

University of Hertfordshire

School of Computer Science

BSc Honours in Computer Science

6WCM0029-0105-2022 - Computer Science Project

Final Report

April 2023

Author: Anais Urlichs (SID removed)

Supervised by Mike Watkins

**Designing a Kubernetes Operator to Apply
Security Chaos Engineering on Containerised
Workloads**

Abstract

Chaos Engineering is a practice that was first implemented in 2010 by Netflix to make their cloud services more resilient to failure. This was done by turning off different components of the application and infrastructure stack and monitoring the system's response. Since then, the term has become well known across the cloud computing industry alongside the development of several projects that enable the testing of infrastructure, applications, and other development resources. In 2017, the first projects were developed to expand on Chaos Engineering principles to test the security of these resources. The practice is encompassed in the term 'Security Chaos Engineering', defined by *Rinehart and Shortridge (2021)* as "the identification of security control failures through proactive experimentation to build confidence in the system's ability to defend against malicious conditions in production."

This report is divided into two parts. The first part provides further insights into the theory of Security Chaos Engineering, Kubernetes and cloud native concepts. The second part details the design and development of a Kubernetes Operator based on the concepts introduced in the first part. Kubernetes is a container orchestration tool to run and manage containerised workloads and related tooling. To test the Operator and gain further insights into the Kubernetes processes, a four-node Raspberry Pi cluster has been set up with the Kubernetes distribution micok8s. Kubernetes Operator are a type of Kubernetes application that continuously monitor specific resources inside a Kubernetes cluster. If predefined conditions are met, the Operator can perform tasks inside the cluster, including modifying existing resources.

The Operator developed for this project can be configured to apply Misconfiguration to running containerised workloads inside a Kubernetes cluster. A user, also called a cluster administrator, can specify the Kubernetes Deployment Resources that should be misconfigured by the Operator and the type of Misconfiguration that should be applied. Once the Operator has access to a Kubernetes Deployment, it will apply and manage the Misconfiguration. This will allow cluster administrators to build Security Chaos Engineering experiments following Kubernetes best practices.

Keywords: Kubernetes, Operator, Security, Chaos Engineering, Containerisation, microk8s, Raspberry Pi

Acknowledgements

Many thanks to Mike Watkins for his supervision and guidance throughout the project. His feedback and perspectives helped shape this project and significantly increased my motivation.

Additionally, I would like to thank my grandfather Bernhard Urlichs for encouraging me towards completing my Bachelor degree.

Table of Contents

Abstract.....	1
Acknowledgements.....	2
Table of Contents.....	3
List of Figures.....	5
1.0 Introduction.....	6
1.1 Problem Overview.....	6
1.2 Motivations for the Project.....	7
1.3 Main Aim and Objectives.....	7
1.4 Approach.....	8
1.5 Report Structure.....	8
2.0 Background Research and Overview of Technologies used.....	9
2.1 Containerisation, Containers and Container Registries.....	9
2.2 Kubernetes.....	10
2.4 Kubernetes Deployment.....	11
2.5 Kubernetes Operators.....	11
2.6 Golang IDE and development tools.....	12
3.0 Kubernetes cluster comparison and Raspberry Pi cluster built.....	12
3.1 Raspberry Pi Cluster vs Managed Kubernetes by Cloud Providers.....	12
3.2. Raspberry Pi Cluster Development.....	13
4.0 Chaos Engineering and Security Chaos Engineering.....	14
4.1 Chaos Engineering.....	14
4.2 Security Chaos Engineering.....	15
5.0 Introduction to kubebuilder.....	16
5.1 Choosing a development path.....	16
5.2 Kubebuilder.....	16
6.0 Operator Design.....	17
6.1 Operator Requirements.....	17
6.2 Kubernetes Resources to modify.....	17
6.3 Kubernetes CRD to implement.....	18
6.4 User Flow for Kubernetes cluster administrators.....	21
7.0 Operator Development.....	23
7.1 Initialising the operator with kubebuilder.....	23
7.2 Directory 'apis/api/v1alpha1'.....	28
7.3 Directory 'controllers/api'.....	29
7.4 Directory 'controllers/apps'.....	33
7.5 File 'main.go'.....	35
7.6 File 'Dockerfile'.....	35
7.7 File 'Makefile' and deploying the Operator to a Kubernetes cluster.....	36
7.8 Directory 'config'.....	37
7.9 Testing and Debugging during Development.....	38
8.0 Operator Usage.....	39

8.1 Define Security Chaos Experiment.....	40
8.2 Install the Operator into the Kubernetes cluster.....	40
8.3 Set the Operator CRD to introduce misconfiguration into the Deployment.....	43
9.0 Future Work.....	44
10.0 Conclusion.....	44
11.0 Bibliography.....	45
12.0 Appendices.....	48
Appendix 1 Chaos Engineering Projects.....	48

List of Figures

1. <i>Figure 1 Overview of containers and container registries</i>	<i>Page 10</i>
2. <i>Figure 2 Image of four-node Raspberry Pi Cluster</i>	<i>Page 13</i>
3. <i>Figure 3 YAML manifest of a Kubernetes Deployment</i>	<i>Page 18</i>
4. <i>Figure 4 YAML metadata of Kubernetes Deployment</i>	<i>Page 19</i>
5. <i>Figure 5 Steps that the Operator will follow</i>	<i>Page 20</i>
6. <i>Figure 6 Controller and Operator interaction</i>	<i>Page 21</i>
7. <i>Figure 7 Activity Diagram of interaction between cluster administrator and the Operator</i>	<i>Page 22</i>
8. <i>Figure 8 Boilerplate directory layout of the operator</i>	<i>Page 24</i>
9. <i>Figure 9 new directory structure with the Deployment controller</i>	<i>Page 25</i>
10. <i>Figure 10 Boilerplate Deployment Controller logic</i>	<i>Page 26</i>
11. <i>Figure 11 PROJECT file of the kubebuilder boilerplate</i>	<i>Page 27</i>
12. <i>Figure 12 Project structure with the two different controllers</i>	<i>Page 27</i>
13. <i>Figure 13 Misconfiguration Spec with the input for the Operator CRD</i>	<i>Page 28</i>
14. <i>Figure 14 Libraries in the API Controller</i>	<i>Page 30</i>
15. <i>Figure 15 If statement to verify Operator CRD is running and not being deleted</i>	<i>Page 30</i>
16. <i>Figure 16 Access Deployments with Operator annotation</i>	<i>Page 31</i>
17. <i>Figure 17 Main if statement to update Deployments</i>	<i>Page 31</i>
18. <i>Figure 18 finishReconcile function</i>	<i>Page 32</i>
19. <i>Figure 19 SetupWithManager function</i>	<i>Page 33</i>
20. <i>Figure 20 Controller accessing the Kubernetes Deployments inside the cluster</i>	<i>Page 34</i>
21. <i>Figure 21 Setting the Deployment Annotation back to true</i>	<i>Page 34</i>
22. <i>Figure 22 Controller setup logic</i>	<i>Page 35</i>
23. <i>Figure 23 Dockerfile copy logic from the source code of the controller into the container image</i>	<i>Page 36</i>
24. <i>Figure 24 Operator CRD</i>	<i>Page 38</i>
25. <i>Figure 25 VSCode, Operator running in Debug mode</i>	<i>Page 39</i>
26. <i>Figure 26 Querying the nodes of the Kubernetes cluster</i>	<i>Page 40</i>
27. <i>Figure 27 running microk8s from the main node</i>	<i>Page 40</i>
28. <i>Figure 28 Output of the 'make deploy' command</i>	<i>Page 41</i>
29. <i>Figure 29 query Kubernetes namespaces</i>	<i>Page 41</i>
30. <i>Figure 30 Kubernetes Deployment before the Operator modifies it</i>	<i>Page 42</i>
31. <i>Figure 31 Deployment after the Operator applied the changes</i>	<i>Page 43</i>
32. <i>Figure 32 Comparison of Chaos Engineering Projects</i>	<i>Page 44</i>
33. <i>Figure 33 Comparison of Chaos Engineering Projects</i>	<i>Page 48</i>

1.0 Introduction

Security Chaos Engineering is defined by *Rinehart and Shortridge (2021)* as “the identification of security control failures through proactive experimentation to build confidence in the system’s ability to defend against malicious conditions in production”. The idea of Security Chaos Engineering (SCE) is to apply the better-known principles of Chaos Engineering to proactively test and verify a system's security tooling and processes. This makes it possible to gain higher confidence in the system and discover any unknown variables that could impact either the way security issues are discovered or mitigated.

1.1 Problem Overview

In the past 10 years, the server industry has seen a clear shift towards cloud-native applications. These “refers to a set of technologies and design patterns that have become the standard for building large-scale cloud applications,” as shared by *Gannon, Barga, and Sundaresan (2017)*. With this new trend came new deployment strategies and ways to manage the application stack to utilise the resources from cloud providers more efficiently. Some of these trends include utilising Containerisation and Container Orchestration, which have been further explained in Section 2.

These technologies do not only come with innovative approaches to application management and hosting but also provide novel challenges. One of the problems of running distributed workloads at scale in remote environments, such as in the infrastructure of cloud providers, is verifying the configuration of workloads (*Wood and Pereira, 2011*).

Configuration may refer to the configuration used to set up the application in the cloud provider, such as the scale of resources required. Any configuration that is not defined according to industry best practices is called a ‘Misconfiguration’. Misconfigurations may result in security issues for companies. To identify misconfiguration before they become an issue, companies implement security scanners.

Various security scanners exist for different types of resources. These security scanners have a database with a set of best practices for different configurations. If best practices are not followed in the configuration of the resource, then the security scanner will flag it as a misconfiguration. Subsequently, a human looking at the security scan is then able to analyse further and mitigate the issue.

Usually, misconfigurations are not set up on purpose. Instead, they become introduced by accident into the deployment resources such as the application code. Thus, the business is highly dependent on its security scanners and workflows, including human processes, working as expected to mitigate security issues in a timely manner. If the security scanner does not identify misconfigurations, or the person who is looking at the scan output does not know how to respond, then the misconfigurations will not be mitigated and could be deployed to a production environment.

To verify that tools and processes work as expected requires proactive testing. Testing might include introducing misconfigurations on purpose to observe the response of the security tooling. If the misconfiguration is known, it is possible to predict how tools and processes

should respond. If the response is not aligned with expectations, it is possible to improve processes.

Misconfigurations can be introduced manually into the development stack or through automated processes and tooling. However, in the case of large-scale environments, it becomes rather difficult to introduce misconfigurations manually. If things do not go as expected, the engineers and system operators need an automated way to revert any changes that have been done for testing purposes. This is not possible if changes are introduced manually, as engineers might forget the steps that have been taken to create the misconfigurations.

At the time of writing, no open source project has been identified that is focused on Security Chaos Engineering to proactively test security tooling (*Rinehart and Shortridge, 2021*). The only open source project that has been developed for this purpose is called ChaoSlingr, which has been archived on GitHub. As shared by *Rinehart and Shortridge (2021)*, “the majority of organizations utilizing ChaoSlingr have since forked the project and constructed their own series of security chaos experiments using the framework provided by the project as a guide.” This leaves a gap in the industry for a tool that enables engineers to test their application and infrastructure configuration proactively for security issues.

1.2 Motivations for the Project

Containerisation and Container Orchestration, further explained in Section 2, are emerging technologies in the cloud computing industry. However, security expertise is rare, and many smaller companies often do not have access to experts in the field. Thus, engineers who are specialised in working with other software components will have to work on security. Security scanners, among other security tooling, can make it easier to test infrastructure components and workloads for security issues. However, these have to evolve with the infrastructure and application deployment strategies which they are supposed to test. Otherwise, security tooling will not be able to identify security issues, such as misconfiguration, in the application and infrastructure configuration.

1.3 Main Aim and Objectives

The goal of this project is to provide a Proof of Concept on how Chaos Engineering can be applied to Infrastructure Security as part of Security Chaos Engineering. A Proof of Concept has been defined by *C. Kending (2014)* as “research in the beginning stages, at the cutting edge of new applications or technologies, and is a buzzword used to mark out scientific research as potentially extendable and/or scalable.”

As part of this Proof of Concept, a tool will be developed to implement the principles of Security Chaos Engineering. This tool will introduce misconfigurations into application deployments that are running inside a Kubernetes cluster, defined in section 2.2. The application deployment hereby refers to a containerised workload that is deployed to a Kubernetes cluster. The tool has to automate the application configuration change. Additionally, users should be able to configure the misconfigurations that should be applied to the Kubernetes deployment through Kubernetes YAML manifests. YAML is defined by the official documentation as “a human-friendly, cross language, Unicode based data serialization

language designed around the common native data types of dynamic programming languages.”

1.4 Approach

The research and development of this project can be divided into three different parts. The first part was concerned with the background research around containerisation, Kubernetes clusters and workloads running on a Kubernetes cluster. In addition, the existing literature around Chaos Engineering and Security Chaos Engineering (SCE) needed to be reviewed to gain a better understanding of the landscape and existing work. It was important to understand whether projects of this sort already existed and whether they could be further enhanced to implement SCE principles or whether a new project would have to be developed. With the obtained information, a Kubernetes Operator was designed and developed. This process required further research into Software Development Kits (SDKs) that are commonly used to develop Applications on Kubernetes environments and, specifically, Kubernetes Operators. The third part of the project was concerned with the development of the Kubernetes Operator. The term SDK has been defined in the Gartner Glossary as “a set of development utilities for writing software applications, usually associated with specific environments (e.g., the Windows SDK).”

1.5 Report Structure

This report is divided into *10 sections*, excluding Bibliography and Appendices. The sections are representative of the process followed during the development of the project and have been written largely alongside the practical work done, such as developing the artefacts.

Section 1 provides the foundation of the project with information on the problem space, the goals of the project and an overview of the approach used to develop the project and technical artefacts.

Section 2 provides an overview of the background research done to gain a better understanding of the different technologies involved in the development of the project. The terms discussed are important to understand the design and implementation of the practical work in Sections 3, 6 and 7.

Section 3 details the development of a four-node Raspberry Pi Kubernetes cluster. This Kubernetes cluster is used for the testing and development of the Kubernetes Operator.

Section 4 goes into further detail on Chaos Engineering and Security Chaos Engineering.

Section 5 introduces the SDK that will be used to develop the Kubernetes Operator and the reasoning behind the decision.

The Kubernetes Operator design is described in *section 6*.

Section 7 shares the details of how the Operator design has been implemented.

Section 8 showcases the usage of the Kubernetes Operator.

