Politécnico de Viseu

Escola Superior de Tecnologia e Gestão de Viseu

Security and Privacy Framework for a Cloud Native Platform

Yuka Mouro Takagi

Trabalho de Projeto

Mestrado em Engenharia Informática - Sistemas de Informação

Trabalho efetuado sob a orientação de
Professor Doutor Filipe Manuel Simões Caldeira
Professor Doutor João Pedro Menoita Henriques

February 2025

Politécnico
de Viseu

Escola Superior
de Tecnologia
e Gestão de Viseu

Security and Privacy Framework for a Cloud Native Platform

Yuka Mouro Takagi

Trabalho de Projeto

Mestrado em Engenharia Informática - Sistemas de Informação

Trabalho efetuado sob a orientação de

Professor Doutor Filipe Manuel Simões Caldeira
Professor Doutor João Pedro Menoita Henriques

February 2025

"Security is a process, not a product."
Bruce Schneier

Acknowledgements

I would like to express my deep gratitude to the people who, in different ways, contributed to the completion of this work.

Firstly, my mother and grandfather for providing me with not only financial support but also the emotional support necessary to overcome the challenges of this journey. I thank them for always being by my side and believing in me unconditionally.

To my brother, for the inspiration to move on and pursue a master's degree, and to my boyfriend, for the daily motivation and for never doubting my ability. Your constant presence and words of encouragement were fundamental for me to continue believing in my dreams.

To my supervisors, Professor Doctor Filipe Manuel Simões Caldeira and Professor Doctor João Pedro Menoita Henriques, who were always available and ready to help. Your academic support and guidance were indispensable for the completion of this project.

Abstract

There has been a growing adoption of cloud-native platforms as an essential trend in developing and operating modern applications. These applications stand out for their benefits, such as agility and flexibility in the development and implementation of services, support for dynamic scalability, and reduction in operational costs, as well as the possibility of integration with distributed and high-availability environments. However, the distributed and automated nature of the platforms has increased the attack surface and the complexity of protecting sensitive data in an environment where multiple services and users are integrated.

Risks include the possibility of data breaches through incorrect or vulnerable configurations in containers and microservices. To avoid privilege escalation, the need arose to manage identity and access efficiently. Guaranteed confidentiality, integrity and availability of data in compliance with global data protection regulations, such as the General Data Protection Regulation, mitigating legal risks and protecting user privacy.

This scenario raises the need to improve their security and privacy mechanisms to mitigate vulnerabilities and protect cloud-native environments. In that aim, the proposal of this project aims to carry out a state-of-the-art on privacy and security in cloud-native applications, proposing and evaluating a security framework.

To analyse the results, the framework was validated through specific test scenarios that demonstrated the effectiveness of the applied security methods in mitigating vulnerabilities and enforcing robust security policies.

Keywords: Security; Privacy; Cloud-native applications.

Resumo

Tem havido uma crescente adoção de plataformas nativas na nuvem como uma tendência essencial no desenvolvimento e operação de aplicações modernas. Estas aplicações destacam-se pelos seus benefícios, como a agilidade e flexibilidade no desenvolvimento e implementação de serviços, o suporte à escalabilidade dinâmica e a redução de custos operacionais, bem como a possibilidade de integração com ambientes distribuídos e de alta disponibilidade. No entanto, a natureza distribuída e automatizada das plataformas aumentou a superfície de ataque e a complexidade de proteger dados sensíveis num ambiente onde estão integrados vários serviços e utilizadores.

Os riscos incluem a possibilidade de violações de dados através de configurações incorretas ou vulneráveis em contentores e microsserviços. Para evitar o escalonamento de privilégios, surgiu a necessidade de gerir a identidade e o acesso de forma eficiente. Garantia de confidencialidade, integridade e disponibilidade dos dados em conformidade com as normas globais de proteção de dados, como o Regulamento Geral de Proteção de Dados, mitigando os riscos legais e protegendo a privacidade do utilizador.

Este cenário levanta a necessidade de melhorar os seus mecanismos de segurança e privacidade para mitigar vulnerabilidades e proteger os ambientes nativos na nuvem. Com este objetivo, a proposta deste projeto visa realizar um estado de arte sobre a privacidade e segurança em aplicações nativas da nuvem, propondo e avaliando um framework de segurança.

Para analisar os resultados, a estrutura foi validada através de cenários de teste específicos que demonstraram a eficácia dos métodos de segurança aplicados na mitigação de vulnerabilidades e na aplicação de políticas de segurança robustas.

Palavras-Chave: Segurança; Privacidade; Aplicações nativas na nuvem.

Contents

List of Tables

List of Figures

List of Acronyms

ABAC *Attribute-Based Access Control*

AI *Artificial Intelligence*

AKS *Azure Kubernetes Service*

API *Application Programming Interface*

AR *Augmented Reality*

AWS *Amazon Web Services*

CAMS *Culture, Automation, Measurement, Sharing*

CC *Cloud Computing*

CI *Continous Integration*

CI/CD *Continous Integration/Continous Delivery*

CIS *Center for Internet Security*

CLI *Command-Line Interface*

Cloud KMS *Google Cloud Key Management Service*

CNCF *Cloud Native Computing Foundation*

CPU *Central Processing Unit*

CUI *Controlled Unclassified Information*

CVE *Common Vulnerabilities and Exposures*

CVSS *Common Vulnerability Scoring System*

CWE *Common Weakness Enumeration*

DAC *Discretionary Access Control*

DCT *Docker Content Trust*

DDoS *Distributed Denial of Service*

DevOps *Development and Operations*

DevSecOps *Development, Security, and Operations*

DoS *Denial-of-service*

DTMA *Dynamic Traffic Management Assistant*

eBPF *Extended Berkeley Packet Filter*

EKS *Amazon Elastic Kubernetes Service*

GDPR *General Data Protection Regulation*

GKE *Google Kubernetes Engine*

HPC *High performance computing*

HTML *HyperText Markup Language*

IaaS *Infrastructure as a Service*

IAM *Identity and Access Management*

IBM *International Business Machines*

IoT *Internet of Things*

IP *Internet Protocol*

ISMS *Information Security Management System*

JSON *JavaScript Object Notation*

LXC *Linux Containers*

MAC *Mandatory Access Control*

MITM *Man-in-the-middle*

NAS *Network-Attached Storage*

NIST *National Institute of Standards and Technology*

NVD *National Vulnerability Database*

OS *Operation System*

OSI *Open Systems Interconnection*

OWASP *Open Web Application Security Project*

PaaS *Plataform as a Service*

PbD *Privacy by Design*

PoC *Proof of Concept*

PoLP *Principle of Least Privileged*

PSA *Pod Security Admission*

PSP *Pod Security Policies*

PSS *Pod Security Stantards*

RBAC *Role-Based Access Control*

SaaS *Software as a Service*

SAN *Storage Area Network*

SBOM *Software Bill of Materials*

SDK *Software Development Kit*

SDS *Software-Defined Storage*

seccomp *secure computing mode*

SLR *Systematic Literature Review*

SOA *Service-Oriented Architecture*

SPOF *Single point of failure*

TBAC *Task-Based Access Control*

TLS *Transport Layer Security*

TMAC *Team-Based Access Controll*

TUF *The Update Framework*

URLLC *Ultra-Reliable Low-Latency Communication*

UX *User Experience*

VM *Virtual Machine*

VMM *Virtual Machine Monitor*

VR *Virtual Reality*

YAML *YAML Ain't Markup Language*

ZTA *Zero Trust Architecture*

Chapter 1

Introduction

This initial chapter introduces the project's fundamental concepts in cloud-native applications. It presents the background and motivation regarding the security and privacy challenges in these environments, leading to the definition of the objectives. In addition, it outlines the bibliographic research and details a structured work plan. Finally, the structure of this document is presented.

1.1 Background

The evolution of software architecture and computing infrastructure has undergone several transformations to meet scalability, flexibility, and efficiency. This move to modern applications increased their attack surface, imposing new requirements regarding their security and privacy, requiring new approaches to protect them.

In the past, monolithic applications [Villamizar et al., 2015] were developed in a single codebase shared between multiple developers for a long time. When they needed to add or change to a given application part, they needed to ensure the other parts were still working. Such an approach increased their complexity and constraints when extending those applications with new features. Moreover, as soon they were deployed to production, all their services increased the chances of problem *User Experience* (UX). In addition, if the application fails, all its services will also fail, representing a *Single point of failure* (SPOF).

