UNIVERSIDADE Đ
COIMBRA

Miguel de Sousa Nunes Teixeira

µDetector: An Intrusion Detection Tool for Microservices

Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Professor Nuno Antunes and José Flora and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2022

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE Ð
COIMBRA

DEPARTMENT OF INFORMATICS ENGINEERING

Miguel de Sousa Nunes Teixeira

µDetector: An Intrusion Detection Tool for Microservices

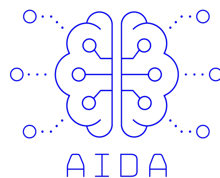
Dissertation in the context of the Master in Informatics Engineering, specialization in Software Engineering, advised by Prof. Nuno Antunes and José Flora, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.

July 2022

The work presented in this thesis was carried out within the Software and Systems Engineering (SSE) group of the Centre for Informatics and Systems of the University of Coimbra (CISUC)

This work is partially supported by the project AIDA: Adaptive, Intelligent and Distributed Assurance Platform FCT (CMU-PT) (`POCI-01-0247-FEDER-045907`), co-funded by the Portuguese Foundation for Science and Technology (FCT) and by the *Fundo Europeu de Desenvolvimento Regional* (FEDER) through *Portugal 2020 - Programa Operacional Competitividade e Internacionalização (POCI)*.

This work has been supervised by Prof. Nuno Antunes and José Flora, and presented to the Department of Informatics Engineering of the Faculty of Sciences and Technology of the University of Coimbra.



Cofinanciado por:

UNIÃO EUROPEIA
Fundo Europeu
de Desenvolvimento Regional

Acknowledgements

To begin with, I would like to thank my supervisors Professor Nuno Antunes and José Flora for the opportunity to work with them and for all the support given before and during this dissertation.

I want to thank Adriana for her love, care, and support. You mean the world to me.

I would also like to thank my friends for all the good moments we spent and for helping me throughout this journey.

Finally, I would like to thank my family - dad, mom, brothers, and aunt - for always being there for me, for the unconditional love and sacrifice, and for making sure I had everything I ever needed.

This thesis is dedicated to all of you.

Thank you.

Abstract

In recent years, there has been an increasing adoption of microservice-based applications in organizations and businesses. This software architectural style divides an application into small independent services that communicate using lightweight mechanisms, improving flexibility and scalability in dynamic DevOps environments. Containers are often used to deploy these applications with the help of orchestration tools such as Kubernetes. However, the growing popularity of microservices raises concerns related to dependability and security of these systems. The rising number of attacks and the lack of intrusion detection tools that target microservices applications aggravate these problems. Thus, the development of solutions that can be deployed in real-world scenarios and whose purpose is to keep applications and businesses secure is of the utmost importance. Monitoring and identifying suspicious activities with the help of Intrusion Detection tools are a possible approach to making microservices applications trustworthy in real-world scenarios.

In this work, we propose ***µDetector***, an intrusion detection tool for microservice-based applications. This tool relies on proof-of-concept techniques of Intrusion Detection, previously researched in the group, and automates their functioning for Kubernetes and KubeEdge deployments. In practice, after the user has provided a configuration, the tool uses monitoring agents to automatically collect system calls from the desired containers and transfers them over to the Intrusion Detection module of the tool. This module covers all the stages of anomaly-based intrusion detection and all activity classified as anomalous will trigger alarms in real-time indicating a possible intrusion in the microservices' application. The user can interact with the tool and its monitoring capabilities through a command-line interface or a web dashboard. Following the development phase, the *µDetector* tool was validated using functional testing, and performance and scalability tests. The results showed that the *µDetector* tool performs well and does not impact the proper functioning of the microservices application: in scenarios where there were over 100 000 system calls being collected per second, the Central Processing Unit (CPU) and memory usage of the worker nodes did not exceed 10% of the total resources available. This work presents itself as a further step in strengthening container security and microservices applications in cloud environments through the use of state-of-the-art intrusion detection techniques.

Keywords

Microservices, Cloud Computing, Kubernetes, Intrusion Detection, Containers, Monitoring

Resumo

Nos últimos anos, tem havido uma grande adoção de aplicações baseadas em microsserviços por parte de organizações e empresas. Este estilo arquitetural de software divide uma aplicação em pequenos serviços independentes que comunicam entre si utilizando mecanismos *lightweight*, melhorando assim a flexibilidade e a escalabilidade em ambientes de *DevOps*. Os *containers* são frequentemente utilizados para implementar estas aplicações juntamente com a ajuda de ferramentas de orquestração, tais como o Kubernetes. Contudo, a crescente popularidade dos microsserviços suscita preocupações relacionadas com a confiabilidade e segurança destes sistemas. O número crescente de ataques e a falta de ferramentas de detecção de intrusão direcionadas a aplicações de microsserviços agravam estes problemas. Assim, o desenvolvimento de soluções que podem ser implementadas em cenários reais e cujo objectivo é manter as aplicações e as empresas seguras é de extrema importância. A monitorização e identificação de actividades suspeitas com a ajuda de ferramentas de detecção de intrusão são uma abordagem possível para tornar as aplicações de microsserviços confiáveis.

Neste trabalho, propomos ***µDetector***, uma ferramenta de detecção de intrusões para aplicações baseadas em microsserviços. Esta ferramenta faz uso de técnicas de Detecção de Intrusão, previamente pesquisadas no grupo de investigação, e automatiza o seu funcionamento para Kubernetes e KubeEdge. Na prática, após o utilizador ter fornecido uma configuração, a ferramenta usa agentes de monitorização nos *nodes* pretendidos da aplicação, recolhe *system calls* e transfere-os para o módulo de Detecção de Intrusão da ferramenta. Este módulo cobre todas as fases da detecção de intrusão baseada em anomalias e toda a actividade classificada como anómala desencadeará alarmes em tempo real indicando uma possível intrusão na aplicação. O utilizador pode interagir com a ferramenta através de uma interface de linha de comandos ou através de uma *Web Dashboard*. Após a fase de desenvolvimento, a ferramenta *µDetector* foi validada recorrendo a testes funcionais, de desempenho e de escalabilidade. Os resultados mostram que a ferramenta oferece um bom desempenho e não afeta o funcionamento normal da aplicação de microsserviços: em cenários onde mais de 100 000 system calls estão a ser recolhidas por segundo, a percentagem de utilização de CPU e de memória dos worker nodes não ultrapassou os 10% do total de recursos disponíveis. Este trabalho pretende contribuir para o reforço da segurança em *containers* e aplicações de microsserviços em ambientes de computação em nuvem, através da utilização de técnicas de detecção de intrusão de última geração.

Palavras-Chave

Microsserviços, Computação na Nuvem, Kubernetes, Deteção de intrusão, Containers, Monitorização

List of Publications

This dissertation is partially based on the work presented in the following publications:

- José Flora, Paulo Gonçalves, **Miguel Teixeira**, Nuno Antunes, "My Services Got Old! Can Kubernetes Handle the Aging of Microservices?", *The 13th International Workshop on Software Aging and Rejuvenation (WOSAR 2021)*, Wuhan, China, October 25-28, 2021.
  - ***Abstract:*** *The exploding popularity of microservice based applications is taking companies to adopt them along with cloud services to support them. Containers are the common deployment infrastructures that currently serve millions of customers daily, being managed using orchestration platforms that monitor, manage, and automate most of the work. However, there are multiple concerns with the claims put forward by the developers of such tools. In this paper, we study the effects of aging in microservices and the utilisation of faults to accelerate aging effects while evaluating the capacity of Kubernetes to detect microservice aging. We consider three operation scenarios for a representative microservice-based system through the utilization of stress testing and fault injection as a manner to potentiate aging in the services composing the system to evaluate the capacity of Kubernetes mechanisms to detect it. The results demonstrate that even though some services tend to accumulate aging effects, with increasing resource consumption, Kubernetes does not detect them nor acts on them, which indicates that the probe mechanisms may be insufficient for aging scenarios. This factor may indicate the necessity for more effective mechanisms, capable of detecting aging early on and act on it in a more proactive manner without requiring the services to become unresponsive.*

Contents

1 Introduction **1**
- 1.1 Contributions . . . 3
- 1.2 Document Structure . . . 4

2 Background and Related Work **5**
- 2.1 Microservices Architecture . . . 5
  - 2.1.1 Microservices vs Monolithic Applications . . . 6
  - 2.1.2 Microservices Setups . . . 8
- 2.2 Containers . . . 10
  - 2.2.1 Definition . . . 11
  - 2.2.2 Docker . . . 12
  - 2.2.3 Kubernetes . . . 13
  - 2.2.4 KubeEdge . . . 15
  - 2.2.5 Docker Swarm . . . 17
  - 2.2.6 Apache Mesos/Marathon . . . 17
  - 2.2.7 Nomad . . . 18
- 2.3 Communication Mechanisms . . . 19
  - 2.3.1 TCP Sockets . . . 19
  - 2.3.2 Websockets . . . 20
  - 2.3.3 REST . . . 20
  - 2.3.4 RPC . . . 21
  - 2.3.5 Message Queues . . . 22
  - 2.3.6 Comparison of communication mechanisms . . . 22
- 2.4 Intrusion Detection Systems . . . 22
  - 2.4.1 Detection Target . . . 23
  - 2.4.2 Detection Approach . . . 23
  - 2.4.3 Anomaly Detection Algorithms . . . 24
- 2.5 Monitoring . . . 25
  - 2.5.1 Tracing Solutions . . . 25
  - 2.5.2 Monitoring Solutions . . . 27
  - 2.5.3 Comparison of Related Systems . . . 32

3 Requirements Analysis **35**
- 3.1 User Stories . . . 35
- 3.2 Mockups . . . 36
- 3.3 Functional Requirements . . . 37
- 3.4 Non-Functional Requirements . . . 41
- 3.5 Technical Restrictions . . . 41

4 A Tool to Automate Intrusion Detection in Microservices **43**
4.1 Proposed Architecture 43
4.2 Implementation 46

5 Validation and Experimentation **53**
5.1 Validation Plan 53
5.2 Functional Testing 54
5.3 Non-Functional Requirements Validation 58

6 Project Management **65**
6.1 Methodology 65
6.2 Work Plan 66
6.3 Risk Assessment 66

7 Conclusion **75**

Appendix A User Stories and Mockups **87**

Acronyms

API Application Programming Interface. 12, 14, 17, 19, 20, 22, 28, 30, 31, 44, 45, 46, 47, 49, 51

BoSC Bags of System Calls. 24, 44, 45, 50, 59

CISUC Centre for Informatics and Systems of the University of Coimbra. 3

CLI Command-Line Interface. xvii, 4, 13, 18, 19, 36, 37, 38, 40, 44, 45, 47, 50, 54, 56, 75, 87, 88, 90

CNCF Cloud Native Computing Foundation. 12, 13, 29

COTS Commercial off-the-shelf. 28

CPU Central Processing Unit. v, vii, 1, 3, 11, 41, 53, 59, 60, 62

CRI Container Runtime Interface. 14

CSS Cascading Style Sheets. 45, 50

eBPF Extended Berkeley Packet Filter. xv, 25, 26, 27, 32, 33

GUI Graphical User Interface. 32, 33, 47

HIDS Host Intrusion Detection System. 2, 23

HTML HyperText Markup Language. 45, 47, 50

HTTP Hypertext Transfer Protocol. 9, 17, 20, 29

IDS Intrusion Detection System. 2, 22, 23, 44, 45

IoT Internet of Things. 15

JSON JavaScript Object Notation. 14, 20, 36, 38, 39, 40, 43, 44, 45, 48, 49, 51, 56, 57, 90

KNN K-Nearest Neighbour. 24

ML Machine Learning. 24

NIDS Network Intrusion Detection System. 2, 23

OCI Open Container Initiative. 12, 13

OS Operating System. 12, 25, 32

PNG Portable Graphics Format. 39

RAM Random Access Memory. 1, 12, 58, 59

REST Representational State Transfer. 5, 8, 9, 12, 17, 19, 20, 22, 28, 31, 43, 45, 46, 49

RPC Remote Procedure Call. 19, 21

SOAP Simple Object Access Protocol. 19

STIDE Sequence Time-Delaying Embedding. 24, 44, 45, 50

SVM Support Vector Machines. 24

TCP Transmission Control Protocol. 19, 22

UI User Interface. 18

VM Virtual Machine. 1, 11, 12

YAML YAML Ain't Markup Language. 14, 30, 58

List of Figures

2.1 Monolithic architecture vs Microservices architecture from [1]. . . . 7
2.2 TeaStore service architecture from [26]. 9
2.3 Sock Shop service architecture from [27]. 10
2.4 TrainTicket service architecture from [28]. 11
2.5 Virtualization in virtual machines and containers from [36]. 12
2.6 Docker architecture from [41]. 13
2.7 Kubernetes architecture from [46]. 15
2.8 KubeEdge architecture from [48]. 16
2.9 Docker Swarm architecture from [49]. 17
2.10 Apache Mesos/Marathon architecture from [52]. 18
2.11 Nomad architecture of a single region from [55]. 19
2.12 gRPC architecture from [66]. 21
2.13 Strace workflow from [85]. 26
2.14 Sysdig workflow from [85]. 26
2.15 Extended Berkeley Packet Filter (eBPF) workflow from [87] 27
2.16 Prometheus architecture from [98]. 29
2.17 Falco architecture from [13]. 30
2.18 KubAnomaly architecture from [100]. 31

3.1 Mockup of the dashboard page. 37

4.1 General Architecture. 44
4.2 IDS Architecture. 46
4.3 Dashboard page. 52

5.1 Experimental Setup of the Non-Functional Requirements Validation. 58
5.2 Locust workloads. Scenarios 1 and 3 used the constant-wl and Scenarios 2 and 4 used the variable-wl. 60
5.3 System calls collected 61
5.4 Requests Throughput 61
5.5 Median Response Time and 95% Percentile Response Time 61
5.6 CPU Usage - *µDetector* machine (first graph), CPU Usage - Master node machine (second graph) and CPU Usage - worker-1 node machine. 63
5.7 Memory Usage - *µDetector* machine (first graph), Memory Usage - Master node machine (second graph) and Memory Usage - worker-1 node machine (third graph) 64

6.1 Gantt chart for the expected work plan. Grey bars represent a major chapter and orange bars represent smaller task. Each bar has a number that corresponds to the duration in days. 67
6.2 Gantt chart for the actual work plan. Grey bars represent a major chapter and orange bars represent smaller task. Each bar has a number that corresponds to the duration in days. 68
6.3 Risk Exposure Matrix. 73

A.1 Mockup of the dashboard page. 92
A.2 Mockup of the alarms page. 92
A.3 Mockup of the resources page. 93
A.4 Mockup of the help page. 93

List of Tables

2.1 Comparison of Related Systems. 33

3.1 System Functional Requirements - Command-Line Interface (CLI). 38
3.2 System Functional Requirements - Dashboard. 39
3.3 System Functional Requirements - Optional. 40
3.4 Technical restrictions of the project. 41

4.1 API Server Endpoints 48

5.1 General case tests. 55
5.2 Functional Requirements Validation - CLI case tests. 56
5.3 Functional Requirements Validation - Dashboard case tests. 57
5.4 Experimental Scenarios 59

6.1 Risk Assessment for the project at the beginning of the first semester. 70
6.2 Risk Assessment for the project at the end of the first semester/beginning of the second semester. 71
6.3 Risk Assessment for the project at the end of the second semester. . 72

Chapter 1

Introduction

In recent years, the software development industry has witnessed a shift from the traditional large monolithic application where components are strongly coupled, to their decoupling resulting in a collection of relatively small and independent services where each one provides a unique business capability while communicating with each other using lightweight mechanisms [1]. Applications that follow these principles are said to follow a **microservice-based architecture**. While working with a small application, the monolithic architecture naturally appears as a simple and efficient solution. However, as the complexity of the application increases, changes require the entire monolith to be rebuilt and deployed, teams become dependent on a single technology stack, modularity becomes tough to achieve and scaling the entirety of the application instead of single components is usually the only alternative [1]. The consequences reflect across the whole software development life cycle translating into increased costs for the companies. Microservice-based applications allows businesses and organizations to mitigate the problems related to monolithic applications and achieve higher **availability**, **flexibility** and **scalability** of resources on cloud infrastructures [2]. In this architectural style, services comprise smaller and independent codebases that allow introducing automation for faster deployments. Additionally, the failure of a single module does not affect the entirety of the application. This catalyzes the delivery of software products to maximize profits by automating the software development life cycle.

Microservice-based applications usually **leverage container technologies** to aid in the deployment of their services [3]. Containers are packages that contain software and bundle up all the dependencies needed to run an application. They differ from a Virtual Machine (VM) in the sense that they only virtualize the user space of an existing operating system, whereas VMs virtualize an entire machine including Central Processing Unit (CPU), Random Access Memory (RAM), file systems and network resources. Containers adoption has been taking off recently due to Docker release in 2013 [4] allowing users to easily manage containers and bridging the gap between developers and operations teams to ship software. Also, their lightweight properties allow them to achieve fast instantiation and minimize the use of resources as when compared to VMs, containers work with sizes in the order of megabytes instead of gigabytes. The usage of containers al-

lows developers to efficiently build applications that run anywhere using fewer resources and still maintain the principles of the microservices approach where application components can be deployed and scaled more granularly [5].

Still, with the continuous growth of applications, managing containers becomes a difficult task. It is not feasible to manage individual containers. Therefore, the use of tools and methods that automate and simplify this cumbersome process becomes a fundamental necessity. Orchestration platforms such as **Kubernetes** [6], **Apache Mesos** [7] or **Docker Swarm** [8] emerged as an approach to solve this problem and provide support to the infrastructure of cloud applications. They usually follow a master-slave approach, where a small number of master nodes controls the majority of the other nodes, referred to as slave or worker nodes. Orchestrators add a layer of abstraction that implements a large set of features, such as service discovery and load balancing.

These functionalities try to improve the process of managing and deploying containers. However, even though Kubernetes provides features for creating a secure cluster, it lacks built-in security and security decisions usually rely on the operator to ensure the containers and code running on the cluster are safe [9].

Furthermore, edge computing solutions such as KubeEdge which extends Kubernetes capabilities to host at Edge (i.e., closer to the user and the data source) are being considered to complement and improve the current cloud infrastructure increasing performance and minimizing the need for data to be processed in remote data centers.

Stemming from the fact that microservices and containers technologies have rapidly gained popularity [4], new security challenges arise in both cloud and edge environments. It is important to make sure containers and applications running inside containers are safe from threats and operate in a secure environment. The use of vulnerable container images, privilege escalation and poor container isolation are examples of security risks that can harm an organization [10]. Moreover, edge computing expands the potential attack surface by storing sensitive data stored across more systems [11]. Therefore, security plays an important role in these environments and techniques based on Intrusion Detection are a possible solution to approach the security concerns of microservices and containers.

Intrusion detection is defined as the process of monitoring events occurring in a computer system or network and analyzing them for signs of intrusions, defined as attempts to compromise the confidentiality, integrity, availability, or bypass the security mechanisms of a computer or network [12]. An Intrusion Detection System (IDS) automates this process and when it comes to where the detection takes place, IDSes can be classified as Network Intrusion Detection System (NIDS) or Host Intrusion Detection System (HIDS). NIDSes analyze packets moving across the network and IDSes analyze system configurations and application activity on an individual machine. Even though there is a vast set of solutions for Intrusion Detection in general, the number of options drastically diminishes when it comes to the application of intrusion detection systems in microservices-based environments that make use of containers technologies. In this domain, there are few solutions and the ones available are either in early versions or are distributed

as paid services by specialized companies. Current open-source solutions such as Falco [13] use predefined rules which transfer the responsibility to the user. Therefore, novel automated approaches are encouraged to improve security in microservices applications and further contribute to the observability and monitoring stack of these systems.

This work focuses on designing and developing an intrusion detection tool for microservice-based applications called *μDetector*. This **Host-based Intrusion Detection** tool provides monitoring capabilities by implementing proof-of-concept intrusion detection techniques researched at Centre for Informatics and Systems of the University of Coimbra (CISUC) and automates their functioning for Kubernetes and KubeEdge deployments. Through the use of a command-line interface or a web dashboard, the user can interact with the tool and provide a configuration of the system he wants to monitor. The tool will then configure the necessary probes to gather data in the form of system calls and transfer them to the Intrusion Detection module of the tool. This module uses anomaly detection techniques to control both the training and detection phase. It works by searching for deviations in the systems calls collected, following the principle that something that is not normal is considered to be suspicious, instead of looking for predefined signatures.

To validate the tool and certify that *μDetector* meets the requirements, the tool was deployed on Sock Shop, a microservice-based application. Kubernetes served as the infrastructure and 1 Master Node and 3 Worker Nodes were used, where one of them was operating on the Edge with the help of KubeEdge. Functional tests were made to make sure the requirements were fulfilled. Additionally, performance and scalability experiments were conducted to demonstrate our solution is scalable and performance-efficient. We used Locust to simulate users accessing the Sock Shop application and collected metrics including CPU and memory usage, requests throughput and response time, and system calls per second. The results showed that the performance of the machines of the Kubernetes cluster is not affected and there is a negligible impact on the proper functioning of the microservices application: in scenarios where there were over 100 000 system calls being collected per second, the CPU and memory usage of the worker nodes did not exceed 10% of the total resources available. Also, there were no failed requests and no visible difference in the response time of the requests.

The *μDetector* tool allows development and operation teams to improve the security and reliability of their solution in an automated fashion by monitoring their microservices-based application in real-time. It also leverages proof-of-concept intrusion detection techniques, which present as a further step in strengthening container security and microservices applications in cloud environments.

1.1 Contributions

The main contributions of this work are:

- **Design and Implementation of an intrusion detection tool that monitors**

microservice-based applications with the help of proof-of-concept techniques developed by the research group. After a configuration is provided and the probes are running on cluster nodes, the tool automatically collects system calls from the hosts and uses anomaly-based intrusion detection algorithms to trigger alarms in real-time.

- **Development of a Web Dashboard that allows users to communicate with the tool.** While the CLI focuses on simplicity and provides a fast and efficient way to run commands, the Web Dashboard improves usability by displaying charts and information more appealingly. In both interfaces it is possible to configure the monitoring of the application, view the status of the monitoring phases, view the alarms generated and resources of the cluster.
- **Extension of the tool to Edge Computing.** The *µDetector* tool was designed to operate in cloud and edge environments. Therefore, it is possible to install the probes on nodes from both Kubernetes and KubeEdge deployments, extending the domain of the solution to edge environments.
- **Validation of the tool.** The *µDetector* tool was subjected to functional, performance and scalability tests. The goal is to prove the requirements were fulfilled and that the tool did not impact the normal functioning of the microservices application.

1.2 Document Structure

The remainder of this document is structured as follows:

Chapter 2 presents the fundamental concepts and technologies related to this work. This includes an overview of microservices, containers and containers orchestrators. edge computing, communication mechanisms, intrusion detection, monitoring concepts and related systems are also presented in this chapter.

Chapter 3 documents and explains the whole requirement gathering process. This includes the user stories, mockups, requirements and technical restrictions of the project.

Chapter 4 explains the proposed architecture and the implementation details of the *µDetector* tool.

Chapter 5 describes the validation plan followed to certify the tool meets the requirements and the respective results.

Chapter 6 documents the methodology used during this thesis. It also includes the work plan for both the first and second semesters and a risk assessment for the project.

Chapter 7 summarises the work performed during the course of this dissertation.

Chapter 2

Background and Related Work

In this chapter, we discuss important concepts, technologies and work related to the project. This information comes from an extensive analysis of articles, web searching and knowledge acquired throughout the academic years. In particular, we focus on the microservices architecture and compare it to the monolithic approach. Afterward, we present microservices applications that can be used to run tests and benchmarks. Next, we explain containers concepts and present technologies such as Docker and Kubernetes. We also focus on communication mechanisms including Representational State Transfer (REST) and gRPC. Finally, we explain concepts related to intrusion detection and present related monitoring systems.

2.1 Microservices Architecture

In this section, we provide some background related to the microservices architecture. We explain the concept of microservices and compare it to monolithic applications, and also present some open-source microservices setups that try to simulate real-world applications.

The earliest documented reference to microservices occurred in 2005 when Dr. Peter Rodgers used the term "Micro-Web-Services" in a presentation at the Web Services Edge conference [14]. A few years later, in 2011, at a software architecture workshop held near Venice, Italy, participants used this term to describe an architectural style based on a collection of small and independent services and, in the following year, they formally adopted it as the most appropriate term for the architecture. It was also around this time that companies like Netflix started using this architecture describing it as a "fine-grained SOA" [1].

Microservices can be described as a type of software architecture that splits an application into a collection of small, lightweight and independent services where each one provides a unique business capability while communicating with each other through lightweight mechanisms [1]. It is inspired by service-oriented computing and emerged as an alternative to the traditional monoliths where all modules of an application are bound together [15]. Microservices try to solve the

problem of building and maintaining complex applications by making the components of an application modular and independent.

In the past years, microservices have been on the rise and have gained a significant amount of supporters. A survey [16], conducted in 2020, showed that around 76% of the respondents (n=1502) claim they have adopted microservices in their organization.

2.1.1 Microservices vs Monolithic Applications

A monolith is described as a software application whose modules cannot be executed independently [15]. Usually, this is a good approach for simple or small-sized projects because it enables software teams to quickly write code and test the application. Since the application is viewed as a single package, the deployment is relatively easy as is not required to orchestrate modules separately. The scaling of these types of applications is usually straightforward because we can simply increase the number of resources or run several instances behind a load balancer.

However, problems arise when businesses start growing and larger teams start working on the monolith. Due to the strong cohesion of the modules that constitute the application, scaling, maintaining and accelerating deployments becomes a challenging task [15]. For example, as the software matures, more and more features are added and the monolithic application is likely to become complex and heavy which impacts the speed at which teams can deliver new releases and improvements. This usually leads to difficult challenges such as refactoring the whole monolith and the need to rethink the architecture. Furthermore, technologies evolve at a relatively fast pace. Some require frequent updates and others become obsolete forcing teams to change or update the technology stack [15].

Figure 2.1, explains the difference between the monolithic architecture and microservices architecture when it comes to scaling. Owing to the strong cohesion of components in a monolith, when it is required to scale the application, one must scale the entire system. In a microservices architecture, we can scale specific services, thus, making better use of resources. In addition, the loosely coupled and modular characteristics of microservices allow developers to have the freedom to choose from different programming languages and frameworks for each service. They can promptly develop a small portion of an application (i.e., a service) without interfering with other parts or needing to refactor the whole infrastructure.

Contrasting with the monolithic solution, an easier deployment is also possible to achieve using microservices because each module is independently deployable and fault tolerance should be easier to achieve as long as the full application does not crash when a service is not responding. To achieve this there are several microservices patterns that describe a possible approach to a certain problem. For example, to improve the reliability of a system we can use the microservices circuit breaker pattern [17] which prevents the failure of one service from having a cascade effect on other services throughout the application.

Microservices combine well with the DevOps culture and increase the speed at which engineering teams build and deploy software [18]. DevOps is a set of practices, tools, and a cultural philosophy that automates and integrates the processes between software development and IT teams [19]. Fundamentally, it tries to improve the communication between Development and Operation teams because in the traditional software development model these teams are organizationally and functionally apart [19]. It makes use of practices such as **Continuous Integration** and **Continuous Delivery** which uses tools and automated processes to quickly deliver software to the customers.

The benefits of microservices have been proved by some of the largest technology companies, such as Netflix, which started with a monolithic application and later migrated to microservices precisely to overcome the limitations of the monolithic architecture [20].