Lastly, *section 9* provides an insight into future work planned to improve the Operator, and *section 10* summarises the work completed.

2.0 Background Research and Overview of Technologies used

This section provides the key information on the background research that is relevant for the development and analysis of the technical artefacts, namely the Kubernetes Operator.

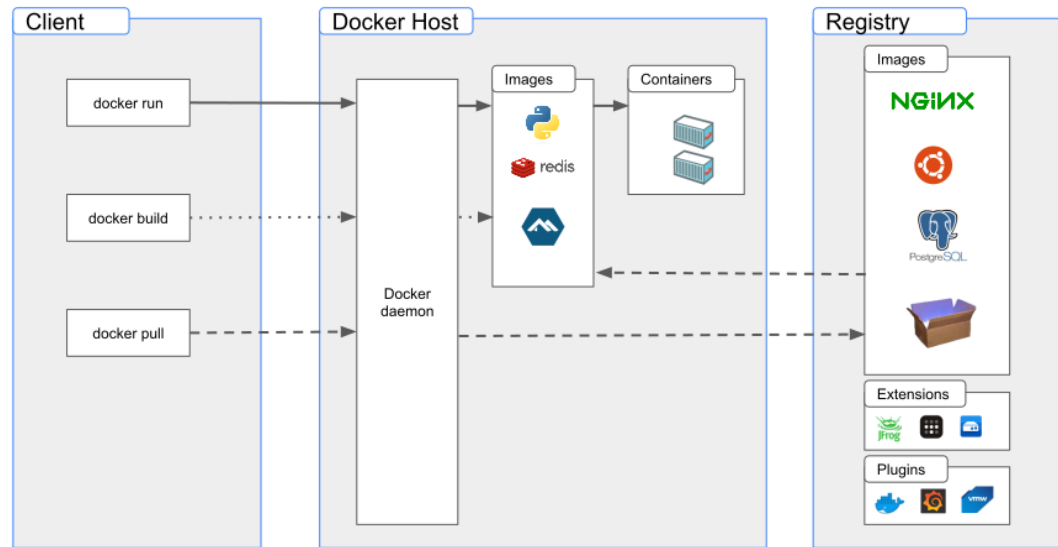
2.1 Containerisation, Containers and Container Registries

Software applications consist of several different components. These might be libraries, software packages or development frameworks. In most cases, these components require a custom installation that varies between different operating systems. Resulting, once the application is developed, it is difficult to install and run it on different machines.

The term containerisation has been defined by *C. Pahl et al. (2017)* as “a technology to virtualise applications in a lightweight way.” This is done through the use of container technologies. “A container holds packaged self-contained, ready-to-deploy parts of applications and, if necessary, middleware and business logic (in binaries and libraries) to run the applications” (*C. Pahl et al., 2017*). Containers make it possible to package application dependencies in a portable manner. These containers can then run in different environments through a container engine without the need to install any application dependencies as separate software components. Different to Virtual Machines (VMs), containers do not include the Operating System on which the application runs. Instead, multiple containers can utilise the same host Operating System (*C. Pahl et al., 2017*).

It is important to differentiate between a container image, which defines what has to go into a container, and the running container, which is an instance of the container image and includes the installed components that are necessary to run the application (*F. Paraiso et al., 2016*). To facilitate the management and distribution of container images, container images can be stored in container registries. A Client, such as installed in the command line, can be used to interact with the different components, including accessing container images from the registry and running them in a local environment as a container. Figure 1 provides an overview of the interaction between containers and container registries.

Figure 1 Overview containers and container registries



Source: Docker Overview (2023) Docker Documentation. Available at: <https://docs.docker.com/get-started/overview/> (Accessed: March 7, 2023).

Docker is one of the tools used to package and run an application as a container. However, Docker is not capable of dynamically managing containers. Meaning it cannot deploy multiple containers of the same application or set up business logic and rules for running these containers. This is the problem that container orchestration systems, such as Kubernetes, aim to solve.

2.2 Kubernetes

As described in the official Kubernetes documentation, Kubernetes is an open source project that is hosted by the Cloud Native Computing Foundation, which is part of the Linux Foundation. Claus Pahl et al. (2017) “define orchestration as constructing and continuously managing possibly distributed clusters of container-based software applications.”

Furthermore, a container orchestration system is responsible for scheduling containers across a cluster of different compute nodes, turning containers on and off, and managing components used to facilitate running the containers on the nodes, such as authentication, networking and scaling of containers.

Kubernetes can be used in several different ways. While it is possible to install Kubernetes components individually, it is not a common approach as it becomes easily very difficult to manage the Kubernetes cluster. Some companies are running Kubernetes clusters that consist of several hundred nodes. For instance, JD, one of China’s largest retailers, shared in 2020 that they are running “Kubernetes and container clusters on roughly tens of thousands of physical bare metal nodes.”

To optimise the management of large-scale Kubernetes clusters, there are several managed Kubernetes versions available. These have been divided into the following two categories to provide a better overview:

1. Cloud-hosted Kubernetes clusters, such as the options provided by Google Cloud, AWS, and Microsoft Azure. In this case, the user specifies the size of the cluster and any additional resources that are required. The Cloud Provider will then provision these resources. The user will connect to the Cluster through the Kubernetes and Cloud Provider API.
2. Independent distributions of Kubernetes that are installed on custom hardware. Several enterprises and open source communities provide their own Kubernetes distribution. These distributions can be installed similarly to a package manager on user-owned hardware. Usually, each distribution has been developed for a specific purpose, such as running on edge devices. This gives the user greater autonomy over how and where they would like to run the Kubernetes cluster.

2.4 Kubernetes Deployment

The standard Kubernetes API comes with several different extensions that reference different resource types that can be deployed to the cluster. One of these deployment types is Kubernetes Deployments. Kubernetes Deployments are part of the Kubernetes ‘apps’ API extension. Citing from the Kubernetes Documentation, “a *Deployment* provides declarative updates for Pods and ReplicaSets.” Pods, on the other hand, are resources that run one or more containers. A ReplicaSet is linked to one Pod definition through the Kubernetes Deployment and describes how many instances of the Pod should run inside of the Kubernetes cluster.

The Deployment Specification is defined through a Kubernetes YAML manifest, as are all Kubernetes resources. Users can interact with the Kubernetes API through a CLI tool called kubectl. Kubectl can be installed on the host machine or any other terminal that interacts with the Kubernetes cluster. Once the Kubernetes Deployment is installed in the cluster, a Deployment controller is responsible for provisioning the components.

2.5 Kubernetes Operators

Dobies and Wood (2020) define Kubernetes Operators in their book *Kubernetes Operators* “as a way to package, run, and maintain a Kubernetes application.” Kubernetes Operators are used to automate the management of resources inside a Kubernetes cluster. Once the Operator is deployed to the Kubernetes cluster through Kubernetes Custom Resources, it can deploy or delete application resources, modify existing applications and monitor the state of the Kubernetes cluster. To perform these tasks, Kubernetes Operators make use of Custom Resource Definitions (CRDs). Custom Resources make it possible to expand the Kubernetes API to define additional resource types that are not part of the default Kubernetes API.

As described in the Operator Whitepaper, a user should be able to describe the desired state of a resource, such as a Kubernetes Deployment, and the Operator should then be able to act upon the state request in the orchestration system. One of the key components of a Kubernetes Operator is the controller. The Operator Whitepaper defines controllers as “daemons that run inside the orchestrator like any other but connect to the underlying API and provide automation of common or repetitive tasks.” To know when to perform these tasks, the controller runs a continuous loop to monitor the state of the resources that the operator is responsible for managing inside the cluster. The process of running the continuous loop is

called a reconciliation loop. If new resources are deployed, or existing resources are changed, the Operator can then perform state updates on those resources.

2.6 Golang IDE and development tools

To develop this project, Visual Studio Code (VSCode) has been used with the additional extensions installed for Golang, Docker, and Kubernetes. These extensions make it easier to run debugging programs for the code that help analyse the code and potential errors. VSCode has been installed on a MacBook Pro. The development of the artefacts has been done locally. Furthermore, the artefacts that have been developed were tracked through Git version control and published to a public GitHub repository. Git is defined on the website as “a free and open source distributed version control system.”

The GitHub repository for the technical artefacts can be found under the following link:

<https://github.com/AnaïsUrlichs/security-controller>

The project structure explained and used in section 7 and 8 is based on the code in the GitHub repository.

3.0 Kubernetes cluster comparison and Raspberry Pi cluster built

Raspberry Pis are small, single-board computers, which make it possible to run application-specific logic in different environments, as described in the Raspberry Pi documentation. For distributed computing, and specifically, container orchestration with Kubernetes, Raspberry Pis are commonly used to develop applications in small environments or for custom home-clusters. Home-clusters or “homelabs” are Kubernetes clusters that run on small-scale hardware resources, such as Raspberry Pis, and are usually used for experimentation and development purposes rather than commercial reasons.

This section provides further information on why the decision has been made to develop a Raspberry Pi Kubernetes Cluster over utilising the Kubernetes Services of a Cloud Provider, as well as how the Raspberry Pi cluster has been developed.

3.1 Raspberry Pi Cluster vs Managed Kubernetes by Cloud Providers

It is possible to use local Kubernetes clusters that run on the local machine, such as the laptop used to develop the artefacts of the project. However, in this case, the Kubernetes cluster stops running when the laptop is turned off. It was necessary to have access to a Kubernetes cluster that can run for several days to test the functionality of the Operator. Thus, the decision had to be made to run a Kubernetes cluster on custom hardware or to pay for a managed Kubernetes cluster by a cloud provider.

Vendors can decide how they would like to customise their Kubernetes distribution. This results in Vendor-specific application deployments (*E. Truyen et al., 2020*). In comparison, Raspberry Pis make it possible to install any open source distribution to run the Kubernetes cluster. The two most common distributions used for edge computing and small scale infrastructure, such as Raspberry Pis, are microk8s and K3s (*Böhm and Wirtz, 2021*). Both

bring better performance benefits when run on small devices (*Kjorveziroski and Filiposka, 2022*). Since these Kubernetes distributions are designed to run on small-scale, custom hardware, they do not include additional, vendor-specific components.

3.2. Raspberry Pi Cluster Development

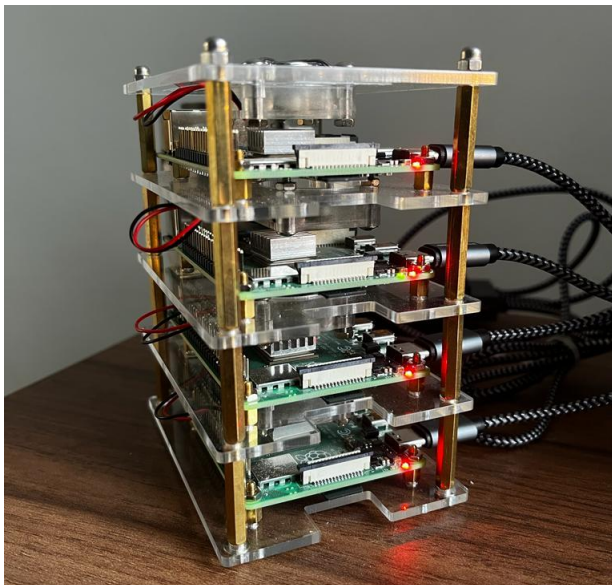
The following components have been purchased to build the Raspberry Pi cluster:

- 4 times: Raspberry Pi 4 Model B
- 1 Raspberry Pi 4 Model B Case
- 4 times: Power cable USB Type C
- 4 times: SanDisk Ultra microSDHC memory card
- USB-C power charging hub

Two tutorials from the official microk8s documentation were used to set up the Raspberry Pi cluster. Microk8s is developed by Canonical, which is also the publisher of Ubuntu, a Linux-based Operating System (*Gkrekos et al., 2019*). The first tutorial describes how to set up a Ubuntu Server on the Raspberry Pi Cluster by installing Ubuntu on the SD cards. The second tutorial walks the user through installing microk8s on the Raspberry Pis and connecting the individual Raspberry Pi compute nodes with each other by connecting the cluster to the local network over the local router.

Figure 2 provides an image of the Raspberry Pi cluster. The cluster consists of four Raspberry Pis that are connected over the host network. Each Raspberry Pi has a microSD card installed that runs Ubuntu and the microk8s code for Kubernetes. The Raspberry Pis are put together in a case with fans so that the Pis do not overheat. Furthermore, each Raspberry Pi has been connected through a USB-C cable to a USB-C power charging hub (not depicted in the photo in Figure 2).

Figure 2 Image of four-node Raspberry Pi Cluster



Each Kubernetes cluster consists of one main node and one or several worker nodes. In one-node clusters, the main node can also provide the functionality of the worker node. Once the Kubernetes cluster is set up on the Raspberry Pis, it is possible to ssh into the main node and then copy the Kubeconfig from the main node and paste it to the local machine. As described in an article by UCL, ssh makes it possible for two computers to communicate securely.

The Kubeconfig file contains information “about clusters, users, namespaces, and authentication mechanisms” as detailed in the official Kubernetes Documentation. Once the Kubeconfig of a Kubernetes cluster is available, kubectl can use it to connect to “communicate with the API server of a cluster.”

4.0 Chaos Engineering and Security Chaos Engineering

The main focus of this thesis project is to apply Chaos Engineering principles on Kubernetes cluster security as part of Security Chaos Engineering and to develop a Kubernetes Operator in accordance. This section provides an introduction to both Chaos Engineering and Security Chaos Engineering to showcase in a later section how the design of the Kubernetes operator aids the implementation of Security Chaos Engineering.

A characteristic that both Chaos Engineering and Security Chaos Engineering have in common is their aim to make systems more resilient through Chaos Engineering experiments. According to *Kazuo Furuta (2015)* “Resilience is the intrinsic ability of a system to adjust its functioning prior to, during, or following changes and disturbances so that it can sustain required operations under both expected and unexpected conditions.” Chaos Engineering experiences are built around a hypothesis that will be tested on the system (*Basiri et al., 2016*).

4.1 Chaos Engineering

Chaos Engineering has been defined in the ‘Principles of Chaos Engineering’ as “the discipline of experimenting on a system in order to build confidence in the system’s capability to withstand turbulent conditions in production”. A lot of Kubernetes-related projects rely on configuration and installation processes, which are highly dependent on the custom environment that they will be installed into. As specified by *Dart et al. (1987)* “Environment refers to the collection of hardware and software tools a system developer uses to build software systems.” Even if the tooling is thoroughly tested in staging environments that imitate the production environment, it is difficult for developers to predict how the tools will behave once installed into large scale enterprise environments. The production environment refers to the infrastructure and workload that is made accessible to users and customers of a business or project. In comparison, the staging environment serves developers to test their software in an environment where it is possible to have more control over external variables, such as the number of user requests on the application. Most tools have to be configured to interact with other tools and real customer data to become more useful.