Service-Oriented Architecture (SOA) emerged as a high-level standard for developing and managing distributed software systems, highly linked to services that can

be seen from a business term as a unit of transaction or value or in a technical way in which these are units of functionality implemented in interfaces. These features include platform or language independence, operating under a of being modular and reusable, tract defined by the expectations of being modular and reusable, and offering high flexibility and agility [Perrey and Lycett, 2003].

Microservices is another approach trying to overcome the limitations of monolithic applications, such as the fragmentation of the application into smaller and independent services [Alshuqayran et al., 2016]. Each service is responsible for only one specific function, communicating with each other through *Application Programming Interface* (API)s. Moreover, they can be developed, deployed, and scaled independently. Overcoming the previously imposed challenges and presenting greater modularity, scalability, and flexibility in development. However, such benefits raise some challenges, including the need for efficient management of communication, authentication, and security between distributed services. Microservices are small, single-response applications that can be deployed, scaled, and tested independently. There is a decomposition of the monolith since the system is broken down into a more granular form, introducing the concept of SOA [Larrucea et al., 2018].

In 1960, the concept of *Virtual Machine* (VM) was created to provide simultaneous interactive access to a mainframe computer by *International Business Machines* (IBM) corporation. Each VM represents a replica of the underlying physical machine, and users have the illusion that they are running directly on the physical machine. It brings benefits such as isolation, resource sharing, and the ability to run multiple flavors and configurations of *Operation System* (OS)s different sets of software technology and configuration [Ali and Meghanathan, 2011]. Virtualization is also usually described as the technology that introduces a software abstraction layer between the hardware and the applications that run on top of it. This abstraction layer is called *Virtual Machine Monitor* (VMM) or hypervisor [Daniels, 2009]. It can hide physical resources from OS and at the same time control them directly, opening up the possibility of running multiple and possibly different OSs since these are not regulated by the OS. This process results in partitioning the hardware platform into one or more independent units, known as VMs [Sahoo et al., 2010]. VM is an isolated entity that simulates a physical computer that can run various operating systems and applications composed of virtualized resources such as *Central Processing Unit* (CPU), memory, storage, and input/output devices. Virtualization brings several benefits by reducing the need for dedicated hardware and significantly saving acquisition and maintenance costs. It provides efficiency by enabling dynamic allocation of resources, thus optimizing available capacity and facilitating rapid modularization and reconfiguration of systems, reducing downtime. It should also be noted that it provides flexible scalability, allowing industrial systems to adapt to changes in production demand without significant investments in infrastructure

[Malhotra et al., 2014]. Also, network infrastructure can be virtualized, including physical network components such as switches, routers, and firewalls. Storage is done in virtualized for ms such as *Network-Attached Storage* (NAS), *Storage Area Network* (SAN), and *Software-Defined Storage* (SDS), the first two dependent on dedicated hardware for centralized storage; the first is connected to the network providing file-based access, used to share data between multiple devices, and the second offers block storage, simulating local disks for connected servers. At the same time, the last proposes a storage solution that separates the physical hardware, providing flexibility and scalability [Patil and Modi, 2019].

National Institute of Standards and Technology (NIST) defined *Cloud Computing* (CC) as a model that enables ubiquitous, convenient, and on-demand network access to a shared set of configurable computing resources, such as networks, servers, storage, applications, and services that can be rapidly provisioned and released with minimal management effort or service provider interaction [Mell et al., 2011]. CC is made up of five essential features: On-demand Self-Service, Broad Network Access, Resource Pooling, Rapid Elasticity, and Measured Service, three service models: *Software as a Service* (SaaS), *Plataform as a Service* (PaaS) and also *Infrastructure as a Service* (IaaS) and four implementation models [Radack, 2012]. Different cloud implementation models exist, such as Public, Private, Community, and Hybrid. The public is considered to have the lowest security, and service providers manage it for the general public and industry. Private is for a single organization, and community is for organizations that share the same security policies, a set of rules that define executions considered unacceptable for some reason that can be treated as access control, information flow, and availability [Schneider, 2000] or share similar interests, both considered safer. Furthermore, finally, combining Public and Private, we have Hybrid, which is classified as intermediate security [Rashid and Chaturvedi, 2019]. Cloud service providers such as *Amazon Web Services* (AWS), Microsoft Azure, and Google Cloud Platform, all public examples, have security experts in their ranks who are prepared to deal with threats, guaranteeing metrics on how their customers access the services, such as processing, latency, access time, and data transfer rate. However, the responsibility does not lie entirely with them, as it depends on the users. [Islam and Hasan, 2017]

Virtual containerization has transformed software development by isolating applications and their dependencies in portable and lightweight environments. Unlike traditional virtualization, which uses VMs, containers share the OS kernel, thus resulting in faster startups and lower resource usage. Docker is the leading technology in this area, allowing the creation, management, and execution of containers to be efficient and more scalable [Sobieraj and Kotyński, 2024]. Containers also allow isolated runtime environments for application execution, meaning they can encapsulate the application with its dependencies, such as source code, files, and environment

variables. They only isolate system processes and resources but share de OS kernel, making them lightweight, easier to deploy and migrate, and scalable. An application encapsulated by Docker is distributed as a Docker Image, an executable package that includes everything it needs to run the application[Lin et al., 2020].

Unlike traditional, cloud-native is an application development and architecture approach designed to make the most of a cloud computing environment's unique resources and characteristics [Linthicum, 2017]. This includes automatic scaling, fitting the workload requirements, ensuring optimized resources and elasticity, designing to adjust to sudden changes, and adding or removing resources as needed. Efficiency that promotes a reduction in operating costs. Having an architecture in which applications are developed in a distributed manner promotes resilience and flexibility.

Container orchestration emerged as an approach to deal with the complexity of location, coordination, and selection of resources to meet the desired requirements [De Sousa et al., 2019]. Container orchestration automatically manages container deployment, scaling, networking, monitoring, and lifecycle. Tools for the container include Docker Swarm, Kubernetes, and Mesos [Al Jawarneh et al., 2019].

The concept of observability [Kosińska et al., 2023] came to help in the way to deal with the complexity of applications in cloud-native environments. Thus, a system is observable if its current state can be determined within a finite time using only outputs. Bringing benefits such as identifying deviation from usual behavior, performing root cause analysis, forecasting failures, altering events, optimizing performance, and proactively improving UX.

Regarding security, observability helps to have a more comprehensive view of distributed systems. It allows developers to determine the root cause of problems and, at the same time, improve system performance. Observability is composed of three elements: the metric, a quantitative number measured over time, including computing resource usage and network traffic; the log, a record of an event that occurred at a specific time containing related text or data in the form of a structure; and finally, the trace, used to represent end-to-end request flow [Cha and Kim, 2021].

1.2 Motivation

Virtualization introduced management complexity due to the increased number of abstraction layers. This scenario can become challenging when identifying and solving problems. Although operational costs will be reduced in the long term, this requires a considerable initial investment as well as specialized hardware and software and the training of teams to deal with these difficulties [Fong and Steinder, 2008].

Collecting data from *Internet of Things* (IoT), *Augmented Reality* (AR) e *Virtual Reality* (VR) brings data protection issues and concerns to comply with regulations,

such as *General Data Protection Regulation* (GDPR) [Voigt and Von dem Bussche, 2017] It is paramount to avoid severe consequences for individuals and organizations, including physical harm and financial losses. IoT refers to scenarios in which the network connection and computing capacity are extended to objects, sensors, and objects that are not usually considered computers, allowing these devices to collect and exchange information with as little human intervention as possible [Rose et al., 2015]. Nowadays, countless devices on the market are aimed at concepts such as smart Homes, smart wearables, and smart televisions, which are integrated into everyday life. Despite their function of facilitating and assisting daily, they cause several concerns regarding the massive amount of data they collect. Such data can be confidential or collected without the user's consent, private information such as their identity, location, searches, or even behavior patterns [Chaudhuri, 2016].

Cloud-native applications are increasingly being adopted, reducing services' development and deployment time while increasing scalability. Consequently, this has increased the surface of attacks regarding security-related aspects due to implementing these new concepts. As a result, there is a need to implement a Cloud Native platform that can respond to challenges where security and privacy are crucial. Due to the dynamic and constantly evolving nature of the cloud-native environment, the combination of virtualization, cloud search, and other technologies makes the scenario extremely prone to threats [Ibrahim et al., 2016].