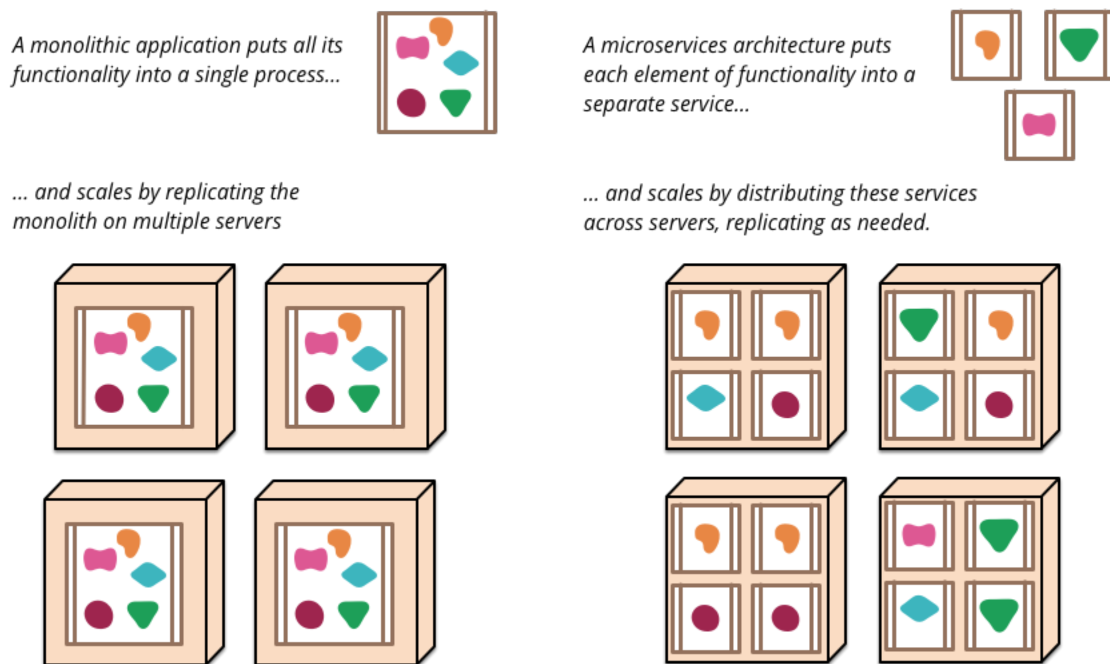


Figure 2.1: Monolithic architecture vs Microservices architecture from [1].

Nevertheless, the microservice architecture also has its drawbacks. Firstly, when starting a new project, it is extremely hard to design and conceptualize an application separated into small independent services. Secondly, doing general changes that affect the entire application becomes a strenuous task even though it is quick and easy to perform small changes. Finally, because services communicate with each other through a network, the communication is subject to failures and delays that impact the system.

The key takeaway is that microservices arise from the necessity to solve common problems identified in traditional monoliths. However, it should not be seen as a replacement for the monolithic architecture but as an alternative instead. Despite the ease of starting new projects with a monolithic architecture, microservices are likely to be the software architecture style of choice in the long run. Even though it does require extra effort in terms of manageability, the benefits it provides, in

the long run, are far greater than the required investment [21].

2.1.2 Microservices Setups

Microservices applications composed of many services are extremely complex to build. We can take a look at real-world applications, usually comprised of hundreds of microservices, that took years to build and required a huge investment. Examples of these applications are Netflix, AWS and Twitter [22]. The dimension of these applications and the fact that they are not open-source makes it difficult (and in most cases impossible) to conduct experiments on these applications. To overcome this problem, smaller microservices applications were developed by academic institutions and other organizations.

There are several open-source microservices setups (often referred to as "testbeds" or "benchmark applications") that try to simulate real-world applications built on a microservices architecture. Their main limitation is usually the size of the applications which usually contain a relatively small number of services.

Examples of these setups include TeaStore [23], Sock Shop [24] and TrainTicket [25].

TeaStore

TeaStore [23] is a microservices testbed that simulates the functioning of an online tea store and is used for benchmarking and testing [26]. It is composed of six different services: WebUI, Registry, Image-Provider, Auth, Persistence and Recommender, each one with its docker image. Every request is done using REST and all the services are deployed on Apache Tomcat as web services.

The services are depicted in Figure 2.2 and an explanation of their functioning is presented below.

- **Registry**. Its main objective is to know which services are alive and which ones are dead. By receiving keep-alive messages from the other services, it can keep a list of their IP addresses, hostnames and ports;
- **WebUI**. Handles all the requests directed to TeaStore. It uses Java Server Pages to provide the web pages;
- **Image provider**. When it starts, it queries the Persistence service for all the images. Then, it waits for the WebUI requests for images. It uses a least frequently used cache to get better performance;
- **Authentication**. Does the authentication of the users that want to login on TeaStore. It also keeps track of the session data of the users that already logged in;
- **Recommender**. The first instance queries Persistence for a generated dataset to train, while new Recommenders ask the remaining for their data-set

to train itself. It uses a rating algorithm to recommend products for the users;

- **Persistence**. It receives requests and queries the database for the information that is requested. It also uses a cache to reduce response times.

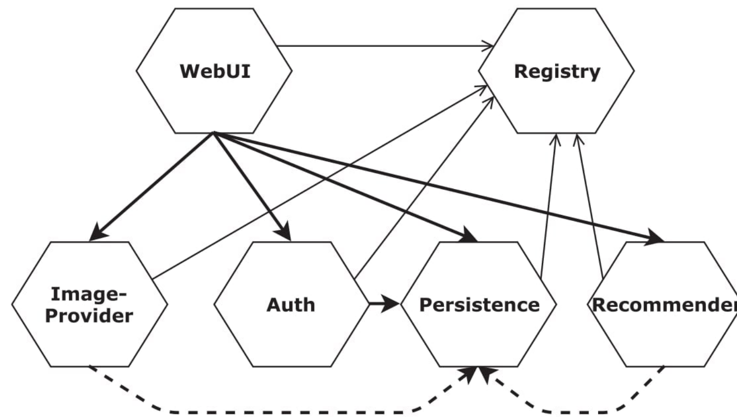


Figure 2.2: TeaStore service architecture from [26].

Sock Shop

Sock Shop [24] is a microservices application that simulates the user-facing part of an e-commerce website that sells socks [27]. It was developed using Spring Boot, Go, and Node.js, with the specific aim of aiding the demonstration and testing of existing microservices and cloud-native technologies. All services communicate using REST over Hypertext Transfer Protocol (HTTP). As we can see in Figure 2.3 it is made up of 8 services: Front-End, Order, Payment, User, Catalogue, Cart, Shipping and Queue-Master.

The services are explained below:

- **Front-end**. This service puts together all of the microservices and exposes the Sock Shop to the users. It is written in Node.js.
- **Order**. This service provides ordering capabilities. It is written in Java and .NET Core and is associated with a MongoDB instance.
- **Payment**. This service provides payment services. It is written in Go.
- **User**. This service covers user account storage, including cards and addresses. It is written in Go and is associated with a MongoDB instance.
- **Catalogue**. This service provides catalog/product information. It is written in Go and is associated with a MySQL instance.
- **Cart**. This service provides shopping carts for users. It is written in Java and is associated with a MongoDB instance.

- **Shipping**. This service provides shipping capabilities. It is written in Java.
- **Queue-Master**. This service provides reading from the shipping queue. It will spawn new docker containers that simulate the shipping process. It is written in Java.

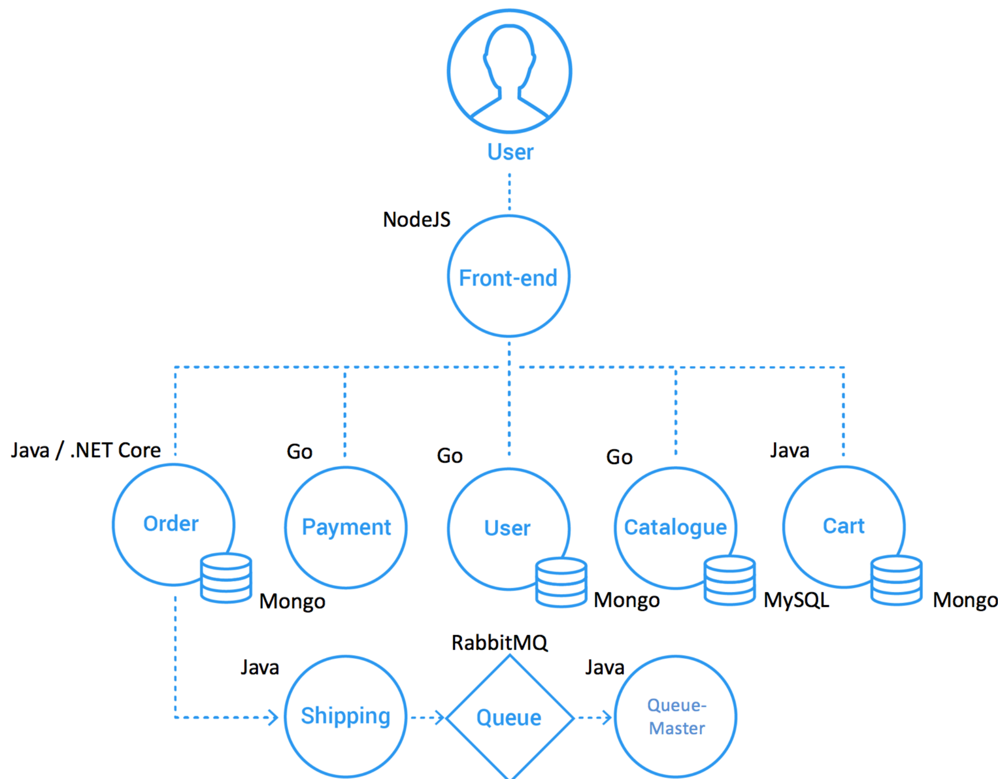


Figure 2.3: Sock Shop service architecture from [27].

TrainTicket

TrainTicket [25] is a booking system for trains based on the microservices architecture [28]. It is a medium-size benchmark system composed of 41 microservices and is written in Java, Go, Node.js and Python. Even though it is considered a medium-size benchmark it is the larger open-source microservice application known to date. Due to the high amount of services, we will not go into detail on the TrainTicket architecture. However, Figure 2.4 shows all the services that compose the application.

2.2 Containers

Microservices applications can be deployed in cloud environments using container technologies [29]. In this section, we explain the concept of containers and present the most used technologies used in the industry.

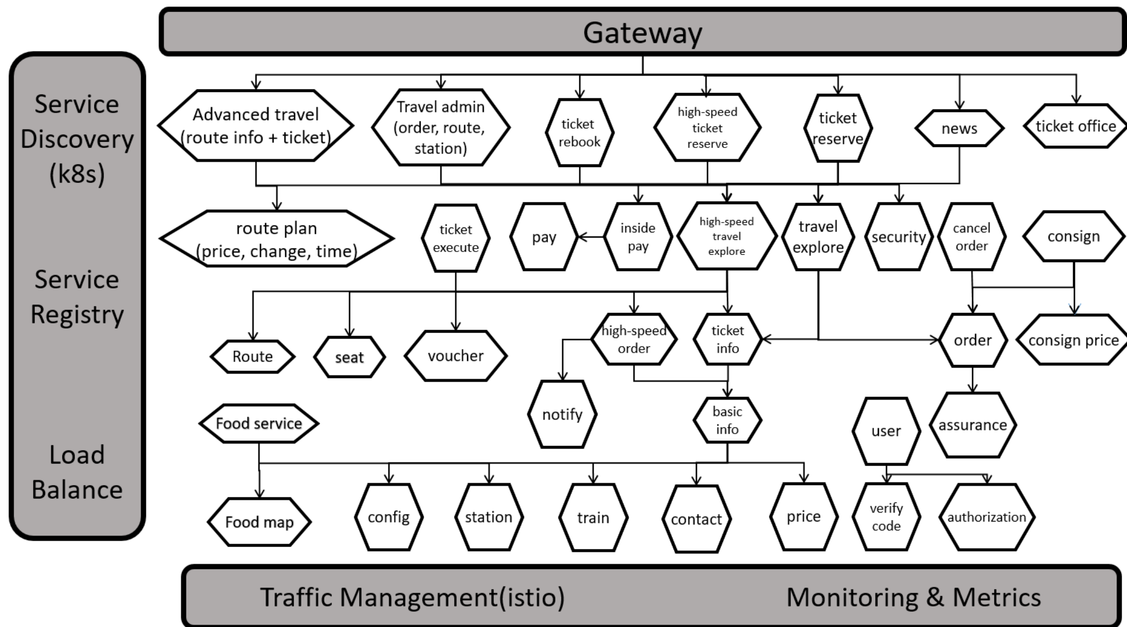


Figure 2.4: TrainTicket service architecture from [28].

2.2.1 Definition

The idea of a container first appeared in the 1970s, when the concept was first employed on Unix systems to better isolate application code [30]. This isolation could be achieved through the use of the command chroot which allowed the user to change the root directory for the currently running process and its children but they still share the resources, like users, hostname and IP address [31].

In 2000, FreeBSD extended chroot to FreeBSD Jails allowing the possibility to virtualize users, and network subsystems, among others. In the following years, other technologies like Linux VServer, Solaris Containers and Open VZ were introduced [30].

In 2006, Google launched Process Containers which were designed for limiting, accounting and isolating resource usage (CPU, memory, disk I/O, network) of a collection of processes. It was renamed "Control Groups (cgroups)" the next year and got merged to the Linux kernel [32]. Linux namespaces and cgroups were then used together to implement the LXC (LinuX Containers) technology, in 2008. Linux containers did not require dedicated guest operating systems, instead, they share the host operating system kernel providing less baggage when compared to Virtual Machines (VMs) [33].

In 2013, Docker, Inc. (former dotCloud), launched Docker [34]. It leveraged existing containers concepts to allow users to easily manage containers which lead to massive adoption of the container technology [30].

Essentially, containers are packages that contain software and bundle up all the dependencies needed to run an application [35]. They differ from a VM in the sense that they only virtualize the user space of an existing operating system whereas VMs virtualize an entire machine [5] including Central Processing Unit

(CPU), Random Access Memory (RAM), file systems and network resources. As shown in Figure 2.5, VMs use hardware virtualization and containers use Operating System (OS) virtualization [36]. This difference in the type of virtualization allows containers to "reuse" existing hardware and virtualize only the software layers above the OS level, becoming thinner than virtual machines. The lightweight properties of containers allow them to use far fewer resources than virtual machines [36]. Additionally, they can be instantiated very quickly when compared to VMs, which is extremely important in cloud environments where they try to guarantee the on-demand availability of resources [37, 38]. Organizations, such as Cloud Native Computing Foundation (CNCF) and Open Container Initiative (OCI), try to create open industry standards around container formats and also support open-source projects to make them even more portable [39] and available to everyone [40].

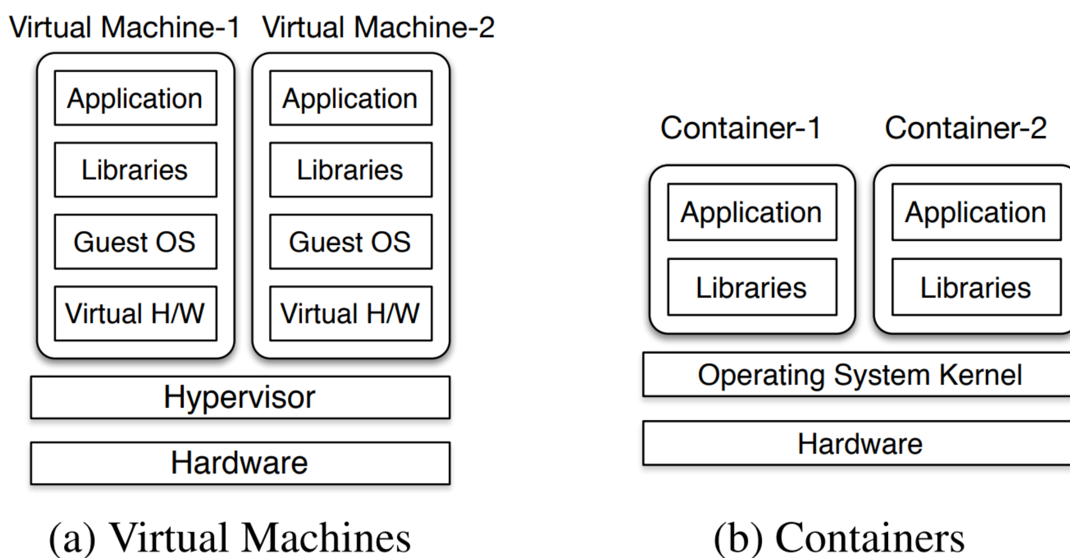


Figure 2.5: Virtualization in virtual machines and containers from [36].

2.2.2 Docker

Docker (also referred to as "Docker Engine") [41] is a software platform that follows a client-server architecture and makes use of containers technology. The isolation achieved by Docker is made possible thanks to three concepts: **Namespaces** which limit what the container can see; **Control groups** (cgroups) that limit an application to a specific set of resources allowing the Docker Engine to share available hardware resources to containers enforcing limits and constraints; **Union file systems** that operate by creating layers, Docker image is made up of file systems layered over each other making it very lightweight and fast.

Docker architecture is shown on Figure 2.6. It is composed of five components described below:

- **Docker Daemon.** Responds to the requests from the Docker REST Ap-

plication Programming Interface (API) and acts accordingly by managing Docker objects such as images, containers, networks and volumes. It is considered the server in the Docker client-server architecture. On a deeper level, the Daemon communicates with **containerd** through gRPC calls. Containerd is a container runtime, responsible for pulling images, managing network, storage and running containers using **runc**. Runc is a tool for spawning and running containers on Linux according to the OCI which creates open industry standards around container formats and runtimes.

- **Docker Client.** Allow users to interact with Docker daemon through a Command-Line Interface (CLI).
- **Docker Host.** Provides an environment to execute and run applications. It comprises of the Docker daemon, Images, Containers, Networks, and Storage.
- **Docker Images.** Read-only templates that contain information about the application and its dependencies. Containers are created and executed from these images. Docker images can be described in a Dockerfile.
- **Docker Registry.** Stores Docker images privately or publicly (e.g., Docker Hub).

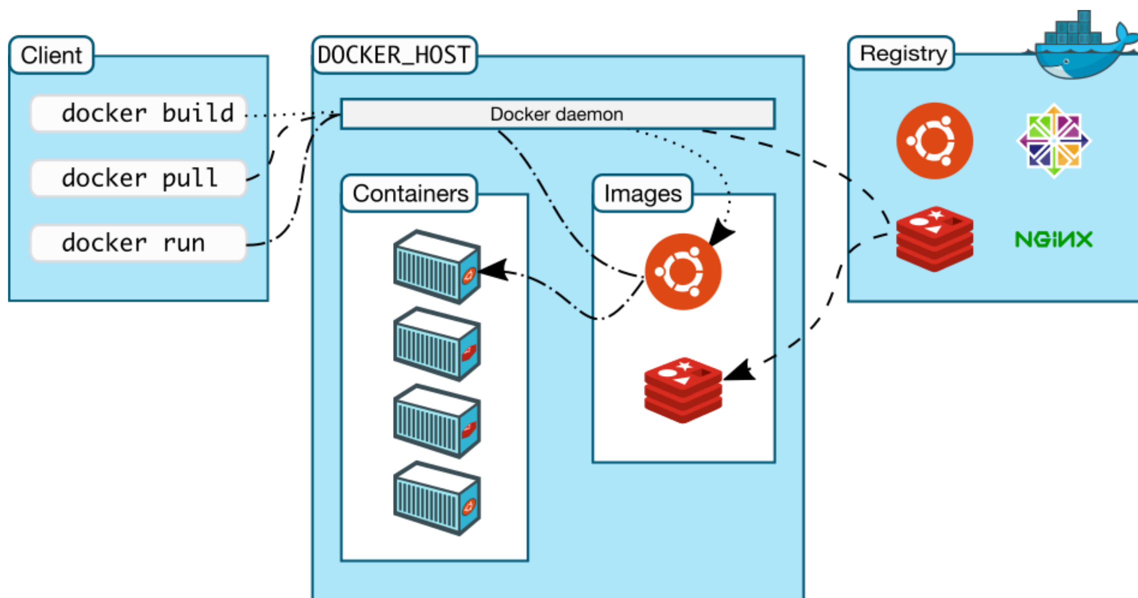


Figure 2.6: Docker architecture from [41].

2.2.3 Kubernetes

Kubernetes [6], also known as k8s, is an open-source system for automating deployment, scaling, and management of containerized applications [42]. It was originally developed by Google based on an internal cluster management system called Borg [43]. It is currently maintained by the CNCF [40].

Kubernetes allows users to build and scale container-based applications, schedule containers across a cluster and manage their health over time eliminating many of the manual processes involved in deploying and scaling containerized applications across public, private, or hybrid clouds [44]. A cluster is composed of nodes and is deployed by describing a configuration in a file using either a YAML Ain't Markup Language (YAML) or JavaScript Object Notation (JSON).

In Kubernetes, pods are the smallest deployable units of computing that can be created and managed [45]. A Pod contains one or more containers, with shared storage and network resources, and a specification for how to run the containers [45]. Kubernetes supports multiple container runtimes because it uses Container Runtime Interface (CRI) which defines an API between Kubernetes and the desired container runtime. Therefore, container runtimes such as containerd (which came from Docker) and CRI-O (developed by RedHat, IBM, Intel, etc.) are supported by Kubernetes.

Pods are deployed on nodes (which may be a virtual or physical machine) and are controlled by the Control Plane comprised of the etcd, scheduler, API server and controller manager. Typically, as depicted in Figure 2.7, we have a master node (or more) responsible for the Control Plane and several worker nodes that contain pods that run the workload (i.e., do the "hard work"). Together they form a cluster. Kubernetes components are described in detail below:

- **API server**. Exposes the Kubernetes API so that users can interact with it using a client such as a command-line tool (e.g., `kubectl`) or the Kubernetes dashboard. A request made to the API by the user is validated and then forwarded to another process providing an initial barrier of authentication.

- **Controller manager**. Runs controller processes that are constantly trying to detect changes in the cluster and tries to recover from undesired states. The information is then passed to other processes like the Scheduler

- **etcd**. The key-value store used by Kubernetes to store cluster data such as the resources available. Data produced by applications running in Kubernetes is not stored in etcd.

- **Scheduler**. Manages the scheduling of pods, more specifically, it decides which nodes will host newly created pods, taking into account the available resources.

- **kubelet**. Works on each node of the cluster and executes requests forward by other processes such as pod scheduling. It also makes sure the pods are running and healthy.

- **kube-proxy**. Runs on each node in the cluster and forwards requests to and from the pods ensuring communication in the network.

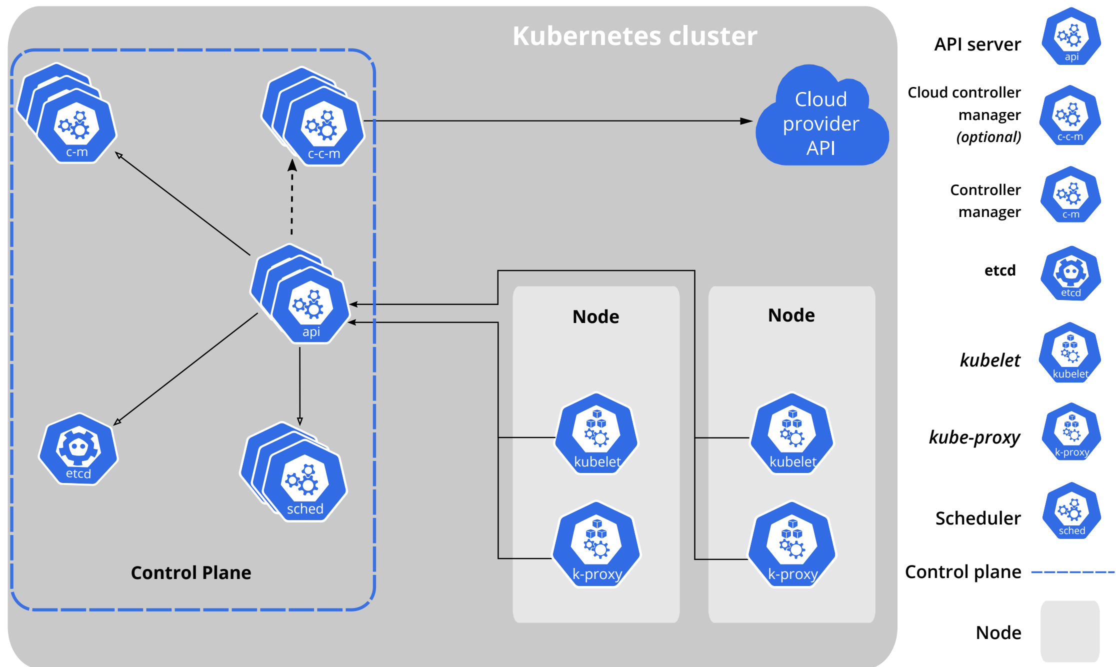


Figure 2.7: Kubernetes architecture from [46].

2.2.4 KubeEdge

Edge deployments are becoming more and more common due to the increasing rise of Internet of Things (IoT) devices. Edge computing further improves the current cloud infrastructure by increasing performance and minimizing the need for data to be processed in remote data centers because they are closer to the user and the data source. KubeEdge [47] is an open-source system that extends Kubernetes to deploy nodes at the Edge. It provides core infrastructure support for networking, application deployment and metadata synchronization between cloud and edge [48]. KubeEdge is divided in the **CloudCore** and the **EdgeCore**. Figure 2.8 presents the architecture of KubeEdge. In more detail, the CloudCore has two controllers responsible for the management of devices (device controller) and pods and nodes on the edge (edge controller). The CloudHub communicates the changes that happen at the cloud side with the EdgeCore (in particular the EdgeHub), through web sockets. On the edge side, the EdgeHub communicates with the CloudHub to sync cloud-side resource updates to the edge and reports changes on the edge-side to the cloud. The EventBus on the EdgeCore allows an MQTT client to interact with MQTT servers. Still, on the EdgeCore, DeviceTwin provides query interfaces for applications and is responsible for storing device status and syncing device status to the cloud. The MetaManager acts as a message processor between edged and EdgeHub and communicates with an SQLite database. Finally, Edged works as a daemon that runs on edge nodes and manages containerized applications.

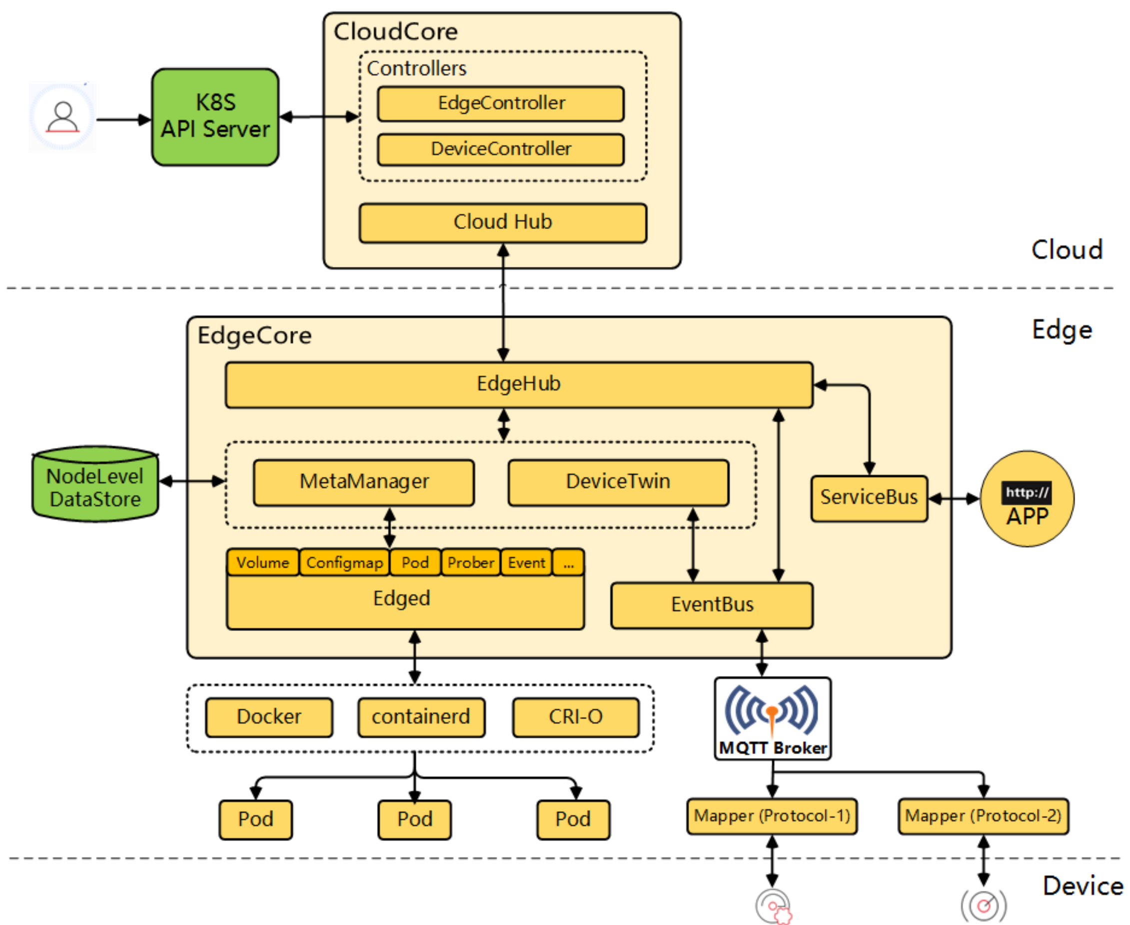


Figure 2.8: KubeEdge architecture from [48].