The first tool that has been created to implement Chaos Engineering in large-scale systems is Chaos Monkey. Chaos Monkey was developed in 2010 by the engineering team at Netflix to ensure that the loss of compute instances hosted through Amazon Web Services would not compromise its streaming service. As noted by *Basiri et al. (2016)*, Chaos Monkey “randomly selects virtual machine instances that host their production services and terminates them.” This made it possible to analyse how different services respond to outages and design systems that can withstand partial failure. The source code for Chaos Monkey was made available in a public GitHub repository in 2012. Since then, several more projects have been released. The projects that are open source and under the governance of the Cloud Native Computing Foundation include Chaos Mesh, Litmus Chaos, and ChaosBlade. Appendix 1 provides a comparison with additional details on the different projects.

4.2 Security Chaos Engineering

According to *Rinehart and Shortridge (2021)*, the two main principles of Security Chaos Engineering are to expect system failure to happen in one way or another, and the second part is to prepare accordingly to be able to mitigate the issues as effective as possible. Failure hereby refers to any component of the application, infrastructure and workloads, as well as human processes, that might not work as expected. However, it is only possible to know adequate responses to different failure types by building an understanding of what those might be. Every part of the infrastructure and application stack should be tested for failure. Depending on the situation, the failure will have different causes. These can only be identified if system failure is triggered and observed.

In theory, the process of implementing Security Chaos Engineering is the following. First, the system’s state has to be documented during best-case scenarios, answering the question of ‘how the system should behave if nothing were to go wrong.’ Once the healthy state of the system is known, it is possible to formulate a hypothesis of different scenarios that could happen if specific aspects of the application or infrastructure change. Once the hypothesis has been defined, it is possible to verify the assumption by introducing errors into the system. In the case of a Kubernetes Deployment, errors can be introduced by changing the YAML configuration of the Kubernetes Deployment.

Security Chaos Engineering is not a new term. Work that has used the same term ‘Security Chaos Engineering’ was already released in 2017. The most well-known project to implement Security Chaos Engineering was ChaoSlingr. ChaoSlingr “is a security experiment and reporting framework originally created by a team at UnitedHealth Group led by Aaron Rinehart” (*Rinehart and Shortridge, 2021*). The tool itself has been archived and the GitHub repository hosting the source code does not provide further information on the reasons. However, one of the core maintainers mentioned in the same GitHub repository, Aaron Rinehart, is one of the co-authors of ‘Security Chaos Engineering’, a book that will be published by O’Reilly in April 2023. At the time of writing this report, no well-known, open source Security Chaos Engineering project has been identified.

5.0 Introduction to kubebuilder

This section will provide an overview of the two different development options that have been evaluated and then further elaborate on why a custom Operator will be developed with the use of a Kubernetes Operator SDK called kubebuilder.

5.1 Choosing a development path

Two different options were evaluated to build the software artefacts for this project. The first option would have been to use the Software Development Kit (SDK) of an existing Chaos Engineering-focused project. Appendix 1 showcases the different Chaos Engineering projects available and highlights which ones provide an SDK. Depending on the SDK, the project could then be extended from Chaos Engineering focused experiments into Security Chaos Engineering experiments. The benefit of this approach is that it would have been possible to build upon a technology and open source software project that has been developed for several years and has a large open source community around it. For instance, one of the projects called Litmus Chaos has 3600 stars on GitHub. GitHub stars are a common indicator used to gauge the popularity of a project. A larger developer community can provide higher confidence in the software.

However, in this approach, a Chaos Engineering project would have been used for something other than its intended purpose, and the development of the security-focused use cases would have been limited within the SDK of the project. Thus, custom designs and complex scenarios might not be possible to add to the project in the future. This approach has not been further evaluated due to time constraints for developing the software artefact. A Security Chaos Experiment would have to be largely implemented into the software of an existing Chaos Engineering project to evaluate whether this approach is feasible.

The second option entailed building a custom Kubernetes Operator. Kubernetes Operators are usually built with Golang and the use of an Operator SDK (*Dobies and Wood, 2020*). Golang is a popular language for Kubernetes-focused projects, as shared in the Operator Whitepaper. The Operator can access Kubernetes resources and modify those through custom functions. The two main open source projects available to build Kubernetes Operators are kubebuilder and OperatorSDK. The latter uses kubebuilder as a framework. Given that there are more resources available for using kubebuilder directly, the decision was made to use kubebuilder instead of OperatorSDK.

5.2 Kubebuilder

As described in the official GitHub repository of Kuberbuilder, it “is a framework for building Kubernetes APIs using custom resource definitions (CRDs).” Kubebuilder can either be used to build custom operators or as a framework for other projects. Kubebuilder leverages two other libraries that are used to build custom operators. The libraries are called ‘controller-runtime’ and ‘controller-tools’. Both libraries encompass a set of further libraries that are used to aid the development of Kubernetes Operators in kubebuilder and OperatorSDK. These libraries help abstract the underlying logic needed to access and modify Kubernetes resources, such as Kubernetes Deployments.

Kubebuilder provides an abstraction to make Operator Development easier. The main resource for kubebuilder is the kubebuilder book, which serves as its documentation. However, the kubebuilder community has also developed several other tutorials that showcase how to use kubebuilder and optimise the development of Kubernetes Operators through its functionality.

6.0 Operator Design

This section details the requirements and the design of the Kubernetes Operator. Software requirements specify what should be built and which problems it should solve (*Stuart, 1995*). This section will focus on Functional Requirements, which refer to the functionality and features of the system.

Note that from this section onwards, whenever the term ‘Operator’ is used, it will reference not the type of Kubernetes application specified in section 2.5 of this report but the Kubernetes Operator that has been built for this project and its artefacts.

6.1 Operator Requirements

Functional Requirements of the Operator from a user perspective

A user should be able to choose:

1. The Kubernetes Deployment Resource that should be modified by the operator, and
2. The configuration of the Kubernetes Deployment that should be changed, as well as how it should be changed.

Functional Requirements of the Operator

The Operator has to:

1. Run the controller loop inside of the Kubernetes cluster that it is installed to for an indefinite time duration or until the cluster is deleted or the Kubernetes Operator is deleted directly.
2. Accept a Kubernetes Custom Resource Definition (CRD) that tells it which configuration to change on the Kubernetes Deployment. This will be called the Operator CRD.
3. If no Operator CRD is installed inside the Kubernetes cluster, then the Kubernetes Operator should not take any actions towards modifying Kubernetes Deployments.
4. Know which resources it is allowed to access and change the configuration.
5. If the Operator is no longer supposed to access and modify the resource, it should leave it as is in the Kubernetes cluster.

6.2 Kubernetes Resources to modify

The main purpose of the Operator is to introduce misconfigurations on selected Kubernetes Deployments that are running inside the Kubernetes cluster. Kubernetes Pods define the configuration to run a containerised application. By misconfiguring the Kubernetes Pod

specification, it is possible to misconfigure the way the containerised application is supposed to run. This makes Kubernetes Pods a good first use case for the Operator to introduce misconfiguration. However, it is better practice to define Pods through Kubernetes Deployments. Kubernetes Deployments do not only make it possible to configure the running Pods but also the ReplicaSet that defines how many instances of the Pod should be running inside the cluster. Figure 3 is an example YAML manifest for a Kubernetes Deployment.

Figure 3 YAML manifest of a Kubernetes Deployment

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5   namespace: demo
6   annotations:
7     anaisurl.com/misconfiguration: "true"
8   labels:
9     app: nginx
10 spec:
11   replicas: 2
12   selector:
13     matchLabels:
14       app: nginx
15   template:
16     metadata:
17       labels:
18         app: nginx
19     spec:
20       containers:
21       - name: nginx
22         image: nginx:1.23
23         resources:
24           limits:
25             memory: "128Mi"
26             cpu: "500m"
27           requests:
28             memory: "64Mi"
29             cpu: "250m"
30       securityContext:
31         allowPrivilegeEscalation: false
32         runAsNonRoot: false
33         readOnlyRootFilesystem: false
34       ports:
35       - containerPort: 80
```

This YAML manifest has been used throughout the development of the Kubernetes Operator. It is based on the Deployment YAML manifest provided in the official Kubernetes documentation.

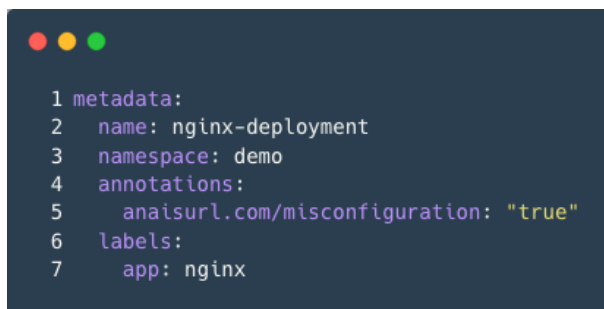
6.3 Kubernetes CRD to implement

Once the Operator is running inside of the Kubernetes cluster, it will look for an implementation of its Kubernetes Custom Resource (CRD). The Operator CRD make it possible for users to specify which configuration should be changed on the Kubernetes

Deployment. If no CRD is installed inside of the Kubernetes Cluster, then the Operator should not modify any resources.

Once the Operator is provided with the Operator CRD, it has to know which Kubernetes Deployments to change. Users can define additional metadata, including labels and annotations in the form of 'key:value' pairs in the Deployment YAML manifest. This information is not used for anything but to allow other applications to discover the Deployment, connect to it, or modify configuration inside the cluster. Figure 4 provides an example of the metadata in a Kubernetes Deployment.

Figure 4 YAML metadata of Kubernetes Deployment



```
1 metadata:
2   name: nginx-deployment
3   namespace: demo
4   annotations:
5     anaisurl.com/misconfiguration: "true"
6   labels:
7     app: nginx
```

The Operator will know which Kubernetes Deployment it is supposed to modify if it contains the following annotation:

```
anaisurl.com/misconfiguration: "true"
```

Once the Kubernetes Operator is installed inside the Kubernetes cluster, it will go through the following steps:

1. Check whether it can find an Operator CRD that tells it which modifications to make on the Kubernetes Deployment. If it is not provided with a CRD, it will not look for a Kubernetes Deployment as it does not know which misconfiguration to apply.
2. Access all Deployments through the Kubernetes API that are running inside the cluster.
3. Check which Deployments contain the annotation needed to make modifications. If no Kubernetes Deployments are identified, the Operator will not apply the misconfigurations specified in the Operator CRD to any Deployment.
4. If the Operator identifies Deployments that contain the annotation, it will add them to a separate list of Deployments.
5. If the list is not empty, the Operator will iterate through the new list of Deployments and make modifications to each Deployment in accordance with the misconfiguration provided in the Operator CRD.
6. Lastly, it will set the value of 'anaisurl.com/misconfiguration' to false. This will prevent the Operator controller loop from trying to access the same Deployment again. The Operator CRD might be modified, which does not mean that the Operator should apply the new misconfiguration to the Deployment that has already been changed.

A separate Kubernetes Controller will run alongside the Operator. Both the Kubernetes Operator and the additional Controller run a reconciliation loop inside the Kubernetes cluster. If the Kubernetes Controller identifies any resources that has the annotation set to 'false' it will set it back to 'true' after 10 hours. This way, the Kubernetes Operator is able to reaccess the resource and make further configuration changes.

10 hours is the default duration, after which the Operator will run again. The Operator will run every 10 hours. While it is possible to change the duration when the Operator runs, it is not recommended by the kubebuilder project. Changing the time for the controller loop is a more difficult process that can result in issues with the controller libraries that kubebuilder implements.

Figure 5 details the different steps described above. Figure 6 showcases the interaction between the main Kubernetes Operator and the controller that runs alongside the Operator to change the value of the annotation.

Figure 5 Steps that the Operator will follow

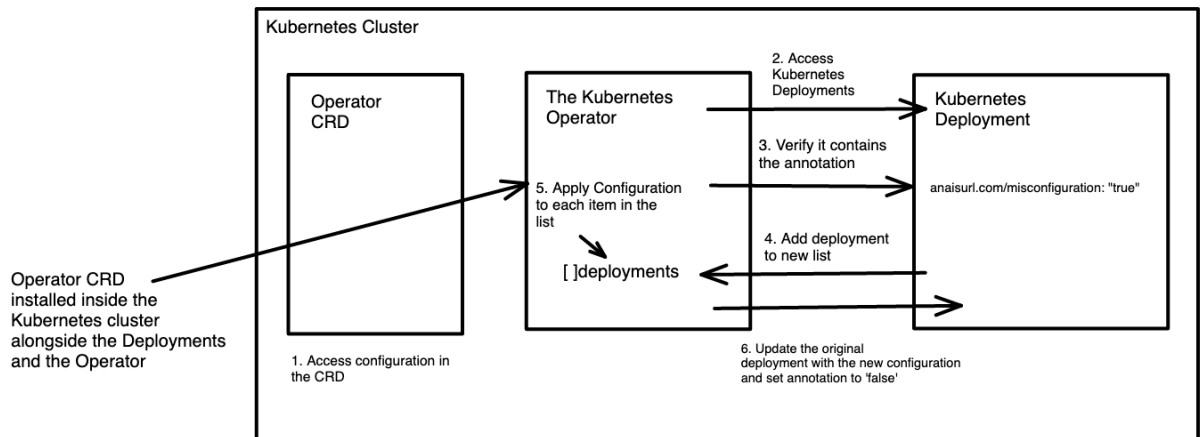
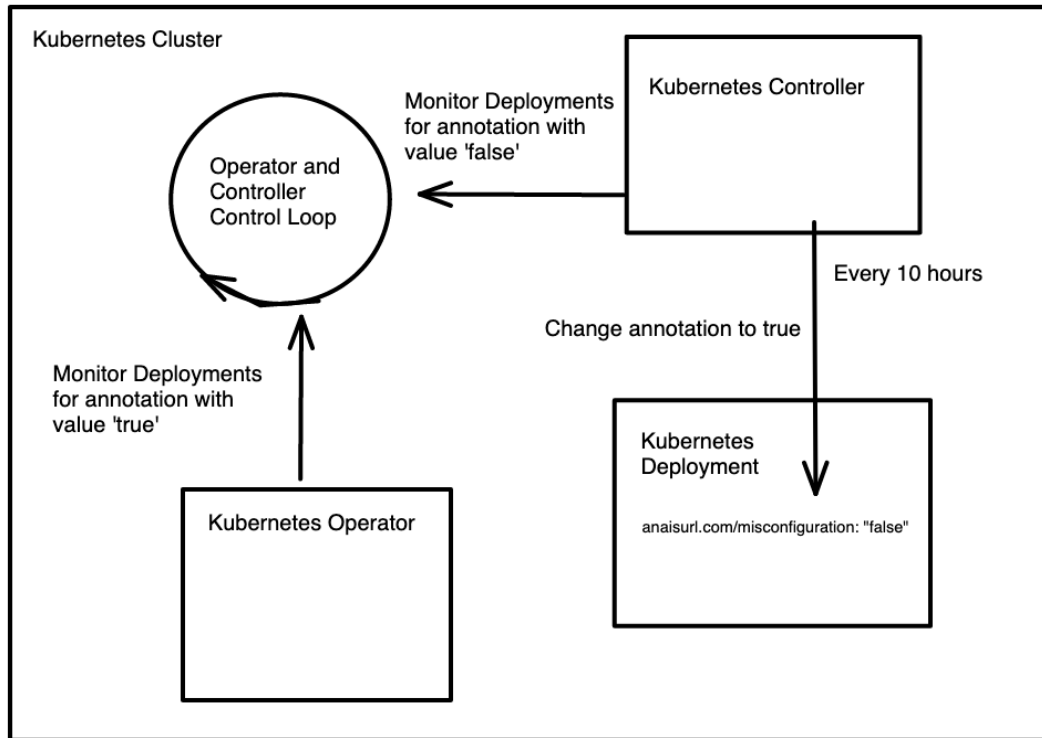


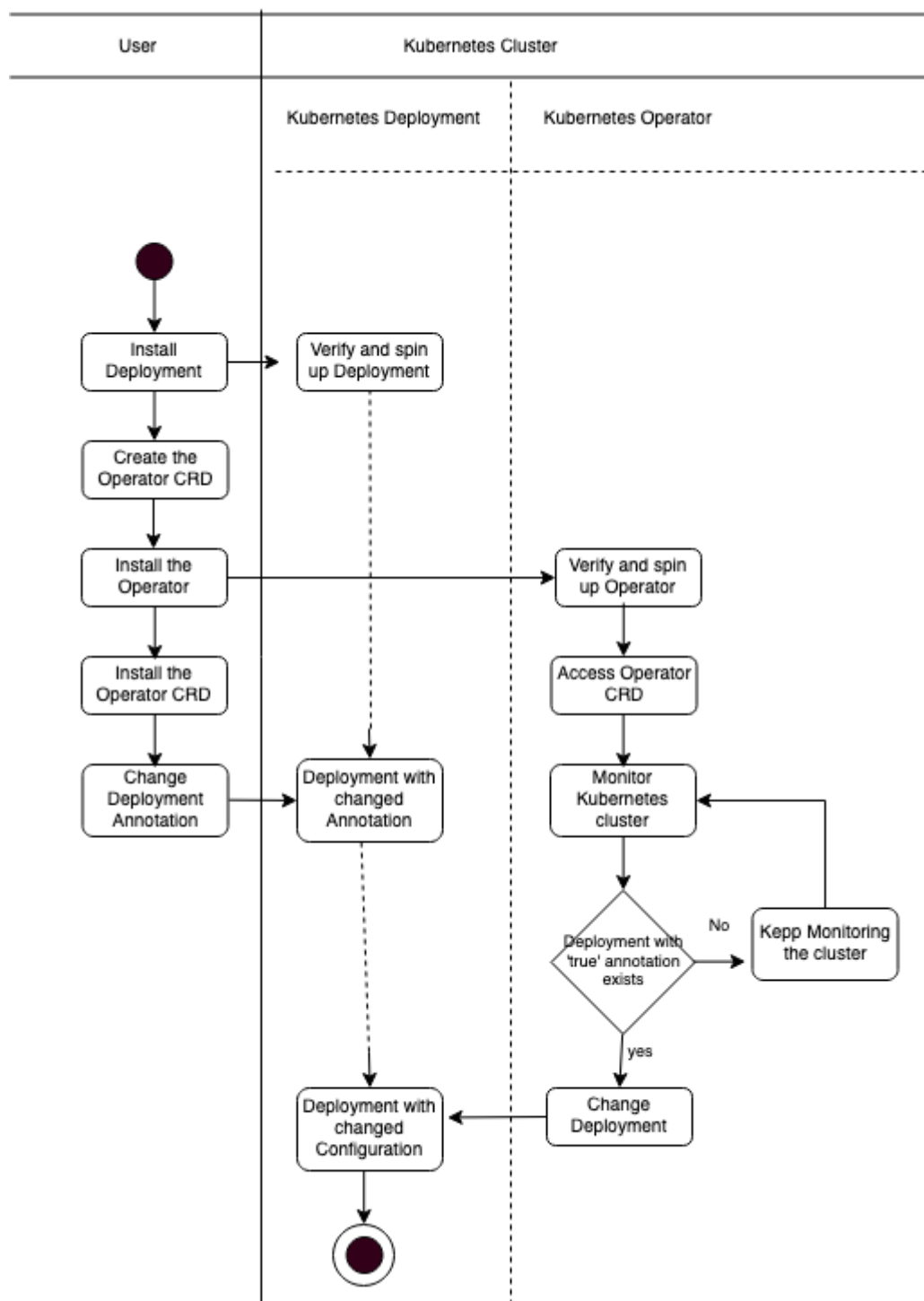
Figure 6 Controller and Operator interaction



6.4 User Flow for Kubernetes cluster administrators

A cluster administrator has the authorisation to access the Kubernetes cluster and can make changes to the Kubernetes resources installed inside the cluster. The activity diagram in Figure 7 provides an overview of how a cluster administrator is supposed to interact with the Operator.

Figure 7 Activity Diagram of interaction between cluster administrator and the Operator



First, the cluster administrator will decide which configuration to set in the Operator CRD. This is based on the misconfiguration that the cluster administrator would like to introduce to the Deployment. Next, they will install the Operator inside the cluster. Once the Operator is installed, the Operator CRD can now be installed to the Kubernetes cluster. At this point, the Operator does not have any Kubernetes Deployments that it is supposed to modify. Once the cluster administrator would like any Deployments to be modified by the Operator, they can add the annotation from section 6.3 to one of their existing Deployments or create a new Kubernetes Deployment with this annotation. Once the resources are installed inside of the

cluster, the administrator should then monitor their security tooling, such as security scanners, to report on the new misconfiguration inside of the cluster. In smaller Kubernetes clusters, the Deployment and other resources can also be monitored manually without additional scanners to see how the Deployment resources are changing.

7.0 Operator Development

This section describes the process followed to set up the Kubernetes Operator through kubebuilder as well as the key components of the Operator that have been developed to implement the functionality described in section 6.

7.1 Initialising the operator with kubebuilder

Setting up the project with kubebuilder required the installation of the kubebuilder CLI and of a Kubernetes tool called Kustomize. Kustomize, as described on their website, “introduces a template-free way to customize application configuration that simplifies the use of off-the-shelf applications.” This tool is used by kubebuilder to make changes to the Kubernetes manifests that are provided upon initialising the project. Both projects have been installed through Homebrew in the local development environment. Homebrew is a package manager for MacOS and Linux-based operating systems.

Once the kubebuilder CLI has been installed in the local development environment, it was used to create the following resources:

1. The boilerplate kubebuilder project.
2. A Deployment controller.
3. An API controller

Boilerplate kubebuilder project

Kubebuilder makes it possible to generate a template project. This project provides a simplified example of a Kubernetes Operator with the file structure required to get started.

Since kubebuilder uses Golang to build the project, the repository in which the boilerplate should be installed has to be initialised. This is done with the following command:

```
> go mod init <link to remote repository>
```

The remote repository was set to the following GitHub repository:

github.com/AnaisUrlichs/security-controller.git

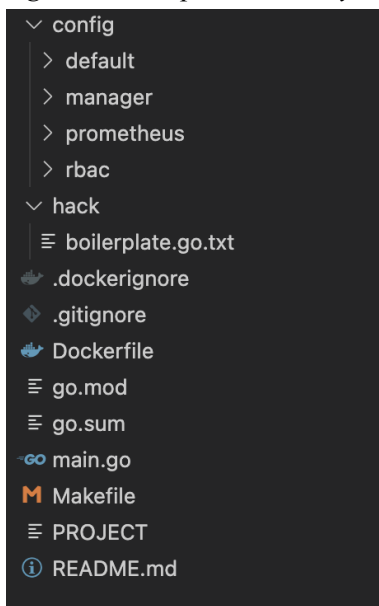
Next, the template for the Operator has been created with the following kubebuilder command:

```
> kubebuilder init --owner AnaisUrlichs --domain  
core.anaisurl.com
```


- The ‘owner’ flag specifies who is owning this repository. Ideally, this is going to be the GitHub handle if the project is made available on GitHub. If people only have access to the owner name, they will be able to find the public repository under the same-named GitHub handle.
- The ‘domain’ flag specifies the API domain used as the prefix for the operator API resources. It is worth noting that this flag might be confused with an actual webdomain and no webdomain is required to set the ‘domain’ flag.

After completing the first steps describe above, the project structure will be as shown Figure 8. Figure 8 is a screenshot of the repository from VSCode.

Figure 8 Boilerplate directory layout of the operator



The ‘config’ directory contains boilerplate Kubernetes manifests. These manifests currently do not correlate to any Operator logic. The ‘Dockerfile’ provides the basic logic for packaging the operator into a container image. Lastly, the ‘Makefile’ contains commands on creating the Kubernetes resources from the ‘config’ directory through Kustomize so that they can be installed inside of the Kubernetes cluster.

Deployment controller

It is best practice, as defined in the Operator Whitepaper, for each controller to correlate to one task only in the Kubernetes Operator. However, an Operator can have multiple controllers. To modify Kubernetes Deployments independent of the main code logic, a Deployment controller was set up with the following command:

```
> kubebuilder create api --group apps --version v1 --kind
Deployment
```

The flags reference the information on Kubernetes Deployments in the Kubernetes API. It is possible to access this information through kubectl by running the following command:

```
> kubectl api-resources
```

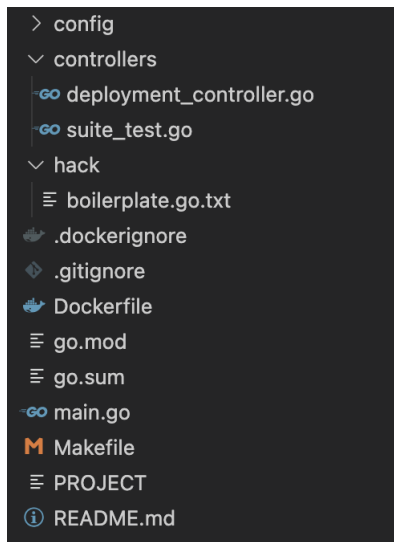
This will return a list of all the built-in Kubernetes API types. The important information is provided next to the Deployment type:

```
> deployments                                deploy      apps/v1
```

Once the command is run, kubebuilder will ask the developer two questions through the command line. The first one is whether custom resources should be created. In this case, the Deployment resources which is part of the main Kubernetes API will be used, so no custom resources should be created. The second question will ask whether a controller should be created. In this case, we needed a controller to program it to make changes to existing Deployments.

Once the command is run, kubebuilder will add a new directory, called 'controllers' to the existing file structure as shown in Figure 9.

Figure 9 new directory structure with the Deployment controller



At this point, the 'deployment_controller.go' file contains the main functions required to run the controller:

1. `Reconcile()`: This is the function in which most of the controller logic takes place. Kubernetes resources can be observed and state updates can be performed.
2. `SetupWithManager()`: This function adds the Deployment controller to the main controller manager, specified in `main.go`

Figure 10 provides a screenshot of the logic that is provided in both functions after the Deployment controller is initialised by kubebuilder.

Figure 10 Boilerplate Deployment Controller logic

```
// For more details, check Reconcile and its Result here:
// - https://pkg.go.dev/sigs.k8s.io/controller-runtime@v0.14.1/pkg/reconcile
func (r *DeploymentReconciler) Reconcile(ctx context.Context, req ctrl.Request) (ctrl.Result, error) {
    _ = log.FromContext(ctx)

    // TODO(user): your logic here

    return ctrl.Result{}, nil
}

// SetupWithManager sets up the controller with the Manager.
func (r *DeploymentReconciler) SetupWithManager(mgr ctrl.Manager) error {
    return ctrl.NewControllerManagedBy(mgr).
        For(&apps1.Deployment{}).
        Complete(r)
}
```

API controller

The API controller is the main part of the Operator. It is responsible to access Kubernetes Deployments inside the Kubernetes cluster and run them through the Operator logic specified in section 6 to apply the Operator CRD with the misconfiguration to the Deployment.

Before the API controller was added to the existing directory structure, two changes had to be made. These are listed below:

1. **Creating a subdirectory for each controller**

A subdirectory was created within the 'controllers' directory, called 'apps' and the two files of the Deployment Controller were moved into this directory. It was called 'apps' to reference the name of the Kubernetes API used for Deployments. This change will make it possible to better structure the project.

2. **Modifying the PROJECT file**

The second step took a lot of trial and error as this step has not been documented very well in the kubebuilder documentation. The different controllers that are used in the Operator are referenced in the PROJECT file. Figure 11 shows the code of this file as provided by the kubebuilder template. While the comments of the file explicitly state to not edit it, editing the file was necessary to create the API controller in a project that already had one controller initialised.

Figure 11 PROJECT file of the kubebuilder boilerplate

```
1 # Code generated by tool. DO NOT EDIT.
2 # This file is used to track the info used to scaffold your project
3 # and allow the plugins properly work.
4 # More info: https://book.kubebuilder.io/reference/project-config.html
5 domain: core.anaisurl.com
6 layout:
7 - go.kubebuilder.io/v3
8 projectName: security-chaos-engineering
9 repo: github.com/AnaisUrlichs/security-chaos-engineering.git
10 resources:
11 - controller: true
12   group: apps
13   kind: Deployment
14   path: k8s.io/api/apps/v1
15   version: v1
16 version: "3"
```

Before the 'projectName' field, an additional key has to be added to specify that the Operator is responsible for multiple controllers:

```
multigroup: true
```

Next, it was possible to create the API controller with the following kubebuilder command:

```
> kubebuilder create api --kind Configuration --version
v1alpha1 --group api
```

In this case, it was necessary to create both a custom resource for the controller and the controller.

Figure 12 Project structure with the two different controllers