Adopting cloud-native platforms is a key strategy in accelerating the development and deployment of virtualized services while ensuring scalability. However, its adoption increases the attack surface while introducing security and privacy challenges. Addressing these challenges is critical to protecting sensitive data and maintaining the operational integrity of systems, minimizing risks such as unauthorized access and financial repercussions.

Although the security and privacy of cloud-native platforms are improving, the problem must be solved. Pairing with rapid technological evolution means a constant need to develop more robust and effective approaches [Rahaman et al., 2023].

1.3 Objectives

The expected results include the survey of the state-of-the-art, a framework specification meeting the security and privacy challenges, and the experimental work with a *Proof of Concept* (PoC) demonstrating its effectiveness.

The main objective of this project is to design and validate a security and privacy framework that mitigates vulnerabilities and enhances the protection of cloud-native platforms against potential threats. The objectives listed below are presented in the order they are addressed throughout the document rather than by priority.

- (O1) - Survey of the state of the art regarding security and privacy for Cloud Native applications.

  The first objective focuses on reviewing the state of the art, where a survey is carried out on security and privacy on cloud-native platforms, identifying the best practices, gaps, and current challenges, exemplifying successful cases in the application of security and privacy.

- (O2) - Specify a security and privacy framework for Cloud Native platforms.

  The second objective is to establish guidelines and recommended practices to ensure security and privacy in cloud-native platforms, providing a structured approach for specifying the framework.

- (O3) - Demonstrating the security and privacy framework into a Cloud Native platform.

  Finally, the third objective is focused on data and operations.

1.4 Bibliographical research

In addition to the sources suggested by the supervisors in the project topic proposal, the research was carried out by carefully analyzing other sources to gain a broader perception of the topic and in-depth research into what exists today.

The primary source of information for the research was Google Scholar. This widely used tool allows access to resources with many citations.

Semantic Scholar and IEEE Xplore were also used in this literature search, which has been running since the beginning of October and will be an ad hoc activity until the conclusion of the work, as they contain similar functions to Google Scholar [Hannousse, 2021].

Departing from the title of this work,'Framework for Security and Privacy for a Cloud Native Platform,' by breaking it down into parts, it is possible to remove the keywords 'Security," Privacy,' and 'Cloud Native.' As the Google Scholar tool is universal, it was concluded that searching for these keywords in Portuguese and English would deliver more results and relevant information for developing this document.

The current study is based on the *Systematic Literature Review* (SLR) useful for extracting empirical evidence from the available literature and the analysis to support the research [Dikhanbayeva et al., 2020].

The evaluation method used to gather relevant information is based on a hybrid concept between the quality and quantity of the document. In terms of quality, the timeliness of the article, the clarity and good organization of the article's structure, and whether the conclusions reached were valid with the objectives presented and the relevance of its content to the research for this project.

The number of citations and the scope of the key keywords were considered, as Google Scholar prioritizes search results based on these factors. However, many citations do not always reflect a source's quality or strategic relevance for this research.

1.5 Work Plan

For the first step, objectives and planning were defined, and an initial discussion with supervisors was held to align the achievement goals and work expectations. Time frames were established, detailed planning was provided, and steps and deadlines were defined for each task identified in the work plan. Starting with research for reviewing the state-of-the-art, where a bibliographical survey is carried out using not only the references provided by a guiding document given in the work proposal as a starting point but also complemented with recent publications on academic platforms, such as Google Scholar.

After identifying requirements, the goal is to propose a security and privacy framework supported by the challenges found after reviewing the state-of-the-art. Then, the framework's demonstration will be supported by the use of a PoC.

In developing evaluation scenarios, practical cases demonstrate the framework's effectiveness under some security and privacy scenarios. Results from each phase will be analyzed and compared, reaching conclusions and proposing ways to make the framework even more robust. During all this, the documentation of the thesis takes place in parallel, structured with the division of clear chapters and, over time, revised and assisted as necessary.

The adopted methodology can be described as a hybrid approach combining elements of structured project management, such as waterfall, a traditional way in which the objectives are initially well defined and followed linearly, meaning that the project only moves on to the next step when the previous one is complete. However, despite the objectives being well-defined, they should be improved. Here comes the part that makes the methodology hybrid since there is an iterative refinement with continuous feedback, allowing adjustments based on findings or challenges encountered. Parallel documentation also emphasizes maintaining comprehensive records during the process, providing consistency and complementary reducing the risk of gaps [Fagarasan et al., 2021].

The phases were divided into four tasks. A fundamental and continuous one will be the writing of the thesis. To simplify the table 1.2, it was created ahead of a table 1.1 that describes what each task represents.

Task ID	Description
T1	Review of the state-of-the-art
T2	Specification of a security and privacy framework
T3	Integration of the security and privacy framework in a cloud-native platform
T5	Writing the thesis

Table 1.1: Project tasks description

As the name suggests, the first task is to identify the state-of-the-art concerning security and privacy for cloud-native applications. To this end, there is a need to carry out a detailed analysis of existing security and privacy technologies and methods, including scientific articles, digital books, and technical reports. Could you explain key topics such as container virtualization, orchestration, and service mesh?

For the second task, now that significant information has been gathered on the topic and which technologies, security, and privacy concerns and how to overcome them, we have a basis to start specifying the security and privacy framework, mentioning which tools were chosen, which security methods were selected so that it was created with the most excellent robustness and scalability. It defined parameters such as functional and non-functional requirements and specified which procedure to integrate into the cloud-native platform's security and privacy framework.

After all the conditions are defined, it is possible to move on to the third task of the project, that is, the development of the framework and how it was integrated into the cloud-native platform. In this phase, methods that help guarantee security and privacy are selected, presenting practical cases and test scenarios that prove their validation. At the end of this task, there is an analysis of the results achieved and a comparison between the results, such as what already exists in the current market and the proposal for improvements.

Finally, the last task represents the writing of the thesis and how this is always present throughout the other tasks, as it occurs in parallel and is defined at the beginning of the project as it will be structured. Changes may occur after revisions with the advisors throughout time. This process reports the entire development to produce the final document.

As Table 1.2 shows, the work schedule is organized with columns for each month and rows for each project task. The "X" in each cell represents the planned completion of a task during that specific month.

Task/Date	2023			2024												2025	
	Oct	Nov	Dec	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Jan	Feb
T1	X	X	X	X	X												
T2						X	X	X	X	X	X						
T3												X	X	X	X	X	X
T4	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Table 1.2: Work Plan

Firstly, tasks 1 and 2 occurred from October 2023 to March 2024 and February to August 2024, respectively. Subsequently, task 3 started in September 2024. and completed in February 2025. Having occurred in parallel, taking advantage of the results of the other functions.

1.6 Structure of the document

The document presents an overview of security and privacy challenges in cloud-native platforms. Chapter 2 explores the state of the art, covering key concepts such as container-based virtualization, orchestration, and observability, emphasizing their security implications. Chapter 3 details the proposed security and privacy framework specification, the requirements for best practices for securing cloud-native environments, and the architecture. Chapter 4 focuses on validating the framework, demonstrating its implementation through security mechanisms following the best practices, and discussing results. Finally, the last chapter synthesizes this work with main conclusions and future work.

1.7 Summary

This chapter provided the background and motivation for this work. It highlighted the challenges posed by emerging technologies and outlined the project objectives, methodology, and research strategy.

The next chapter will review the state-of-the-art, exploring key concepts and technologies that underlie the framework, such as container virtualization, orchestration, and observability. It discusses how these impact security and privacy challenges in cloud-native environments.

Chapter 2

State-of-the-Art

This chapter presents the main topics on improving security and privacy for cloud-native applications. It covers the technologies and related work related to the increasing adoption of virtualized services. It also explores the security and privacy challenges in cloud-native environments. Moreover, it analyzes container virtualization.It also discussed how container orchestration facilitates managing and automating complex environments and security challenges. Then, the security and privacy frameworks to be followed by organizations are presented. Finally, it ends with an analysis of the use of service mesh.