2.2.5 Docker Swarm

Docker introduced in version v1.12.0 a cluster management and orchestration feature called Swarm mode. Instead of deploying a standalone container, Swarm mode allows developers to deploy containers in nodes called workers and perform orchestration and cluster management using managers nodes. A Docker host can be a manager, a worker, or both.

Manager node's responsibility is to maintain cluster state, schedule services and serve swarm mode HTTP API endpoints [49]. An implementation of Raft[1] is used if there is more than one manager node. Raft is a fault-tolerant algorithm that tries to guarantee consensus via an elected leader [50]. Worker nodes, on the other hand, run tasks instructed by the manager nodes as depicted in Figure 2.9.

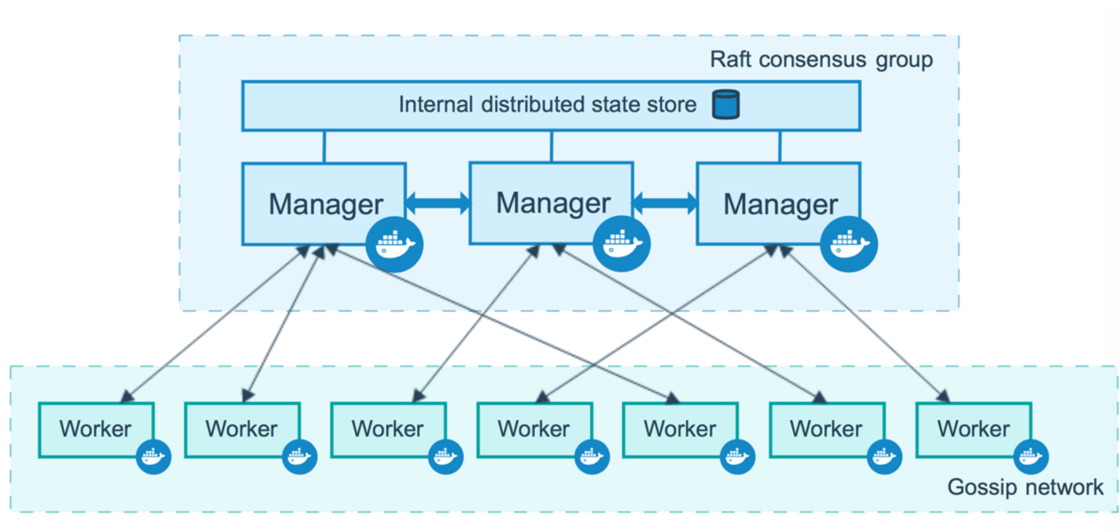


Figure 2.9: Docker Swarm architecture from [49].

2.2.6 Apache Mesos/Marathon

Apache Mesos [7] began as a research project developed by UC Berkeley RAD Lab Ph.D. students and later presented at the Usenix Symposium on Networked Systems Design and Implementation Conference. However, it was not until 2016 that the first version was released under the Apache Software Foundation. Mesos is a cloud-native distributed systems kernel that provides applications with APIs resource management and scheduling [51]. Marathon[2] is a framework for Apache Mesos that acts as a container orchestration platform and provides features such as high availability, health checks, service discovery and load balancing. It also supports Docker and provides a REST API for scaling applications and collecting metrics. Figure 2.10 illustrates the Apache Mesos / Marathon architecture composed of: Mesos Master which manages resources in the cluster; Mesos Slave that runs agents that report to the Master; Marathon is responsible for running and

[1] https://raft.github.io/

[2] https://mesosphere.github.io/marathon/

maintaining other frameworks (such as Chronos); Zookeeper assures the cluster high availability. The user interacts with the Marathon Scheduler and Master via the CLI or User Interface (UI).

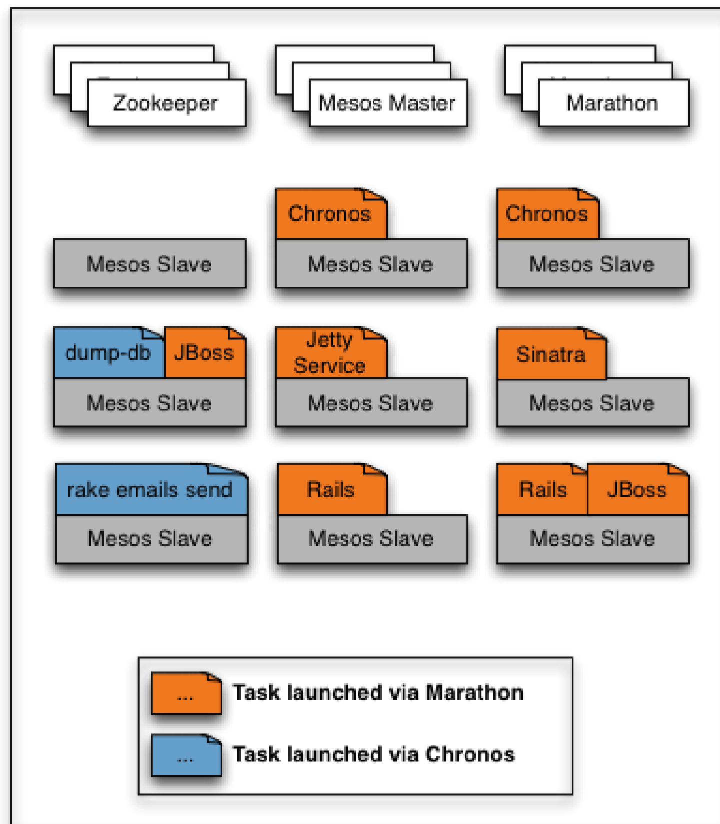


Figure 2.10: Apache Mesos/Marathon architecture from [52].

2.2.7 Nomad

Nomad [53] is a container orchestrator technology developed by HashiCorp that can deploy a mix of microservice, batch, containerized, and non-containerized applications. It has native Consul and Vault integrations and relies on plugins to execute. Nomad supports technologies such as Docker, containerd, LXC, rkt and Java. The majority of features are available in an open-source version, and some more advanced features require upgrading to Nomad Enterprise [54].

As we can observe in Figure 2.11, a Nomad region is composed of servers and clients. Regions are fully independent of each other and do not share jobs, clients, or state. They are loosely coupled using a gossip protocol, which allows users to submit jobs to any region or query the state of any region transparently.

Servers elect a single leader (who has more responsibilities) using an implementation of Raft and are responsible for accepting jobs from users, managing clients, and computing task placements.

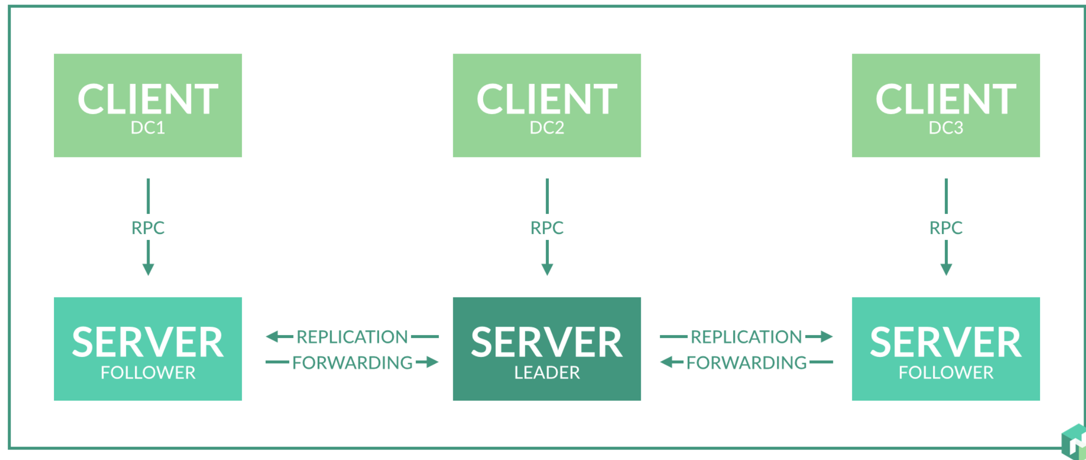


Figure 2.11: Nomad architecture of a single region from [55].

Clients communicate with their regional servers through Remote Procedure Call (RPC) to register themselves, send heartbeats, wait for new allocations and update the status of allocations. A client registers with the servers to provide the resources available, attributes, and installed drivers. Servers use this information for scheduling decisions and create allocations to assign work to clients.

Finally, users make use of a CLI or API to submit jobs to the servers that provide the set of tasks that are run by the clients [55].

2.3 Communication Mechanisms

This section presents some of the mechanisms used to communicate over distributed systems. This includes TCP Sockets, WebSockets, REST, RPC and Messaging. We discard other technologies such as Simple Object Access Protocol (SOAP) due to being outdated or less common in real-world applications. At the end of this section, a brief comparison is made between the communication mechanisms.

2.3.1 TCP Sockets

Sockets allow the communication between two different processes on the same or different machines [56]. TCP Sockets are sockets that make use of Transmission Control Protocol (TCP) to guarantee the delivery of network packets. It is an implementation of a communication mechanism that offers little abstraction thus being considered a low-level technology. This makes it possible to achieve high throughput and low latency communication between server and client.

2.3.2 Websockets

Websockets is an API [57] that makes it possible to open a two-way interactive communication session between the user's browser and a server [58]. They are primarily used (though not necessarily) in web applications that require a permanent connection to the server. It is based on the protocol with the same name. This protocol, by definition, enables two-way communication between a user agent running untrusted code running in a controlled environment to a remote host that has opted-in to communications from that code [59]. It is layered over TCP and appears as an alternative to opening multiple HTTP connections. There are several frameworks that implement websockets such as Flask-SocketIO [60] and ws4py [61].

2.3.3 REST

REST is an architectural style that provides guidelines for designing web APIs. It follows a stateless approach where each request from the client should contain all of the information necessary for the server to understand and attend to the request. It uses HTTP methods like GET, POST, PUT, and DELETE to interact with resources located on the server. REST typically uses the JSON format to transfer data. Python Web frameworks such as *Flask* [62], *Django REST* [63] and *Falcon* [64] take advantage of this architectural style and provide the necessary tools to build powerful web aplications.

Falcon is a lightweight Python Web API framework for building high-performance microservices, app backends, and higher-level frameworks. According to their documentation [64], aside from being extremely simple to use, Falcon requests performance is several times faster than most other Python frameworks. It also transfers the responsibility of the implementation details to the developer allowing greater flexibility.

Flask is a micro framework written in Python designed to be simple, lightweight and powerful. Similar to Falcon, most decisions and implementation details rely solely on the developer. There are several extensions to Flask developed by the python community that add value to the base functionalities. Flask makes use of the Jinja template engine and the Werkzeug WSGI toolkit [62]. Flask has a more mature community due to being more popular when compared with Falcon.

Django is a high-level Python web framework that encourages rapid development and clean, pragmatic design [63]. It offers more functionalities when compared to Falcon and Flask. This comes with the downside of being a heavier framework which might not be desirable for simpler and smaller projects. Also, customizing details in the Django framework becomes a harder task due to less freedom and minimalism.

2.3.4 RPC

RPC is the synchronous language-level transfer of control between programs in disjoint address spaces whose primary communication medium is a narrow channel [65]. It is used as a form of client-server interaction where a client calls a subroutine on a server. It can be implemented in different ways. Our particular focus is on gRPC, the high-performance, open-source RPC framework initially developed by Google. It is based on a client-server model where the client can directly call methods on a server application as if it was a local object. gRPC uses protocol buffers which are an extensible and powerful mechanism for serializing structured data. It works, by first defining a service in a *.proto* file and generating the server and client code using the protocol buffer compiler. Then the server and client applications are created making it possible to execute RPC calls using generated stubs. As depicted in Figure 2.12, the client and the server may use different technologies and still communicate with each other. The steps that make the communication possible are the following:

1. A client application makes a local procedure call to the client stub containing the parameters to be passed on to the server and the stub serializes the parameters.
2. The client stub forwards the request to the local client time library which then forwards the request to the server stub.
3. The server run-time library receives the request and calls the server stub procedure which unmarshalls parameters.
4. The server stub calls the actual procedure and sends back a response to the client stub using the same process.

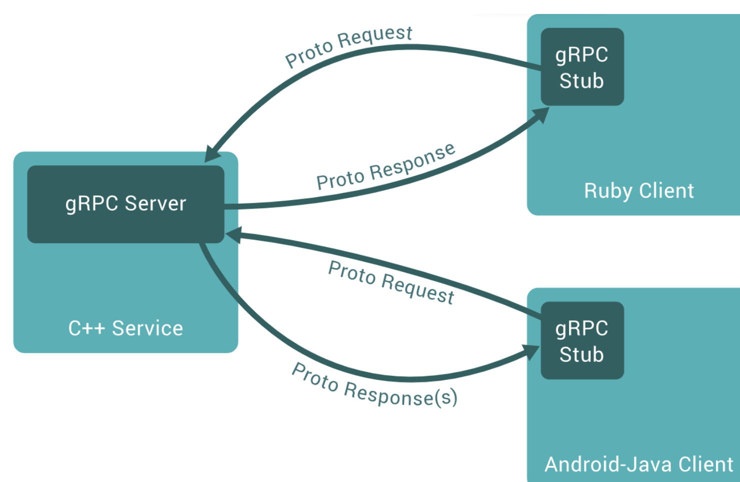


Figure 2.12: gRPC architecture from [66].

2.3.5 Message Queues

A message queue is responsible for asynchronously transferring data from one application to another. In a messaging system, producers create messages and deliver them to the message queue while consumers connect to the queue and retrieve the messages to be processed [67]. Messages are stored in a queue until the consumer retrieves them. Examples of message queues systems include RabbitMQ [68] and ZeroMQ [69].

2.3.6 Comparison of communication mechanisms

The communication mechanisms mentioned above are used in different scenarios. For the transfer of system calls, we could consider TCP sockets. Even though they achieve higher throughput, TCP sockets offer no security as it is up to the programmer to implement most of the functionalities. This can be a time consuming task and solutions such as Websockets already solve these problems while still having a high throughput [70]. Websockets add a Web "origin"-based security model for the browser and can pass through most firewalls without any reconfiguration. Websockets are a framing mechanism on top of TCP that eliminates the length limits imposed by plain TCP protocol [59]. They simplify socket programming and are also suitable for real-time applications that deal with lots of information such as the transfer of a high volume of system calls. Message queues could also be considered, however, we discard them since they have a larger overhead and are used to store messages for later use when there is no need for immediate action.

When creating APIs, gRPC and REST are the two most popular architectural styles. Both are viable solutions that could be used. gRPC offers better performance due to the use of protocol buffers to serialize data and HTTP/2 to solve latency issues. However, REST is a more generic and mature technology that provides universal browser support.

2.4 Intrusion Detection Systems

Intrusion detection is the process of monitoring system events and analyzing them for possible incidents, automated through an Intrusion Detection System (IDS) [71]. The main purpose of an IDS is to detect malicious activities and protect computers and the network infrastructure. An IDS will capture and analyze network traffic or host data to discover suspicious activities and report them to an administrator, human or computational system, which will then decide what to do to that activity [72]. To put it simply, an IDS uses rules and heuristics to help identify unauthorized host or network activity that may or may not be an intrusion. The deployment of IDSes as a security countermeasure is advised [71] and provides a good mechanism against external and internal attacks. This is extremely important in microservices scenarios where a collection of services interacts with each other through the network. It is also a challenge due to the

isolation of the services.

This section presents intrusion detection concepts such as the types of intrusion detection systems when it comes to their nature and approach. Also, we briefly present anomaly detection algorithms because it is of particular interest to this work.

2.4.1 Detection Target

The detection or monitoring target refers to the nature of the target being monitored. It can be **Host Intrusion Detection System (HIDS)**, **Network Intrusion Detection System (NIDS)** or **Hybrid**.

HIDS are usually installed on servers and their focus is on verifying the integrity of system files, monitoring unusual resource utilization or unauthorized access, or other system activity as long as it resides on the host machine [73]. They log any activities discovered to a secure database and check to see whether the events match any malicious event record listed in the knowledge base. They are often critical in detecting internal attacks directed towards an organization's servers. An example of an HIDS is OSSEC [74].

NIDS are dedicated network devices distributed within networks that monitor and inspect network traffic flowing through the devices [73]. They monitor network infrastructure to identify attacks, analyze the flow of packets, inspecting headers and contents. Instead of analyzing information that originates and resides on a host, NIDS use packet sniffing techniques to pull data from TCP/IP packets or other protocols that are traveling over the network. Most Network-based IDS log their activities and report or alarm suspicious events. An example of a NIDS is Snort [75].

Hybrid systems combine both HIDS (which monitors events occurring on the host system) and NIDS (which monitors network traffic) functionalities on the same system [73]. They monitor the host in which they are placed and also use the network interfaces the host is connected to, to monitor the ongoing traffic similar to a NIDS. They try to combine the best of the two detection target methods. An example of an hybrid IDS is Prelude SIEM [76].

2.4.2 Detection Approach

The detection approach classification uses intrusion detection principles to classify the type of intrusion. It divides the type of approach into a **Signature-based** process and an **Anomaly-based** process [77].

The **signature-based** process continuously analyzes the incoming data and attempts to identify patterns of well-known and established attacks [73]. Because they only detect known attacks, a signature must be created for every attack. The false negatives rate is higher if the attack is not identified properly and there is not a corresponding signature to that attack [78]. Therefore, this approach fails to

detect unseen attacks [73].

The **anomaly-based** process focuses on finding abnormalities in the traffic in question by following the principle that something that is not normal is considered to be suspicious [78]. Therefore, anomaly detection uses a baseline behavior to determine a "normal" state for the system and compare events that occur to that profile [73]. A new attack can be detected if it falls out of the "normal" profile. False positives alarms may appear if the "normal" profile is inaccurate or not representative of the system [78]. Overall, this approach overcomes the limitations of the signature-based approach at the cost of relying heavily on the data used during the training phase subjecting to false positives [73].

2.4.3 Anomaly Detection Algorithms

This subsection presents the most used algorithms for anomaly detection. The algorithms researched in the group focus only on anomaly detection. In particular, the algorithms researched in the group are the **Bags of System Calls (BoSC)** and **Sequence Time-Delaying Embedding (STIDE)**. However, there are others such as K-Nearest Neighbour (KNN) and Support Vector Machines (SVM). Signature Detection, which usually uses rule-based approaches, will not be covered.

The **BoSC** algorithm uses a sliding window over a collection of system calls to detect intrusions and define a baseline behavior database, which contains bags of system calls considered normal [78]. The order in which the system calls are made is not taken into account because this algorithm counts the frequency of each system call every time it appears in the sliding window [79].

The **STIDE** algorithm uses a window sliding over a collection of system calls to define a baseline behavior database that will be used in the detection phase to find possible intrusions [78]. Unlike BoSC, it maintains the original order of system calls.

KNN is a Machine Learning (ML) classifier that relies on information acquired during a training to identify anomalous events [78]. To group the events into different classes uses distance heuristics between new data and classified data and tries to find event similarities. Every time a new event is processed the K nearest neighbors to it are calculated and the most frequent label among them is assigned to the new event [80].

SVM is a classifier method that relies on information acquired during training to identify anomalous events [78]. The events collected are split into different classes separated by a hyperplane defined by several support vectors used to define boundaries [81].

2.5 Monitoring

Monitoring is fundamental to ensure the performance of the system and guarantee the application is running as intended. It relies on collecting metrics and analyzing data to obtain information about the resource usage or behavior of the system. Low-level metrics provided by the operating system such as CPU usage, memory usage and system calls, or higher-level metrics related to the business logic of an application such as the number of user requests help development and operation teams understand the current state of the whole infrastructure and make decisions based on historic trends, patterns or unexpected behaviors. The events obtained are collected with the help of tracing tools which provide more visibility into the runtime behavior of a system. They can be further analyzed by monitoring systems that usually leverage tracing technology. This section describes existing tracing and monitoring solutions relevant to the area. A comparison between the tracing solutions is not relevant since our focus is only on Sysdig.

2.5.1 Tracing Solutions

This subsection describes in more detail three of the most popular tracing tools: Strace, Sysdig and Extended Berkeley Packet Filter (eBPF). Other tools such as LTTng [82] and SystemTap [83] also belong to this category but will not be covered due to being less commonly used and because the work being conducted will focus only on Sysdig to extract system calls.

Strace

Strace [84] is a diagnostic, instructional and debugging tool that collects events such as system calls and signals received by processes. It is based on a Linux kernel feature called "ptrace" that listens for the system calls of a given process. As shown in Figure 2.13, the ptrace mechanism makes it possible for strace to interrupt the traced process every time a system call is invoked to capture and decode the call, and then resume the execution of the initial traced process. Every time a system call is invoked a transition between user mode and kernel mode is required which can be time consuming. During the collection of the OS events, multiple transitions will occur making strace not very efficient [85].

Sysdig

Sysdig [86] is an alternative to strace. It is a tool that captures OS events with the help of the *sysdig-probe* driver that works directly in the Linux kernel. All the system calls coming from applications and containers in the host can be captured. To reduce the amount of data captured, sysdig supports filtering of incoming OS events. These can be based on specific system calls, the source of an event (such as specific containers or processes) or attributes of the respective event. As shown in

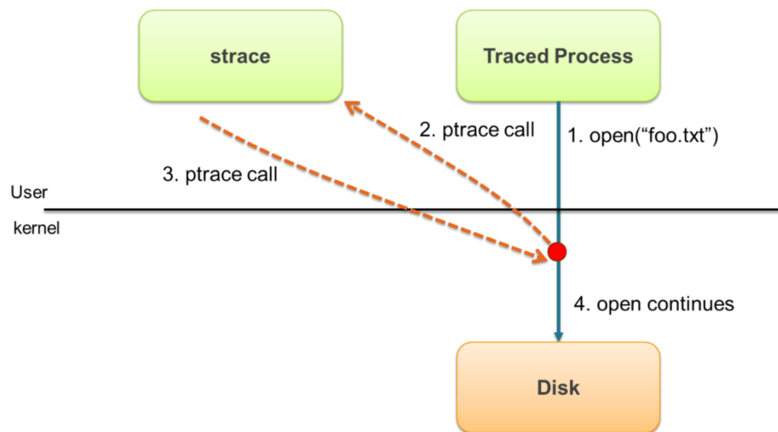


Figure 2.13: Strace workflow from [85].

Figure 2.14, and starting from the bottom, the sysdig-probe is responsible to capture the events at the kernel level and store them in an event buffer. If the event buffer gets full the sysdig-probe starts dropping the incoming events to guarantee that the performance of the other processes is not affected. Afterwards, libscap and libsinsp ("scap" and "sinsp" in the figure, respectively) actuate, providing support for reading, decoding, and parsing events. Sysdig works as a wrapper around these libraries and provides a command-line interface to ease the interaction.

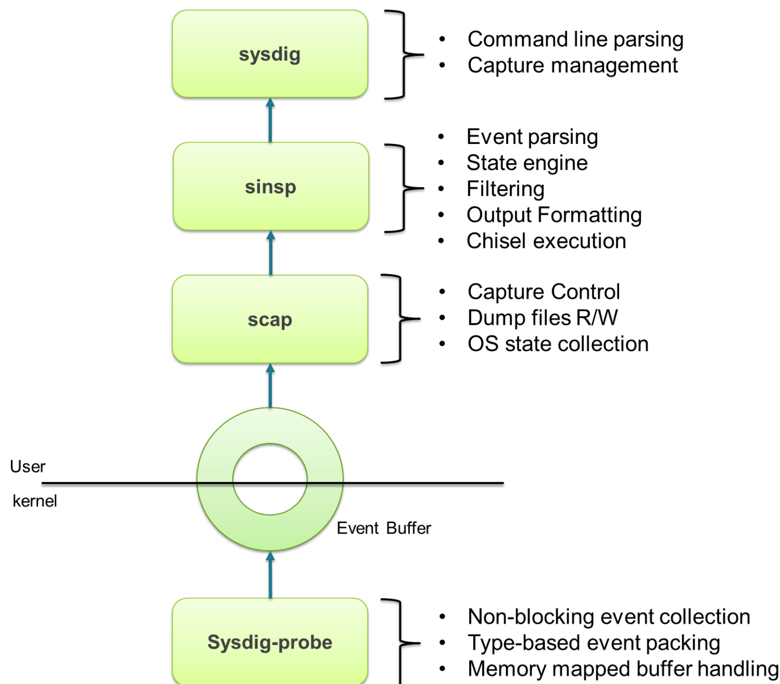


Figure 2.14: Sysdig workflow from [85].

eBPF

eBPF is a subsystem in the Linux kernel in which a virtual machine can execute programs passed from the user space to the kernel [87]. It was originally used

for network packet filtering but later became a powerful tool for kernel developers and production engineers as they saw an opportunity to get a better insight into their system through the collection of custom metrics. As shown in Figure 2.15, eBPF allows a user-space application to inject code into the kernel without the need to recompile the kernel (i.e., at runtime) or install additional kernel modules [88]. eBPF programs are event-driven and run when the kernel or an application passes a certain hook point such as system calls, function entry/exit, kernel tracepoints and network events. eBPF programs can be written using eBPF assembly instructions and converted to bytecode or in C language and compiled using the LLVM Clang compiler, as shown in the development part of Figure 2.15. When the hook is triggered the eBPF program is loaded into the Linux kernel using the *bpf()* system call and goes through a sanity-check process to enforce security. eBPF provides a fast and non-intrusive way of monitoring events from the kernel to improve observability in the cloud.

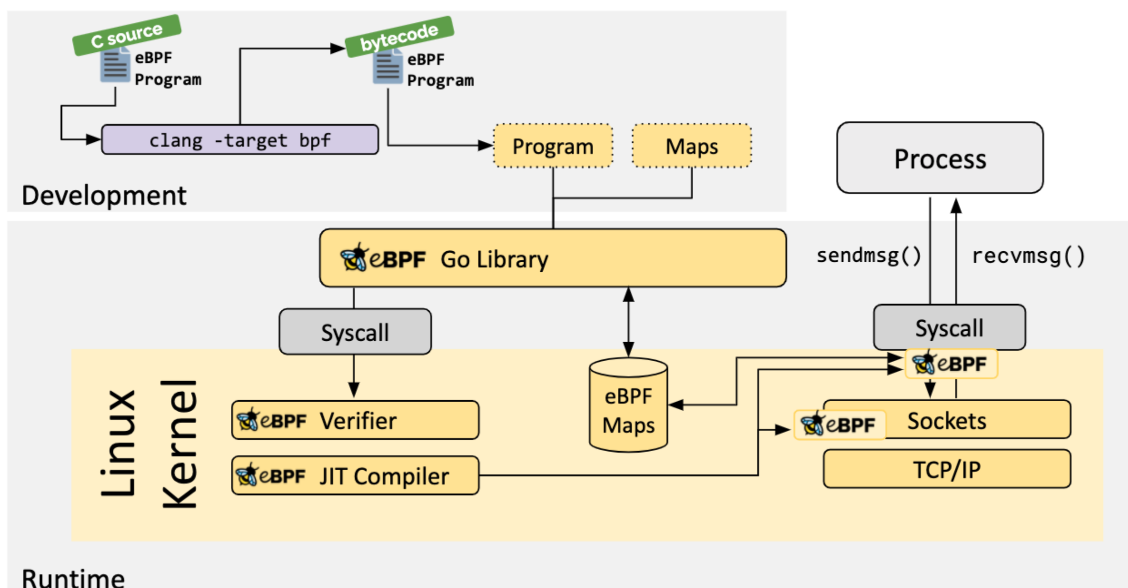


Figure 2.15: eBPF workflow from [87]

2.5.2 Monitoring Solutions

Monitoring solutions provide mechanisms to collect and analyze a given system. The now considered legacy monitoring tools used to be a good approach to monitoring. However, with the rising of microservices applications and orchestrators like Kubernetes, the paradigm has changed. Each service can be distributed across multiple instances through different nodes. The use of containers poses new challenges because of their ephemeral properties and because of that, after a container exits, usually, the information is lost. Additionally, the number of metrics collected increases exponentially as more services and containers are deployed and legacy systems cannot keep up with the continuous growth. Thus, verifying that all the instances are healthy as well as the performance of the application becomes a difficult and heavy task. Modern solutions must pivot on the performance and scaling of the collection of metrics and provide an effi-

cient mechanism to analyze an enormous amount of data and alert the operations teams. The existence of tools like sysdig only provide functionalities to collect metrics. Other solutions like cAdvisor and Falco extend these tools to support the analysis of the data collected and take some sort of action in later stages. Companies are adopting these solutions to have better control of the infrastructures that support their businesses.