```

  apis/api/v1alpha1
  ├── configuration_types.go
  ├── groupversion_info.go
  └── zz_generated.deepcopy.go
  bin
  config
  ├── crd
  ├── default
  ├── manager
  ├── prometheus
  ├── rbac
  └── samples
  controllers
  ├── api
  │   ├── configuration_controller.go
  │   └── suite_test.go
  └── apps
      ├── deployment_controller.go
      └── suite_test.go
```

This will create several new directories, namely:

- ‘apis/api/v1alpha1’ which will hold the definition for the custom resources of the Operator. This is further elaborate on in section 7.2.
- The ‘config’ directory now also holds a ‘crd’ and a ‘samples’ subdirectory, which contain the operator Kubernetes YAML manifest and example Operator CRDs respectively. This is further detailed in section 7.8.
- The logic for the api controller is set in the ‘controllers/api/configuration_controller.go’. This file currently contains the same boilerplate functions as the ‘deployment_controller.go’ file.

After the project directory has been set up, it becomes possible to start editing the files to implement the API Controller and the Deployment Controller logic for the Operator. The following sections will provide details with explanations on each file that has been modified in the kubebuilder boilerplate code.

7.2 Directory ‘apis/api/v1alpha1’

The API for the Custom Resource Definition used by the API Controller is specified in the ‘controllers/api’ directory. The most important part in this directory is the ‘ConfigurationSpec’ in the ‘configuration_types.go’ file. This defines a new type with the state of the input Misconfiguration for the Operator. Figure 13 provides an overview of the different types that can be defined through the Operator CRD.

Figure 13 Misconfiguration Spec with the input for the Operator CRD

```
// ConfigurationSpec defines the desired state of the Misconfiguration to be applied to deployments
type ConfigurationSpec struct {

    // Set Container ImageTag
    ImageTag string `json:"imageTag,omitempty"`

    // Set ContainerPort
    ContainerPort int32 `json:"containerPort,omitempty"`

    // Set allowPrivilegeEscalation
    AllowPrivilegeEscalation bool `json:"allowPrivilegeEscalation,omitempty"`

    // Set readOnlyRootFilesystem
    ReadOnlyRootFilesystem bool `json:"readOnlyRootFilesystem,omitempty"`

    // Set runAsNonRoot
    RunAsNonRoot bool `json:"runAsNonRoot,omitempty"`

    // CPU limits
    CPULimits resource.Quantity `json:"limits,omitempty"`

    // Memory limits
    MemoryLimits resource.Quantity `json:"memorylimits,omitempty"`

    //CPU requests
    CPURequests resource.Quantity `json:"requests,omitempty"`

    // Memory requests
    MemoryRequests resource.Quantity `json:"memoryrequests,omitempty"`
}
```

One of the rules for defining Kubernetes APIs is that the serialisation filed, meaning the filed that is passed into the Go program for processing, must be camelcase. ‘Omitempty’ specifies that if the field is not set, it will not be processed. Thus, when a user defines the Operator CRD, they do not have to set all of the fields.

The fields that can be set to modify the corresponding fields in the Kubernetes Deployment were chosen based on common Misconfiguration that Kubernetes administrators unintentionally make (*Rahman et al., 2023*). For instance, it is a common mistake to set the container image tag in the Deployment to the ‘latest’ tag instead of a specific container image version. Running a container image with the ‘latest’ tag means that the engineer does not know which container image will be pulled from the container registry. If an updated version is pushed to the container image in the container registry, then the Pod will automatically run that latest image. However, it could happen that the latest image is not compatible with the other tools, which are running inside the cluster or the application. By Misconfiguring the container image tag to “latest”, it is possible to test whether security scanners and other security tooling can identify that misconfiguration and set the container image back to the correct version.

The ‘ContainerPort’ can be misconfigured to test Firewall rules and NetworkPolicies. A Firewall is configured to restric the network traffic between two networks (*He, Xinzhou, 2021*). Similarly, a NetworkPolicy, as stated on the Kubernetes documentation, restricts the communication between the Pod and other endpoints in the cluster. A NetworkPolicy can be set through Kubernetes YAML manifests, using the Kubernetes Network Plugin. Both NetworkPolicies and Firewalls are important to restrict network connections of an application.

‘AllowPrivilegeEscalation’, ‘ReadOnlyFilesystem’ and ‘RunAsNonRoot’ can be modified to test the security context configuration of the Pod. Restricting the privileges of the container through these configuration is considered best practice. Thus, it will be useful to test changes to these configuration on different applications.

The last four fields, CPULimits, CPURequests, MemoryLimits, and MemoryRequests specify how many resources the Pod is allowed to access from the Kubernetes nodes. Depending on the application, different resource limits will be set.

7.3 Directory ‘controllers/api’

The custom API of the Operator is defined in the ‘controllers/api’ directory. The file containing the main controller logic is the ‘configuration_controller.go’ file. This section walks step by step through the logic of the api controller.

At the top of the file, all of the go libraries that are needed for the controller are imported as shown in Figure 14. The first three packages are go built-in libraries. The next block mainly consists of Kubernetes-specific libraries.

The last three libraries make it possible to access the Kubernetes APIs. ‘apiv1alpha1’ refers to the custom api specification of the Operator, which has been detailed in section 7.2. This will make it possible for the api Controller to access the Operator CRD from the Kubernetes

cluster. 'kapps' and 'kcore' reference the two Kubernetes APIs 'apps/v1' and 'core/v1' which make it possible to access the Deployments and fields in the Pod specification respectively.

Figure 14 Libraries in the api Controller

```
1 import (
2     "context"
3     "strings"
4     "time"
5
6     "github.com/go-logr/logr"
7     "k8s.io/apimachinery/pkg/api/errors"
8     "k8s.io/apimachinery/pkg/runtime"
9     "k8s.io/apimachinery/pkg/types"
10    ctrl "sigs.k8s.io/controller-runtime"
11    "sigs.k8s.io/controller-runtime/pkg/client"
12    "sigs.k8s.io/controller-runtime/pkg/log"
13
14    apiv1alpha1 "github.com/AnaisUrlichs/security-controller/apis/api/v1alpha1"
15    kapps "k8s.io/api/apps/v1"
16    kcore "k8s.io/api/core/v1"
17 )
```

The next section of the code, shown in Figure 15, has two main functions

1. Check if an Operator CRD exists, only if the CRD exists, the api controller should run.
2. Check whether the Operator CRD is being deleted, in which case, the api controller should also not run inside the cluster.

Figure 15 If statement to verify Operator CRD is running and not being deleted

```
1     if err := r.Client.Get(ctx, req.NamespacedName, mdConf); err != nil {
2         if errors.IsNotFound(err) {
3             // taking down all associated K8s resources is handled by K8s
4             r.Log.Info("No Misconfiguration Configuration found.")
5             return r.finishReconcile(nil, false)
6         }
7         r.Log.Error(err, "Failed to get the Misconfiguration Configuration")
8         return r.finishReconcile(err, false)
9     }
10
11     if !mdConf.ObjectMeta.DeletionTimestamp.IsZero() {
12         // Stop reconciliation as the item is being deleted
13         return r.finishReconcile(nil, false)
14     }
```

The next image, Figure 16, shows how the operator accesses and filters the Deployments in the Kubernetes cluster that should be modified.

Figure 16 Access Deployments with Operator annotation

```
1 // Get list of deployments
2 deploymentList := &kapps.DeploymentList{}
3 var mdDeploymentList []kapps.Deployment
4
5 if err := r.List(ctx, deploymentList); err != nil {
6     return r.finishReconcile(err, false)
7 }
8
9 cmExists := false
10 // Get list of deployments with annotation
11 for _, cm := range deploymentList.Items {
12     val, ok := cm.GetAnnotations()["anaisurl.com/misconfiguration"]
13     if ok && val == "true" {
14         cmExists = true
15         mdDeploymentList = append(mdDeploymentList, cm)
16     }
17 }
```

In line 2, the variable ‘deploymentList’ access all of the deployments inside of the Kubernetes cluster. Next, a new object is defined, called ‘mdDeploymentList’ which is an array of Kubernetes Deployments that is empty at this point. If the api controller is unable to access the ‘deploymentList’ from the cluster, the ‘finishReconcile’ function will be called. For all the Kubernetes Deployments in the ‘DeploymentList’, line 11 to 17, if the Deployment contains the Operator annotation and the annotation is set to “true”, then the Deployment is added to the ‘mdDeploymentList’ object. Additionally, a boolean variable, called ‘cmExists’ is set to true. This will make it easier further down in the code to verify, whether any Deployments have been found that should be modified.

Figure 17 Main if statement to update Deployments

```
101 if cmExists == true {
102     for _, cm := range mdDeploymentList {
103
104         err := r.Get(ctx, types.NamespacedName{Name: cm.Name, Namespace: cm.Namespace}, &cm)
105         deploymentStatus := cm.Status.Conditions[0].Type
106
107         // Update Deployment Spec
108         log.Info("Reconciling deployments" + cm.Name)
109         if err != nil && errors.IsNotFound(err) {
110             log.V(1).Info("Deployment is not found")
111             return r.finishReconcile(err, true)
112         } else if err == nil && deploymentStatus == "Available" {
113             cm.Spec.Template.Spec.Containers[0].Ports[0].ContainerPort = mdConf.Spec.ContainerPort
114             cm.Spec.Template.Spec.Containers[0].Image = strings.Split(cm.Spec.Template.Spec.Containers[0].Image, ":")[0] + ":" + mdConf.Spec.ImageTag
115             cm.Spec.Template.Spec.Containers[0].SecurityContext.AllowPrivilegeEscalation = &mdConf.Spec.AllowPrivilegeEscalation
116             cm.Spec.Template.Spec.Containers[0].SecurityContext.RunAsNonRoot = &mdConf.Spec.RunAsNonRoot
117             cm.Spec.Template.Spec.Containers[0].SecurityContext.ReadOnlyRootFilesystem = &mdConf.Spec.ReadOnlyRootFilesystem
118             cm.Spec.Template.Spec.Containers[0].Resources.Requests[kcore.ResourceCPU] = mdConf.Spec.CPURequests
119             cm.Spec.Template.Spec.Containers[0].Resources.Limits[kcore.ResourceCPU] = mdConf.Spec.CPULimits
120             cm.Spec.Template.Spec.Containers[0].Resources.Requests[kcore.ResourceMemory] = mdConf.Spec.MemoryRequests
121             cm.Spec.Template.Spec.Containers[0].Resources.Limits[kcore.ResourceMemory] = mdConf.Spec.MemoryLimits
122             cm.Annotations["anaisurl.com/last-updated"] = time.Now().Format(time.RFC3339)
123
124             val := "false"
125             cm.Annotations["anaisurl.com/misconfiguration"] = val
126
127             err := r.Client.Update(ctx, &cm)
128             if err != nil {
129                 r.finishReconcile(err, true)
130             }
131         }
132     }
133 } else {
134     if err := r.List(ctx, deploymentList); err != nil {
135         return r.finishReconcile(err, false)
136     }
137 }
```


Figure 17 is a screenshot that has been taken directly from the code and showcases the part of the Operator api that is used to update the Deployments. This section will walk through the different components of the if statement.

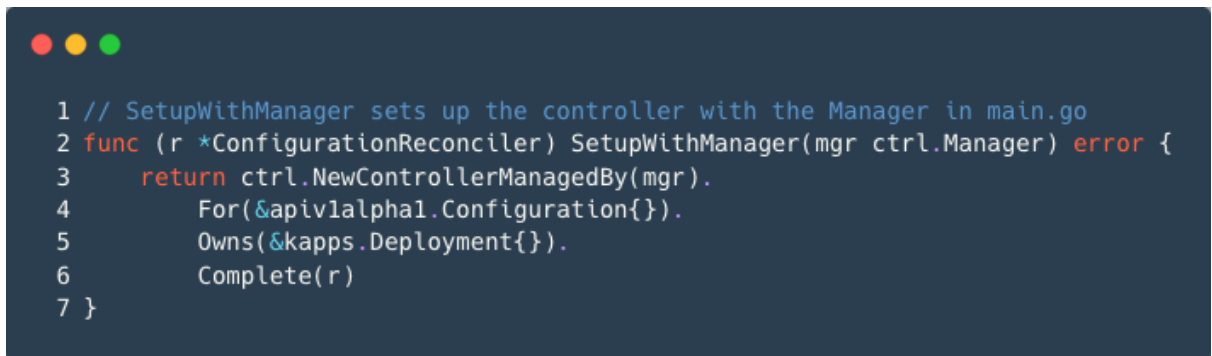
1. Line 101: If Deployments exists and have been added to the 'mdDeploymentList', the logic of the if statement will be executed.
2. Line 102: A for-loop is set up to iterate through each Kubernetes Deployment in the 'mdDeploymentList'.
3. Line 104: The name and then namespace of the Deployment is accessed. If this cannot be done, then an error will be thrown.
4. Line 105: The status of the Deployment is accessed to verify in Line 112 that the Deployment should only be modified if it is 'Available' and not being deleted.
5. Line 109 to 111: This if statement will make sure that the Deployment can still be found in the cluster and has not been deleted since it got added to the 'mdDeploymentList' list. If it cannot be found, then the 'finishReconcile' function is called.
6. Line 112: If there is no error at this point and the Deployment is 'Available', then each field of the Deployment Specification will be updated with the Misconfiguration provided in the Operator CRD.
7. Line 123 to 124: A new annotation is added to the Deployment Manifest with the current time and the Operator Annotation is set to false. Thus, the api controller will not run on this Deployment again unless the value of the Operator Annotation is set back to true and 10 hours have passed. The annotation referencing the current time is required in the apps controller, described in section 7.3.
8. Line 126 to 129: In this section, the Deployment in the cluster is updated with the new information. If it cannot be updated and an error is thrown, then the 'finishReconcile' function is called.
9. Line 132 to 136: Similarly, if the no Deployments have been found that should be modified, then the Controller will continue reconciling. Otherwise, if an error has been found, the controller will enter the 'finishReconcile' function.

Figure 18 finishReconcile function