2.1 Virtualization in containers

In the late 1970s, container-based virtualization had its roots in creating the chroot system call. It may have been the first step towards process isolation, changing the root directory of a process and its children to a new location on the file system. From the beginning till this day, container technology has become essential in modern computer environments, evolving from an abstraction of multiple Linux kernel features to sophisticated frameworks powered by tools specifically developed to support the execution, management, and orchestration of containers [Queiroz et al., 2023]. Container-based virtualization is described as a technique where the kernel of the OS host is modified to support multiple OS guests encapsulated in containers. Each container operates in isolation, with dedicated access to

resources such as IP addresses, memory, and root permissions, but shares the same host OS kernel [Babu et al., 2014].

In comparison VM, containers allow the development and development of light and fast applications, whereas complete isolation is not essential. Such an approach maximizes hardware efficiency whereas VMs are more appropriate for complete isolation or the ability to run different OSs are required. While the kernel is shared with the host system in containers, each VM has its kernel, and isolation is complete through hypervisors, while in containers, it is at the level of processes and namespaces. Containers end up being faster in initialization and lighter, having greater resource efficiency since they only package the necessary libraries and dependencies. At the same time, in VMs, they include the entire OS. Thus, the use of containers ends up being better for the execution of distributed applications and microservices as VMs are a good alternative for legacy systems or more complex infrastructures [Seo et al., 2014]. One of the benefits of containers is that they require significantly less disk storage and memory than VM. This results in a smaller energy consumption footprint. Startup is almost instantaneous, given its containerized software providing portability [Potdar et al., 2020]. The absence of a guest OS significantly reduces overhead. They are ideal for cloud environments and microservices-based architectures, providing agility in application development and execution [da Silva et al., 2018].

Several container-based technologies exist, such as *Linux Containers* (LXC). It proposes isolation through the kernel Linux namespaces and groups with features such as container migration, route/bridge network, and fair resource scheduling. Singularity is also a great container-based option focusing on portability to meet the demands of *High performance computing* (HPC) environments, where traditional solutions face security and compatibility issues [Kurtzer et al., 2017]. However, the most popular is Docker, the successor to LXC, since it evolved from it and used it as the underlying mechanism for creating and managing containers [Arango et al., 2017].

Docker is an open-source platform that runs applications, making the process easier to develop and distribute [Boettiger, 2015]. These applications built in the docker have all the supporting dependencies packaged as containers. Although container technology has been around for a while, Docker streamlines building and managing containers; the benefits were speed, portability, scalability, rapid delivery, and density [Rad et al., 2017].

Due to the need for more efficient resource management, a shift has led to rapid growth in cloud computing, particularly container technology, due to its advantages over traditional VMs. As containers have numerous benefits, such as the need for fewer resources and speed, they quickly became the popular choice, with their number one platform being Docker for creating and controlling these containers, which run from images downloaded from different sources, the most popular being Docker

Hub. Due to container technology sharing the same OS kernel, there is a greater need to monitor the security of containers and the images from which they run [Andersson and Hysing Berg, 2022].

Though docker can quickly share application build environments among developers through container technology, it does not provide security guarantees for known docker images. Since docker images are shared without the means of a vulnerability diagnostic, polluted docker images can be distributed so that the docker-based application can easily collapse [Kwon and Lee, 2020].

The NIST defines a vulnerability as a weakness in the computational logic found in software or hardware components that, when exploited, negatively impact confidentiality, integrity, or even availability [Honkaranta et al., 2021]. Therefore, in docker images, bad actors could be allowed to commit cyber attacks when these vulnerabilities are present. Besides *National Vulnerability Database* (NVD), there are also GitHub advisories that store hundreds of known vulnerability details [Boles et al., 2024].

Many non-profit organizations have emerged, such as MITRE and *Open Web Application Security Project* (OWASP), with the concern of tracking vulnerabilities and publishing defense recommendations in standardized formats since producing data in formats manually is very time-consuming. MITRE introduced *Common Vulnerabilities and Exposures* (CVE), the most recognized vulnerability standard, enabling software vendors and consumers to map specific vulnerabilities to their patches [Haddad et al., 2023].

Containerization has changed the way applications are deployed and managed. One of the crucial components of image security is identifying vulnerabilities. Since these images are usually drawn from open or closed repositories, they can uncover application security flaws, which, if left undetected, can lead to bad outcomes such as data theft, system failure, or loss of service. The *Center for Internet Security* (CIS) came up with a document that describes a secure Kubernetes environment covering guidelines for container vulnerability management [Chintale and Kethireddy, 2024].

The meteoric rise of containerization has also given rise to a wide variety of security problems, as it is said that the images available on Docker Hub may contain critical vulnerabilities due to outdated packets. Thus, these can be impacted in various ways, such as by a high scale of privileges. Although containers have been a great help for developers in creating modern applications for Industry 4.0, they have also led to security measures being introduced, one of which is integrating vulnerability scanning. So before the images are deployed, they can be analyzed using image scanning tools [Jaisinghani, 2022].

The importance of container security is disregarded due to the increasing adoption of technology and its benefits, such as portability and efficiency. There are associated risks due to image vulnerabilities and incorrect configurations, which lead to

insecure environments. To achieve this, good practices are adopted, such as reducing the duration of containers, exposing only necessary ports, and incorporating vulnerability analysis tools such as Trivy, Grype, Clair, and Snyk [Ugale and Potgantwar, 2023].

Trivy is a widely used open-source tool for detecting known vulnerabilities in OS packages, language-specific libraries, container images, and Kubernetes components. Based on trusted vulnerability databases, such as vendor advisories and community-maintained repositories. It can also find misconfigurations and secrets, locate which ones are exposed and even licenses, and check that they comply with GDPR, therefore mitigating legal risks [Bhatia, 2024].

Grype by Anchore is also an open-source tool to scan container images for vulnerabilities. It analyses these, comparing packages, dependencies, and collected data in its primary database and for scanning the NVD, a popular choice due to its effectiveness in identifying threats. However, it has limitations due to false positives or negatives, just like other tools. It can also identify outdated or misconfigured packages, protect container environments against threats such as *Distributed Denial of Service* (DDoS) or malicious code injection, and even improve security before shipping containers to production [Kalaiselvi et al., 2023].

Software Bill of Materials (SBOM) is an inventory of all software components used in an application or system, acting as a list of ingredients for enumerating dependencies, libraries, and tools involved. Image scanning tools generate reports that analyze SBOM components to identify vulnerabilities, checking against databases such as NVD or Github advisors and providing information on severity and impact. Helping to prioritize correcting critical vulnerabilities and identifying outdated or high-risk components [O'Donoghue et al., 2024].

The figure 2.1 represents the CVSS, which provides a numerical score, ranging from 0.0 to 10.0, that reflects the vulnerability potential impact, helping prioritize responses based on severity, making sure the ones categorized as critical or high are updated and the medium or low can be ignored [Xia et al., 2023].

Score Range	**Severity Level**
0.0	None
0.1–3.9	Low
4.0–6.9	Medium
7.0–8.9	High
9.0–10.0	Critical

Table 2.1: CVSS scores are categorized into severity levels

Table 2.2 presents the severity ratings used by the Grype vulnerability scanning tool. This is because depending on which tool is used, the levels may differ, for

example in the case of Trivy it does not have the ´NEGLIGIBLE' parameter. The meaning of each level describes the impact that each vulnerability may present.

Severity	**Description**
Critical	Vulnerabilities that can be exploited remotely with significant impact on confidentiality, integrity, or availability. Immediate attention is required.
High	Vulnerabilities that could lead to serious consequences if exploited, but may require certain conditions to be met.
Medium	Vulnerabilities that are harder to exploit or have a lower impact on the system.
Low	Vulnerabilities that present minimal risk and are unlikely to be exploited in real-world scenarios.
Unknown	Vulnerabilities without a clearly defined severity classification due to lack of sufficient information.
Negligible	Vulnerabilities with minimal impact that do not pose a significant security risk. Often ignored in automated security policies.

Table 2.2: Severity classifications in the Grype vulnerability scanning tool

To provide a more effective effort in terms of remediation the NVD is using *Common Weakness Enumeration* (CWE) as a classification mechanism by hierarchical taxonomy of weaknesses, managed by MITRE, where each CWE represents a single vulnerability type, defining a standardized description of software weakness to improve communication between developers, security professionals and vendors. The taxonomy categories are divided into input validation, data handling, resource management, security features and architectural issues [Ambala, 2024].

