure microblogging platform that serves as a testing ground for Dynatrace.

Prior to adopting GitOps, deploying and upgrading Unguard was a tedious and error-prone manual process that utilized Kustomize and Skaffold. Each time engineers installed or updated Unguard, they had to connect to the correct cluster and manually start the building and deployment process with Skaffold. Furthermore, it was necessary to provide Skaffold with the appropriate cluster configuration, which was a common cause of errors. Skaffold built all images from scratch for every installation, resulting in unnecessary resource expenditure. Kustomize was used as a configuration management tool; however, Unguard implemented a nested Kustomize structure spanning multiple repositories, which made it challenging to keep an overview of the final Kubernetes manifest's rendering.

GitOps, on the other hand, relies on automation to synchronize the Kubernetes infrastructure with a defined state in a Git repository, using Git as the single source of truth. In its most extreme form, the GitOps model even prohibits manual changes to the infrastructure. Therefore, instead of making manual changes, an engineer adjusts the configuration files that represent the intended state which are then automatically applied to the cluster by software agents. Using Git as a version control system for both application and infrastructure code improves traceability, collaboration, auditability, and overall code quality.

To apply GitOps principles in Unguard, a Helm chart was developed as a prerequisite, which substantially simplifies the installation process. Both engineers and non-engineers can install Unguard without Skaffold and without building Unguard themselves since the chart uses pre-built container images for Unguard's services. This eliminates the need to rebuild the images during every installation, saving a significant amount of time. A new Git repository containing the defined state of Unguard's infrastructure was created, as prescribed by GitOps principles. The actual state is automatically synchronized with the desired state by the GitOps tool, Argo CD.

In conclusion, Unguard now predominantly adheres to GitOps principles. The implemented solutions have facilitated the developers' workflow and reduces the risk for human error. All of the implemented changes described in this thesis can be reviewed in the corresponding pull request[26] in Unguard's repository as well as in the newly created infrastructure repository[27].

[26] `https://github.com/dynatrace-oss/unguard/pull/47`
[27] `https://github.com/MfCrizz/unguard-infra/tree/main`

List of Figures

Listings

List of Tables

References

1. Blinowski, G. J., Ojdowska, A. & Przybylek, A. Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation. *IEEE Access* **10,** 20357–20374. https://doi.org/10.1109/ACCESS.2022.3152803 (2022).
2. Harris, C. *Microservices vs. monolithic architecture* https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith. (accessed: 29.07.2023).
3. Luksa, M. *Kubernetes in Action* ISBN: 9781617293726. https://www.manning.com/books/kubernetes-in-action (Manning, 2018).
4. Lutkevich, B. *DEFINITION cloud-native application* https://www.techtarget.com/searchcloudcomputing/definition/cloud-native-application. (accessed: 19.08.2023).
5. Ibryam, B. & Huß, R. *Kubernetes Patterns: Reusable Elements for Designing Cloud-Native Applications* ISBN: 9781492050230. https://www.oreilly.com/library/view/kubernetes-patterns/9781492050278/ (O'Reilly Media, 2019).
6. *Use containers to Build, Share and Run your applications* https://www.docker.com/resources/what-container/. (accessed: 19.08.2023).
7. Velayudhan, N. *What are container runtimes?* https://opensource.com/article/21/9/container-runtimes. (accessed: 19.08.2023).
8. *What is container orchestration?* https://cloud.google.com/discover/what-is-container-orchestration. (accessed: 19.08.2023).
9. *What is container orchestration?* https://www.redhat.com/en/topics/containers/what-is-container-orchestration. (accessed: 19.08.2023).
10. *What is container orchestration?* https://www.vmware.com/topics/glossary/content/container-orchestration.html. (accessed: 19.08.2023).
11. Yuen, B., Matyushentsev, A., Ekenstam, T. & Suen, J. *GitOps and Kubernetes: Continuous Deployment with Argo CD, Jenkins X, and Flux* ISBN: 9781617297274. https://www.manning.com/books/gitops-and-kubernetes (Manning, 2021).
12. Kubernetes. *Kubernetes — Wikipedia, The Free Encyclopedia* [Online; accessed: 19.08.2023]. 2023. https://en.wikipedia.org/wiki/Kubernetes.
13. *Overview* https://kubernetes.io/docs/concepts/overview/. (accessed: 19.08.2023).
14. Poulton, N. *The Kubernetes Book* ISBN: 1916585000. https://leanpub.com/thekubernetesbook (Leanpub, 2021).
15. *What is the Kubernetes Control Plane?* https://www.armosec.io/glossary/kubernetes-control-plane/. (accessed: 19.08.2023).
16. Patel, A. *Kubernetes — Architecture Overview* https://medium.com/devops-mojo/kubernetes-architecture-overview-introduction-to-k8s-architecture-and-understanding-k8s-cluster-components-90e11eb34ccd. (accessed: 20.08.2023).
17. Arundel, J. & Domingus, J. *Cloud Native DevOps with Kubernetes: Building, Deploying, and Scaling Modern Applications in the Cloud* ISBN: 9781492040767. https://www.oreilly.com/library/view/cloud-native-devops/9781492040750/ (O'Reilly Media, Incorporated, 2019).
18. Authors, T. K. *Kubernetes Documentation* https://kubernetes.io/docs/home/. (accessed: 07.08.2023).