```
1 func (r *ConfigurationReconciler) finishReconcile(err error, requeueImmediate bool) (ctrl.Result, error) {
2     if err != nil {
3         interval := reconcileErrorInterval
4         if requeueImmediate {
5             interval = 0
6         }
7         r.Log.Error(err, "Finished Reconciling Deployments with error: %w")
8         return ctrl.Result{Requeue: true, RequeueAfter: interval}, err
9     }
10    interval := reconcileSuccessInterval
11    if requeueImmediate {
12        interval = 0
13    }
14    r.Log.Info("Finished Reconciling Deployment")
15    return ctrl.Result{Requeue: true, RequeueAfter: interval}, nil
16 }
```

If any step throughout the controller fails because an object is either not accessible through the Kubernetes API or fails to update changes to the running Deployment in the Kubernetes cluster, the ‘finishReconcile’ function will be called. The main responsibility is to requeue the Deployment resources without stopping the controller from running inside the cluster. Even if one Deployment cannot be modified for various reasons, it should not prevent the controller from updating the other Deployments.

Figure 19 SetupWithManager function



```
1 // SetupWithManager sets up the controller with the Manager in main.go
2 func (r *ConfigurationReconciler) SetupWithManager(mgr ctrl.Manager) error {
3     return ctrl.NewControllerManagedBy(mgr).
4         For(&apiv1alpha1.Configuration{}).
5         Owns(&kapps.Deployment{}).
6         Complete(r)
7 }
```

The function ‘SetupWithManager’ shown in figure 19 is part of the kubebuilder boilerplate code. This function is required to register the Configuration api ‘apiv1alpha1’ with the Manager. The Manager is an object provided by kubebuilder that makes it possible to add controllers to it. Whenever the configuration is changed, it will reconcile through the Kubernetes Deployments.

7.4 Directory ‘controllers/apps’

The controller in the ‘controllers/apps’ directory is responsible for managing the Deployment annotation that makes it possible for the api controller to access the Deployment and apply the misconfiguration. The api controller should run on the Deployment resources either

1. Once they are deployed and first reconciled
2. If an existing Deployment changes configuration and the annotation is set to true
3. If the Operator annotation in the Deployment is set from “false” to “true” and 10 hours have passed

Once the api controller has run on a Deployment and misconfigured it, the Operator annotation will be set to false:

```
anaisurl.com/misconfiguration: "false"
```

Additionally, an annotation has been added by the api controller with the time at which the configuration change took place. The ‘apps’ controller will take these two information and set the annotation back to true if either of the following conditions are true:

- After 10 hours of the api controller running and misconfiguring the deployment
- The Deployment configuration changes and at least 10 hours have passed

Figure 20 Controller accessing the Kubernetes Deployments inside the cluster

```
1 deployment := &kapps.Deployment{
2
3 // Get list of deployments with annotation
4 if err := r.Get(ctx, req.NamespacedName, deployment); err != nil {
5     return ctrl.Result{}, client.IgnoreNotFound(err)
6 }
7
8 l.Info("Deployment", "name", deployment.Name, "namespace", deployment.Namespace, "annotations",
    deployment.Annotations)
```

This controller will reconcile each Deployment individually as it is Deployment-specific controller that uses the Kubernetes Deployment API. As shown in Figure 20, a new Deployment is accessed through the Kubernetes 'apps' api. The controller will access additional details of the Deployment, namely the Name and the Namespace it is in. This information is added to the controller log. The logging information makes it possible to understand processes of the controller better once the Operator has been deployed to the cluster. For instance, users would be able to share the logs of the Operator if it does not work as expected, which can aid the improvement of the Operator.

Figure 21 Setting the Deployment Annotation back to true

```
1     lastUpdated := deployment.GetCreationTimestamp().Time.Format(time.RFC3339)
2     val, ok := deployment.GetAnnotations()["anaisurl.com/misconfiguration"]
3
4     // check if lastUpdated is more than 1 minutes
5     if ok && val == "false" {
6
7         lastUpdatedTime, err := time.Parse(time.RFC3339, lastUpdated)
8
9         if time.Now().Sub(lastUpdatedTime) > 5*time.Minute {
10             val = "true"
11             // Update deployment
12             deployment.SetAnnotations(map[string]string{"anaisurl.com/misconfiguration": val})
13             deployment.Annotations["anaisurl.com/last-updated"] = time.Now().Format(time.RFC3339)
14
15             err := r.Client.Update(ctx, deployment)
16             if err != nil {
17                 return ctrl.Result{}, err
18             }
19         }
20
21         if err != nil {
22             return ctrl.Result{}, client.IgnoreNotFound(err)
23         }
24     }
25 }
```

The if-condition in Figure 21 is responsible to set the Deployment annotation back to 'true' so that the Deployment will be reconciled again by the API controller. To do so, it will access the annotation and the time that the annotation was last changed by the API controller or a cluster administrator. If the annotation is set to false, then the if statement will continue to run. It will then check that at least 5 minutes have passed since the Deployment was last changed. This is

merely an additional measure to ensure that the Deployment is only misconfigured if 10 hours have passed.

Without checking the time, the Deployment controller will run right after the api controller has modified the Deployment resource, and set the annotation back to 'true'. In this case, the Deployment has been misconfigured but the annotation is set to 'true' again as if it still has to be misconfigured. By checking the time, the Deployment Operator will run right after the API controller has misconfigured the Deployment. However, since 5 minutes have not passed yet, the Deployment controller will not be able to change the annotation to 'true' right away. After 10 hours, the Deployment Operator will run again. At this point, more than 5 minutes have passed and it can set the annotation back to 'true'.

7.5 File 'main.go'

The main.go file is the starting point for the operator. This file has largely been setup by kubebuilder as part of the boilerplate and as the controllers have been added. The two main sections regarding the controller setup logic are shown in Figure 22.

Figure 22 Controller setup logic



```
1 if err = (&appscontrollers.DeploymentReconciler{
2     Client: mgr.GetClient(),
3     Scheme: mgr.GetScheme(),
4 }).SetupWithManager(mgr); err != nil {
5     setupLog.Error(err, "unable to create controller", "controller", "Deployment")
6     os.Exit(1)
7 }
8 if err = (&apiconfigcontrollers.ConfigurationReconciler{
9     Client: mgr.GetClient(),
10    Scheme: mgr.GetScheme(),
11 }).SetupWithManager(mgr); err != nil {
12    setupLog.Error(err, "unable to create controller", "controller", "Configuration")
13    os.Exit(1)
14 }
```

Both if statements set up the custom controllers with the Manager in kubebuilder. The Manager is a sort of wrapper of the different controllers in the Operator. The manager is used to install the controllers into the Kubernetes cluster as detailed in the kubebuilder book.

7.6 File 'Dockerfile'

The Dockerfile provides the instructions on packaging the Controller logic into a container image. This file has not been modified from the boilerplate kubebuilder setup. However, it is worth noting that the Dockerfile has to be verified throughout the development of the controller to ensure that the directory structure of the project is accurately referenced even after manual changes have been made. For instance, the logic of the Deployment Controller has been moved into its own subdirectory 'controllers' as detailed in section 7.1. After this change, it is important to verify that the Dockerfile correctly copies the controller code into the container image that will be built as shown in Figure 23. The Dockerfile is used through the 'Makefile', which is further elaborated on in section 7.7.

Figure 23 Dockerfile copy logic from the source code of the controller into the container image

```
1 # Build the manager binary
2 FROM golang:1.19 as builder
3 ARG TARGETOS
4 ARG TARGETARCH
5
6 WORKDIR /workspace
7 # Copy the Go Modules manifests
8 COPY go.mod go.mod
9 COPY go.sum go.sum
10 # cache deps before building and copying source so that we don't need to re-download as much
11 # and so that source changes don't invalidate our downloaded layer
12 RUN go mod download
13
14 # Copy the go source
15 COPY main.go main.go
16 COPY apis/ apis/
17 COPY controllers/ controllers/
18
19 # Build
20 # the GOARCH has not a default value to allow the binary be built according to the host where the command
21 # was called. For example, if we call make docker-build in a local env which has the Apple Silicon M1 SO
22 # the docker BUILDPLATFORM arg will be linux/arm64 when for Apple x86 it will be linux/amd64. Therefore,
23 # by leaving it empty we can ensure that the container and binary shipped on it will have the same platform.
24 RUN CGO_ENABLED=0 GOOS=${TARGETOS:-linux} GOARCH=${TARGETARCH} go build -a -o manager main.go
25
26 # Use distroless as minimal base image to package the manager binary
27 # Refer to https://github.com/GoogleContainerTools/distroless for more details
28 FROM gcr.io/distroless/static:nonroot
29 WORKDIR /
30 COPY --from=builder /workspace/manager .
31 USER 65532:65532
32
33 ENTRYPOINT ["/manager"]
```

7.7 File ‘Makefile’ and deploying the Operator to a Kubernetes cluster

The Makefile defines how the Operator should be built into Kubernetes CRDs and deployed inside the Kubernetes cluster. This file has not been modified besides the first line, which references the name of the container image and the tag that will be used to package the operator:

```
IMG ?= anaisurlichs/security-controller:0.0.1
```

It is worth noting that ‘hard-coding’ the image tag does not support releases long-term as with each change to the operator, the image tag has to be updated. Thus, using a release tool would be a better alternative in the future.

The different commands of the Makefile are invoked through:

```
> make <command>
```

The command hereby refers to any command set in the Makefile. Note that this file was not used during the development and testing of the Operator but to package the Operator at the end with the following commands:

Build and push your image to the registry specified in the \$IMG variable:

```
> make docker-buildx
```

This runs the following command from the Makefile:

```
PLATFORMS ?= linux/arm64,linux/amd64,linux/s390x,linux/ppc64le
.PHONY: docker-buildx
docker-buildx: test ## Build and push docker image for the manager for cross-platform support
# copy existing Dockerfile and insert --platform=${BUILDPLATFORM} into Dockerfile.cross, and preserve the original Dockerfile
sed -e '1 s/\(^FROM\) /FROM --platform=${BUILDPLATFORM} /; t' -e '1, / s/ FROM --platform=${BUILDPLATFORM} /' Dockerfile > Dockerfile.cross
- docker buildx create --name project-v3-builder
- docker buildx use project-v3-builder
- docker buildx build --push --platform=${PLATFORMS} --tag ${IMG} -f Dockerfile.cross .
- docker buildx rm project-v3-builder
rm Dockerfile.cross
```

Once the container image for the operator is built and available in DockerHub, it is possible to deploy the operator the Kubernetes cluster:

```
> make deploy
```

This runs the following command from the Makefile:

```
.PHONY: deploy
deploy: manifests kustomize ## Deploy controller to the K8s cluster specified in ~/.kube/config.
cd config/manager && ${KUSTOMIZE} edit set image controller=${IMG}
${KUSTOMIZE} build config/default | kubectl apply -f -
```

Lastly, the sample configuration for the Operator and a Kubernetes Deployment need to be installed inside of the cluster. This is done directly through kubectl:

```
> kubectl apply -f config/samples/
```

This will install the following two files:

- api_v1alpha1_configuration.yaml
- deployment.yaml

The Kubernetes YAML manifests in the ‘config’ directory are further defined in the next section.

7.8 Directory ‘config’

The config directory contains the boilerplate YAML manifests from kubebuilder as well as the Operator YAML specification and the Operator CRD in which the misconfiguration are set.

The ‘config/CRD/bases/api.core.anaisurl.com_configurations.yaml’ defines the YAML specification for the Operator CRD manifest based on the information that has been set in the ‘apis/api/v1alpha1’ directory. Only if these specification are provided to the Kubernetes cluster, it is possible to install the Operator CRD and run the Operator.

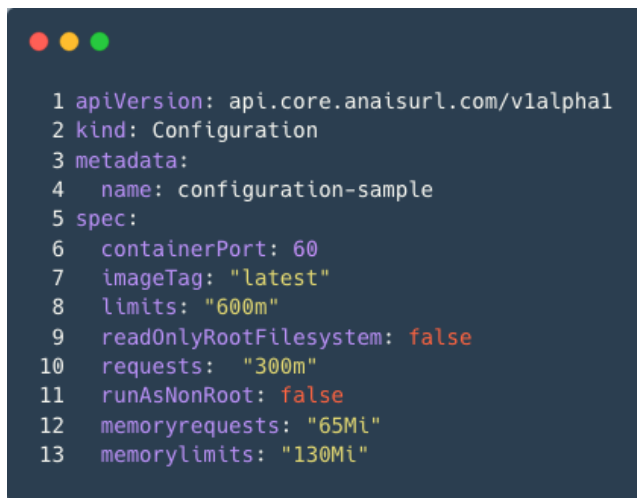
The directories ‘default’, ‘manager’ and ‘rbac’ are used to generate the Kubernetes manifests for the Operator through the Kustomize commands defined in the Makefile. Furthermore, the ‘prometheus’ directory installs the necessary components to scrape the Kubernetes Operator endpoint for metrics. Metrics provide further details into the processes and results of the

Kubernetes Operator. These can then be queried through Prometheus, “an open-source systems monitoring and alerting toolkit originally built at SoundCloud” as defined on the Prometheus documentation. When the Operator is run inside the cluster, it will create a Kubernetes Service. This Service has a ‘/metrics’ endpoint, which Prometheus is able to scrape. This functionality is provided by kubebuilder out of the box.

Lastly, the ‘config’ directory also contains the ‘samples’ directory with an example of the Operator CRD YAML manifest and the ‘deployment.yaml’ manifest that can be used to test the functionality of the operator.

The YAML manifest shown in Figure 24 defines the Operator CRD, which corresponds to the API specification set up in section 7.2.

Figure 24 Operator CRD