There are some Commercial off-the-shelf (COTS) alternatives in this domain. COTS is a type of software usually built by companies to reach the mass market. Most of the time the implementation is straightforward, it has a vast set of functionalities and customer support is usually provided. However, this type of software typically comes at a certain cost and users must be willing to pay the price for it. They also do not provide access to the code as this software is generally not open-source. This makes it difficult to use the product if you are not willing to pay for it. Examples of COTS monitoring solutions are Sysdig Secure [89] and PrismaCloud [90].

The subject of this dissertation focuses on using intrusion detection systems in microservices applications and Kubernetes, therefore all the solutions presented below are narrowed to this scope.

cAdvisor, Heapster and Kubernetes built-in solutions

cAdvisor [91] is a metrics collection tool that allows users to obtain information about the resource usage and performance characteristics of containers. cAdvisor works as a daemon that collects, aggregates, processes, and exports information about running containers [91]. Specifically, for each container, it keeps resource isolation parameters, historical resource usage, histograms of complete historical resource usage and network statistics. Finally, the data can be exported to various storage drivers such as Prometheus, Kafka, stdout and a few more. The data collected by cAdvisor also can be viewed with the help of a web-based UI which it exposes at its port. It also exposes its data via a REST API. When working with Kubernetes, cAdvisor is integrated into the Kubelet binary. The main characteristics are that it provides auto-discovery of all the containers running and collects CPU and memory usage, file system and network usage statistics. cAdvisor used to be used together with Heapster [92] which runs as a pod in the cluster and groups the information collected. However, since around 2018, Heapster has become deprecated on versions of Kubernetes greater than v1.11 making it a nonviable solution [93].

The alternative provided by the Kubernetes development team was Kubernetes built-in metrics-server [94]. Metrics-server collects resource metrics from Kubelets and exposes them in the Kubernetes API server through Metrics API. Its main purpose is to gather data for the autoscaling functionalities of Kubernetes which include the Horizontal Pod Autoscaler and Vertical Pod Autoscaler. Even though it is a limited alternative, it allows consumers to access resource metrics (CPU and memory) for pods and nodes. Other solutions built into Kubernetes, such as the Kubernetes Dashboard [95], kube-state-metrics [96] and Kubernetes probes [97] can also be take into consideration. When used together they can provide a

deeper insight into the application under monitoring.

Prometheus

Prometheus [98] is an open-source monitoring and alerting toolkit originally built at SoundCloud and in 2016 joined the CNCF as the second hosted project, after Kubernetes. Prometheus collects and stores its metrics as time-series data whereas the name suggests changes are recorded over time. it is a very extensive solution with an entire monitoring stack around it. Tools like Grafana (https://grafana.com/) can be integrated with Prometheus to provide dashboards with the help of "sidecar" containers that transform service metrics into Prometheus metrics following a specific format. To monitor services using Prometheus, it is necessary to expose a Prometheus endpoint which is an HTTP interface that exposes a list of metrics and the current value of the metrics. It is also possible to write queries in the PromQL language to extract metric information. As depicted in Figure 2.16, Prometheus is composed of the *Prometheus server*, which scrapes and stores time series data; client libraries, for instrumenting application code; a push gateway, for supporting short-lived jobs special-purpose exporters for services like HAProxy, StatsD and Graphite; an alert manager to handle alerts and integrate them with other tools [98]. It is a toolkit that when combined with other solutions comprises a powerful monitoring stack.

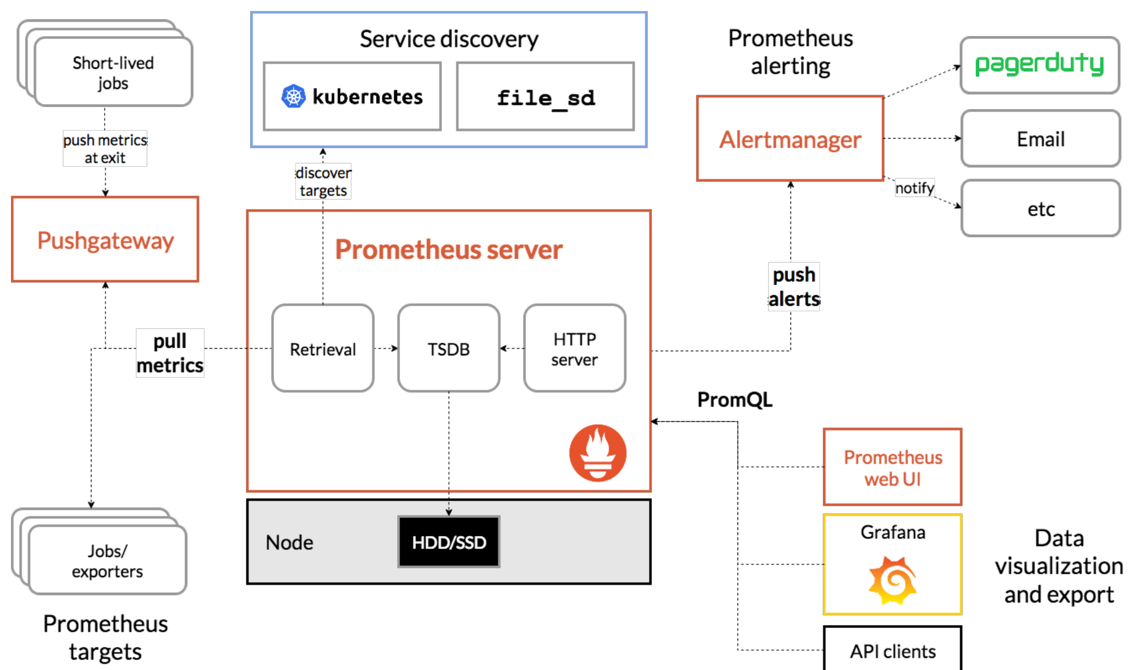


Figure 2.16: Prometheus architecture from [98].

Falco

Falco [13] is a cloud-based security tool originally developed by Sysdig. Later was donated as an incubation-level project to the CNCF. Falco claims to be the *"de*

facto" Kubernetes threat detection engine. It is a container native runtime security solution that focuses on Intrusion detection and uses the *sysdig_probe* module to monitor Linux system calls and collect Kubernetes audit logs to generate alerts based on a custom rules and macros engine. As depicted in Figure 2.17, Falco complements the sysdig tool and adds some functionalities such as behavioral activity monitor whose policies can be defined based on the sysdig filter options as conditions. The output can be accessed in files, stdout or through a gRPC API. The configurations and rules are written in YAML files, and a vast set of rules is already available as part of the open-source initiative. However, Falco presents some limitations. For example, it is only a monitoring tool and does not take action on the alarms generated. Another limitation of Falco is that it is only suitable for detecting point anomalies, such as the occurrence of a certain event, as starting of a certain application, or the use of an unauthorized system call, meaning there is no possibility to map contextual or collective anomalies through the available rules set [99]. Recent approaches and techniques that use for example collection of system calls are not incorporated in Falco as the rules defined are based on conditions that system calls have to match. Also, because Falco is still in an early version (< v1.0) this might indicate that this monitoring solution is not mature enough for production.

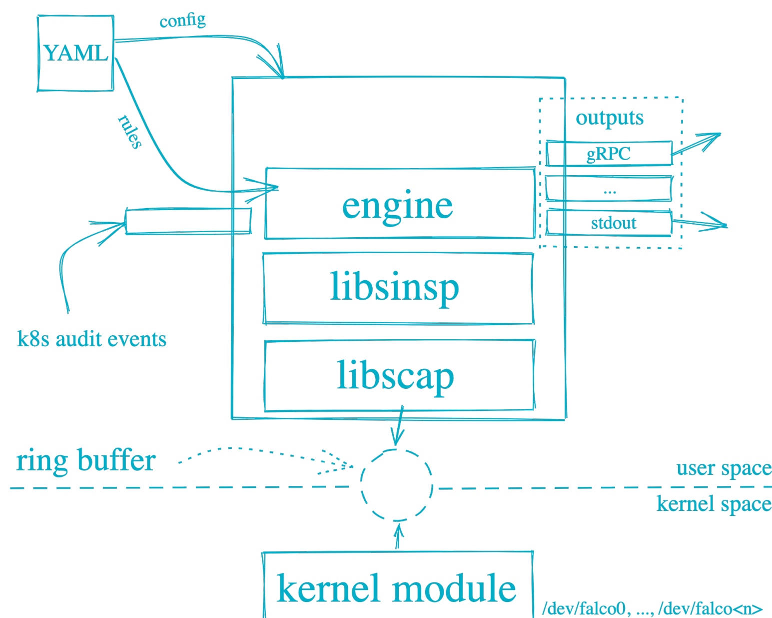


Figure 2.17: Falco architecture from [13].

Enterprise solutions like Sysdig Secure [89] extend Falco by adding out-of-the-box workflows for security and compliance. Even though the core of the technology is open-source, these solutions are usually costly as their primary target audiences are businesses and companies.

KubAnomaly

KubAnomaly [100] is a system developed by a research team from the Cybersecurity Technology Institute, Institute for Information Industry in Taiwan, that pro-

vides security monitoring capabilities for anomaly detection on the Kubernetes orchestration platform. Through the use of neural network approaches, they created classification models to find abnormal behaviors such as web service attacks and common vulnerabilities and exposures attacks. KubAnomaly architecture is depicted in Figure 2.18 and divides the system into three main components: Center, Web Portal and Agent Service. The Center component is the main component of the system and consists of several modules. Its primary purpose is the collection of data and detection in case a container shows anomaly behavior. It also exposes a REST API so that the Web Portal component can display information and inform users about the container status. The Agent Service component is used to collect logs from Docker-based containers. It also supports Kubernetes. The accuracy and performance results obtained in the evaluation of KubAnomaly showed that the accuracy of the proposed model is around 96% for detecting anomaly behavior in containers and it is capable of detecting anomalous events such as web service attacks and CVE attacks originating from different places with a minimal impact in performance.

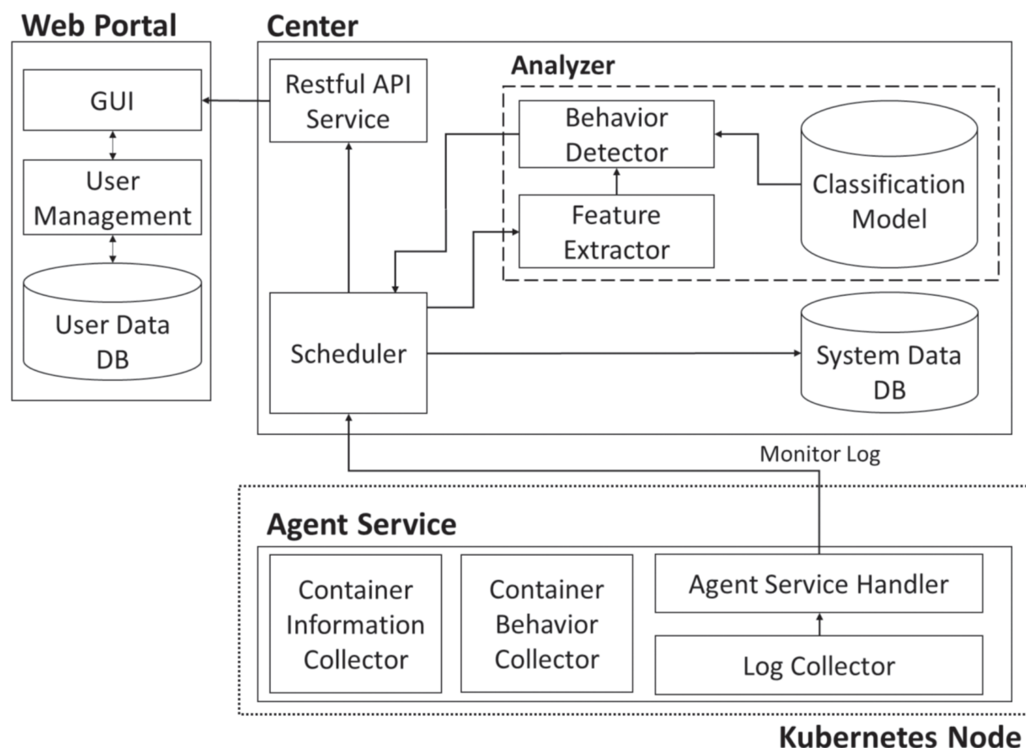


Figure 2.18: KubAnomaly architecture from [100].

Kohyarnejadfard et al

Kohyarnejadfard et al [101] propose an anomaly detection framework that reduces troubleshooting time and highlights anomalous parts in trace data. It works by collecting streams of system calls using LTTng [82] and sending the data over to a machine learning module that reveals anomalous patterns of system calls based on their execution times and frequency and generates an alarm. Their find-

ings suggest that the framework is an effective tool for automated anomaly detection and they intend to integrate this work as an open-source Trace Compass extension [101].

Tracee

Tracee [102] is a runtime security and forensics tool for Linux. It is an open-source project developed by Aqua Security and uses Linux eBPF technology to trace systems and applications at runtime, and analyze collected events to detect suspicious behavioral patterns based on predefined rules. Tracee is available as a Docker image that monitors the OS. It is divided into two components, the Tracee-eBPF and Tracee-Rules. Tracee-eBPF allows to independently run the eBPF module without involving the detection engine. Tracee-Rules is a rule engine that applies rules over the input and alerts are communicated over stdout, posted to a webhook or integrated with external systems.

2.5.3 Comparison of Related Systems

In this subsection we compare related systems to *µDetector*, the monitoring solution to be developed during this dissertation. It is important to mention that cAdvisor, Heapster, Kubernetes built-in solutions and Prometheus do not use intrusion detection techniques and therefore are not included in the comparison. However, they could be used in the validation phase of this project to collect important metrics. In order to compare the related system we focused on the following parameters:

- **Open-source.** This parameter distinguishes open-source systems from proprietary systems. Open-source systems are usually preferred because the code is publicly available and it is free to use. Closed-source systems are difficult to have access to unless we are willing to pay for the product.
- **Graphical User Interface (GUI).** This parameter distinguishes systems that have an integrated GUI from the ones that do not. A GUI is usually much easier to use and could provide some extra visualization that is not available in simpler interfaces.
- **Extensibility.** This parameter distinguishes systems that can support and easily incorporate new algorithms from the ones that do not or have but at a higher cost. These systems provide a wide range of options from which the users can choose or even add new techniques.
- **Intrusion Detection Approach.** This parameter refers to the type of intrusion detection approach used to detect abnormal behavior. It can be anomaly-based, signature-based or both. Anomaly-based techniques can overcome the limitations of the signature-based approach, which requires the use of a set of rules, at the cost of relying heavily on the data used during the training phase. While anomaly-based techniques can increase the

number of false positives alarms, signature-based techniques can increase the number of false-negative alarms.

- **Tracing.** This parameter identifies the tracing tool used. Tracing tools use different approaches to collect events from different sources (such as the Linux kernel or user applications).

The results of the comparison are summarized in Table 2.1 as we can observe. Most of the systems included in the comparison were open-source due to the vast information available obtained from articles, blogs and official documentation. Sysdig Secure [89] uses Falco under the hood and appears in the table as the most complete solution, however, it is a closed-source product with associated costs. In addition, both the framework proposed by Kohyarnejadfard et al [101] and KubAnomaly [100] are still a work in progress and, therefore, are not ready for production environments. Tracee and Falco both use Signature-based techniques (rules) to detect intrusions and neither provide a GUI. We consider our tool to be the only one out of the systems presented to guarantee the possibility of easily incorporating new anomaly detection techniques into the tool. It will be possible to add a new algorithm at a small cost. This comparison provides some motivation for the design and development of an alternative solution.

Table 2.1: Comparison of Related Systems.

System	Open-source	GUI	Extensibility	Approach	Tracing
Kohyarnejadfard et al	✓			Anomaly	LTTng
Falco	✓			Signature	Sysdig/eBPF
KubAnomaly	✓	✓		Anomaly	Sysdig
Tracee	✓			Signature	eBPF
Sysdig Secure		✓		Both	Sysdig/eBPF
µDetector	✓	✓	✓	Anomaly	Sysdig

Chapter 3

Requirements Analysis

In this chapter we cover the whole process of requirements gathering. We start by describing the User Stories and Mockups. Then we present functional and non-functional requirements and finish with the technical restrictions of the project.

3.1 User Stories

A user story is an agile methodology tool used in project management that provides users with a more informal and less detailed explanation of a software feature written from the end user's perspective, emphasizing communication with the client. It is a less cumbersome process when compared to traditional methods such as use case descriptions. Therefore, it was defined that user stories would be used in the requirements elicitation process because it is a flexible and adaptable way of linking the user to the requirements.

The template to be used for each user story and corresponding acceptance criteria is the following:

"**As a** [type of user], **I want to** [perform something] **so that** [I can achieve some goal]."

"**Given** [pre-conditions], **when** [key action], **then** [draw conclusions]."

For the user story description, we start by identifying the individual that interacts with the system. Then, we describe the action that represents the behavior of the system and the goal that the individual wants to achieve with the action performed. Finally, we define acceptance criteria detailing the conditions that the software product must meet to be accepted by the type of user. Due to the high number of user stories (i.e. over 20 user stories), the complete list is presented in **Appendix A**. Below we present two user stories as an example.

US-1: Configure the detection phase

As a user, **I want to** configure the detection phase **so that** I can keep my application secure by detecting intrusions in real-time.
Acceptance Criteria: Given the user has his application running and the tool installed, when he runs a command (via CLI or dashboard) to setup the detection phase and specifies a configuration file containing information about the application deployment, the desired services to be monitored, the algorithm to be used, the location to retrieve and store logs and the duration of the detection phase, then the tool should configure the probes on the desired services and start monitoring the application for possible intrusions.

US-15: Upload a configuration file
As a user, **I want to** upload a configuration file to the dashboard **so that** I can start monitoring my application via the dashboard
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he clicks the button to upload a file, then he should be able to choose a JSON file from a specific location and upload it to the dashboard.

3.2 Mockups

The possibility to develop a Web Dashboard, as an alternative to a CLI, was planned and discussed since the beginning of this dissertation. Therefore, we felt the need to draw some mockups to have an overall idea of the software product design, space allocation and prioritization of content, functionalities available and user experience. Figure 3.1 shows the idea for the Web Dashboard, specifically the main page. In this particular situation, mockups were designed using low-fidelity representations. Therefore a more adequate term would be "wireframe" or "prototype" but we decided to maintain the term "mockups" for simplicity's sake. The rest of the Mockups appears in the **Appendix A** along with the User Stories.

The Mockups divide the Web Dashboard into four main pages: **Dashboard**, **Alarms**, **Resources** and **Help**. The **Dashboard** page shows an overview of the system such as node information, if any phase is active, expected time and a few more configurations. It also shows a chart with the alarms generated in real-time and possibly a list of the latest alarms. The **Alarms** page shows a list of all the alarms. The **Resources** page shows the resources such as pods and services of the cluster. The **Help** page shows information that allows the user to properly use the Web Dashboard. A Header with two buttons is available on all pages whose purpose is to configure and add a monitoring phase by providing a JSON file or stop monitoring.

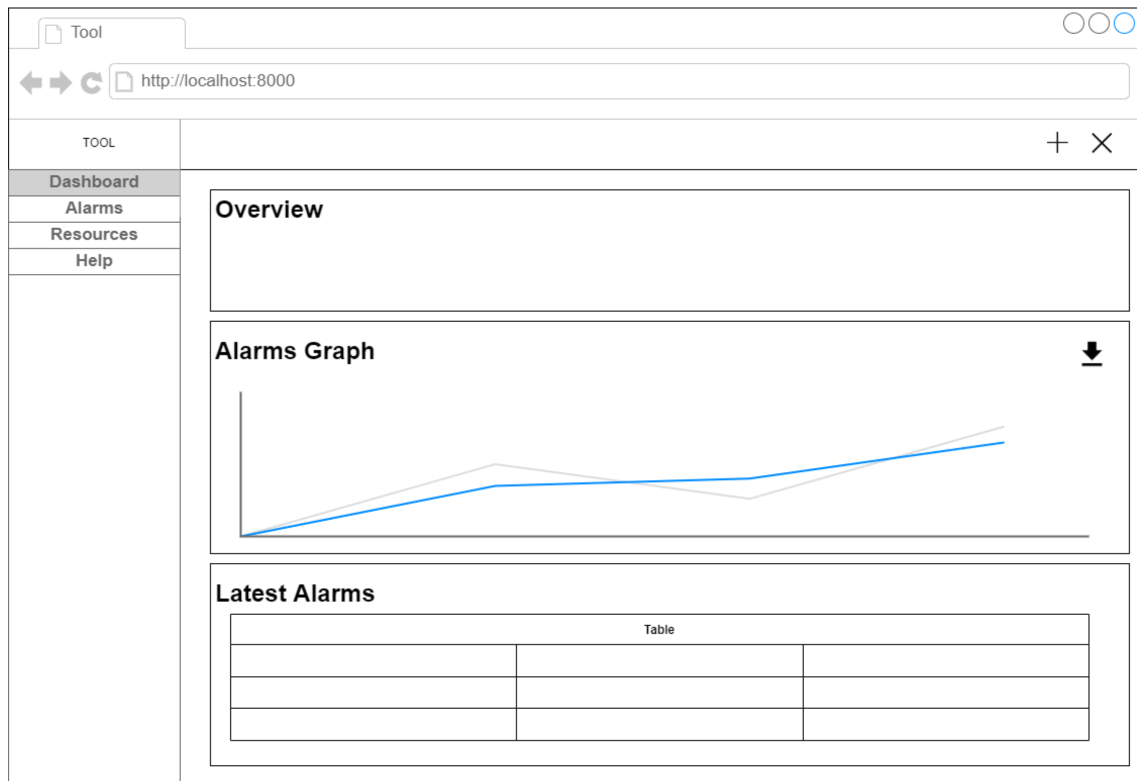


Figure 3.1: Mockup of the dashboard page.

3.3 Functional Requirements

After the user stories have been written, we need to extract the requirements. Each requirement has a well-defined priority obtained using the MoSCoW method. This method divides priorities into four categories:

- **Must** have. Requirements that are necessary to the successful completion of the project.
- **Should** have. Requirements that are important but are not necessary to the project.
- **Could** have. Requirements that are a good addition to the project but are considered non-important to the project.
- **Won't** have. Requirements that could eventually be added. They are not a priority right now to the project.

The functional requirements are listed in the Table 3.1, Table 3.2 and Table 3.3. Each requirement has an identifier, a title in bold followed by a description and the priority according to the MoSCoW method. In more detail, Table 3.1 presents the CLI functional requirements, Table 3.2 presents the Web Dashboard functional requirements and Table 3.3 presents optional requirements that will probably not be accomplished but are still worth to mention.

Table 3.1: System Functional Requirements - CLI.

Id	Title and Description	Priority
REQ-1	**Start monitoring.** The user enters a command to start monitoring his application specifying a JSON file with configurations that include the sysdig format and parameters, compression and window size. The system validates the JSON and starts collecting the system calls.	Must
REQ-2	**Stop monitoring all services.** The user enters a command to stop monitoring all services and the system communicates with the monitoring agents to stop collecting the system calls.	Must
REQ-3	**Inspect the system calls.** The user enters a command to start inspecting the system calls by providing a JSON file that specifies the type of phase (detection or training), the algorithm to be used and the duration.	Must
REQ-4	**View status.** The user enters a command to view the current status of the tool. The tool could be idle, collecting (monitoring) system calls or collecting and analyzing the system calls with the help of an intrusion detection module that generates alarms. Additional information associated with each phase (such as the duration) can be provided.	Must
REQ-5	**List alarms.** The user enters a command to list alarms and the system presents the desired list. It should be possible to filter the alarms by latest (show `n` latest alarms)	Must
REQ-6	**List pods.** The user enters a command to list pods of a cluster and the system presents the desired list along with basic pod information. It should be possible to view pods from different namespaces.	Must
REQ-7	**List services.** The user enters a command to list services of a cluster and the system presents the desired list along with basic service information. It should be possible to view services from different namespaces.	Must
REQ-8	**List deployments.** The user enters a command to list deployments of a cluster and the system presents the desired list along with basic deployment information. It should be possible to view deployments from different namespaces.	Must
REQ-9	**List namespaces.** The user enters a command to list namespaces of a cluster and the system presents the desired list.	Must
REQ-10	**List nodes.** The user enters a command to list nodes of a cluster and the system presents the desired list.	Must
REQ-11	**List algorithms.** The user enters a command to list algorithms available and the system presents the desired list.	Must
REQ-12	**List functionalities.** The user enters a command to list functionalities of the tool and the system presents the desired list.	Must

Table 3.2: System Functional Requirements - Dashboard.

Id	Title and Description	Priority
REQ-13	**Start monitoring.** The user clicks on a button to start monitoring the application and provides a valid JSON file (using an input file button) with configurations that include the sysdig format and parameters, compression and window size. The system validates the JSON and starts collecting the system calls displaying a successful alert.	Must
REQ-14	**Stop monitoring all services.** The user clicks on a button to stop monitoring all services. The system communicates with the monitoring agents to stop collecting the system calls.	Must
REQ-15	**Inspect the system calls.** The user clicks on a button to start inspecting the system calls by providing a JSON file that specifies the type of phase (detection or training), the algorithm to be used and the duration. The system starts inspecting the system calls.	Must
REQ-16	**List all intrusion detection instances.** The user navigates to the dashboard page and clicks on a button that lists all the intrusion detection instances and presents details such as the id, type of detection, algorithm, start time, elapsed time and duration.	Must
REQ-17	**Stop all intrusion detection instances.** The user clicks on a button to stop all intrusion detection instances without stopping the collection of the system calls.	Must
REQ-18	**Stop specific intrusion detection instances.** The user clicks on a button to stop all intrusion detection instances without stopping the collection of the system calls.	Could
REQ-19	**View general information.** The user navigates to the dashboard page and the system presents information about the system being monitored. This information includes everything specified in the JSON configuration. If no monitoring is currently taking place then a list of the agents running on the nodes is presented.	Must
REQ-20	**View monitoring statistics.** The user navigates to the dashboard page and the system presents real-time monitoring information that includes the total number of alarms generated, the total number of systems calls, their size in bytes and the number of system calls' batches.	Must
REQ-21	**View a graphical representation of the number of system calls over time.** The user navigates to the dashboard page and the system presents a real-time chart of the number of system calls collected over time.	Must
REQ-22	**Export chart.** The user navigates to the dashboard page and clicks on the save chart button that immediately starts the download of the chart as an image in the Portable Graphics Format (PNG) format.	Should
REQ-23	**View latest alarms.** The user navigates to the dashboard page and the system presents a list of the latest five alarms. This list is constantly being updated.	Should
REQ-24	**View alarms.** The user navigates to the alarms page and the system presents a list of all the alarms detected.	Must
REQ-25	**Search alarms.** The user navigates to the alarms page and an input placeholder is presented that allows to search for any alarm.	Should
REQ-26	**Delete alarms.** The user navigates to the alarms page and clicks on a button to delete the current list of alarms.	Should
REQ-27	**Export alarms.** The user navigates to the alarms page and clicks on a button to export the list of alarms as a JSON.	Should
REQ-28	**View resources.** The user navigates to the resources page and the system presents a list of all the resources such as pods, services, deployments, namespaces and nodes.	Should
REQ-29	**View Algorithms.** The user navigates to the help page and the system presents a list of all the algorithms supported as well as a short description of how they work.	Should
REQ-30	**View Help.** The user navigates to the help page and the system presents information that guides the user on how to use the tool correctly.	Could

Table 3.3: System Functional Requirements - Optional.