19. Burns, B., Beda, J., Hightower, K. & Safari, a. O. M. C. *Kubernetes: Up and Running* https://www.oreilly.com/library/view/kubernetes-up-and/9781491935668/ (O'Reilly Media, Incorporated, 2019).

20. Jaiswal, A. *Kustomize Tutorial: Comprehensive Guide For Beginners* https://devopscube.com/kustomize-tutorial/. (accessed: 20.08.2023).

21. *Kustomize Documentation* https://kubectl.docs.kubernetes.io/references/kustomize/kustomization/. (accessed: 20.08.2023).

22. *Helm Documentation* https://helm.sh/docs/. (accessed: 20.08.2023).

23. Choudhary, A. *Effortless Cloud-Native App Development Using Skaffold: Simplify the development and deployment of cloud-native Spring Boot applications on Kubernetes with Skaffold* ISBN: 9781801076951. https://www.packtpub.com/product/effortless-cloud-native-app-development-using-skaffold/9781801077118 (Packt Publishing, 2021).

24. *Skaffold Documentation* https://skaffold.dev/docs/. (accessed: 20.08.2023).

25. Vinto, N. & Bueno, A. *GitOps Cookbook* ISBN: 9781492097440. https://www.oreilly.com/library/view/gitops-cookbook/9781492097464/ (O'Reilly Media, 2022).

26. *Argo CD Documentation* https://argo-cd.readthedocs.io/en/stable/. (accessed: 21.08.2023).

27. *Understanding GitHub Actions* https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions. (accessed: 27.08.2023).

28. LLC, D. *Kubernetes monitoring* https://www.dynatrace.com/technologies/kubernetes-monitoring/. (accessed: 07.08.2023).

29. LLC, D. *Unguard GitHub Page README* https://github.com/dynatrace-oss/unguard#readme. (accessed: 07.08.2023).

30. LLC, D. *Unguard GitHub Page* https://github.com/dynatrace-oss/unguard. (accessed: 07.08.2023).

31. *GitOps Principles* https://opengitops.dev/. (accessed: 24.08.2023).

32. *Kubernetes Demystified: Solving Service Dependencies* https://www.alibabacloud.com/blog/kubernetes-demystified-solving-service-dependencies_594110. (accessed: 07.08.2023).

33. Tarandach, I. & Coles, M. *Threat Modeling: A Practical Guide for Developing Teams* ISBN: 9781492056553. https://www.oreilly.com/library/view/threat-modeling/9781492056546 (O'Reilly Media, Incorporated, 2020).

34. Shostack, A. *Shostack's 4 Question Frame for Threat Modeling* https://github.com/adamshostack/4QuestionFrame. (accessed: 04.08.2023).

35. Kohnfelder, L. & Garg, P. *The Threats To Our Products* https://adam.shostack.org/microsoft/The-Threats-To-Our-Products.docx. (accessed: 04.08.2023).

36. *Renovate Documentation* https://docs.renovatebot.com/. (accessed: 07.08.2023).