Since containerization promotes the grouping of applications from their dependencies into a single image, the Docker framework facilitates the process of sharing images publicly, but this can lead to users creating and sharing images that include private keys or API secrets that either through error or negligence can be leaked, putting authentication and confidential use of privacy-sensitive data at stake an even allow attacks [Dahlmanns et al., 2023].

Sabotage is a significant concern for industries, especially in Industry 4.0, where sensitive and proprietary data is increasingly exposed due to digital connectivity and automation. This can lead to interruption of manufacturing processes, damage to equipment, and loss of confidence in the integrity of industrial systems. In extreme cases, induced failures can result in catastrophic consequences such as industrial accidents or failures of products in use, putting lives at risk [Prinsloo et al., 2019].

To guarantee the integrity and authenticity of docker images, *Docker Content Trust* (DCT) enables image signing. Allowing the verification of the origin and integrity of images before they are run in production. The digital signature of images guarantees that they have not been altered since their creation, protecting against code injection attacks [Xu et al., 2017]. This is to ensure that only images signed by their authors are accessible, whether they are pulled from a public or private registry. DCT is a functionality introduced in docker that allows checking the integrity of container images through the digital signature of their creator. This ensures that images are not manipulated before use. DCT uses digital signatures to authenticate images, thus protecting environments against possible attacks that could compromise the reliability of services running in containers [De Benedictis and Lioy, 2019].

DCT is built on Notary, a tool for publishing and managing trusted content collections by implementing *The Update Framework* (TUF), a framework for securing software update systems that follows a set of guidelines such as separation of responsibilities for each different keys and metadata [Elamin and Kierkels, 2021]. The root role is the trust point that delegates authority to other roles stored offline to reduce the risk of compromise. The targets role signs metadata of trusted files, the snapshot role guarantees integrity and prevents mix-and-match attacks, and the timestamp role ensures the freshness of the data [Kuppusamy et al., 2018].

A robust solution to increase the security of Docker images is presented by DCT. However, there is a need for additional care in highly automated pipelines where multiple entities are involved. The complexity of key management is highlighted as a challenge in using DCT since external entities can manipulate images between the developer and the production environment, breaking the single trust model [Combe et al., 2016]. Although DCT was designed to detect attacks that alter images in the docker registry, it is unable to detect image manipulation if this occurs in *Continous Integration* (CI), thus making it vulnerable [Moriconi et al., 2023].

The term runtime refers to the period in which a program or application runs when the code is being executed after being compiled. In the first phase, the security was focused before this moment, starting with the detection of vulnerabilities in the images of the content and the use of DCT, which verifies the integrity of the author of the images through the verification of digital signatures. Container runtime security then becomes crucial after these checks.

To have better case-control for Industry 4.0 manufacturing systems, there needs to be a focus on implementing policies that align with *Principle of Least Privileged* (PoLP)—aiming to improve cybersecurity by minimizing the risks of internal and external attacks and adapting industrial systems to dynamic and interconnected scenarios. Involving a proposal that controls restrictive access to the necessary privileges at each specific moment in the workflow.

Several security mechanisms are available to improve system security by restricting individual program actions only to allow authorized activities. There are three main models of traditional access control. *Mandatory Access Control* (MAC), where access is determined by system-enforced policies set by administrators, *Discretionary Access Control* (DAC) where the end-user determines policies, and *Role-Based Access Control* (RBAC) is based on user roles within an organization [Ubale Swapnaja et al., 2014].

In addition, others are also implemented industrially, such as *Attribute-Based Access Control* (ABAC), in which access decisions are based on the user, resource, and environmental attributes such as location, time, and job title. There is also the need for hybrid models such as *Task-Based Access Control* (TBAC) and *Team-Based Access Controll* (TMAC), in which access is guaranteed dynamically as tasks progress in workflow and in which access comes from the concept that teams are dynamic groups collaborating on a specific task [Tolone et al., 2005].

It is possible to restrict container operations to reduce the attack surface, which is crucial for maintaining a secure cloud-native environment. Tools like *secure computing mode* (seccomp) and AppArmor are available to enforce these restriction policies. Customizing and maintaining security profiles can be complex due to potentially error-prone manual policy configurations, requiring careful testing to avoid breaking container functionality; therefore, default profiles are available to overcome this problem and reduce complexity.

As the use of container technology is ubiquitous, security must be increased. Docker offers several options to improve this security, one of them being the use of seccomp profile, which the community has not entirely accepted. However, it can mitigate many attacks and some zero-day vulnerabilities [Lopes et al., 2020], that is, security flaws in software or hardware where the vendor has zero days to fix the flaw before it can be exploited. Thus, these profiles consist of several syscalls that either will not or will not be relieved to execute on the system [Zarić,].

seccomp profile is used as a security mechanism that defines system calls for containers. This then allows the blocking or permission of the call system, like chmod, used to change permissions of files and directories in the system, as containers having access to this call can easily make changes that put the system at risk by exposing sensitive data and escalating privileges unnecessarily. This process then reinforces the PoLP, a key element for security and privacy frameworks, as it ensures that they operate with permissions restricted to what is necessary.

To tighten the security of docker containers, some solutions integrate the creation and maintenance of seccomp profiles directly into the pipeline to facilitate the use of this technology and allow constant updates without human intervention. These profiles are challenging to create and maintain due to the complexity and frequency of necessary updates that lead to configuration failures that can result in vulnerabilities

or even service interruptions [Lopes et al., 2020].

There is currently an alternative to the traditional individual blocking of system calls. It proposes an advanced mechanism that analyzes the sequence of calls inspired by pattern detection methods used for malware, resulting in greater precision in detecting attacks and blocking them in time. However, the existence of sequences similar to exploits can generate unnecessary blocks [Song et al., 2023].

2.2 Security and Privacy for Cloud-Native Applications

Microservices are cohesive and independent processes that interact with others through messages. This approach reflects the basic principles of distributed systems, where different components or nodes exchange information through networks. Each microservice is an autonomous model that implements a specific functionality, allowing the construction of more flexible and agile distributed systems [Dragoni et al., 2017]. The isolation of microservices is because, although they offer significant advantages in modularity and scalability, they can become vulnerable if they are not properly isolated.

Cloud-native applications emerged from the evolution of building and operating software in distributed and scalable environments. Through adopting technologies such as containers and microservices, cloud-native platforms allow *Continous Integration/Continous Delivery* (CI/CD) and dynamic management of applications [Chen, 2018]. However, this flexibility and agility introduce new challenges in security and privacy, as these increase the attack surface and the complexity of the environment. Due to its distributed nature and the intensive use of automation, security must be incorporated into all infrastructure layers [Nascimento et al., 2024].

Cloud-native services, designed to run seamlessly in distributed, scalable, and dynamic environments, face unique security challenges due to their architecture and operational characteristics. The excerpt highlights three significant threats: malware, *Denial-of-service* (DoS) attacks, and *Man-in-the-middle* (MITM) attacks [Theodoropoulos et al., 2023].

The main security challenges include *Identity and Access Management* (IAM) [Singh et al., 2022], wherein a highly distributed environment ensures that only authorized entities access critical resources, which is fundamental to preventing cyber attacks, such as credential theft and privilege escalation. An attack on a single microservice can quickly spread throughout the network if security barriers are inadequate. The dynamic nature of Industry 4.0 requires continuous vigilance to detect and mitigate attacks in real-time, ensuring that industrial operations are not interrupted [Singh et al., 2024].

When dealing with security challenges in distributed systems, such as privilege escalation attacks, the need for mechanisms that ensure integrity and efficient

interaction between microservices becomes evident. In these environments, each component can be created, scaled, or replaced dynamically; the complexity of interactions and the need to align business processes create unique demands. Thus, in this context, orchestration emerges as a crucial approach to managing workflows, providing a centralized structure that not only simplifies the integration of large-scale services but also allows dynamic and real-time control of complex processes [Monteiro et al., 2018].

Cloud-native environments introduce unique security challenges that must be addressed on multiple fronts. Using containers and microservices fragments applications into smaller parts, each subject to different vulnerabilities. One of the main problems is access and permissions control, as an incorrect configuration can expose sensitive containers and data to external attacks. Furthermore, container orchestration can allow a vulnerability in a single microservice to compromise the entire system without adequate isolation. Another critical challenge is the management of identities and credentials, which becomes more complex in environments with multiple instances of services. Authentication and authorization need to be strictly implemented and managed, as privilege escalation attacks or token theft can compromise the security of the entire application. Furthermore, it is necessary to protect communication between microservices through robust encryption, ensuring that sensitive data is not exposed [Souppaya et al., 2017].