Id	Title and Description	Priority
REQ-31	**Submit a monitoring configuration using a form** On the dashboard, the user is presented with a form that he can complete as an alternative to providing a JSON file. The system guarantees that the input fields are valid, provides feedback and starts monitoring the system calls.	Could
REQ-32	**Submit an intrusion detection configuration using a form** On the dashboard, the user is presented with a form that he can complete as an alternative to providing a JSON file. The system guarantees that the input fields are valid, provides feedback and starts an intrusion detection instance.	Could
REQ-33	**Stop monitoring a specific service.** On the CLI the user enters a command to stop a specific intrusion detection instance without stopping the collection of the system calls.	Won't
REQ-34	**Restart a specific pod.** On the dashboard the user clicks on a button associated with a specific pod to restart it and the system communicates with the cluster to restart the pod.	Won't
REQ-35	**Download a profile.** On the dashboard, after the training phase of an intrusion detection instance ends, the user clicks on a button to download a profile with information from the training and the system starts downloading the file.	Could
REQ-36	**Upload a profile.** On the dashboard, the user clicks on a button to upload a profile (obtained from a training phase), selects the file, and confirms the choice. The system uploads the file and uses it in the detection phase.	Could

3.4 Non-Functional Requirements

In this section, we present the Non-Functional Requirements. These correspond to quality constraints that the system must satisfy and were defined at the early stages of requirements gathering during the reunions with the client (the supervisors of this dissertation).

Performance

The tool must not interfere with the performance of the host (node). To achieve this, we define that the resource usage of the nodes under monitoring, in particular the CPU and memory usage, must not exceed 10% of the total resources available. Additionally, when users are interacting with the application, the request throughput should not be affected by the *µDetector* tool.

Scalability

The tool must be able to function properly if the number of system calls increases in a short period and if the number of pods and nodes increases as well, for example, an increment of one replica/node to three replicas/nodes. There should be no impact on the requests throughput of a microservices application that uses the *µDetector* tool.

3.5 Technical Restrictions

In this section, we present technical restrictions that were considered in the proposed solution. The technical restrictions listed in Table 3.4 originated from meetings with the client and have an identifier and a description associated.

Table 3.4: Technical restrictions of the project.

Id	Description
TR-1	The domain of the solution will focus on microservices applications.
TR-2	Kubernetes and KubeEdge deployments will only be considered.
TR-3	The intrusion detection algorithms are based on system calls. Sysdig will be used as the tool to extract system calls.

Chapter 4

A Tool to Automate Intrusion Detection in Microservices

This chapter presents *µDetector*, a monitoring tool that was designed to operate in cloud and edge environments. We start by presenting the proposed architecture and then we move to the explanation of the implementation details.

4.1 Proposed Architecture

In this section, we explain the proposed architecture. Diagrams of the initial architecture are depicted in Figure 4.1 and Figure 4.2. The former represents a more generalized view of the *µDetector* tool and how it interacts with the other systems and the latter details the functioning of the Intrusion Detection component of the tool.

The *µDetector* tool will interact with Kubernetes clusters using different technologies. When we refer to the Kubernetes cluster, we also **include support for KubeEdge** meaning that edge nodes can also be monitored. The communication between the Edge and the Cloud is made through the KubeEdge components CloudCore and EdgeCore, as shown in Figure 4.1.

Firstly, to obtain information about the cluster, such as the pods and nodes that are running on the system, and be able to deploy the sysdig probes in the correct nodes, it is necessary to run a **Daemon** on the master node that will communicate with the tool through REST. Then the user can specify in a JSON file configurations of how the tool should behave when running. For example, the user can specify the services he wants to monitor, the format in which the system calls are collected, the size of the window of collection of system calls and whether the system calls are compressed or not. The configuration file is then validated by the tool and the instructions are sent to the master node (through the daemon) which is responsible for communicating with the monitoring agents (also referred to as "probes"). These monitoring agents will then run an instance of sysdig to retrieve the system calls and transfer them to the machine that is running the *µDetector* tool where they are temporarily stored in a Redis instance for later use by the

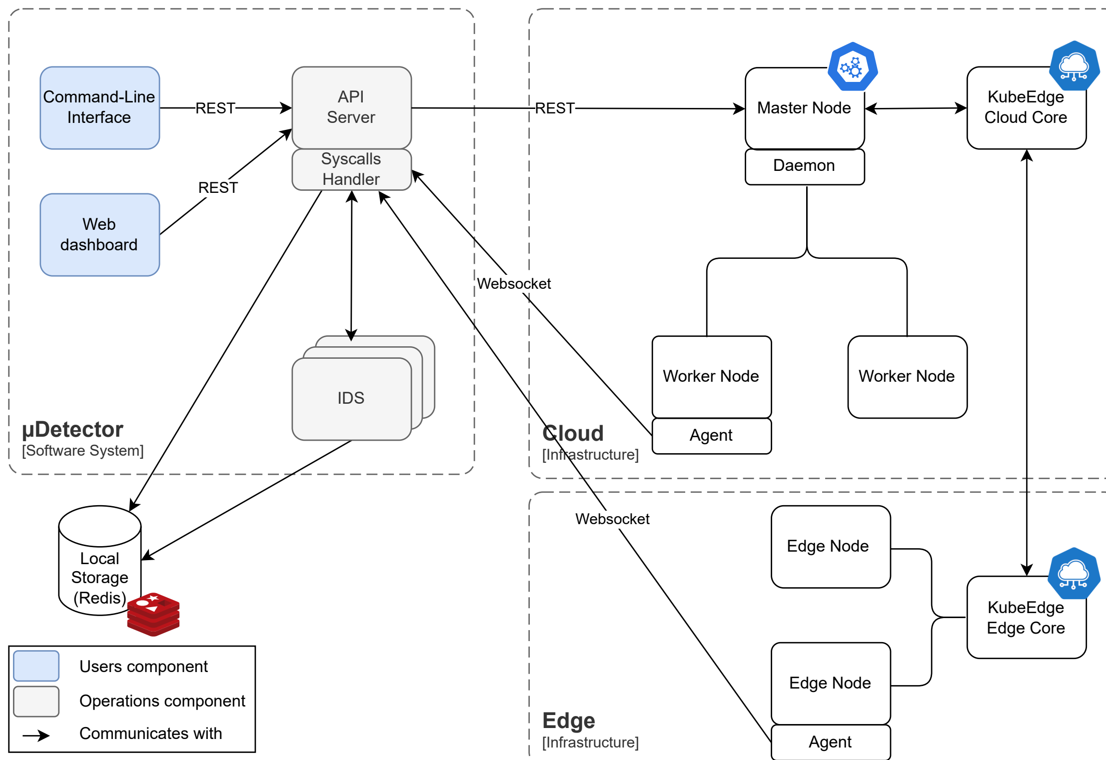


Figure 4.1: General Architecture.

IDS module. After the system calls are being monitored (i.e.: extracted from the worker nodes and transferred to the machine that is running the *µDetector* tool), the user can spawn intrusion detection instances by providing a configuration file in a JSON format. The configuration file requires a **type** (training or detection), an **algorithm** (for example BoSC or STIDE) and a **duration** (in seconds). When an Intrusion Detection instance is running, the IDS module is responsible for analyzing the system calls according to a certain algorithm and generating a profile if it is a training phase or generating alarms if it is the detection phase and anomalous behavior is detected.

As shown in Figure 4.1, the *µDetector* tool has two user components that ease the interaction of the user with the tool and improve the observability of the system. These are the CLI and the Web Dashboard. The API Server and the IDS are operations components because they do not interact directly with the user and are responsible instead for the inner workings of the tool.

In depth, the tool is comprised of the following components:

- **Master Daemon.** This component is deployed on a Master node of the Kubernetes cluster. It works as a proxy which allows the *µDetector* tool, running on an external machine, to communicate with the Kubernetes API to retrieve information about the cluster such as available pods and nodes [103]. Additionally, when a user provides a specific monitoring configuration, the daemon is responsible for communicating with the monitoring agents in the worker nodes. It is written in Python using the Flask web framework [62].

- **Monitoring Agent.** Each Worker node can have a monitoring agent (also referred to as a "probe") running in the background whose job is to collect system calls and transfer them to the external machine that is running the *μDetector* tool through Websockets. It is written in Python using the Flask web framework [62].

- **API Server.** Provides endpoints that both the interfaces (CLI and Dashboard) will use to perform actions such as view the resources of the cluster or start and stop monitoring system calls. Communication is achieved through REST. For example, if the user pretends to list the pods in the cluster, he uses one of the two interfaces available (CLI or the web Dashboard) and sends a request to the API Server which will then communicate with the Daemon in the Master node of the cluster to retrieve the list of pods requested. It is tied together with a system calls handler whose job is to receive the system calls directly from the worker nodes and store them in a Redis instance locally. It is written in Python using the Flask web framework [62].

- **Intrusion Detection System.** This component can have multiple instances (with different algorithms) and is deployed when the user provides a configuration file in a JSON format. It reads the system calls from the Redis instance and uses them to detect threats with the help of algorithms such as BoSC and STIDE. The IDS, as shown in Figure 4.2 is divided in two phases that the users can choose. The **training phase**, creates profiles that represent the baseline behavior of the application, based on the system calls collected. The **detection phase** uses the generated profiles to assess in real-time if the microservices application shows signs of abnormal behavior. In case this happens, an alarm is generated. It is written in Python.

- **Command-Line Interface.** Connects the user to the tool and allows them to interact with it by typing in commands. The main purpose is to provide a simple interface that provides faster results. It is written in Python using the Click framework [104].

- **Web Dashboard.** It is a more appealing alternative to the CLI to aid in the monitoring of the microservice application. It supports the same functionalities and some others, such as visualization of a system calls chart, statistics and alarms' information in real-time. It is written using standard HyperText Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript.

- **Local Storage**. A Redis instance is responsible for storing all the information required to run the tool correctly. This includes the batches of system calls, the profiles and alarms generated as well as some statistics of the tool and configuration files.

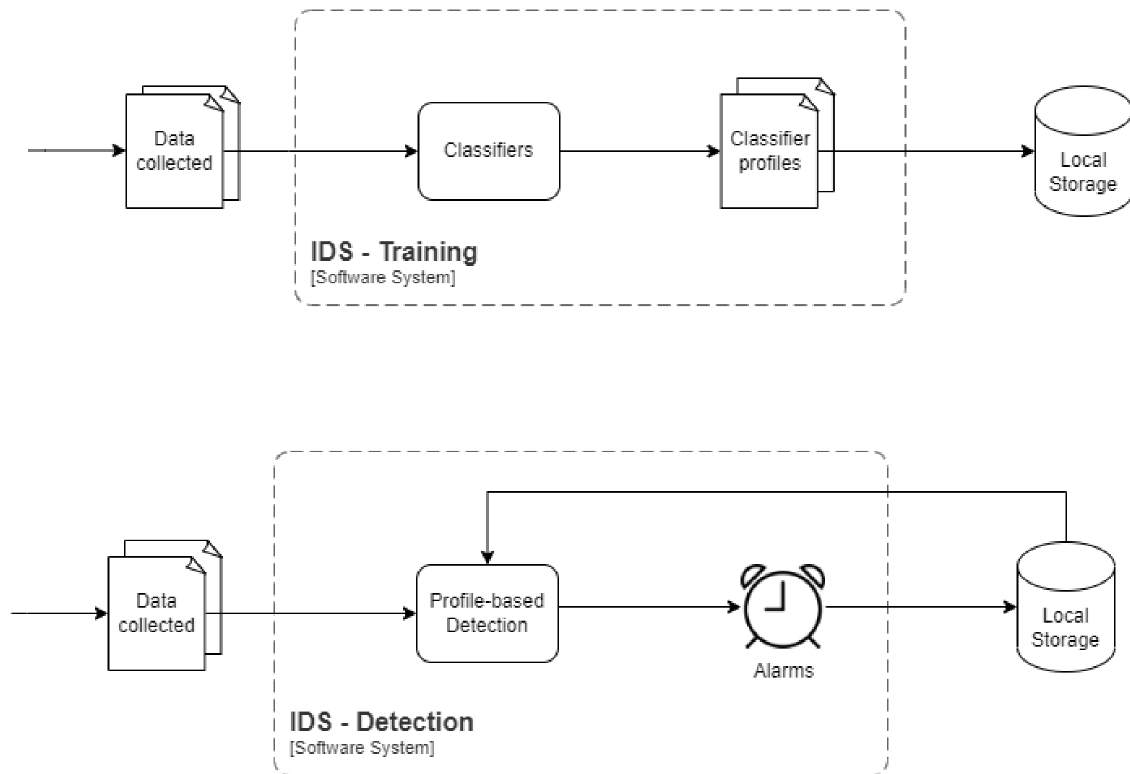


Figure 4.2: IDS Architecture.

4.2 Implementation

This section describes the implementation details considered during the development phase of the application.

Initially, we need a Kubernetes cluster comprised of a master node and several worker nodes. On the master node, we run the daemon in the background and, to have access to the Kubernetes REST API we run *kubectl* in proxy mode since it is the recommended way according to [103]. On the worker nodes, we need to have Sysdig [86] installed and then we run the monitoring agents in the background.

The external machine that runs the *μDetector* tool needs to have a Redis server configured and the *μDetector* tool (more specifically the API server) running in the background. The user can then interact with the tool using the *cli.py* or accessing `http://localhost:5001`.

A general decision that impacts the whole architecture is that, aside from the front-end of the Dashboard, all the components are written in Python. This was decided mostly because the intrusion detection module is also written in Python and, aside from our experience with the language, there are good Python libraries that make network programming easier (for example *AsyncIO*), and the compatibility with technologies such as Websockets, Redis and Kubernetes are assured. Therefore we discarded other language frameworks that are also fit to build web applications such as the JavaScript framework *Node.js*. Even though Python is known for being a slow language, if there is a bottleneck it is on the transfer of

the system calls and this occurs over the network.

Ideally, we planned to develop the dashboard and API Server with the help of the Django framework, the Master Daemon using Flask and the monitoring agents using sockets. This would require getting familiar with three different technologies and would negatively impact the delivery of the project due to time constraints. To mitigate this risk, we decided to use Flask as the solution to all these three APIs because it has no dependencies on external libraries making it a very suitable lightweight and performant framework that we are already familiarized with it.

On the front-end side of the Web Dashboard, we chose to use standard HTML, CSS and JavaScript and keep the application "thin". This means that JavaScript is only used to display the information and almost no processing logic is involved in the frontend. Therefore, we opt out of JavaScript frameworks to reduce the number of dependencies and keep the application lightweight. Nevertheless, we use a few JavaScript libraries. This include *Bootstrap* and *jQuery*, to make the GUI more responsive and appealing. *eCharts.js* is used to build the charts because it is a powerful and flexible open-source JavaScript visualization library provided by Apache [105].

Below, we explain each component in greater detail.

API Server

The API Server acts as the middleman between the interfaces (the CLI and Dashboard) and the Kubernetes cluster. Table 4.1 shows all the endpoints. This component has also associated a system calls handler responsible for establishing a connection with the agents and retrieving the system calls. The transfer of the system calls occurs over Websockets as was discussed in the section 2.3 of Chapter 2. The system calls are retrieved in 5 seconds (the default value) groups called batches. Every batch is stored in a Redis instance and, to avoid storing many batches, a configurable max value limits the number of batches. This way we can guarantee that only the latest 10 (the default value) batches are stored and keep the system ready to inspect system calls in real-time. It is also possible for the user to choose if he wants to compress the system calls using *Blosc*, a high-performance compressor optimized for binary data [106], to reduce the size of the data being transferred over the network. This might cause delays due to the time necessary to compress but might also improve the transfer of system calls. A compromise between both these scenarios must be achieved and the responsibility falls to the user.

Table 4.1: API Server Endpoints

Method	URL	Description
GET	/api	Returns the string “The detector AP”
GET	/api/daemon/status	Establishes a connection to the Daemon and returns status code 200
GET	/api/daemon/agents	Establishes a connection to the Daemon and returns a JSON with information about the nodes and whether they are running agents or not
POST	/api/monitoring/start	Establishes a connection to the Agent and sends a request to start collecting system calls
POST	/api/monitoring/stop	Establishes a connection to the Agent and sends a request to stop collecting system calls
POST	/api/inspecting/start	Starts an intrusion detection instance
POST	/api/inspecting/stop	Stops all intrusion detection instances or a single one if an id is provided
GET	/api/alarms	Returns a list of all the alarms sorted by most recent. It is also possible to retrieve the latest “N” alarms
DELETE	/api/alarms	Deletes the list of alarms
GET	/api/algorithms	Returns a list of the algorithms available
GET	/api/resources/pods	Returns information about the pods of the Kubernetes cluster
GET	/api/resources/services	Returns information about the services of the Kubernetes cluster
GET	/api/resources/deployments	Returns information about the deployments of the Kubernetes cluster
GET	/api/resources/namespaces	Returns information about the namespaces of the Kubernetes cluster
GET	/api/nodes	Returns information about the nodes of the Kubernetes cluster
GET	/api/stats	Returns a list of all the statistics from the monitoring phase

Daemon

The Daemon works as a proxy which allows the *μDetector* tool to communicate with the Kubernetes API and the monitoring agents of the worker nodes. It runs on the master node of the Kubernetes cluster and it is a Flask app written in Python. It provides 5 REST endpoints:

- `/status` (GET) - This method always returns 200 and is used to verify that the connection to the daemon is working.
- `/agents` (GET)- This method is used to verify if the monitoring agents are running on the worker nodes. It returns a JSON with information about the nodes and whether they are running agents or not.
- `/start` (POST) - This method forwards a POST request to the monitoring agents on the worker nodes to start collecting system calls.
- `/stop` (POST) - This method forwards a POST request to the monitoring agents on the worker nodes to stop the collection of system calls.
- `/proxy` (Any) - Acts as a proxy and allows the tool to access the Kubernetes API and retrieve information about the resources of the cluster.

Agents

The monitoring agent (also called "agent" or "probe") is responsible for collecting system calls and transferring them through Websockets to an external machine that is running the *μDetector* tool. It runs on each worker node of the Kubernetes cluster and it is a Flask app written in Python. It provides three REST endpoints:

- `/` (GET) - This method always returns 200 and is used to verify that the connection to the agent is working.
- `/start` (POST) - This method spawns a sysdig process with the parameters from the request data such as filters and args. The system calls collected by this process are then piped to another process responsible for sending them over the network using websockets to the *μDetector* tool. Before they are sent, the system calls are grouped in batches defined by a configurable window size which defaults to 5 seconds. This means that every 5 seconds a batch of system calls is sent through over websockets.
- `/stop` (POST) - This method stops the sysdig and websocket processes that were spawned by the agent in the `/start` method where the purpose is to stop the collection of system calls.

Intrusion Detection

The intrusion detection component is responsible for analyzing the system calls collected and creating profiles (training phase) or generating alarms (detection

phase). Every time the user commands a new intrusion detection instance to be spawned, the API Server spawns a new process that runs the intended algorithm, the type (training or detection) and the duration. All of the described components above were implemented during this project. This includes the Intrusion Detection component but the algorithms were already implemented and it was only required to modify some parts to be compatible with the tool. This way, the tool has a folder called algorithms where it is possible to add new algorithms. Each file corresponds to a new algorithm and the system can detect the file and incorporate the algorithm into the tool. This way we can guarantee the extensibility of the tool since the level of effort required to implement the extension (add more algorithms) is low. The implemented algorithms were BoSC and STIDE.

Command-Line Interface

The CLI was developed in Python with resort to Click, a Python package that provides the necessary tools to build a command-line interface [104]. The CLI provides 7 commands that are explained below:

- *status* - prints the status of the tool. Whether is monitoring, idle or inspecting system calls
- *start* - requires a monitoring configuration to be provided and starts the collection of system calls based on the configuration.
- *stop* - stops collecting system calls and/or inspecting the system calls
- *inspect* - requires a configuration to be provided and starts inspecting the system calls based on the configuration and by deploying an intrusion detection instance
- *alarms* - prints the alarms generated
- *algorithms* - prints the algorithms available
- *list* - prints information about the Kubernetes cluster. Lists information about the pods, services, deployments, namespaces and nodes of the cluster.

As previously explained, the CLI has fewer functionalities when compared to the Dashboard because the goal is to provide a simple interface that provides faster results.

Dashboard

The Dashboard was developed in Python for the backend and HTML, CSS and JavaScript for the frontend. Additionally, JavaScript libraries such as *Bootstrap*, *jQuery*, *Axios*, *eCharts.js* and *sweetalert2.js* were used. The Dashboard is composed of 5 pages that follow the common style used in the Dashboard where there is a side panel on the left. The pages are:

- Dashboard. The main page of the Dashboard presents all information about the current status of the tool, statistics about the system calls collected including a chart with the system calls collected over time, alarms generated and information about the latest 5 alarms. It also allows the user to start collecting system calls, stop the collection and deploy intrusion detection instances by uploading a configuration file. Every 5 seconds, the page refreshes the alarms list and the statistics

- Alarms. A page that displays the complete list of alarms generated and updates it every 5 seconds. The user can search for any alarm by typing in a search box. It is also possible to download the alarms as JSON file and also clear the list of alarms.

- Resources. A page that connects to the Kubernetes API and retrieves information about the resources of the cluster including pods, services, namespaces, deployments and nodes.

- Help. A static page with information about the tool such as what it is and how to deploy it.

- Error. A static page that indicates an error has occurred.

For the backend, the Dashboard uses Flask and Jinja2 template to render the pages. Requests to the API Server are made to separate the viewing part from the information part. This way, the Dashboard focus only on retrieving the information from the API Server and displaying it to the user. An example is the use of the JavaScript library Axios to make a request to the endpoint `/api/stats` that returns information that is later used by the *echarts.js* library to build a system calls graph and present statistics to the user. The *Bootstrap* and the *sweetalert2.js* library help make the dashboard more appealing to the users. Figure 4.3 shows the dashboard page and its components such as the configuration overview, the system calls chart, the statistics and the latest alarms. Other screenshots were omitted due to not being relevant (the dashboard is the most important of all the pages).

Storage

Redis was chosen as the storage mechanism due to being an in-memory data structure store that offers a rich set of features. Its ease of use and good documentation makes it also a very suitable option. Other alternatives such as Memcached and Kafka were also considered. However, Memcached does not provide support for reliable queues while Redis does. This pattern is important and is used to store the batches of system calls. Redis is also more powerful, more popular, and better supported than Memcached [107]. Kafka is usually not as fast as Redis [108] and for our use case, we need to guarantee the processing of system calls is not affected by performance.

Figure 4.3: Dashboard page.

Chapter 5

Validation and Experimentation

In this chapter, we describe the validation plan we followed to make sure that the *µDetector* tool meets the requirements defined in Chapter 3. The solution was validated in a two-fold process: first, we followed a functional testing approach; then a more experimental approach that includes performance and scalability tests. At the end of this chapter, we present and discuss the results obtained.

5.1 Validation Plan

The **first approach** uses Functional Testing to make sure the functional requirements are fulfilled. We deploy the tool along with a Kubernetes cluster that runs a microservices testbed such as Sock Shop [24]. We then perform the tests according to a predefined list that contains the inputs and respective outputs and makes sure it passes the tests.

The **second approach** consists of deploying the tool in a microservices setup where we will use scripts and Locust [109] to collect metrics such as CPU usage, memory usage, response times and requests throughput. With the metrics obtained, we will then make sure that when running the *µDetector* tool, the performance of the microservices application is not affected and the nodes running the monitoring agents are not affected.

The complete process is described in the sections below.

It is important to mention that the purpose of this dissertation is not to evaluate the detection of abnormal behavior in microservices applications but to develop a tool that supports and automates intrusion detection and the entire process of the collection of system calls. It is out of the scope of this dissertation and will not be covered in the validation stage.

5.2 Functional Testing

After the requirements have been defined we planned to prepare test cases to cover the requirements and make sure they were properly implemented. To achieve this, we use functional testing which is a kind of black-box testing that is conducted to confirm that the functionality of an application or system is behaving as expected [110]. After the execution of the test cases has been completed, we must make sure that all the tests are passed successfully. For each listed (and implemented) requirement we plan on having at least two tests: one default case that shows the normal behavior of the application and a negative case that shows the behavior of the application when something goes unexpected. An example of a negative case could be when we want to test the upload of a file to a website and provide an invalid file. The default case associated is to provide a valid file.

Our black-box approach tries to cover the interaction of the user with the tool. Thus, the tests will initiate through the CLI and the Dashboard which are the components that the users can interact with. The monitoring agents, the daemon, and the detector, even though they belong to the tool they do not interact with the user except when starting or stopping the program which we will also cover.

More detailed tests such as unit tests were not thought of, due to being a very time-consuming process that usually covers modules individually, while functional testing checks the working of an application against the intended functionality described in the system requirement specification [111]. To conduct the tests we defined the following structure:

- Test ID - to identify the test.
- Requirements Addressed - the requirement IDs that the test covers.
- Case Type - The type of the test. Example: Default, Negative, Equivalence, Boundary)
- Test Case - Describes how to conduct the test. Refers to the necessary inputs.
- Expected Result - Describes what is the expected output.

We also defined the following test conditions: Sysdig, Docker, Kubernetes and Redis are prerequisites that were properly installed and configured; the *µDetector* tool was configured and points to the right hosts and ports; a Kubernetes cluster with three worker nodes and a master node was setup where one of the nodes is running KubeEdge. Sock Shop was the testbed chosen. TeaStore was another alternative as both are mature testbeds.

The CLI and the Dashboard tests appear in Table 5.2 and Table 5.3, respectively. Table 5.1 lists tests that do not fit in any the tables mentioned (Table 5.2 and Table 5.3).

Even though the optional requirements from Table 3.3 were not implemented due to being optional and time constraints arise, the tool passed all the tests successfully for the requirements that were implemented (from Table 3.1 and Table 3.2).

Table 5.1: General case tests.

Test ID	Test Case	Expected Result
T_0_1	Send a signal to stop Agent.	The Agent exits successfully cleaning up any child process spawned.
T_0_2	Send a signal to stop Daemon.	The Daemon exits successfully cleaning up any child process spawned.
T_0_3	Send a signal to stop. *µDetector* (API Server)	The *µDetector* exits successfully cleaning up any child process spawned.
T_0_4	Execute the following valid command on the master node “python3 daemon/main.py”	A Flask instance of the daemon starts on the master node
T_0_5	Execute the following valid command on one of the worker nodes “python3 agent/main.py”	A Flask instance of the monitoring agent starts on the worker node
T_0_6	Execute the following valid command on the user’s machine “python3 detector/main.py”	A Flask instance of the API Server starts on the user’s machine

Table 5.2: Functional Requirements Validation - CLI case tests.