```
1 apiVersion: api.core.anaisurl.com/v1alpha1
2 kind: Configuration
3 metadata:
4   name: configuration-sample
5 spec:
6   containerPort: 60
7   imageTag: "latest"
8   limits: "600m"
9   readOnlyRootFilesystem: false
10  requests: "300m"
11  runAsNonRoot: false
12  memoryrequests: "65Mi"
13  memorylimits: "130Mi"
```

The important part to note here is the ‘apiVersion’ and the ‘kind’ set in the Operator CRD. ‘api.core.anaisurl.com/v1alpha1’ references the API naming convention used during the kubebuilder setup and as such the custom Kubernetes API extension of the Operator. The kind ‘Configuration’ is the name provided for the CRD.

7.9 Testing and Debugging during Development

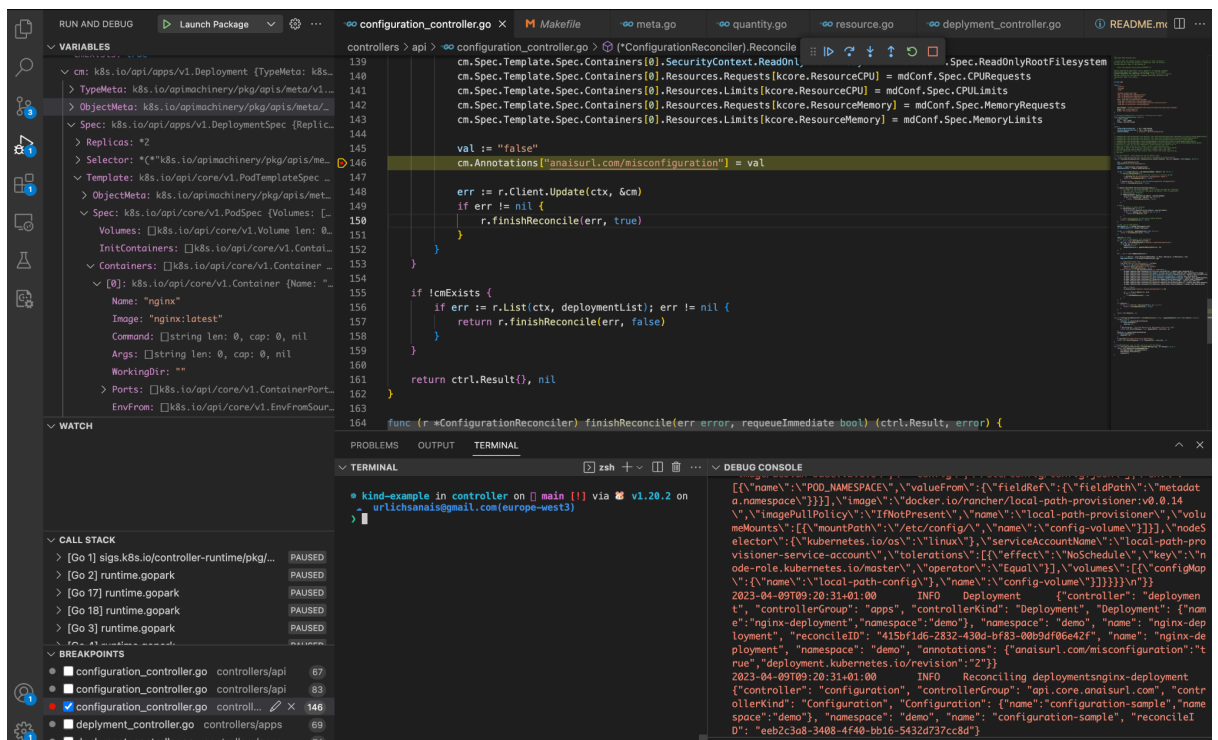
Testing the Operator and debugging its functions was a crucial part of the development of the artefacts. At the beginning, it has been very difficult to understand the exact behaviour of different functions and only through running the Operator and analysing its behaviour inside the cluster, it was possible to improve its processes. For instance, the following statement will attempt to update the ‘&cm’ object in the api controller. If no error is thrown, it will execute the code that comes next. However, if the Update() method will throw an error, then the ‘finishReconcile’ function is called, which will stop the Reconciliation loop as an error occurred.

```
err := r.Client.Update(ctx, &cm)      if err != nil {
    r.finishReconcile(err, true)
}
```


Knowing when to position the ‘Update()’ method in the code was not clear at first. If the method is called at the wrong place and the object that has to be updated changed in the meantime, the code would panic and exit.

This is one example in which debugging was crucial. Debugging was done through the VSCode Debugger, using the Golang plugin from the VSCode marketplace. A ‘launch.json’ file sets the parameters for VSCode to launch the debugger, this is done through running the Golang project locally. It is then possible to run the project in debugging mode and open individual objects as the code runs. Figure 25 provides an example for this.

Figure 25 VSCode, Operator running in Debug mode



Furthermore, to the right bottom of the screenshot is the logging information from the reconciliation loop. This makes it possible to further understand the chronological order in which functions are executed by the controller.

8.0 Operator Usage

While the previous sections described how the Operator design has been implemented in code, it did not showcase how the Operator is running inside of the Kubernetes cluster. This section describes with screenshots how a cluster administrator would interact with the Kubernetes Operator during a Security Chaos Engineering Experiment.

8.1 Define Security Chaos Experiment

Each Security Chaos Experiment as defined in the book on Security Chaos Engineering by *Rinehart and Shortridge (2021)* should consist of the following:

1. A hypothesis
2. Defined variables that influence the experiment, the cause and the outcome
3. A clear timeframe
4. Actions to be taken once the experiment is completed

A hypothesis could consist of the following:

If the container image defined in the deployment is set to the latest tag, the security scanner running inside the cluster should throw an alert to notify cluster administrators of the change.

The Operator can then be used to modify the container image, running inside the Deployment, to the latest tag. Once done, the cluster administrators can observe and review the response of the security tooling to see whether it responded as defined in the hypothesis. If it did not respond as expected, the configuration of the security tooling can be modified.

8.2 Install the Operator into the Kubernetes cluster

For this example, the Raspberry Pi Kubernetes cluster that has been built and detailed in section 3 will be used. The Kubeconfig of the cluster is set to the Kubeconfig used by kubectl. Thus, running kubectl commands connects through the Kubernetes API to the cluster. An example is shown in Figure 26.

Figure 26 Querying the nodes of the Kubernetes cluster

```
* microk8s in ~/code/thesis on  urlichsanais@gmail.com(europe-west3) took 42s
> kubectl get nodes
NAME     STATUS    ROLES    AGE   VERSION
pi2      Ready     <none>    110d  v1.26.3
pi0      Ready     <none>    110d  v1.26.3
pi3      Ready     <none>    110d  v1.26.3
pi1      Ready     <none>    110d  v1.26.3
```

Figure 26 shows the 4 Raspberry Pi nodes that make up the cluster. ‘Pi0’ is the main node in the cluster. Once ssh into the main node, it is possible to modify the cluster configuration and to interact with microk8s to edit the cluster as shown in Figure 27.

Figure 27 running microk8s from the main node

```
pi0@pi0:~$ microk8s status
microk8s is running
high-availability: no
  datastore master nodes: 192.168.0.30:19001
  datastore standby nodes: none
```

Next, the Operator Kubernetes manifests, which have been built through the commands specified in section 7.7, can be installed. This is done by utilising the Makefile:

```
> make deploy
```

Figure 28 displays the output of the command as it installs the Operator CRDs and related resources to the cluster.

Figure 28 Output of the ‘make deploy’ command


```
* microk8s in controller on  main via  v1.20.2 on  urlichsanais@gmail.com(europe-west3) took 6m46s
> make deploy
test -s /Users/anaisurlichs/code/thesis/controller/bin/controller-gen && /Users/anaisurlichs/code/thesis/controller/bin/controller-gen --version | gre
p -q v0.11.1 || \
    GOBIN=/Users/anaisurlichs/code/thesis/controller/bin go install sigs.k8s.io/controller-tools/cmd/controller-gen@v0.11.1
/Users/anaisurlichs/code/thesis/controller/bin/controller-gen rbac:roleName=manager-role crd webhook paths="./..." output:crd:artifacts:config=config/
crd/bases
test -s /Users/anaisurlichs/code/thesis/controller/bin/kustomize || { curl -Ss "https://raw.githubusercontent.com/kubernetes-sigs/kustomize/master/hac
k/install_kustomize.sh" | bash -s -- 3.8.7 /Users/anaisurlichs/code/thesis/controller/bin; }
cd config/manager && /Users/anaisurlichs/code/thesis/controller/bin/kustomize edit set image controller=anaisurlichs/security-controller:0.0.1
/Users/anaisurlichs/code/thesis/controller/bin/kustomize build config/default | kubectl apply -f -
namespace/controller-system created
customresourcedefinition.apiextensions.k8s.io/configurations.api.core.anaisurl.com created
serviceaccount/controller-controller-manager created
role.rbac.authorization.k8s.io/controller-leader-election-role created
clusterrole.rbac.authorization.k8s.io/controller-configuration-editor-role created
clusterrole.rbac.authorization.k8s.io/controller-configuration-viewer-role created
clusterrole.rbac.authorization.k8s.io/controller-manager-role created
clusterrole.rbac.authorization.k8s.io/controller-metrics-reader created
clusterrole.rbac.authorization.k8s.io/controller-proxy-role created
rolebinding.rbac.authorization.k8s.io/controller-leader-election-rolebinding created
clusterrolebinding.rbac.authorization.k8s.io/controller-configuration-rolebinding created
clusterrolebinding.rbac.authorization.k8s.io/controller-manager-rolebinding created
clusterrolebinding.rbac.authorization.k8s.io/controller-proxy-rolebinding created
service/controller-controller-manager-metrics-service created
deployment.apps/controller-controller-manager created
```

Since it is Kubernetes best practices to use specific namespaces that will contain the deployments, the next step will create a new demo namespace in the cluster:

```
> kubectl create ns demo
```

All the namespaces can be queried through kubectl as shown in Figure 29. The ‘controller-system’ namespace has been created that runs the pod of the Kubernetes Operator. All the additional Operator resources have also been added to this namespace. This makes it possible to remove the Operator-related resources by deleting the namespace.

Figure 29 query Kubernetes namespaces

```
* microk8s in controller on  main via  v1.20.2 on  
> kubectl get namespace
NAME                STATUS    AGE
kube-system         Active   110d
kube-public          Active   110d
kube-node-lease      Active   110d
default             Active   110d
controller-system    Active   5m8s
demo                Active   23s
```

To finish the setup, a Deployment manifest has to be installed inside of the Kubernetes cluster. In this case, the example deployment in the ‘config/samples’ folder will be used and applied to the ‘demo’ namespace through kubectl:

```
> kubectl apply -f config/samples/deployment.yaml -n demo
```

This will install the deployment shown in Figure 30 into the Kubernetes cluster.

Figure 30 Kubernetes Deployment before the Operator modifies it



```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: nginx-deployment
5   namespace: demo
6   annotations:
7     anaisurl.com/misconfiguration: "true"
8   labels:
9     app: nginx
10 spec:
11   replicas: 2
12   selector:
13     matchLabels:
14       app: nginx
15   template:
16     metadata:
17       labels:
18         app: nginx
19     spec:
20       containers:
21       - name: nginx
22         image: nginx:1.23
23         resources:
24           limits:
25             memory: "128Mi"
26             cpu: "500m"
27           requests:
28             memory: "64Mi"
29             cpu: "250m"
30       securityContext:
31         allowPrivilegeEscalation: false
32         runAsNonRoot: false
33         readOnlyRootFilesystem: false
34       ports:
35       - containerPort: 80
```

A cluster administrator could use any Deployment inside of the Kubernetes cluster. The only aspect that would have to be changed is to add the annotation for the Operator to discover the Deployment. This could be done with another kubectl command that modifies an existing deployment:

```
> kubectl annotate deployment/nginx-deployment -n demo
anaisurl.com/misconfiguration="true"
```

8.3 Set the Operator CRD to introduce misconfiguration into the Deployment

The Operator will only run and introduce misconfiguration to the Deployment when it has an Operator CRD. The Operator CRD can be installed through another kubectl command:

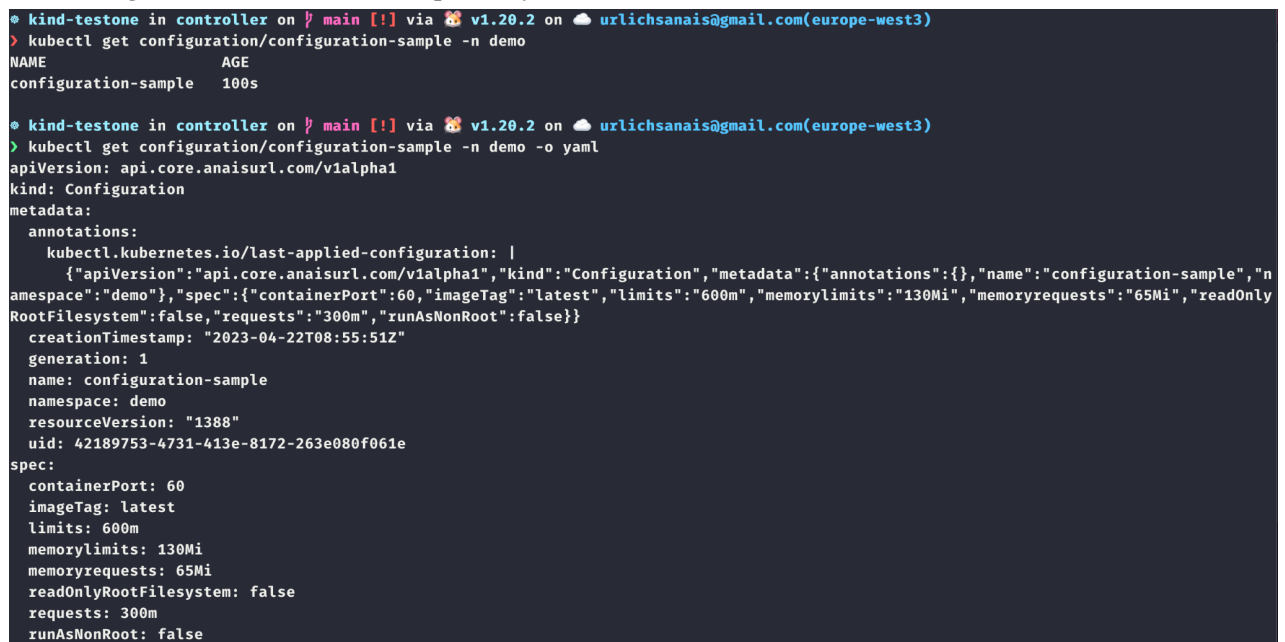
```
> kubectl apply -f
config/samples/api_v1alpha1_configuration.yaml -n demo
```

It is possible to view the Operator specification through the following kubectl command:

```
> kubectl get configuration/configuration-sample -n demo
```

The configuration can also be displayed in YAML as shown in Figure 31.

Figure 31 Kubernetes CRD queried from the cluster



```
• kind-testone in controller on  main [!] via  v1.20.2 on  urlichsanaais@gmail.com(europe-west3)
> kubectl get configuration/configuration-sample -n demo
NAME              AGE
configuration-sample  100s