The study, Securing Cloud-Native Applications: Addressing Security Challenges in Containerization and Microservices Architectures, explores how containment of vulnerabilities in container images, inadequate privilege management, and lack of control over container communication are critical factors that increase the surface of attack. Furthermore, it also suggests that vulnerability scanning tools and the adoption of strict communication controls within container orchestration are essential to mitigate threats [Ali, 2023].

Data privacy [Dhinakaran et al., 2024] is another critical aspect of cloud-native applications. With the increase in collecting and processing large volumes of data through IoT devices and other sensors, exposing sensitive data to third parties is a growing concern. Cloud-native platforms must comply with regulations such as GDPR [Voigt and Von dem Bussche, 2017], which impose strict restrictions on how personal data must be treated and protected. Implementing mechanisms is necessary to ensure data is processed and stored securely. Encrypting data at rest and in transit is essential, as is implementing data retention policies that respect local and international regulations. Another recommended practice is using anonymization and pseudonymization to protect user privacy and minimize the impact if a data breach occurs.

Due to factors such as massive data generation, continuous collection of sensitive data IoT devices, remote storage, and processing, in which this data is transferred

to cloud platforms, increasing the risk of unauthorized access and heterogeneous sources, that is, the data coming from a wide variety of devices, each with its vulnerabilities, causes them to raise specific privacy concerns. In this way, techniques have emerged to preserve it, such as dynamic anonymization [Dhinakaran et al., 2024], a method that masks sensitive data before sharing or storing it, or homomorphic encryption, that is, through calculations performed directly on encrypted data without the need for decryption. They leave sensitive information confidential throughout its life cycle, even during processing.

Privacy by Design (PbD) [Kostova et al., 2020] is a principle that emphasizes the integration of privacy as a fundamental element in the design and development of systems. In the case of IoT systems, where there is a vast amount of data to be generated and processed, this ensures that privacy practices are incorporated, such as minimization of data collection, application of anonymization and security measures by default, and also ensuring that devices and systems respect privacy throughout their life cycle. The combination of PbD with advanced data protection technologies aims to not only protect users' privacy but also ensure compliance with GDPR regulations [Li and Palanisamy, 2018].

Security frameworks are structured sets of best practices and tools to protect applications and infrastructure; in a cloud-native environment, the complexity and distribution of resources require clear guidelines to ensure security is implemented throughout its entire life cycle. Continuous monitoring promotes real-time security by identifying suspicious behavior and responding to incidents before they cause damage, done through tools such as Falco and the Aqua Security security platform. These tools allow the identification of threats in real-time, generating alerts and detailed logs for analysis and automation of responses to contain threats quickly.

Zero Trust Architecture (ZTA) is a security model grounded in the principle that no user, device, or system is trustworthy by default, even if operating within the network. This model enforces strict security protocols by requiring all users and programs to operate with the least privileges necessary, following the PoLP [Schneider, 2003]. It mandates authentication and continuous verification for every access request, regardless of its origin or the user's previous interactions.

As the name suggests, microservices divide an application into small independent services, each with specific functionality, bringing benefits such as scalability and modularity. However, this architecture introduces new security challenges such as increasing the attack surface, isolation insufficient since the failure of a microservice can compromise the entire application or system due to its interdependence with other services, and also the contention of resources since several microservices share the same resources, which can be impacted due to DoS attacks. To mitigate risks, open-source tools allow accessible and adaptable security implementation, such as Falco, which monitors the behavior of containers to identify abnormalities, or Trivy,

which detects vulnerabilities in container images [Theodoropoulos et al., 2023].

Many challenges when implementing CI/CD systems result from a lack of collaboration and communication between operations and development teams. To align priorities to achieve a common goal of successful, safe project execution, DevSecOps results in alignment with the principle *Culture, Automation, Measurement, Sharing* (CAMS) of DevOps [Myrbakken and Colomo-Palacios, 2017].

Security and privacy challenges that must be addressed proactively by adopting modern security practices, such as DevSecOps, ZTA and encryption policies, companies can ensure that their operations and data are protected, allowing the industry to continue to grow in a secure and scalable way.

2.3 Orchestration

The need for container orchestration tools has grown with the increasing complexity of distributed applications, especially in microservices and cloud-native platforms. Kubernetes has emerged as the dominant orchestration solution, enabling efficient management of container clusters. Furthermore, it automates container scheduling, deployment, and management, thus guaranteeing high availability and recovery capacity in the event of a failure [Luksa, 2017].

The open-source container orchestration system, initially created by Google and later donated to the *Cloud Native Computing Foundation* (CNCF), is known as Kubernetes, or "K8s" and even as its short "kube". Allowing the users to declare desired states of an application using concepts such as deployments and services. The installation can be done standalone or through various distributions like Red Hat OpenShift [Kubernetes, 2019].

The leading virtualization technology is containerization, which enables resource sharing and maintains user isolation. While the workloads increase, cloud service providers use clusters. This basic management unit consists of managers and workers, and when a job arrives, the managers select a worker to host the incoming job [Fu et al., 2019].

A cluster is a collection of physical and/or VMs, called nodes, deployed as a single system running distributed workloads since one node could not be enough to fulfill the needed performance. One of the characteristics of orchestration is the ability to scale, which can be vertically by adding more nodes or horizontally by replicating the resources inside the cluster [Fevereiro, 2023].

Pod is a concept specific to Kubernetes and is a group of one or more containers that share the same resources and can communicate via localhost or using interprocess communication methods [Balla et al., 2020].

Kubernetes offers built-in security features such as RBAC, network policies, and secrets management, thus ensuring the security of components by monitoring various microservices and containers. Furthermore, its automation capacity through container operations such as scheduling, scaling and self-healing reduces human intervention since manually configuring security may lead to inconsistencies among policies defined in different clusters. It may require knowledge that the administrator of each domain could not have.

Based on a client-server architecture, the Kubernetes cluster comprises a set of physical and virtual machines and other infrastructure resources responsible for running applications. The machines responsible for managing the cluster are called Masters, one acting as the primary and the other as replicas, and those that run the containers are called Nodes.

A namespace in Kubernetes is a logical abstraction that isolates resources within the same physical cluster, functioning as a separate virtual cluster. It enables the organization and separation of different environments or teams, isolating resources and names and avoiding conflicts. If a resource is not explicitly associated with a namespace, it is assigned to the default namespace, known as "default." Using namespaces promotes better management and reduces vulnerability exposure by isolating failures and limiting the impact of malicious actions [Shamim et al., 2020].

In Kubernetes, a pod is an abstraction that aggregates a set of containers with some shared resources at the same host machine [Medel et al., 2018]. It represents a group of one or more containers that share the same network namespace, making them communicate via localhost using the same IP address. Thus, there is seamless inter-container communication and consistent application behavior. Also, the state of the pods is constantly monitored with health checks if the probes are ready or alive and automatically restarted in case a failure occurs, ensuring high availability.

Because by default, Kubernetes clusters are prone to dangerous privilege escalation attacks, which allows attackers to receive root permissions unintentionally, this has an embedded admission control, *Pod Security Policies* (PSP) for limiting pod privileges [van der Slik et al., 2021].

A fundamental part of container orchestration involves ensuring their security during execution. PSA is a functionality in Kubernetes that allows security control at the pod level, establishing security policies that ensure that only pods with appropriate permissions are executed. This prevents pods from running with insecure configurations, such as unnecessary root permissions, privilege escalation, or in running containers in less secure environments. The three existing *Pod Security Stantards* (PSS) are privileged, baseline, and restricted, with the ancestor of PSA that enforces this standard and takes action on the pods that don't meet the standard [Nordell, 2022].

Secrets are special, unique objects that store this sensitive information in a coded

format, thus allowing for centralized and secure management. Some applications in Kubernetes need secrets, that is, usernames, passwords, API keys, or tokens to authenticate with external services and databases. By default, Kubernetes stores these secrets in etcd, also known as the cluster database, and they are usually in plaintext, without granular access control, which leads to the need to implement security measures for this data that is at rest.

In any orchestration environment, managing secrets such as passwords or cryptographic keys is critical to ensuring application security. Kubernetes secret management provides a secure way to store and use secrets, ensuring they are not exposed to visible text or environment variables.