Test ID	Requirement Addressed	Case Type	Test Case	Expected Result
T_1_1	REQ-1	Default Case	Execute the following valid command "python3 cli.py start -f monitoring.json".	A success message and the start of the collection of the system calls.
T_1_2	REQ-1	Negative	Execute the following valid command "python3 cli.py start -f monitoring.json" but "detector.py" (entry point to the API Server) is not running	Error message and the collection of the system calls does not start.
T_1_3	REQ-1	Negative	Execute the following invalid command "python3 cli.py start". No file is provided.	Error message and the collection of the system calls does not start.
T_1_4	REQ-1	Negative	Execute the following valid command "python3 cli.py start -f monitoring.yaml". The command is valid but the file provided is not because the file extension is different from a JSON.	Error message and the collection of the system calls does not start.
T_1_5	REQ-1	Negative	Execute the following valid command "python3 cli.py start -f monitoring.json" where monitoring.json has an invalid content.	Error message and the collection of the system calls does not start.
T_1_6	REQ-2	Default Case	Execute the following valid command "python3 cli.py stop" and agents are collecting system calls.	Success message and the system calls stop being collected.
T_1_7	REQ-2	Negative	Execute the following valid command "python3 cli.py stop" and agents are not collecting system calls.	Success message and the system calls stop (forcefully) being collected.
T_1_8	REQ-3	Default Case	Execute the following valid command "python3 cli.py inspect -f training.json" and provide a valid inspecting file. The agents are collecting the system calls.	Success message and the start of the inspection of the system calls by deploying an intrusion detection instance.
T_1_9	REQ-3	Negative	Execute the following valid command "python3 cli.py inspect -f training.json". inspecting file but no system calls are being collected.	Error message and no intrusion detection instance is deployed.
T_1_10	REQ-3	Negative	Execute the following valid command "python3 cli.py inspect -f training.json" where the file has an invalid structure.	Error message and no intrusion detection instance is deployed.
T_1_11	REQ-3	Boundary	Execute the following valid command "python3 cli.py inspect -f training.json" but the file content is invalid. Invalid cases: 1. Duration is not an integer 2. Type is different from "detection" or "training" (ex.: "abc" instead of "training") 3. Algorithm is different from the algorithms list (ex.: "abc" instead of "bosc")	Error message and no intrusion detection instance is deployed.
T_1_12	REQ-4	Default Case	Execute the following valid command "python3 cli.py status" and the agents are collecting system calls.	Message with the current status.
T_1_13	REQ-4	Negative	Execute the following valid command "python3 cli.py status" but the "detector.py" (entrypoint to the API Server) is not running.	Error message.
T_1_14	REQ-5	Default Case	Execute the following valid command "python3 cli.py alarms".	Complete list of alarms.
T_1_15	REQ-5	Default Case	Execute the following valid command "python3 cli.py alarms -n 5".	List of the latest 5 alarms.
T_1_16	REQ-5	Equivalence and Boundary	Execute the following invalid commands: 1. "python3 cli.py alarms -n ". 2. "python3 cli.py alarms -n -1". 3. "python3 cli.py alarms -n 9223372036854775808". 4. "python3 cli.py alarms -n abc".	Complete list of alarms.
T_1_17	REQ-6 to REQ-10	Default Case	Execute the following valid command "python3 cli.py list –pods". (Do this for every resource).	List of pods (or other resources) that exist in the cluster.
T_1_18	REQ-6 to REQ-10	Negative	Execute the following valid command "python3 cli.py list –pods" but daemon is not running or connection fails.	Error message.
T_1_19	REQ-11	Default Case	Execute the following valid command "python3 cli.py algorithms".	List of algorithms available.
T_1_20	REQ-11	Negative	Execute the following valid command "python3 cli.py algorithms" and the API server is not running.	Error message.
T_1_20	REQ-11	Negative	Execute the following valid command "python3 cli.py algorithms" and the file "algorithms.json" does not exist	Error message.
T_1_21	REQ-12	Default Case	Execute the following valid command "python3 cli.py –help".	Message showing the commands available.
T_1_22	REQ-1 to REQ-12	Negative	Execute an invalid command such as "python3 cli.py abc".	Message showing the commands available.

Table 5.3: Functional Requirements Validation - Dashboard case tests.

Test ID	Requirem ID	Case Type	Test Case	Expected Result
T_2_1	REQ-13	Default Case	Go to the dashboard page, click “Start monitoring” button, click the file input button and choose “monitoring.json” file, click “Continue” (confirm the information is correct) and click “Start Monitoring”.	A successful alert, the page is refreshed and the agents start the collection of the system calls.
T_2_2	REQ-13	Negative	Go to the dashboard page, click the “Start monitoring” button and the “detector.py” (entrypoint to the API Server) is not running.	The button is hidden on the server-side from the user and it is not possible to click it.
T_2_3	REQ-13	Negative	Go to the dashboard page, click “Start monitoring” button, click the file input button and choose an invalid file extension (eg.: ”monitoring.yaml”). Click “Continue”.	The user is presented with the dashboard page making it impossible to continue with the start monitoring steps.
T_2_4	REQ-13	Negative	Go to the dashboard page, click “Start monitoring” button, click the file input button, choose “monitoring.json” file with invalid structure, click “Continue” and “Start Monitoring”.	An error alert is displayed.
T_2_5	REQ-13	Negative	Go to the dashboard page, click the “Start monitoring” button, click the file input button and choose “monitoring.json” file with invalid sysdig args or format and click “Continue” and “Start Monitoring”.	An error alert is displayed.
T_2_6	REQ-13	Boundary	Go to the dashboard page, click the “Start monitoring” button, click the file input button and choose “monitoring.json” file with invalid window size (integer not in the range [1,1000]) and click “Continue” and “Start Monitoring”.	A successful alert is displayed, the page refreshes automatically and the monitoring starts with the default value of 5 seconds.
T_2_7	REQ-14	Default Case	Go to the dashboard page, make sure the agents are monitoring and then click the “Stop monitoring” button and then confirm.	A successful alert is displayed, the page refreshes automatically and the agents stop collecting system calls.
T_2_8	REQ-14	Negative	Go to the dashboard page, make sure the agents are not monitoring anything and then click the “Stop monitoring” button and then confirm.	A successful alert is displayed similar to the default case, the page refreshes and the agents stop forcefully the collection of system calls.
T_2_9	REQ-15	Default Case	Go to the dashboard page, make sure the agents are monitoring, then click the “Start inspecting system calls” button, submit a valid file and confirm.	A successful alert is displayed, the page refreshes automatically and the intrusion detection instance is deployed.
T_2_10	REQ-16	Default Case	Go to the dashboard page and make sure the agents are monitoring and there is at least one intrusion detection instance running. Then click the “Manage instances” button.	A modal appears with a list of all the intrusion detection instances identified by their id. The user can then expand each instance to view more information.
T_2_11	REQ-16	Negative	Go to the dashboard page, make sure the agents are monitoring and there is no intrusion detection instance running.	The “Manage instances” button is blocked making it impossible for the user to click on the “Manage instances” button.
T_2_12	REQ-17	Default Case	Go to the dashboard page and make sure there is at least one instance running and then click the “Stop all instances” button.	A successful message, the page is refreshed and the instance gets deleted.
T_2_13	REQ-18	Default Case	Go to the dashboard page, make sure there is at least one instance running, expand any of the instances and click the “Stop instances” button.	A successful message, the page is refreshed and the instance gets deleted.
T_2_14	REQ-19 to REQ-21	Default Case	Go to the dashboard page, make sure the daemon is running and system calls are being collected.	An overview box with information about the agents running on the nodes and monitoring information such as duration and sysdig configurations is displayed. A real-time graph and statistics of the system calls being monitored are also presented to the user and automatically refresh the data.
T_2_15	REQ-19 to 21, REQ-28 to 30	Negative	Go to the dashboard page, make sure the daemon is not running.	An error message appears saying the daemon is not running or was not detected by the tool.
T_2_16	REQ-19 to REQ-21	Negative	Go to the dashboard page, make sure the daemon is running and no system calls are being collected.	An overview box with information about the agents running on the nodes is displayed.
T_2_17	REQ-22	Default Case	Go to the dashboard page, make sure the daemon is running and system calls are being collected. Click on the export button that appears on the top right of the system calls chart.	The download of the graph in the PNG format starts immediately.
T_2_18	REQ-23	Default Case	Go to the dashboard page and make sure that at least one intrusion detection instance is running and generating alarms.	A partial list in the form of a table with the latest 5 alarms.
T_2_19	REQ-24	Default Case	Go to the alarms page and make sure that at least one intrusion detection instance is running and generating alarms.	A complete list in the form of a table of all the alarms.
T_2_20	REQ-25	Default Case	Go to the alarms page and make sure that at least one intrusion detection instance is running and generating alarms. Make sure alarms have been listed in the alarms table and characters are entered in the search placeholder. An example is to search for “worker-1” and find all alarms that contain information about “worker-1”	Table filters the information and presents all entries where the search text appears.
T_2_21	REQ-26	Default Case	Go to the alarms page and make sure that the table contains alarms. Click the delete alarms button.	The table is cleared and appears empty.
T_2_22	REQ-26	Negative	Go to the alarms page and make sure that the table contains no alarms. Click the delete alarms button.	The table is cleared and appears empty.
T_2_23	REQ-27	Default Case	Go to the alarms page and make sure that the table contains alarms. Click the download alarms button	The download of the alarms.json file starts immediately. The file contains a JSON list of all the alarms.
T_2_24	REQ-27	Negative Case	Go to the alarms page and make sure that the table contains no alarms. Click the download alarms button	The download of the alarms.json file starts immediately. The file contains an empty JSON list.
T_2_25	REQ-28	Default Case	Access the web dashboard, navigate to the resources page and expand the resources tabs (pods, services, . . .).	The user navigates to the resources page and views tables that describe the resources of the cluster such as pods and namespaces.
T_2_26	REQ-29 and REQ-30	Default Case	Access the web dashboard and navigate to the help page.	The user navigates to the help page and views static helpful information.

5.3 Non-Functional Requirements Validation

This section describes the whole Non-Functional requirements validation process. We will focus on performance and scalability which were the two non-functional requirements elicited during the requirements phase. Our goal is to prove the *µDetector* tool does not interfere with the normal functioning of the microservices application. We also want to guarantee that when the number of system calls and resources of the cluster increases, there is no negative impact on the application. Therefore we started by defining the scenarios which we want to test and the setup of the environment.

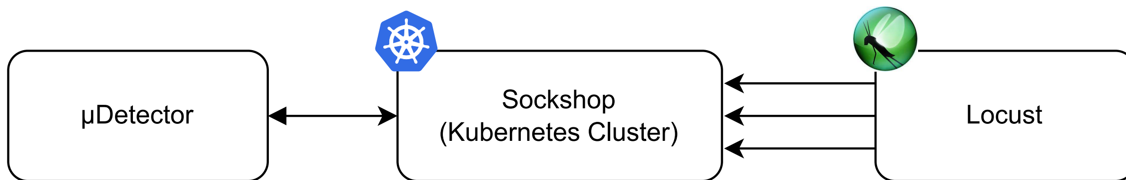


Figure 5.1: Experimental Setup of the Non-Functional Requirements Validation.

Figure 5.1 shows how we will conduct the tests. In more detail, we have a Kubernetes cluster with a variable number of worker nodes (maximum of three worker nodes) that interacts with the *µDetector* tool which is running on another machine. The Kubernetes cluster runs the Sock Shop application while Locust [109], the load testing tool used to simulate users accessing the application, runs on a separate machine.

Concerning the experiments, we had to, first, setup a Kubernetes cluster. To achieve this we used four machines: one Master Node and three worker nodes. Out of the three worker nodes, two are normal Kubernetes worker nodes and the other is an Edge node that relies on KubeEdge. With the help of the *kubeadm* and *kubectl* commands we create the cluster and deploy the Sock Shop [24] testbed. Before deploying the Sock Shop, it is necessary to install a network plugin so that the pods can communicate with each other. We chose Flannel [112] due to being easy to configure as it only requires a YAML file.

For the cluster to support Edge nodes, we have to install KubeEdge (*keadm* more specifically), initialize the cloudcore service on the master node using *keadm init* and retrieve a join token using *keadm gettoken*. This token was used on the Edge node together with the command *keadm join*.

The upload and download speed of the network connection is `1000 Mbit/s` on cable. The specs of the machines are as follows:

- ***µDetector* tool (physical machine)** - Apple M1, 8-core with 16GB of RAM.
- **Master node (virtual machine)** - Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz with 16GB of RAM
- **Worker node 1 (virtual machine)** - Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz with 16GB of RAM

- **Worker node 2 (virtual machine)** - Intel(R) Xeon(R) Silver 4116 CPU @ 2.10GHz with 8GB of RAM
- **Worker node 3 (virtual machine)** - Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with 8GB of RAM
- **Locust tool (physical machine)** - AMD Ryzen 5 3600 6-Core Processor with 16GB of RAM

To guarantee the properties of performance and scalability we increase the collection of system calls, worker nodes and pod replicas. Then, we monitor the requests throughput, response times, and CPU and memory usage of the machines from the cluster (master and worker nodes), and the machine that runs the *µDetector* tool. This happens over a period of 30 minutes. We used a standard configuration of the monitoring of the system calls: we monitor the front-end service of the Sock Shop where we collect the system calls in batches of 5 seconds with syscalls_compression flag set to True. The algorithm for the detection was BoSC. Table 5.4 depicts the 4 scenarios.

Table 5.4: Experimental Scenarios

	1 Worker Node	3 Worker Nodes
Without *µDetector*	Scenario 1	Scenario 2
With *µDetector*	Scenario 3	Scenario 4

Scenarios 1 and 2 serve as stable runs without the *µDetector* tool, while scenarios 3 and 4 have the *µDetector* tool running. Scenarios 2 and 4 will have three worker nodes where one of them is deployed on Edge using KubeEdge. They also have three replicas of the front-end service instead of one to represent a cluster that was scaled up. The workload for Scenarios 1 and 3 where there are fewer resources (only one replica of the front-end service and one Worker node) has a different and less demanding workload than Scenarios 2 and 4. Figure 5.2 shows the two workloads used. Scenarios 1 and 3 used the `constant-wl` and Scenarios 2 and 4 used the `variable-wl`. The `constant-wl` workload means there is always a constant number of users performing requests to the Sock Shop throughout the duration of the experiment. With the `variable-wl` workload we try to simulate a scenario with several high and low intervals of users accessing the Sock Shop to achieve a better representativity of real-world user workloads.

Results and Discussion

In this subsection, we present and discuss the results obtained during the second approach of the validation which consists of an experimental campaign where we evaluate the 4 scenarios previously described.

Figure 5.3 and Figure 5.4 show, respectively, the system calls collected and the requests throughput for each scenario over 30 minutes. For simplicity's sake, we omit the failed requests because there were none in each scenario. Therefore only the successful requests are shown in Figure 5.4. As we can observe, the system

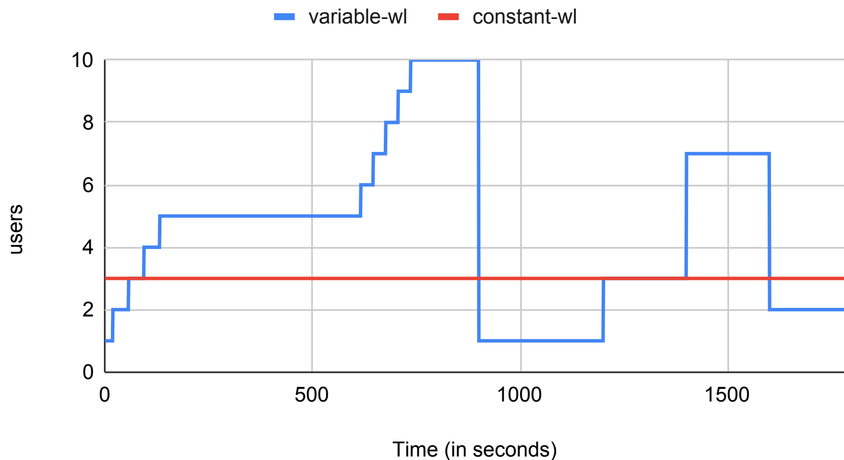


Figure 5.2: Locust workloads. Scenarios 1 and 3 used the constant-wl and Scenarios 2 and 4 used the variable-wl.

calls graph follows the same trend as the requests throughput graph. This is to be expected and in Scenario 3, the *µDetector* tool ended up collecting fewer system calls (around 50 000 system calls per second) due to having a different and less demanding workload. In Scenario 4, there was a peak of around 130 000 system calls per second and the workload was heavier (more system calls being collected per second) than in Scenario 3. Nevertheless, the tool was able to deal with the increasing number of system calls, pods and nodes and there was no negative impact on the requests - 0 failed requests. We also collected the response times for the 50% (median) and 95% percentile. Scenarios 1 and 3 had a slightly lower response time, inferior to 10ms, as we can observe in Figure 5.5. This can be explained due to the workload being less demanding in both scenarios as they also have only one Worker node. This means that the tool was able to deal with the increase in system calls, pods and worker nodes.

Figure 5.6 depicts the CPU usage in the *µDetector* machine, the master node machine and the first worker node machine. We decided not to show the CPU usage for the second and third worker nodes to not overwhelm the reader since the results are similar to the first worker node. As we can observe, for the *µDetector* machine, the CPU usage floated around the 10% and 30% for all the scenarios. Therefore with or without the *µDetector* tool monitoring system calls, we can deduce that performance is not affected on the machine that runs the tool. For the Master node machine we can observe a similar behavior but this time the CPU usage varies between 4% and 10%. This is expected, as this machine is only running a daemon that is idle during the monitoring of the system calls and only takes action if a command is provided by the user.

For the worker node machine, we can already observe differences. From Scenarios 1 and 3 to Scenarios 2 and 4, the CPU usage drops less than 10%. This happens due to the scaling of the worker nodes taking place in Scenarios 2 and 4 - we have the pods of the Sock Shop application split across different nodes, therefore, the

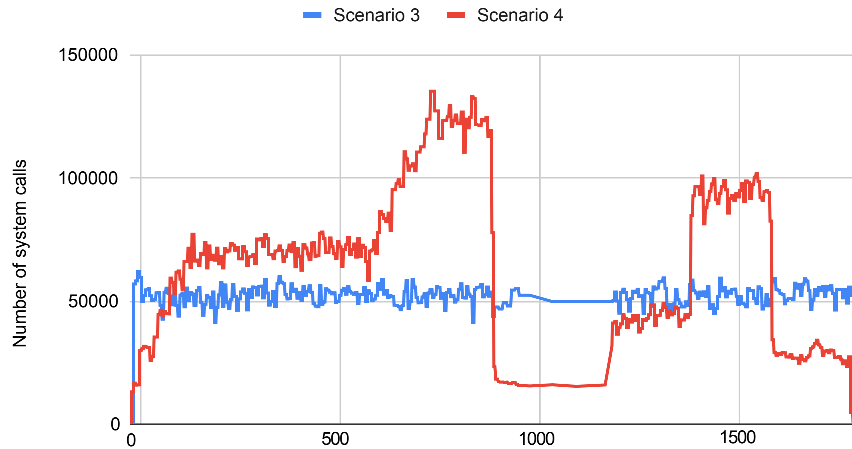


Figure 5.3: System calls collected

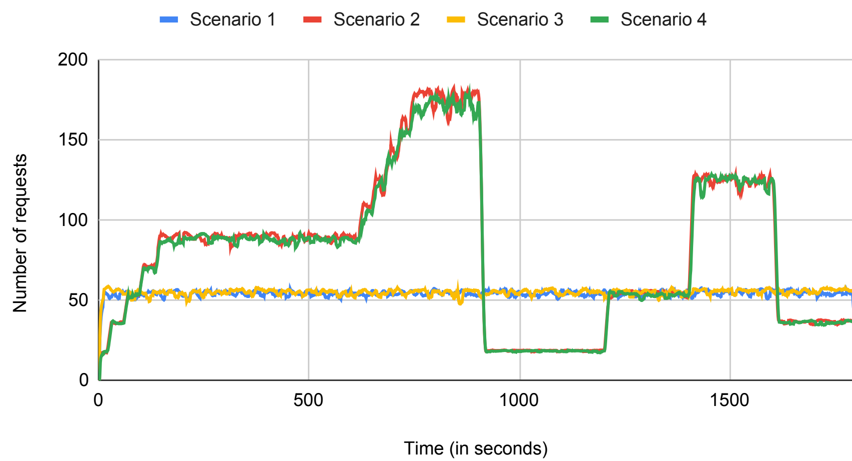
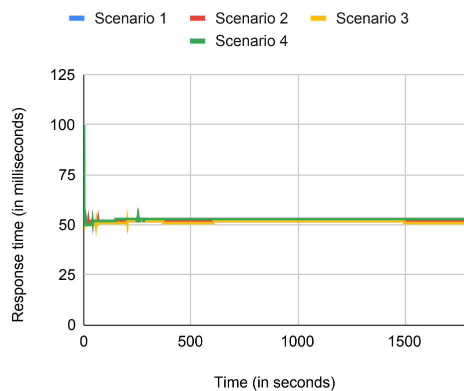


Figure 5.4: Requests Throughput



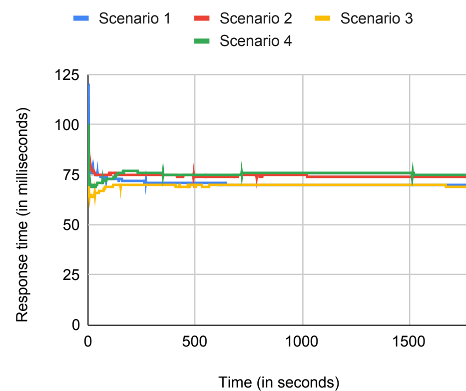


Figure 5.5: Median Response Time and 95% Percentile Response Time

load each node has to deal with, decreases in these scenarios. Nevertheless, if we sum all the work done by the worker nodes in Scenarios 2 and 4 it is still higher than in Scenarios 1 and 3 (as expected) where there is only one worker node, due to a heavier workload. Another difference we can see is that from Scenario 1 to Scenario 3 and from Scenario 2 to Scenario 4, there is an increase of around 5% of CPU usage. Most likely, this has to do with the monitoring agents that are running in the background and collecting the system calls. This increase in CPU usage is expected and represents the cost of using the *µDetector* tool. We consider this to be a low value and that the tool performs well on the worker nodes having a low impact on the system.

We also collected the memory usage. The results appear in Figure 5.7 and are similar to the CPU usage in the sense that the memory usage for the *µDetector* machine and the Master machine is almost the same in every scenario. The only difference is in the worker nodes where the memory usage increases a maximum of 7% and is stable across the entire 30-minute experiment. Again, we consider this to be the cost of having the tool running on the worker nodes and that, nevertheless, it performs well according to the stipulated requirements and therefore, the impact on the system is acceptable.

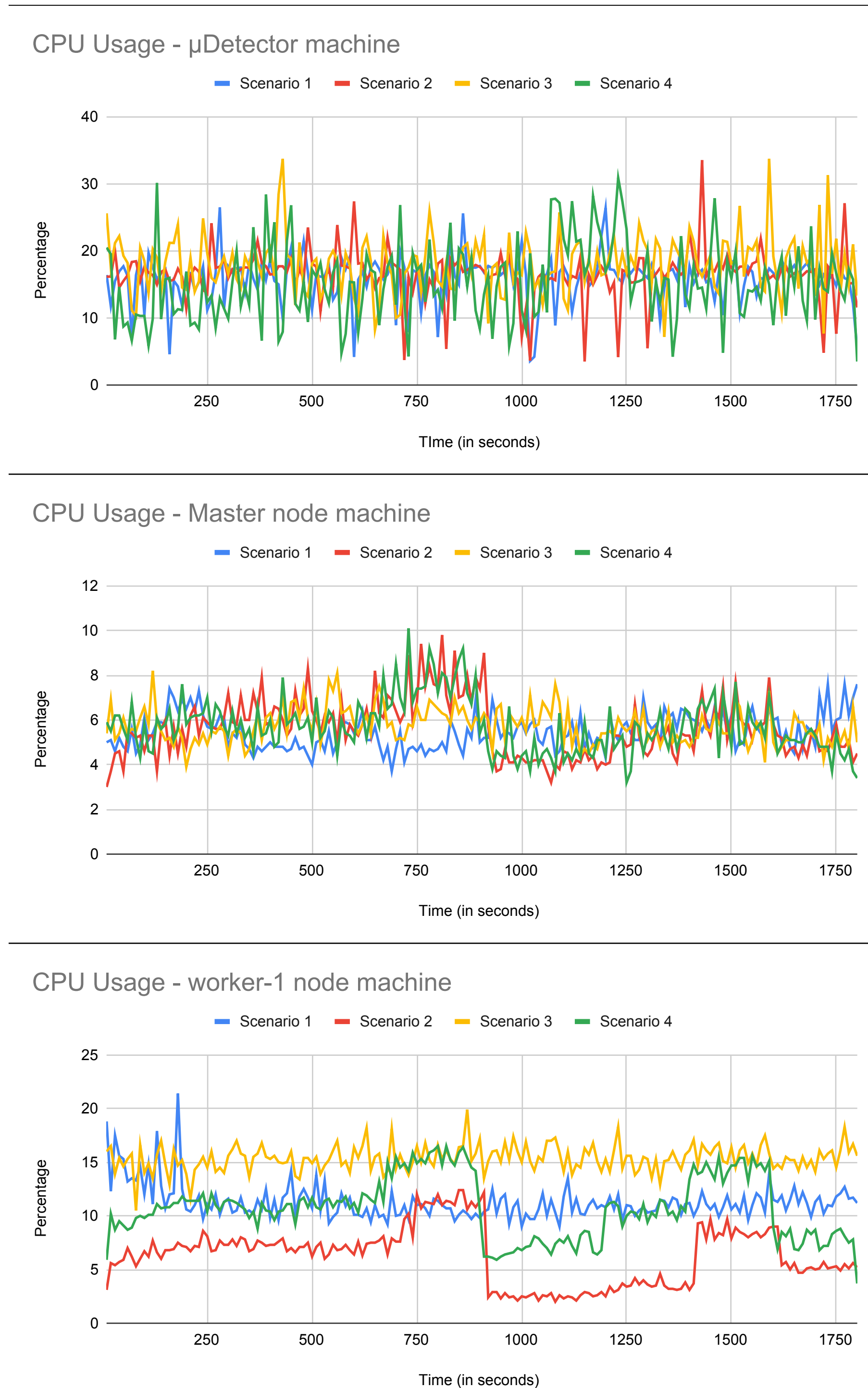


Figure 5.6: CPU Usage - *μDetector* machine (first graph), CPU Usage - Master node machine (second graph) and CPU Usage - worker-1 node machine.

Figure 5.7: Memory Usage - *µDetector* machine (first graph), Memory Usage - Master node machine (second graph) and Memory Usage - worker-1 node machine (third graph)

Chapter 6

Project Management

Project management aims to ensure that a project meets all the requirements within its time and cost constraints. This chapter explains all the details related to the project management of this dissertation. Specifically, the methodology used during the entire dissertation, the work plan for both semesters and how it went, and a risk assessment.

6.1 Methodology

At the start of a software project, there may not be clearly defined goals and it is usually too soon to create a strict plan to follow. Project management involves dealing with change, and given the nature of this dissertation and the state-of-the-art technologies utilized (such as Kubernetes), requirements are subject to change. Therefore, traditional software development methodologies which are based on predefined stages of the software development life cycle become a fallible choice.

The Agile Methodology [113] tries to address the issues of traditional methodologies by encouraging flexible responses to change. For that reason, we decided to follow an Agile approach, using concepts and elements of different Agile frameworks but with a particular focus on the Scrum framework. Scrum is a framework that allows iterative and incremental development by dividing the project into diverse time-boxes called Sprints [114]. In Scrum, there is a Product Backlog that contains a prioritized list of features to be developed in time slots of two weeks (this period may vary) called Sprints. Sprints start with a meeting to choose the items from the backlog that the developers intend to deliver at the end of each Sprint. This is a very flexible methodology that we ended up following during this dissertation.

6.2 Work Plan

During the first semester, informal meetings were held with the two supervisors where we decided what tasks should be executed for the next two weeks until the next meeting. We tried to follow the actual plan depicted in the first half of Figure 6.1 as a Gantt chart. However, there were some setbacks as we can observe in Figure 6.2. The COVID measures lead to remote meetings and the extension of the deadline for the intermediate report by one week. The writing of the intermediate report started early to line up with the background and related work analysis. The requirements took longer than expected due to indecision and inexperience in such a task. Nevertheless, we were able to complete the established plan and acquire valuable insight into project management.

In the second semester, we defined Sprints based on the defined requirements. Every possible change or setback forced us to a new arrangement of the requirements and tasks which we redefined at the beginning of each Sprint. The second half of Figure 6.1 shows the Gantt chart of the expected work plan for the second semester. The second half of Figure 6.2 shows the Gantt chart of the actual work plan for the second semester. As expected, the plan did not go exactly according to Figure 6.1. Unforeseen problems occurred and we had to constantly adapt. Some tasks took longer than expected but we were able to compensate for other tasks. Nevertheless, the requirements were fulfilled, the tool was implemented and validated successfully and the project was completed in time.