• kind-testone in controller on  main [!] via  v1.20.2 on  urlichsanaais@gmail.com(europe-west3)
> kubectl get configuration/configuration-sample -n demo -o yaml
apiVersion: api.core.anaissurl.com/v1alpha1
kind: Configuration
metadata:
  annotations:
    kubernetes.io/last-applied-configuration: |
      {"apiVersion":"api.core.anaissurl.com/v1alpha1","kind":"Configuration","metadata":{"annotations":{},"name":"configuration-sample"},"namespace":"demo"},"spec":{"containerPort":60,"imageTag":"latest","limits":{"memorylimits":"130Mi","memoryrequests":"65Mi","readOnlyRootFilesystem":false,"requests":"300m","runAsNonRoot":false}}
      creationTimestamp: "2023-04-22T08:55:51Z"
  generation: 1
  name: configuration-sample
  namespace: demo
  resourceVersion: "1388"
  uid: 42189753-4731-413e-8172-263e080f061e
spec:
  containerPort: 60
  imageTag: latest
  limits: 600m
  memorylimits: 130Mi
  memoryrequests: 65Mi
  readOnlyRootFilesystem: false
  requests: 300m
  runAsNonRoot: false
```

Once the Operator CRD is added to the cluster, the operator will reconcile the resource and make the changes specified in the Operator CRD to its configuration. To view the modified YAML manifest running inside of the Kubernetes cluster, another kubectl command can be used:

```
> kubectl get deployment/nginx-deployment -n demo -o yaml
```

The output of the command is provided in Figure 32.

Figure 32 Deployment after the Operator applied the changes

```
* microk8s in controller on main via v1.20.2 on urlichsanais@gmail.com(europe-west3)
> kubectl get deployment/nginx-deployment -n demo -o yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    anaisurl.com/last-updated: "2023-04-23T09:28:14Z"
    anaisurl.com/misconfiguration: "false"
    deployment.kubernetes.io/revision: "2"
    kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{"anaisurl.com/misconfiguration":"true"},"labels":{"app":"nginx"},"name":"nginx-deployment","namespace":"demo"},
      "spec":{"replicas":2,"selector":{"matchLabels":{"app":"nginx"},"template":{"metadata":{"labels":{"app":"nginx"},"spec":{"containers":[{"image":"nginx:1.23","name":"nginx","ports":[{"co
ntainerPort":80}],"resources":{"limits":{"cpu":"500m","memory":"128Mi"},"requests":{"cpu":"250m","memory":"64Mi"},"securityContext":{"allowPrivilegeEscalation":false,"readOnlyRootFilesys
tem":false,"runAsNonRoot":false}}}}}}}
  creationTimestamp: "2023-04-23T09:25:44Z"
  generation: 2
  labels:
    app: nginx
    name: nginx-deployment
    namespace: demo
  resourceVersion: "6607636"
  uid: a6f15182-2095-4579-ae1e-97d518bbfcac
spec:
  progressDeadlineSeconds: 600
  replicas: 2
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:
        - image: nginx:latest
          imagePullPolicy: IfNotPresent
          name: nginx
          ports:
            - containerPort: 60
              protocol: TCP
```

By comparing Figure 30 to Figure 32, it is visible that all the fields, which have been set in the Operator CRD, have now changed. For instance, the container image does not use the specific container tag of '1.23' but the 'latest' tag.

9.0 Future Work

This report mentioned several times the possibility of using the Operator developed to test security tooling such as security scanner. Security scanning and the tools available in the cloud native ecosystem are outside the scope of this report. Generally, a security scanner will receive access to the resources in the Kubernetes cluster, it can then scan the configuration provided against a list of best practices. This and other potential use cases for the operator can be evaluated in future work.

10.0 Conclusion

This report and the Operator developed can provide a starting point to explore the principles and implementation of Security Chaos Engineering into a Kubernetes cluster. The cloud native ecosystem, in particular Kubernetes installation, is complex to manage. While the Operator itself is not production ready as it requires additional testing, it can aid cluster administrators to build higher confidence in their security tooling by verifying the response to configuration changes.

11.0 Bibliography

1. Ali Basiri, Niosha Behnam, Ruud de Rooij, Lorin Hochstein, Luke Kosewski, Justin Reynolds, Casey Rosenthal, "Chaos Engineering", *IEEE Software*, vol.33, no. 3, pp. 35 41, May June 2016, DOI:10.1109/MS.2016.60
2. API overview (no date) *Kubernetes API Reference Docs*. Available at: <https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.22/#deployment-v1-apps> (Accessed: March 15, 2023).
3. Böhm, Sebastian & Wirtz, Guido. (2021). *Profiling Lightweight Container Platforms: MicroK8s and K3s in Comparison to Kubernetes*.
4. Cncf (no date) *Tag-app-delivery/operator-whitepaper_v1-0.MD at Main · CNCF/tag-app-delivery*, GitHub. Available at: https://github.com/cncf/tag-app-delivery/blob/main/operator-wg/whitepaper/Operator-White-Paper_v1-0.md (Accessed: March 3, 2023).
5. Dart, S.A. et al. (1987) *Overview of software development environments, Overview of Software Development Environments*. IEEE. Available at: <https://www.ics.uci.edu/~andre/ics228s2006/dartellisonfeilerhabermann.pdf> (Accessed: April 21, 2023).
6. *Deployments (2023) Kubernetes*. Available at: <https://kubernetes.io/docs/concepts/workloads/controllers/deployment/> (Accessed: March 15, 2023).
7. Dobies, J. and Wood, J. (2020) *Kubernetes operators: Automating the Container Orchestration Platform*. 1st edn. Sebastopol, CA: O'Reilly Media.
8. E. Truyen, N. Kratzke, D. Van Landuyt, B. Lagaisse and W. Joosen, (2020) "Managing Feature Compatibility in Kubernetes: Vendor Comparison and Analysis," in *IEEE Access*, vol. 8, pp. 228420-228439, 2020, doi: 10.1109/ACCESS.2020.3045768.
9. Faulk, Stuart. (1995). *Software Requirements: A Tutorial*. 35.
10. Fawaz, Paraiso & Challita, Stephanie & al-dhuraibi, Yahya & Merle, Philippe. (2016). *Model-Driven Management of Docker Containers*. 10.1109/CLOUD.2016.0100.
11. Gannon, Dennis & Barga, Roger & Sundaresan, Neel. (2017). *Cloud-Native Applications*. *IEEE Cloud Computing*. 4. 16-21. 10.1109/MCC.2017.4250939.
12. Gartner_Inc (no date) *Definition of SDK (software development kit) - gartner information technology glossary*, Gartner. Available at: <https://www.gartner.com/en/information-technology/glossary/sdk-software-development-kit> (Accessed: March 29, 2023).
13. (no date) *Git*. Available at: <https://git-scm.com/> (Accessed: March 29, 2023).

14. Gkrekos, I. et al. (2019) *Uses and applications of ubuntu: A technical guide* - researchgate, *USES AND APPLICATIONS OF UBUNTU: A TECHNICAL GUIDE*. Available at: https://www.researchgate.net/publication/336642334_USES_AND_APPLICATIONS_OF_UBUNTU_A_TECHNICAL_GUIDE (Accessed: March 20, 2023).
15. He, Xinzhou. (2021). *Research on Computer Network Security Based on Firewall Technology*. *Journal of Physics: Conference Series*. 1744. 042037. 10.1088/1742-6596/1744/4/042037.
16. (no date) *Homebrew*. Available at: <https://brew.sh/> (Accessed: March 29, 2023).
17. *How to build a raspberry pi kubernetes cluster using Microk8s* (no date) *How to build a Raspberry Pi Kubernetes cluster using MicroK8s*. Available at: <https://ubuntu.com/tutorials/how-to-kubernetes-cluster-on-raspberry-pi> (Accessed: January 12, 2023).
18. *How to install ubuntu server on your raspberry pi* (no date) *How to install Ubuntu Server on your Raspberry Pi*. Available at: <https://ubuntu.com/tutorials/how-to-install-ubuntu-on-your-raspberry-pi> (Accessed: January 12, 2023).
19. (no date) *Introduction - the kubebuilder book*. Available at: <https://book.kubebuilder.io/introduction.html> (Accessed: March 12, 2023).
20. *JD.com case study* (2020) *Kubernetes*. Available at: <https://kubernetes.io/case-studies/jd-com/> (Accessed: April 5, 2023).
21. Jeffery, Andrew & Howard, Heidi & Mortier, Richard. (2021). *Rearchitecting Kubernetes for the Edge*.
22. Kazuo Furuta, "Resilience Engineering," in Joonhong Ahn, Cathryn Carson, Mikael Jensen, Kohta Juraku, Shinya Nagasaki, and Satoru Tanaka (eds), *Reflections on the Fukushima Daiichi Nuclear Accident* (New York: Springer, 2015), sec. 24.4.2.
23. Kendig, Catherine. (2015). *What is Proof of Concept Research and how does it Generate Epistemic and Ethical Categories for Future Scientific Practice?*. *Science and engineering ethics*. 22. 10.1007/s11948-015-9654-0.
24. Kjorveziroski, Vojdan & Filiposka, Sonja. (2022). *Kubernetes distributions for the edge: serverless performance evaluation*. *The Journal of Supercomputing*. 78. 1-28. 10.1007/s11227-022-04430-6.
25. *Kubernetes Documentation* (no date) *Kubernetes*. Available at: <https://kubernetes.io/docs/home/> (Accessed: March 10, 2023).
26. *Kubernetes-Sigs* (no date) *Kubernetes-Sigs/Kubebuilder: Kubebuilder - SDK for building kubernetes apis using CRDS, GitHub*. Available at: <https://github.com/kubernetes-sigs/kubebuilder> (Accessed: March 7, 2023).

27. Microsoft (2021) *Visual studio code - code editing. redefined*, RSS. Microsoft. Available at: <https://code.visualstudio.com/> (Accessed: March 20, 2023).
28. Optum (no date) *Optum/Chaoslingr: Chaoslingr: Introducing security into chaos testing, ChaoSlingr GitHub*. Available at: <https://github.com/Optum/ChaoSlingr> (Accessed: March 5, 2023).
29. *Organizing cluster access using Kubeconfig Files* (2022) Kubernetes. Available at: <https://kubernetes.io/docs/concepts/configuration/organize-cluster-access-kubeconfig/> (Accessed: March 22, 2023).
30. Pahl, Claus & Brogi, Antonio & Soldani, Jacopo & Jamshidi, Pooyan. (2017). *Cloud Container Technologies: A State-of-the-Art Review*. *IEEE Transactions on Cloud Computing*. PP. 1-1. 10.1109/TCC.2017.2702586.
31. Rahman, Akond & Shamim, Md Shazibul Islam & Bose, Dibyendu & Pandita, Rahul. (2023). *Security Misconfigurations in Open Source Kubernetes Manifests: An Empirical Study*. *ACM Transactions on Software Engineering and Methodology*. 10.1145/3579639.
32. Raspberry pi documentation (no date) *Getting started Documentation*. Available at: <https://www.raspberrypi.com/documentation/computers/getting-started.html> (Accessed: March 10, 2023).
33. *Replicated* (no date) *Kubernetes Native Configuration Management, Kustomize*. Available at: <https://kustomize.io/> (Accessed: March 29, 2023).
34. Rinehart, A. and Shortridge, K. 2021 *Security Chaos Engineering*. 2nd edn. Sebastopol, CA: O'Reilly Media.
35. Ucl (2020) *What is SSH and how do I use it?*, Information Services Division. Available at: <https://www.ucl.ac.uk/isd/what-ssh-and-how-do-i-use-it> (Accessed: March 22, 2023).
36. *Yaml Ain't markup language (YAML™) version 1.2* (no date) *YAML Ain't Markup Language (YAML™) revision 1.2.2*. Available at: <https://yaml.org/spec/1.2.2/> (Accessed: March 5, 2023).

12.0 Appendices

Appendix 1 Chaos Engineering Projects

Figure 33 Comparison of Chaos Engineering Projects

Project Name	Chaos Monkey/ Kube Monkey	Litmus Chaos	ChaosBlade	Chaos Mesh
Kubernetes installation available	Yes	Yes	N/A	Yes
Part of the CNCF	No	Yes	Yes	Yes
SDK available	No	Yes Also in Go	N/A	Possible to extend Chaos Experiment Type
Opt-in approach	by application	by application	N/A	by namespace
Started by e.g. company	Netflix	N/A	N/A	N/A
Link to documentation	Link	Link	N/A	Link
Possibility to create Chaos Experiments	No	Yes	N/A	Yes
GUI	No	Yes	N/A	No
CLI	No	Yes	N/A	No
Helm Installation	No	Yes	N/A	yes

The table above, Figure 33, provides an overview of four different Chaos Engineering focused projects; namely, Kube Monkey, Litmus Chaos, ChaosBlade and Chaos Mesh. All projects are open source. While Litmus Chaos, ChaosBlade and Chaos Mesh are donated to the Cloud Native Computing Foundation and as such are under its governing structure, Kube Monkey is a standalone, open source tool.

Overall, Litmus Chaos and Chaos Mesh are the two more advanced tools. Not only do they offer different Kubernetes installation options but also provide ways to expand the type of Chaos Experiments that each project can be used for. Litmus Chaos offers several different SDKs, including an SDK written in Go. As such, it seems like the tool that would likely be the easiest to implement Security Chaos Engineering experiments. In comparison, Chaos Mesh does not have SDKs in different languages available. However, it is possible to implement new Chaos Engineering types to the project, also written in Go. Without trying to implement Security Chaos Engineering experiments in both projects, it is difficult to assess

the limitations of both. Limitations could relate to the type of Kubernetes resources that the experiment can access and modify as well as the way that resources are modified and misconfigured.

In comparison, kube monkey is focused on deleting specific pods within the Kubernetes cluster. It does not offer a way to modify resources nor to configure how those resources should be modified. Lastly, ChaosBlade is a newer project in the CNCF. Its documentation and website is not available in English and thus, it was difficult to assess its functionality.

Bibliography for the Appendix

1. *A powerful Chaos Engineering Platform for kubernetes: Chaos mesh* (no date) *Chaos Mesh* RSS. Available at: <https://chaos-mesh.org/> (Accessed: April 5, 2023).
2. *Chaosblade · help companies solve the high availability problems in the process of migrating to cloud-native systems through Chaos Engineering: Chaosblade* (no date) *ChaosBlade* RSS. Available at: <https://chaosblade.io/> (Accessed: April 5, 2023).
3. *Chaos Engineering projects on the CNCF Cloud Native Interactive Landscape* (no date) *Cloud Native Landscape*. Available at: <https://landscape.cncf.io/card-mode?category=chaos-engineering&grouping=category> (Accessed: April 5, 2023).
4. *Open source chaos engineering platform* (no date) *LitmusChaos*. Available at: <https://litmuschaos.io/> (Accessed: April 5, 2023).
5. Asobti (no date) *ASOBTI/Kube-monkey: An implementation of netflix's chaos monkey for kubernetes clusters*, *GitHub*. Available at: <https://github.com/asobti/kube-monkey> (Accessed: April 5, 2023).

Total Word Count of the main report: 10631