Compared to the local Kubernetes environment, a GKE cluster has more advantages, including monitoring tools, automatic scaling, and security reinforcement through the Google Cloud infrastructure. Also, it adds the ability to configure security policies at an application level instead of from the Kubernetes control plane.

Due to the growth of applications *Ultra-Reliable Low-Latency Communication* (URLLC) in 5G networks, there has been a shift to edge computing and cloud-native, both distributed computing paradigms, the first processing across the edge of the network and the other distributing microservices in the cloud. However, although both allow for scalability, this becomes easier in cloud-native due to its dynamism, while the edge can be scaled horizontally by distributing workloads across multiple edge devices. Security in low-latency applications becomes challenging because the solutions must be fast and efficient without compromising performance [Budigiri, 2023].

Network policies help in low latency environments as they ensure traffic isolation and granular control of communications and minimize overhead, following the PoLP, ensuring that only necessary communications are allowed without impacting performance.

Traffic is then filtered within the cluster, as explicit rules are imposed on which pods can communicate with each other, preventing unauthorized connections and, at the same time, reducing the risk of attacks, such as lateral movement in which an attacker can move within a cluster after compromising a pod.

Extended Berkeley Packet Filter (eBPF) allows custom programs to run inside the kernel without needing code modification, bringing benefits such as discarding unwanted packets before consuming the system resources and, therefore, reducing processing overhead. It replaces traditional networking tools like iptables, though reliable, but they have limitations due to their slow evolution and the challenges of adapting them to rapidly changing network environments. Usually suffering a performance degradation as the number of rules increases, which work through sequential rules, where each packet is compared with the rules in order until a match is found[Scholz et al., 2018].

Regarding network security, Kubernetes and tools like Calico offer a robust solution for implementing network policies. Calico is widely used to provide traffic control policies between pods, restricting access to internal and external resources based on defined permissions. This can leverage eBPF to enhance performance and scalability. Network policies allow the isolation of services or application components, ensuring that only authorized traffic circulates between pods, thus protecting the infrastructure against possible attacks or unauthorized access.

Cilium is a cloud-native networking solution that fully harnesses eBPF to provide security, observability, and load balancing in Kubernetes environments. Additionally, Cilium enhances network logging and enables traffic encryption between Kubernetes nodes [Riegel, 2024].

Integrating security policies into Kubernetes creates a highly secure and scalable container execution environment. Combining PSA, secret management, and network policies with calico, security is applied across multiple layers, from container deployment and execution to network traffic control. Combining these practices mitigates the risk of exploitable vulnerabilities and offers excellent protection for the cluster's sensitive data.

2.4 Observability

The increased complexity of the microservices architecture has made it necessary to adopt a management layer to control communication between services. The service mesh is an infrastructure that facilitates communication, security, and observability between microservices. This allows for more granular control over traffic between services, ensuring secure and efficient communication. One of the solutions suggested by the service mesh is Istio, which provides advanced functionalities for traffic management, security, monitoring, and access policies between microservices [Calcote and Butcher, 2019].

The service mesh is then a level of infrastructure dedicated to providing features to microservices applications. In an instance, there is a single running copy of a microservice on Kubernetes. An example is a small group of containers called pods. The sidecar proxy mechanism consists of running alongside a single instance or pod, allowing the traffic to be routed between containers. Brings numerous advantages, such as resilience, observability, and security. In this way, each application service instance is connected to a sidecar proxy, a component aimed at handling communications between services, monitoring them, and ensuring security [Nevola, 2023].

One of the leading security features offered by Istio is mutual *Transport Layer Security* (TLS) [Pace, 2021], as it allows encrypted communication and authentication between services. This enhances security by ensuring that both parties in a network connection can verify each other's identities through the correct private

key. Therefore, preventing attacks of a MITM nature or other interception attacks [Neves, 2024].

Istio is a platform of service mesh responsible for providing policy-based network services for network-connected workloads by enforcing the desired behavior of the network in the face of constantly changing conditions, such as load, configuration, and resources within or incoming and going [Calcote and Butcher, 2019]. Operating as an extension of Kubernetes, operating at the application layer (Layer 7), which is the highest of seven layers on the *Open Systems Interconnection* (OSI) model, a conceptual framework to understand and standardize communication between systems and providing advanced traffic management and security, thus increasing the observability of microservices without touching the application code. Although Kubernetes provides basic networking capabilities, as seen in the topic focused on network policies where it was possible to determine how pods can communicate between themselves and the outside, Istio can offer more granular capabilities regarding traffic through features such as traffic routing, splitting, or, mirroring.

The Istio architecture is divided into two main components; one is responsible for communication between microservices deployed alongside each service in the service mesh. The other component is the control plane, which manages and configures the data plane and ensures that proxies enforce rules correctly. Inside the control plane exists several key elements, such as the pilot responsible for service discovery and traffic management and the citadel that provides security features such as automatically handling TLS certificates. The galley is the Istio configuration validation, and last but not least importantly, the mixer provides policy enforcement aggregating telemetry data for monitoring [Larsson et al., 2020].

Istio is the most used solution but isn't the only service mesh. Linkerd is a solution to the complexity presented by Istio due to its simplicity and lightweight. Initially, this was written in Java but later completely rewritten in Rust to optimize its performance, offering runtime debugging, observability, reliability, and security without making changes to the code [Adinegoro et al., 2022].

A renowned company in the sustainable ecosystem is HashiCorp [Cultrera, 2022]. It is responsible for several tools, one of which is the service mesh Consul Connect, which not only has features such as mutual TLS [Hahn et al., 2020] but also service discovery and traffic management, just like Istio offers. This company also contains other tools that, when used together, form a cohesive ecosystem to manage infrastructure, services, and security in an automated and efficient way, such as Nomad, responsible for orchestrating workloads, similar to Kubernetes and Vault, a tool for managing secrets and offering protection to sensitive data [Wilken and Eulisse, 2024].

Since Kubernetes lacks features for routing and granularly managing the traffic inside the cluster. Istio traffic management allows the managing of incoming and

outcoming traffic via ingress and egress gateways, providing all the traffic features and also for external services, integrating them inside the cluster seamlessly. For traffic shaping, Istio provides two routing primitives: Virtual services and Destination rules. Virtual services describe how the traffic must be routed to a given destination by creating routing rules [Pace, 2021].

Authorization policies describe controlling access to services using criteria such as user or service identity to ensure security and compliance. Thus, these define who can access certain services within the service network, being even more granular than traffic management techniques.

The authorization policies define the permissions for each identity, specifying what actions they are allowed or denied to perform. These policies can be applied at different levels, including mesh, namespace, or workload. They support four types of actions: ALLOW, which permits a request if a matching rule is found; DENY, which blocks a request if a deny rule applies; CUSTOM, which delegates authorization decisions to an external system; and AUDIT, which logs authorization requests without affecting their execution [Pace, 2021]. Table 2.3 provides a detailed explanation of each action.

Action	Description
ALLOW	The request is allowed if an allow rule is matched.
DENY	The request is denied if a deny rule is matched.
CUSTOM	Custom rules are an experimental feature. They are used to delegating the authorization decision to a custom external authorization system defined in the global mesh options.
AUDIT	Audit actions do not impact the request but determine whether the authorization request should be logged. This feature only works if a supporting audit plugin is enabled.

Table 2.3: Authorization Policy Actions

2.5 Security and Privacy Frameworks

Next, a set of security and privacy frameworks is presented that guide organizations in implementing security solutions suited to their specific operational and regulatory needs.

GDPR was introduced in 2016 to bring protection to personal data [Union, 2016], making it mandatory to obtain consent on the use of personal data.

NIST 800-171 framework [Ross et al., 2019] establishes security requirements to protect *Controlled Unclassified Information* (CUI) within U.S. federal agencies. It provides guidelines to prevent unauthorized access and data breaches.

ISO/IEC 27000 Series [Disterer, 2013] represents a set of international standards for *Information Security Management System* (ISMS) applicable to all industries. These standards help organizations manage and secure their information assets while ensuring confidentiality, integrity, and availability.

NIST SP 800-53 [Force and Initiative, 2013] represents a comprehensive security control framework designed for U.S. federal agencies but also applies to private sector organizations. It offers detailed guidance on managing information security and privacy risks.