6.3 Risk Assessment

The early identification of risks allows us to anticipate future setbacks and come up with a mitigation plan to assure the project goes down the expected path. Therefore, it is a technique that should be used in software projects and is described in this section. To have a clear understanding of the evolution of risks we decided to include only the initial iteration of risks and the final iteration. Every month we would update the risks accordingly. Table 6.1 identify and describe the risks that could prevail in the project during the first months (September and October). Table 6.2 refers to the end of the first semester (January) and Table 6.3 refers to the final months of this dissertation (June and July). For each identified risk there is a corresponding **ID**, **Description** and **Mitigation Plan**. Each risk also has a corresponding **Impact** and **Likelihood**, both on a scale of **Low**, **Medium** and **High**. When it comes to the Impact, Low means that the risk will not have a significant impact and can probably be ignored; Medium means there is a greater impact that can be managed with some effort; High means there is a large impact that can cause, for example, delays and therefore needs to be addressed with some degree of priority. When it comes to the Likelihood, Low means there is a chance of less than 40% of happening at least one time during the project; Medium means there is a chance between 40% and 70% of happening at least one time during the project; and High means there is a chance of higher than 70% of happening at least one time during the project. Figure 6.3 depicts a

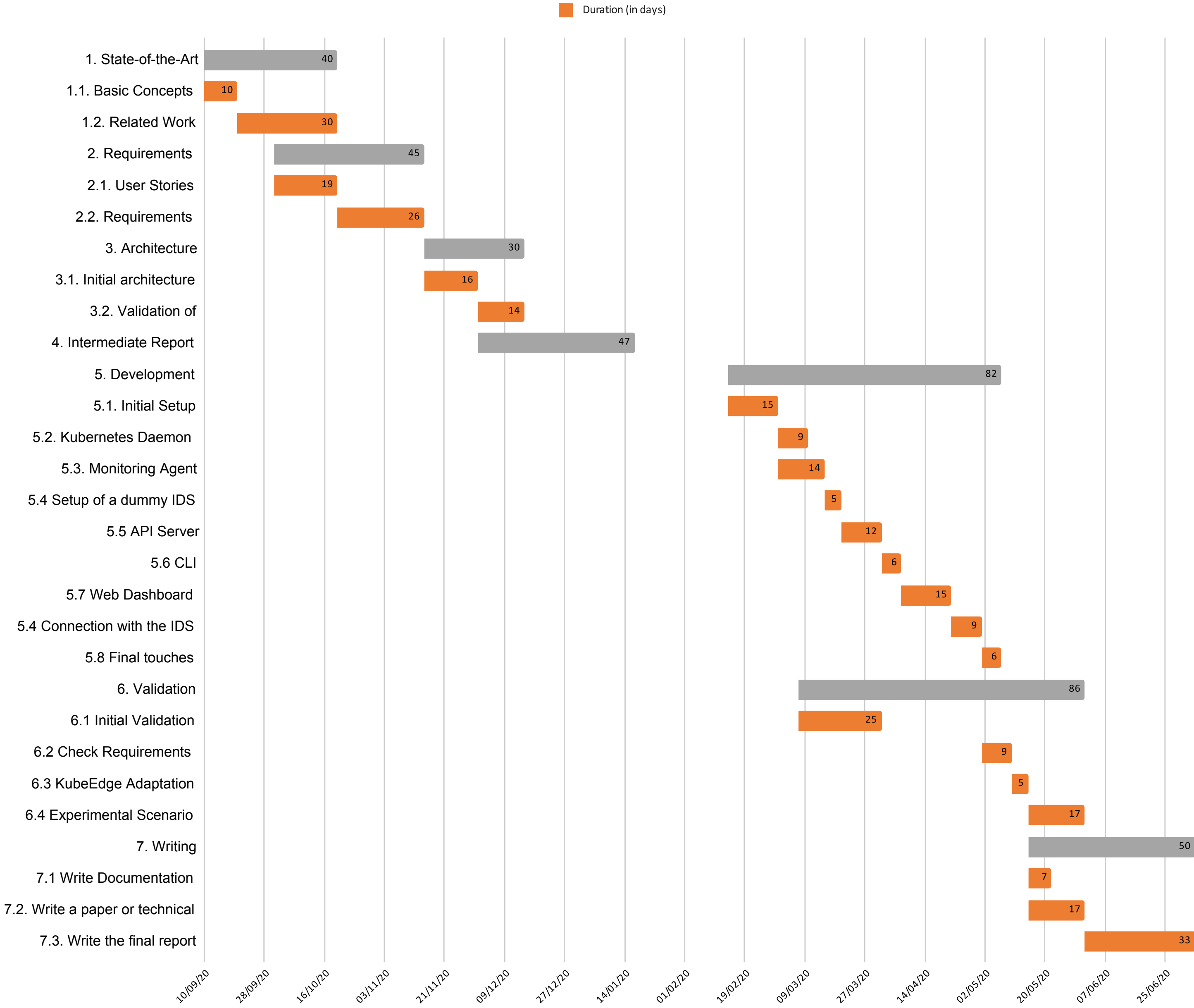


Figure 6.1: Gantt chart for the expected work plan. Grey bars represent a major chapter and orange bars represent smaller task. Each bar has a number that corresponds to the duration in days.

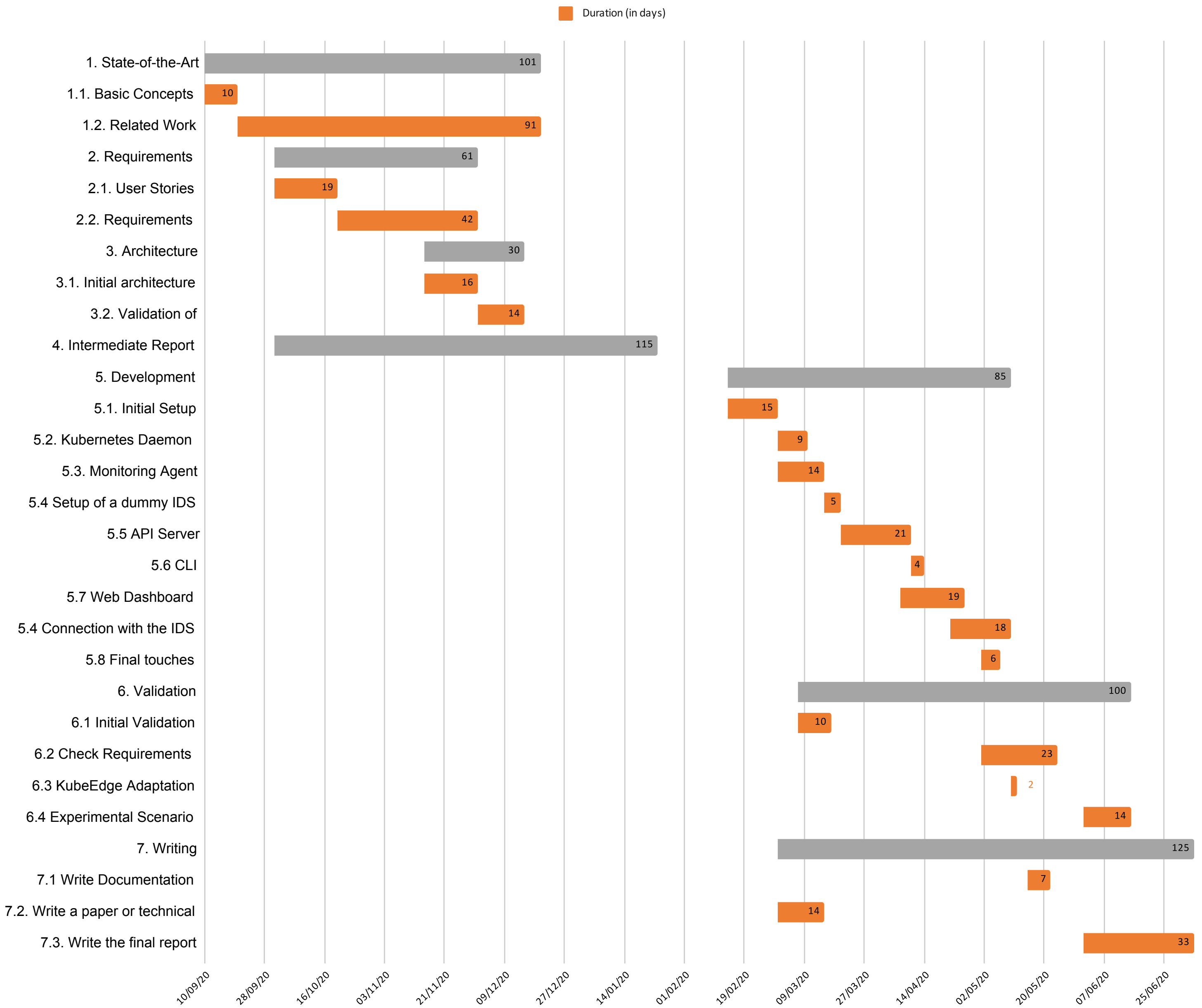


Figure 6.2: Gantt chart for the actual work plan. Grey bars represent a major chapter and orange bars represent smaller task. Each bar has a number that corresponds to the duration in days.

visual representation of the risks at the beginning of the first semester, end of the first semester/beginning of the second semester, and end of the second semester, to better assist in the decision-making process. It also depicts the risks that were completely mitigated or that do not pose a threat to the success of the project anymore. Those risks were removed from the Tables as they were completely mitigated.

At the end of each phase, risks were **partially mitigated** which we can observe through the decrease in the level of likelihood and impact when comparing Table 6.1 with Table 6.2 and Table 6.3. By using the proper mitigation techniques we can reduce the impact and likelihood of the risks to a point where the success of the dissertation is not compromised such as completely mitigating a risk or reducing the impact and likelihood until the risk reaches the "green zone" on the matrix. Risks **R-2**, **R-4**, **R-5**, **R-6**, **R-7** and **R-9** were considered to be completely mitigated at the end of the project as they did not pose a threat to the success of the project anymore according to our analysis. **R-1**, **R-3** and **R-8** were partially mitigated. These were harder to deal with, however, we can still consider the project to be successful as the impact of these risks on the project was arguably low.

Table 6.1: Risk Assessment for the project at the beginning of the first semester.

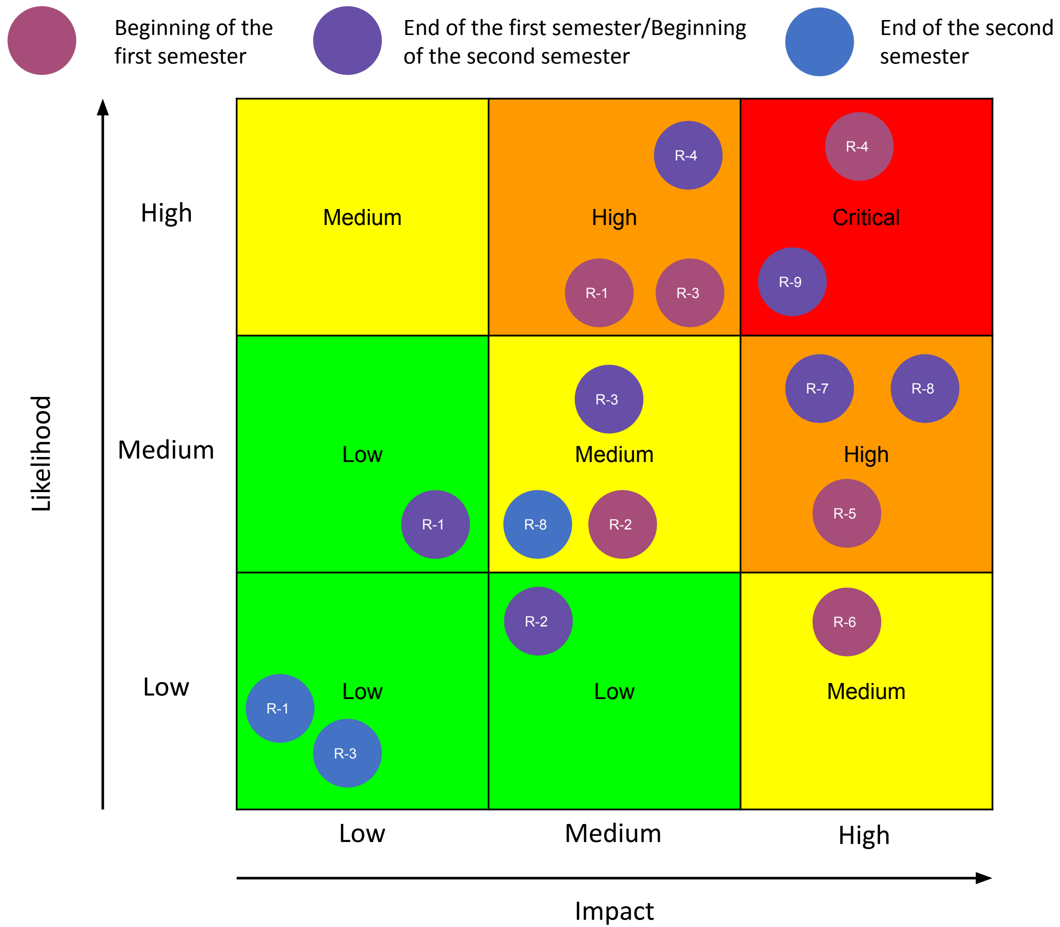
Id	Description	Mitigation Plan	Impact	Likelihood
R-1	**Requirements are too broad.** It is necessary specific knowledge and experience in requirements gathering to conceptualize the solution. The requirements may be too vague to work with and affect the decision making process and development.	Talk to the client, understand the users needs and get a clear vision of the project. Redefine and be more specific in the user stories but not too much as it might give little to no maneuver options in the development phase.	Medium	High
R-2	**Lack of resources.** If the necessary resources are unavailable some objectives could not be attainable. For example, a real scenario where there is a Kubernetes cluster requires many hosts. This could be hard to provide and demonstrate the project in a real-case scenario where scalability is fundamental.	Design the proposed solution to be scalable but make use of a small number of resources such as three hosts (a master, a worker and an "outside" machine).	Medium	Medium
R-3	**Lack of experience and complexity of the technology.** The project requires knowledge about designing web dashboards and working with complex emerging technologies (such as Kubernetes). This could present a steep learning curve that could cause unexpected delays in the development phase and changes in the requirements.	Extensive research and practice with the technology before the development phase starts.	Medium	High
R-4	**Poor estimation.** Lack of experience in planning the duration of tasks causing unaccounted delays.	Apply estimation techniques. Frequently update the estimations.	High	High
R-5	**Poor planning of the submission of the article.** The article might not be submitted in time for the proper conference (that is still to be defined).	Investigate possible conferences and plan ahead of time for an appropriate date	High	Medium
R-6	**Relying on novel techniques and approaches of intrusion detection.** The tool to be developed relies on techniques that are still being researched and investigated.	Abstract from the approaches as much as possible and be aware of possible limitations.	High	Low

Table 6.2: Risk Assessment for the project at the end of the first semester/beginning of the second semester.

ID	Description	Mitigation Plan	Impact	Likelihood
R-1	**Requirements are too broad.** It is necessary for specific knowledge and experience in requirements gathering to conceptualize the solution. The requirements may be too vague to work with and affect the decision-making process and development.	Talk to the client, understand the users' needs and get a clear vision of the project. Redefine and be more specific in the user stories but not too much as it might give little to no maneuver options in the development phase.	Low	Medium
R-2	**Lack of resources.** If the necessary resources are unavailable some objectives could not be attainable. For example, a real scenario where there is a Kubernetes cluster requires many hosts. This could be hard to provide and demonstrate the project in a real-case scenario where scalability is fundamental.	Design the proposed solution to be scalable but make use of a small number of resources such as three hosts (a master, a worker and an "outside" machine).	Medium	Low
R-3	**Lack of experience & complexity of the technology.** The project requires knowledge about designing web dashboards and working with complex emerging technologies (such as Kubernetes and Django). This could present a steep learning curve that could cause unexpected delays in the development phase and changes in the requirements.	Extensive research and practice with the technology before the development phase starts.	Medium	Medium
R-4	**Poor estimation.** Lack of experience in planning the duration of tasks causing unaccounted delays.	Apply estimation techniques. Frequently update the estimations.	Medium	High
R-7	**Difficulty in streaming system calls.** Research and tests conducted showed that the number of system calls to be collected could be huge. This could present an obstacle to the efficient streaming of system calls over the network impacting the performance of the devised solution	Find an alternative solution such as transferring batches of system calls over the network. Use a high-performance communication technology.	High	Medium
R-8	**Poor definition of the validation plan.** A poor definition may cause delays in the experiments and the results might not be relevant.	Extensive research on experimentation methodologies, acquire insight from previous work, and run small tests	Medium	High
R-9	**Slow implementation.** Unexpected errors or challenges could arise during implementation leading to delays in the project	Prioritize tasks and focusing on the most important ones. Also, allocate a chunk of time to unaccounted problems.	High	High

Table 6.3: Risk Assessment for the project at the end of the second semester.

ID	Description	Mitigation Plan	Impact	Likelihood
R-1	**Requirements are too broad.** It is necessary for specific knowledge and experience in requirements gathering to conceptualize the solution. The requirements may be too vague to work with and affect the decision-making process and development.	Talk to the client, understand the users' needs and get a clear vision of the project. Redefine and be more specific in the user stories but not too much as it might give little to no maneuver options in the development phase.	Low	Low
R-3	**Lack of experience & complexity of the technology.** The project requires knowledge about designing web dashboards and working with complex emerging technologies (such as Kubernetes and Django). This could present a steep learning curve that could cause unexpected delays in the development phase and changes in the requirements.	Extensive research and practice with the technology before the development phase starts.	Low	Low
R-8	**Poor definition of the validation plan.** A poor definition may cause delays in the experiments and the results might not be relevant.	Extensive research on experimentation methodologies, acquire insight from previous work, and run small tests	Medium	Medium



Completely Mitigated Risks at the beginning of the project: **None**

Completely Mitigated Risks at the end of the first semester: **R-5, R-6**

Completely Mitigated Risks at the end of the second semester: **R-2, R-4, R-5, R-6, R-7, R-9**

Figure 6.3: Risk Exposure Matrix.

Chapter 7

Conclusion

In recent years, cloud-computing adoption has been increasing rapidly as we have witnessed a massive adoption of containers and microservices throughout organizations worldwide [3]. Their goal is to quickly build and deploy robust applications to answer the increasing customers' demand and generate more profit. Container Orchestrators, such as Kubernetes [6], have gained considerable attention and have become the *de facto* platforms used to manage applications with a large number of containers. The functionalities they provide improve the development and deployment phases, and reduce infrastructure costs through automation, allowing businesses to grow quicker. However, security concerns arise with these emerging technologies and the need to monitor and keep microservices-based applications safe is of the utmost importance [10]. The lack of tools that help keep these systems secure motivates organizations to come up with new solutions and approaches.

In this work, we presented *μDetector*, an intrusion detection tool for microservices applications. *μDetector* supports the techniques previously developed by our research group and incorporates them into a tool that users can leverage to monitor their application and improve the observability of their system. It works by deploying monitoring agents on Kubernetes and KubeEdge applications to collect systems calls from their nodes. Afterward, it transfers the system calls to an intrusion detection module that generates alarms in the presence of abnormal behavior. The users can then interact with the tool through a CLI or a Web Dashboard. The validation of the tool focused on functional testing and an experimental campaign that shows that the *μDetector* tool fulfills its capabilities.

For future work, we could add more functionalities such as the download of the intrusion detection profiles created and the addition of new algorithms. We could also provide support for other orchestration platforms. Additionally, we could leverage a stronger validation plan and focus on making the whole system calls collection process more efficient.

The *μDetector* tool presented itself as a further step in strengthening container security and microservices applications in cloud environments through the use of state-of-the-art intrusion detection techniques.

References

[1] Martin Fowler and James Lewis. Microservices. https://martinfowler.com/articles/microservices.html, 2014. Accessed: 2021-10-09.

[2] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: An experience report, 2015.

[3] Cloud Native Computing Foundation. Cncf survey 2019. https://www.cncf.io/wp-content/uploads/2020/08/CNCF_Survey_Report.pdf, 2019. Accessed: 2021-11-28.

[4] VMWare. Why use containers vs. vms? https://www.vmware.com/topics/glossary/content/vms-vs-containers.html. Accessed: 2022-01-10.

[5] IBM Cloud Team. Containers vs. virtual machines (vms): What's the difference? https://www.ibm.com/cloud/blog/containers-vs-vms. Accessed: 2022-01-10.

[6] Kubernetes. https://kubernetes.io/. Accessed: 2021-11-28.

[7] Apache mesos. http://mesos.apache.org/. Accessed: 2021-11-28.

[8] Docker swarm. https://docs.docker.com/engine/swarm/. Accessed: 2021-11-28.

[9] Snyk. Kubernetes security: Common issues and best practices. https://snyk.io/learn/kubernetes-security/. Accessed: 2022-01-10.

[10] Murugiah Souppaya, John Morello, and Karen Scarfone. Application container security guide, 2017.

[11] Charles Owen-Jackson. What does the rise of edge computing mean for cybersecurity? https://www.kaspersky.com/blog/secure-futures-magazine/edge-computing-cybersecurity/31935/. Accessed: 2022-01-10.

[12] K A Scarfone and P M Mell. Guide to intrusion detection and prevention systems (IDPS), 2007.

[13] Falco. The falco project. https://falco.org/, 2021. Accessed: 2021-12-28.

[14] Service-oriented development on NetKernel- patterns, processes & products to reduce system complexity | CloudEXPO. https://web.archive.org/web/20180520124343/http://www.cloudcomputingexpo.com/node/80883. Accessed: 2021-10-09.

[15] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. In Manuel Mazzara and Bertrand Meyer, editors, *Present and Ulterior Software Engineering*, pages 195–216. Springer International Publishing, 2017. Accessed: 2021-10-09.

[16] Mike Loukides Swoyer, Steve. Microservices adoption in 2020. `https://www.oreilly.com/radar/microservices-adoption-in-2020/`, 2020. Accessed: 2021-10-09.

[17] Chris Richardson. Pattern: Circuit breaker. `https://microservices.io/patterns/reliability/circuit-breaker.html`, 2018. Accessed: 2021-10-22.

[18] Mike Jacobs and Ed Kaim. What are microservices? `https://docs.microsoft.com/en-us/devops/deliver/what-are-microservices`, 2021. Accessed: 2021-10-21.

[19] Atlassian. What is devops? `https://www.atlassian.com/devops`. Accessed: 2021-10-21.

[20] Josh Evans. Mastering chaos - a netflix guide to microservices. `https://www.infoq.com/presentations/netflix-chaos-microservices/`, 2016. Accessed: 2021-10-21.

[21] Alon Girmonsky. Microservices vs monolith at different stages of the sdlc. `https://up9.com/microservices-monolith-sdlc`, 2020. Accessed: 2022-01-05.

[22] Dmitriy Konstantynov. Microservices use cases. `https://alpacked.io/blog/microservices-use-cases/`, 2020. Accessed: 2021-10-21.

[23] DescartesResearch. Teastore. `https://github.com/DescartesResearch/TeaStore`, 2021. Accessed: 2021-10-09.

[24] Weaveworks. Sock shop : A microservice demo application. `https://github.com/microservices-demo/microservices-demo`, 2021. Accessed: 2021-10-09.

[25] FudanSELab. Train ticket: A benchmark microservice system. `https://github.com/FudanSELab/train-ticket`, 2021. Accessed: 2021-10-09.

[26] Joakim von Kistowski, Simon Eismann, Norbert Schmitt, Andre Bauer, Johannes Grohmann, and Samuel Kounev. TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 223–236. IEEE, 2018.

[27] Weaveworks. Sock shop: A microservice demo application. `https://microservices-demo.github.io/`, 2017. Accessed: 2021-10-09.

[28] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chenjie Xu, Chao Ji, and Wenyun Zhao. Benchmarking microservice systems for software engineering research. In Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman, editors, *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 323–324. ACM, 2018.

[29] Nane Kratzke. About microservices, containers and their underestimated impact on network performance, 03 2015.

[30] Rani Osnat. A brief history of containers: From the 1970s till now. `https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016`, 2020. Accessed: 2022-01-10.

[31] Michael Kerrisk. chroot(2) — linux manual page. `https://man7.org/linux/man-pages/man2/chroot.2.html`, 2021. Accessed: 2022-01-10.

[32] Michael Kerrisk. cgroups(7) — linux manual page. `https://man7.org/linux/man-pages/man7/cgroups.7.html`, 2021. Accessed: 2022-01-10.

[33] D2iQ. A brief history of containers. `https://d2iq.com/blog/brief-history-containers`, 2018. Accessed: 2022-01-10.

[34] Docker. Empowering app development for developers. `https://www.docker.com/`. Accessed: 2022-01-10.

[35] IBM Cloud Education. Containerization. `https://www.ibm.com/cloud/learn/containerization`, 2021. Accessed: 2022-01-10.

[36] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and Y. C. Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th International Middleware Conference*, Middleware '16, New York, NY, USA, 2016. Association for Computing Machinery.

[37] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access*, 7:52976–52996, 2019.

[38] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. *Computer Security Division, Information Technology Laboratory, National*, 2011.

[39] The Linux Foundation. Open container initiative. `https://opencontainers.org/`. Accessed: 2021-11-28.

[40] Cloud native computing foundation. `https://www.cncf.io/`. Accessed: 2021-11-28.

[41] Docker. Docker overview. `https://docs.docker.com/get-started/overview/`, 2021. Accessed: 2021-10-16.

[42] Kubernetes. What is kubernetes? `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`, 2021. Accessed: 2021-10-10.

[43] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at google with borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17. ACM, 2015. Accessed: 2021-10-09.

[44] Red Hat. What is container orchestration? `https://www.redhat.com/en/topics/containers/what-is-container-orchestration`, 2019. Accessed: 2021-10-10.

[45] Kubernetes. Kubernetes pods. `https://kubernetes.io/docs/concepts/workloads/pods/`, 2021. Accessed: 2021-10-10.

[46] Kubernetes. Kubernetes components. `https://kubernetes.io/docs/concepts/overview/components/`, 2021. Accessed: 2021-10-10.

[47] KubeEdge. Kubeedge. `https://kubeedge.io/`, 2021. Accessed: 2022-01-10.

[48] KubeEdge. Why kubeedge. `https://kubeedge.io/en/docs/kubeedge/`, 2020. Accessed: 2022-01-10.

[49] Docker. How nodes work. `https://docs.docker.com/engine/swarm/how-swarm-mode-works/nodes/`, 2020. Accessed: 2021-10-11.

[50] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, page 305–320, USA, 2014. USENIX Association.

[51] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI 11)*, Boston, MA, March 2011. USENIX Association.

[52] Mesosphere. Marathon. `https://mesosphere.github.io/marathon/`, 2018. Accessed: 2021-10-11.

[53] Nomad by hashicorp. `https://www.nomadproject.io/`. Accessed: 2021-11-28.

[54] Adrian Todorov. Why you should take a look at nomad before jumping on kubernetes. `https://atodorov.me/2021/02/27/why-you-should-take-a-look-at-nomad-before-jumping-on-kubernetes/`, 2021. Accessed: 2021-10-16.

[55] Hashicorp. Architecture. `https://www.nomadproject.io/docs/internals/architecture`, 2018. Accessed: 2021-10-16.

[56] TutorialsPoint. What is a socket? `https://www.tutorialspoint.com/unix_sockets/what_is_socket.htm`, 2022. Accessed: 2022-01-06.

[57] WHATWG. Html standard. `https://html.spec.whatwg.org/multipage/web-sockets.html`, 2022. Accessed: 2022-01-15.

[58] MDN contributors. The websocket api (websockets). https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API, 2021. Accessed: 2022-01-15.

[59] Ian Fette and Adam Barth. The websocket protocol. https://datatracker.ietf.org/doc/draft-abarth-thewebsocketprotocol/, 2011. Accessed: 2022-01-11.

[60] Miguel Grinberg. Flask-socketio. https://flask-socketio.readthedocs.io/en/latest/, 2018. Accessed: 2022-01-04.

[61] Sylvain Hellegouarch. ws4py - a websocket package for python. https://ws4py.readthedocs.io/en/latest/, 2018. Accessed: 2022-01-04.

[62] Pallets. Welcome to flask's documentation. https://flask.palletsprojects.com/en/2.0.x/, 2021. Accessed: 2022-01-04.

[63] Django Software Foundation. The web framework for perfectionists with deadlines. https://www.djangoproject.com/, 2022. Accessed: 2022-01-04.

[64] Falcon Project. The falcon web framework. https://github.com/falconry/falcon, 2021. Accessed: 2022-01-04.

[65] B.J. Nelson. Remote procedure call. XEROX PARC CSL-81-9, 1981. http://www.bitsavers.org/pdf/xerox/parc/techReports/CSL-81-9_Remote_Procedure_Call.pdf, Accessed: 2022-01-10.

[66] gRPC Authors. A high performance, open source universal rpc framework. https://grpc.io/, 2022. Accessed: 2022-01-04.

[67] CloudAMQP. What is message queuing? https://www.cloudamqp.com/blog/what-is-message-queuing.html, 2019. Accessed: 2022-01-04.

[68] Inc. VMware. Messaging that just works — rabbitmq. https://www.rabbitmq.com/, 2021. Accessed: 2022-01-04.

[69] ZeroMQ authors. Zeromq. https://zeromq.org/, 2021. Accessed: 2022-01-04.

[70] Dejan Skvorc, Matija Horvat, and Sinisa Srbljic. Performance evaluation of websocket protocol for implementation of full-duplex web streams. pages 1003–1008, 05 2014.

[71] Rebecca Bace and Peter Mell. Intrusion detection systems. Special Publication (NIST SP), National Institute of Standards and Technology, Gaithersburg, MD, 2001.

[72] Arkadiusz Warzynski and Grzegorz Kolaczek. Intrusion detection systems vulnerability on adversarial examples. *2018 Innovations in Intelligent Systems and Applications (INISTA)*, pages 1–4, 2018.

[73] Chirag Modi, Dhiren Patel, Bhavesh Borisaniya, Hiren Patel, Avi Patel, and Muttukrishnan Rajarajan. A survey of intrusion detection techniques in cloud. *Journal of Network and Computer Applications*, 36(1):42–57, 2013.

[74] OSSEC PROJECT TEAM. Ossec - world's most widely used host intrusion detection system - hids. https://www.ossec.net/, 2022. Accessed: 2022-01-11.

[75] Cisco. Snort - network intrusion detection & prevention system. https://www.snort.org/, 2021. Accessed: 2022-01-04.

[76] Jean-Philippe Lang. Overview - prelude siem - unity 360. https://www.prelude-siem.org/, 2017. Accessed: 2022-01-04.

[77] Stefan Axelsson. Intrusion detection systems: A survey and taxonomy, 04 2000.

[78] José Eduardo Ferreira Flora. *Container-level Intrusion detection for multi-tenant environments*. PhD thesis, Universidade de Coimbra, 2019.

[79] Dae Ki Kang, Doug Fuller, and Vasant Honavar. Learning classifiers for misuse and anomaly detection using a bag of system calls representation. In *Proceedings from the Sixth Annual IEEE System, Man and Cybernetics Information Assurance Workshop, SMC 2005*, Proceedings from the 6th Annual IEEE System, Man and Cybernetics Information Assurance Workshop, SMC 2005, pages 118–125, 2005. 6th Annual IEEE System, Man and Cybernetics Information Assurance Workshop, SMC 2005 ; Conference date: 15-06-2005 Through 17-06-2005.

[80] Yihua Liao and Rao Vemuri. Use of k-nearest neighbor classifier for intrusion detection. *Computers & Security*, 21:439–448, 10 2002.

[81] Wun-Hwa Chen, Sheng-Hsun Hsu, and Hwang-Pin Shen. Application of svm and ann for intrusion detection. *Comput. Oper. Res.*, 32(10):2617–2634, oct 2005.

[82] The LTTng Project. Lttng: an open source tracing framework for linux. https://lttng.org/, 2018. Accessed: 2022-01-10.

[83] SystemTap. Systemtap. https://sourceware.org/systemtap/, 2021. Accessed: 2022-01-10.

[84] Michael Kerrisk. strace - trace system calls and signals. https://man7.org/linux/man-pages/man1/strace.1.html, 2021. Accessed: 2021-12-28.

[85] Loris Degioanni. Sysdig vs dtrace vs strace: A technical discussion. https://sysdig.com/blog/sysdig-vs-dtrace-vs-strace-a-technical-discussion/, 2014. Accessed: 2021-12-28.

[86] Michael Kerrisk. sysdig - the definitive system and process troubleshooting tool. https://man7.org/linux/man-pages/man8/sysdig.8.html, 2021. Accessed: 2021-12-28.

[87] The Linux Foundation. ebpf. https://ebpf.io/, 2021. Accessed: 2022-01-10.

[88] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8, 2018.

[89] Sysdig. Cloud & container security platform & solutions - sysdig. `https://sysdig.com/products/secure/`, 2021. Accessed: 2021-12-28.

[90] Inc. Palo Alto Networks. Solution overview. `https://docs.paloaltonetworks.com/prisma/prisma-cloud/prisma-cloud-reference-architecture-compute/objectives/solution_overview.html`, 2022. Accessed: 2022-01-11.

[91] Google. cadvisor. `https://github.com/google/cadvisor`, 2021. Accessed: 2021-12-29.

[92] Kubernetes. Heapster. `https://github.com/kubernetes-retired/heapster`, 2018. Accessed: 2021-12-29.

[93] Kubernetes. Heapster deprecation timeline. `https://github.com/kubernetes-retired/heapster/blob/master/docs/deprecation.md`, 2018. Accessed: 2021-12-29.

[94] Kubernetes. Kubernetes metrics server. `https://github.com/kubernetes-sigs/metrics-server`, 2021. Accessed: 2021-12-29.

[95] Kubernetes. Kubernetes dashboard. `https://github.com/kubernetes/dashboard`, 2021. Accessed: 2021-12-29.

[96] Kubernetes. kube-state-metrics. `https://github.com/kubernetes/kube-state-metrics`, 2021. Accessed: 2021-12-29.

[97] Kubernetes. Configure liveness, readiness and startup probes. `https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/`, 2021. Accessed: 2021-12-29.

[98] Prometheus. Overview - prometheus. `https://prometheus.io/docs/introduction/overview/`, 2021. Accessed: 2021-12-29.

[99] Gantikow et al. Rule-based security monitoring of containerized workloads, 2019.

[100] Chin-Wei Tien, Tse-Yung Huang, Chia-Wei Tien, Ting-Chun Huang, and Sy-Yen Kuo. KubAnomaly: Anomaly detection for the docker orchestration platform with neural network approaches, 2019. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/eng2.12080.

[101] Iman Kohyarnejadfard, Daniel Aloise, Michel R. Dagenais, and Mahsa Shakeri. A framework for detecting system performance anomalies using tracing data analysis. *Entropy*, 23(8), 2021.

[102] Aqua Security. Tracee: Runtime security and forensics using ebpf. https://aquasecurity.github.io/tracee/v0.6.5/, 2022. Accessed: 2022-01-11.

[103] Kubernetes. Access clusters using the kubernetes api. https://kubernetes.io/docs/tasks/administer-cluster/access-cluster-api/, 2022. Accessed: 2022-06-06.

[104] Pallets. Welcome to click — click documentation (8.1.x). https://click.palletsprojects.com/, 2014. Accessed: 2022-06-06.

[105] Apache Software Foundation. Apache echarts. https://echarts.apache.org/en/index.html, 2022. Accessed: 2022-01-04.

[106] The Blosc Developers. What is blosc? https://www.blosc.org/pages/blosc-in-depth/, 2014. Accessed: 2022-06-09.

[107] Itamar Haber. Why redis beats memcached for caching. https://www.infoworld.com/article/3063161/why-redis-beats-memcached-for-caching.html, 2017. Accessed: 2022-06-09.

[108] Assad Mahmood. Kafka vs redis pub-sub, differences which you should know. https://blog.containerize.com/2021/04/09/kafka-vs-redis-pub-sub-differences-which-you-should-know/, 2021. Accessed: 2022-06-09.

[109] Locust. Locust. https://locust.io/, 2020. Accessed: 2022-01-08.

[110] softwaretestinghelp. Complete functional testing guide with its types and example. https://www.softwaretestinghelp.com/guide-to-functional-testing/, 2022. Accessed: 2022-06-20.

[111] softwaretestinghelp. The differences between unit testing, integration testing and functional testing. https://www.softwaretestinghelp.com/the-difference-between-unit-integration-and-functional-testing/, 2022. Accessed: 2022-06-20.

[112] flannel io. flannel. https://github.com/flannel-io/flannel, 2022. Accessed: 2022-06-06.

[113] Beck et al. Manifesto for agile software development. http://agilemanifesto.org/, 2001. Accessed: 2022-01-05.

[114] Scrum.org. What is scrum? https://www.scrum.org/resources/what-is-scrum, 2022. Accessed: 2022-01-05.

Appendices

Appendix A

User Stories and Mockups

This appendix contains a template and a list of user stories created to extract the requirements for this project. It also contains the complete mockups designed for the Web Dashboard component of the tool. The template to be used for each user story and corresponding acceptance criteria is the following:

"**As a** [type of user], **I want to** [perform something] **so that** [I can achieve some goal]."

"**Given** [pre-conditions], **when** [key action], **then** [draw conclusions]."

For the user story description, we start by identifying the individual that interacts with the system. Then, we describe the action that represents the behavior of the system and the goal that the individual wants to achieve with the action performed. Finally, we define acceptance criteria detailing the conditions that the software product must meet to be accepted by the type of user. Immediately below, we describe the user stories according to the template. Note that the word "monitoring" and "detection and training phase" are used interchangeably. Also, **US** stands for user story and the series of related and interdependent stories makes up an epic represented as **ES**. Epics are useful to group related User Stories and keep the ideas organized. A number every epic and user story follows to keep track of the listing.

ES-1: As a user, I want to provide configurations so that I can run the tool and monitor my Kubernetes cluster.

- **US-1:** Configure the detection phase
  As a user, **I want to** configure the detection phase **so that** I can keep my application secure by detecting intrusions in real-time.
  Acceptance Criteria: Given the user has his application running and the tool installed, when he runs a command (via CLI or dashboard) to setup the detection phase and specifies a configuration file containing information about the application deployment, the desired services to be monitored, the algorithm to be used, the location to retrieve and

store logs and the duration of the detection phase, then the tool should configure the probes on the desired services and start monitoring the application for possible intrusions.

US-2: Configure the training phase
As a user, **I want to** configure the training phase **so that** I can train an algorithm to recognize intrusions on my application.
Acceptance Criteria: Given the user has his application running and the tool installed, when he runs a command (via CLI or dashboard) to setup the training phase and specifies a configuration file containing information about the application deployment, the desired services to be trained, the algorithm to be used, the location to retrieve and store logs and the duration of the training phase, then the tool should configure the probes on the desired services and start collecting data to train the specified algorithm.

US-3: Configure monitoring for Edge Nodes
As a user, **I want to** configure either the training or the detection phase on Edge Nodes **so that** I can incorporate the monitoring capabilities of the tool in a microservices application that leverages Edge Computing.
Acceptance Criteria: Given the user has his application running and the tool installed, when he runs a command (via CLI or dashboard) to setup the training or the detection phase and specifies a configuration file, then the tool should configure the probes on the desired services that are running on the edge nodes.

ES-2: As a user, I want to perform basic and auxiliary operations that support the main purpose of the tool with the help of a CLI so that I can use the tool properly and take advantage of the tool's functionalities for my microservices application.

US-4: Stop detection/training on all services
As a user, **I want to** stop collecting data and detecting intrusions on all services **so that** I can stop monitoring my application.
Acceptance Criteria: Given the user is running the training phase or detection phase on his application, when he runs a stop command, then the tool should stop the detection/training phase on all services that are being monitored.

US-5: List alarms
As a user, **I want to** see a list of alarms detected by the algorithms **so that** I know the application is behaving correctly.
Acceptance Criteria: Given the user is monitoring his application with the tool, when he runs a command to list the alarms, then the tool displays a list of alarms detected.

US-6: List resources of the cluster
As a user, **I want to** view a list of the cluster's resources, **so that** I can retrieve information without accessing the master node.
Acceptance Criteria: Given the user is running his application and the tool's daemon is running, when he enters a command specifying the

type of resource (such as available nodes, namespaces, deployments, services and pods), then a list should be displayed to the user.

US-7: List algorithms

As a user, **I want to** know which intrusion detection algorithms are available **so that** I am aware of the alternatives provided by the tool.
Acceptance Criteria: Given the user has the tool installed, when he runs a command to list the algorithms, then a list with all the intrusion detection algorithms supported by the tool should be displayed.

US-8: List functionalities of the tool

As a user, **I want to** know the functionalities that the tool provides, **so that** I can choose the functionality that I want to use.
Acceptance Criteria: Given the user has installed the tool, when he enters a specific help command, then a list of the tool's functionalities and corresponding description is shown to the user.

US-9: Access the dashboard

As a user, **I want to** access a dashboard in a web page **so that** I can get an overview of the metrics collected.
Acceptance Criteria: Given the user has the tool installed and his application is running, when he enters a command where he specifies a port, then he should be able to access localhost on a browser and a web dashboard is presented at the address `localhost:port`.

US-10: Stop detection/training on a specific service

As a user, **I want to** stop collecting data and detecting intrusions on a specific service that is being monitored **so that** I can stop monitoring part of my application.
Acceptance Criteria: Given the user is running the training phase or detection phase on his application, when he runs a stop command, then the tool should stop the detection/training phase on a specific service that is being monitored.

US-11: View object detailed info

As a user, **I want to** view detailed information about a specific object, **so that** I can get an overview of the object without needing to access the cluster.
Acceptance Criteria: Given the user is running his application and the tool's daemon is running, when he enters a specific command by specifying a given Kubernetes object, then information related to the object is presented.

US-12: Erase logs

As a user, **I want to** erase logs, **so that** I can free up space in my disk.
Acceptance Criteria: Given the user has logs about the data collected on his machine, when he enters a specific command, then the logs collected are deleted.

US-13: Restart pod automatically

As a user, **I want to** restart automatically the pod that is being attacked **so that** I can mitigate and prevent further damage to the system.
Acceptance Criteria: Given the user has explicitly defined a configuration to automatically restart pods that are under attack, when the

tool is monitoring his application and an intrusion is detected, then the affected pod is restarted automatically and a warning message is displayed to the user.

US-14 Execute a command in a running container
As a user, **I want to** be able to execute a command in a running container **so that** I can interact with the container and eventually perform debugging.
Acceptance Criteria: Given the user is running his application, when he enters a command that specifies a container, then he should be able to execute a command in the container through the CLI (similar to kubectl exec command).

ES-3: As a user, I want to have a dashboard so that I can interact with a graphic interface instead of using a CLI.

US-15: Upload a configuration file
As a user, **I want to** upload a configuration file to the dashboard **so that** I can start monitoring my application via the dashboard
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he clicks the button to upload a file, then he should be able to choose a JSON file from a specific location and upload it to the dashboard.

US-16: View information about the cluster
As a user, **I want to** view information about the cluster on the dashboard **so that** I can have an overview of the cluster.
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he accesses an "overview" tab, then he should be able to view general information about the system.

US-17: View monitored services
As a user, **I want to** view the services being monitored on the dashboard **so that** I can keep track of the application behavior.
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he accesses the web dashboard and accesses the "monitoring" tab, then he should be able to view the services being monitored and time-related information (such as current time, start and end time of the monitoring, duration of the monitoring).

US-18: View alarms
As a user, **I want to** view, on the dashboard, the alarms generated by detecting intrusions on the dashboard **so that** I can keep track of the application behavior.
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he accesses the "alarms" tab, then he should be able to view the alarms generated and the affected services and pods.

US-19: View algorithms
As a user, **I want to** know, via the dashboard, which intrusion detection algorithms are available **so that** I am aware of the alternatives provided

by the tool.
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he clicks a help button to list the algorithms, then a list with all the intrusion detection algorithms supported by the tool should be displayed.

US-20: View graphical representation of the number of alarms over time
As a user, **I want to** view the number of alarms over time as a graphical representation **so that** I can keep track of the application behavior.
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he accesses the "alarms" tab, then he should be able to view the number of alarms over time represented as a chart.

US-21: Export information
As a user, **I want to** export information (such as logs generated by alarms) **so that** I can store the information for future analysis and ensure accountability.
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he chooses the option to export information, then the information collected should be saved in the desired directory.

US-22: Stop detection/training
As a user, **I want to** stop collecting data and detecting intrusions the tool via dashboard **so that** I can stop monitoring my application.
Acceptance Criteria: Given the user is running the training phase or detection phase on his application, when he selects an option to stop the tool on the web dashboard, then the tool should stop the detection/training phase.

US-23: Restart pod
As a user, **I want to** be able to restart a pod that is being attacked **so that** I can mitigate and prevent further damage to the system.
Acceptance Criteria: Given the user is running the tool with the dashboard mode active, when he navigates to the restart pod section and selects a pod, then the selected pod should be restarted and visual confirmation should be displayed to the user.

The complete mockups designed for the Web Dashboard component of the tool are shown here.

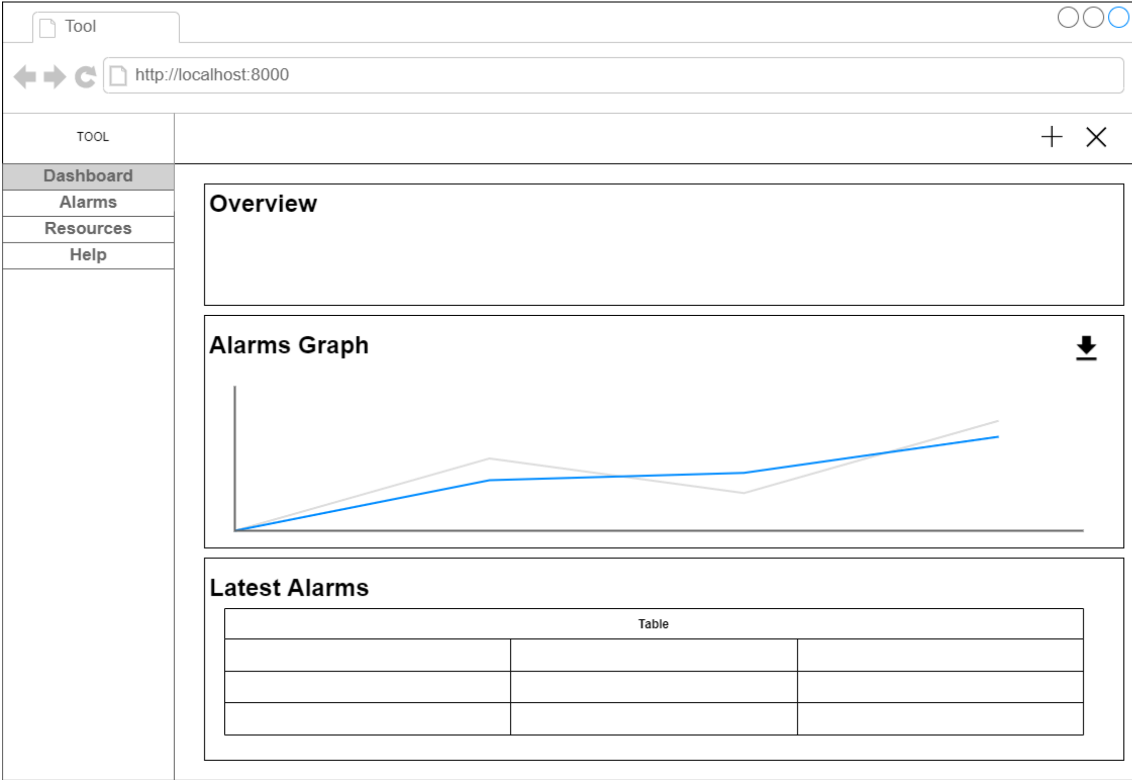


Figure A.1: Mockup of the dashboard page.

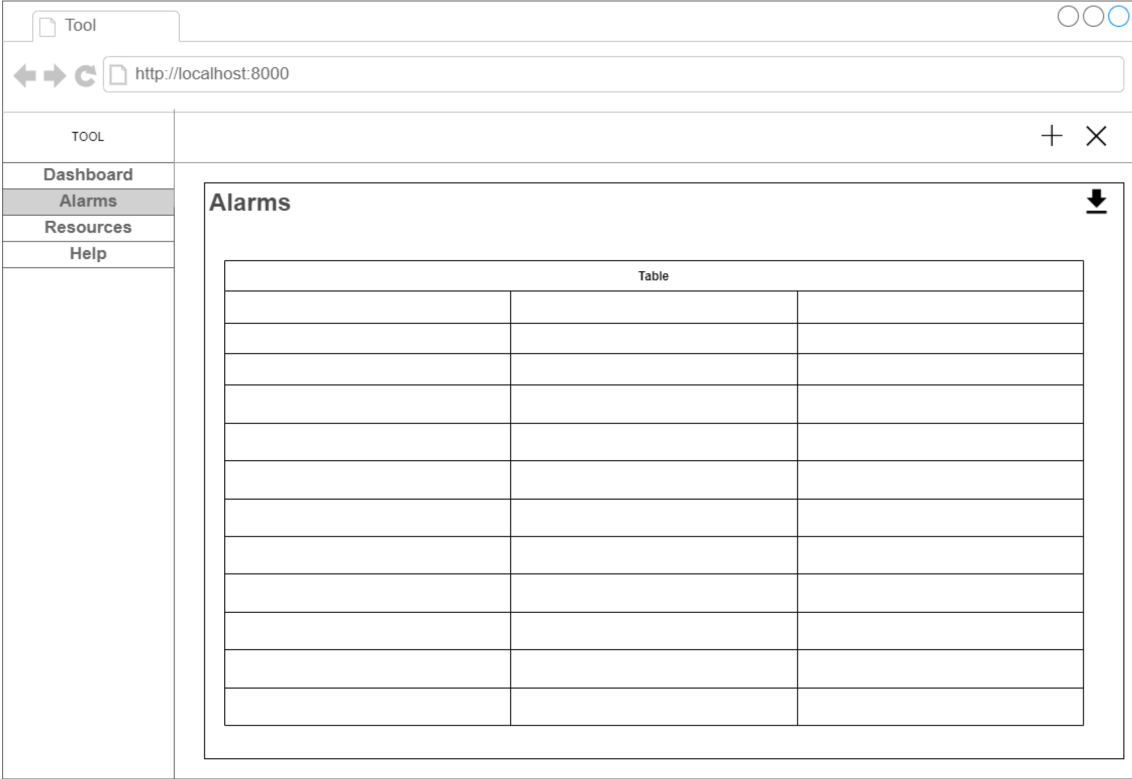


Figure A.2: Mockup of the alarms page.

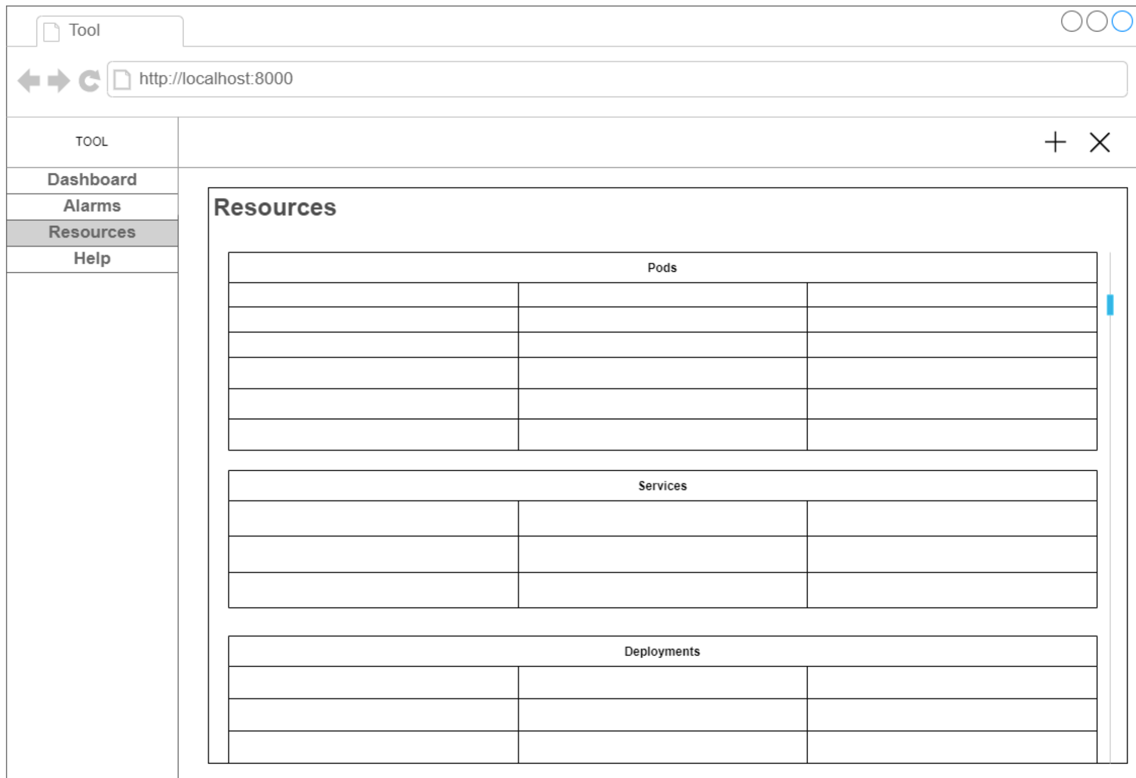


Figure A.3: Mockup of the resources page.

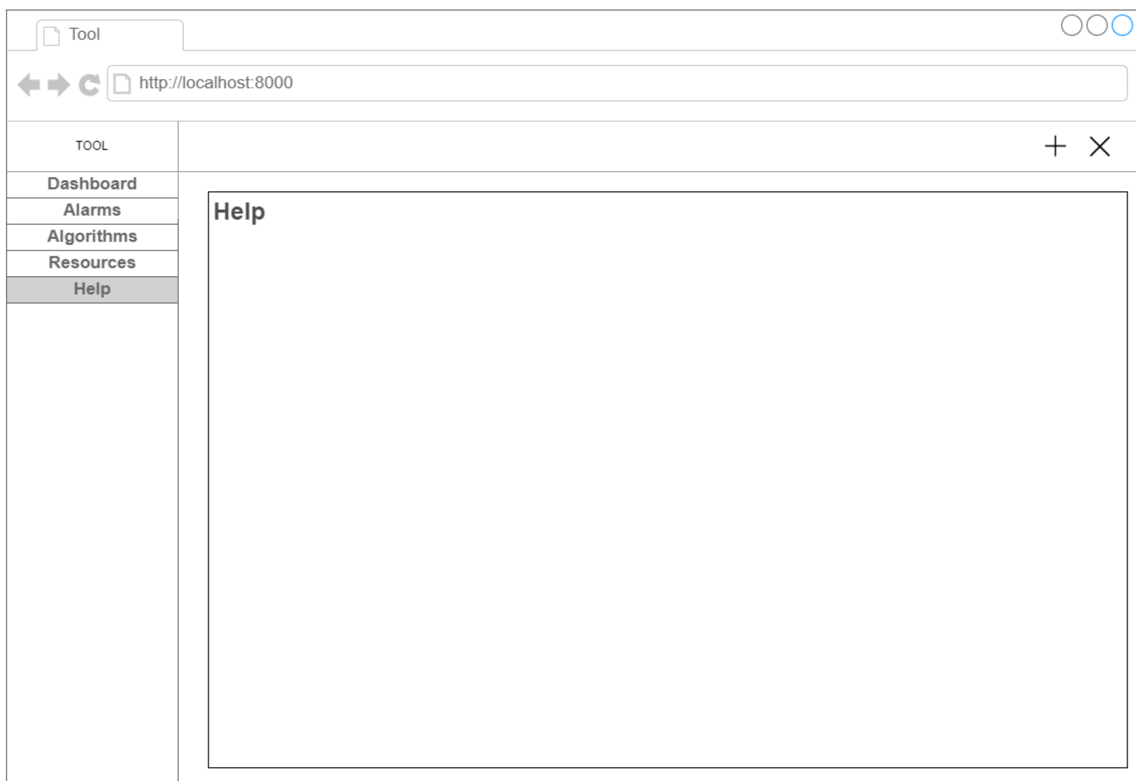


Figure A.4: Mockup of the help page.