Directive (EU) 2022/2555 (NIS2) [of the European Union, 2022] is a European cybersecurity directive that mandates each EU member state to develop a national cybersecurity strategy. It establishes strategic objectives, resource allocation, and regulatory measures to enhance cybersecurity resilience across the EU.

2.6 Related Work

This section reviews relevant studies in the field of security and privacy in cloud-native platforms to contextualize the proposed work and highlight its contributions compared to prior efforts. The following works analyzed are organized in table 2.4 with the order in which they were presented with the respective title, authors, and year of publication.

Table 2.4 shows several documents that were analyzed and how their development and conclusions helped in the process of creating the practical part of the project, as well as the decision-making of which security and privacy methods and associated technologies and the impacts of their use to promote a project that ultimately aims to cover everything from the beginning of container management through vulnerability analysis to secure communication such as the application of authorization policies as well as their automation.

Title [author(s), year]
An Evaluation of Container Security Vulnerability Detection Tools [Javed and Toor, 2021]
Security Auditing of Docker Container Images in Cloud Architecture [Ahamed et al., 2021]
Sequence-based System Call Filtering for Enhanced Container Security: is it beneficial? [Song et al., 2023]
A Systematic Evaluation of CVEs and Mitigation Strategies for a Kubernetes Stack [Nordell, 2022]
Secret Management in Managed Kubernetes Services [Pai and Kunte, 2023]
Network Policies in Kubernetes: Performance Evaluation and Security Analysis [Budigiri et al., 2021]
Building Resilient Microservices with Kubernetes and Istio [Kutsa, 2025]
Balancing Load: An Adaptive Traffic Management Scheme for Microservices [Zhou et al., 2023]
Building Secure Microservices-Based Applications Using Service-Mesh Architecture [Chandramouli et al., 2020]

Table 2.4: Related Works

The document [Javed and Toor, 2021] provides a detailed analysis of security tools for detecting container vulnerabilities. Using two metrics, these are detection coverage, that is, whether it is possible to identify both OS packets and application packets, also called non OS packets. The detection hit rate, measuring the accuracy of the tools, is if the proportion of identified vulnerabilities corresponds correctly to the existing total. Several tools were selected for this study, such as Clair and Anchore, which presented the best detection rate. This study concludes that although these tools performed well in the function they were supposed to fulfill, none provided full coverage. There were still significant flaws in detecting vulnerabilities in application packages.

The images in Docker Hub can be signed or not. However, the user should verify their integrity before using them to promote a more secure environment. Thus, the document [Ahamed et al., 2021] presents several use cases, one promoting a more secure framework, A2, called "Check the trust of the base image." This framework aims to verify the reliability of images extracted from public repositories. Therefore, to implement this use case, it is necessary to enable DCT so that when trying to extract an unsigned image, Docker should display an error informing that it is not

trustworthy and, in turn, its use will not be allowed. This reduces the risk of compromising applications and infrastructure due to altered or vulnerable images.

The third paper [Song et al., 2023], came to help in the decision-making of which approach would be the best for the implementation of the validation of the security and privacy framework since the advantages of using seccomp profiles in blocking system calls were mentioned, to reduce the attack surface. This makes a more in-depth study of existing exploits and the sequence they follow so that the system can intervene as quickly as possible as soon as it suspects the possible existence of an attacker.

PSA introduced as a successor to PSP is widely adopted to harden pod security in Kubernetes. The study [Nordell, 2022] analyzes PSA highlighting its modern approach to applying namespace-based security policies, reducing the attack surface by segmenting workloads into three categories: privileged, baseline, and restricted. This study shows that although PSA improves attacks against privilege escalation, it cannot resolve all vulnerabilities, such as those associated with the runtime.

Secure secrets management is a key concern in Kubernetes environments. Thus, the study [Pai and Kunte, 2023] analyses different approaches used by providers, including *Amazon Elastic Kubernetes Service* (EKS), *Azure Kubernetes Service* (AKS), GKE, among others to protect API credentials and keys. Concluding the selected providers as the way of storing for all of them is through the etcd database, and they offer additional support for Envelope Encryption, which in the case of GKE uses *Google Cloud Key Management Service* (Cloud KMS). Although in the end, the study does not directly point out which is the best approach, it reveals that the most complete would be GKE in terms of security, also highlighting AKS due to the use of Azure Private Link which allows access to services from Azure privately, without exposing traffic to the public internet.

Responsible for investigating the performance and security threats associated with the use of network policies in Kubernetes is the work [Budigiri et al., 2021]. In specific environments, applications from different organizations may share clusters, so it is necessary to restrict communication between pods and namespaces, preventing lateral movement by attackers. It shows the importance of the lack of adequate network segmentation by default, which can lead to serious security vulnerabilities, allowing attackers to compromise multiple services within the cluster and amplify the impact of an attack. To this end, a comparative study is carried out between two plugins, Calico and Cilium, which use eBPF to optimize traffic control—concluding that in inter-node scenarios, calico outperforms due to the use of direct routing via kernel instead of tunneling which adds overhead. It also shows risks identified as misconduct of policies and not following the PoLP.

The paper "Building Resilient Microservices with Kubernetes and Istio" discusses essential tools for managing microservices in a scalable, secure, and efficient way.

Kubernetes offers container orchestration, allowing automatic deployment, load balancing, and scaling; however, it must be complemented with the Istio service mesh, leading to minimization and improved fault tolerance. It includes mutual TLS as one of the leading security features, through the component present in the Citadel control plane, ensuring that all communications between services are encrypted and authenticated and, in turn, preventing MITM type attacks [Kutsa, 2025].

The study proposes a dynamic mechanism to improve service response efficiency by automating real-time traffic policies. When new versions are released, the distribution may overload the old versions by requesting a new request allocation adjustment. The study [Zhou et al., 2023] introduces the *Dynamic Traffic Management Assistant* (DTMA) solution, a plugin that adjusts policies in real-time.

The document [Chandramouli et al., 2020] provides guidelines on authentication and authorization in microservices architectures, paying special attention to implementing security in the service mesh. Within authorization mechanisms, it focuses on implementing identity- and context-based policies such as RBAC and ABAC, one based on roles and the other on attributes. Promoting a ZTA, assuming that no communication within the network should be trusted by default and ensuring that all traffic in microservices should be encrypted through mutual TLS.

2.7 Summary

This chapter presented the state of the art by providing the key concepts, technologies, and relevant related work.

The next chapter will present the proposed framework, which fits the requirements.

Chapter 3

Specification of the Security and Privacy Framework in a Cloud-Native Platform

This chapter defines the proposed security and privacy framework for cloud-native platforms. It presents the key requirements, including vulnerability scanning, runtime security, and secure communication. The framework's architecture outlines its components and security best practices. The framework ensures a safe and resilient cloud-native environment by integrating principles such as ZTA and the PoLP.

3.1 Requirements

Based on the review of the state of the art on security and privacy of cloud-native applications, the main threats and vulnerabilities were identified, leading to the adoption of best practices such as ZTA, PoLP, and security measures in CI/CD environments. The analysis of existing frameworks highlighted gaps and opportunities for improvement. These findings guided the specification of requirements essential for developing a secure and resilient framework.

Considering security since the early design stage was crucial. This is where the principle of Security-by-design comes in, as security is incorporated as an essential

element rather than added later as a stopgap solution [Benzel et al., 2005]. To do this, all potential risks and vulnerabilities related to the system must be identified using fundamental principles such as PoLP, which ensures that only essential permissions are assigned. Defence-in-depth implements multiple layers of security to maintain protection regarding one of the layers ending up being compromised, therefore, secure-by-default configurations, can ensure that the framework starts in the most secured state possible by minimizing exposed ports or unnecessary permissions.

Table 3.1 lists the main requirements identified for implementing the security and privacy framework on the cloud-native platform, as specified in the project. These requirements were defined to cover essential aspects related to security and privacy, as well as a short description of each.

Nº	Name	Description	Citation
1	Privacy by Design	Integrates privacy principles into the framework from the ground up, ensuring data protection and compliance throughout the lifecycle.	[Cavoukian et al., 2021]
2	Vulnerability Scanning	Identifies and mitigates vulnerabilities in container images.	[Holm et al., 2011]
3	Content Trust	Ensures the integrity and authenticity of container images through digital signatures.	[Xu et al., 2017]
4	Runtime Security	Implements policies to prevent privilege escalation and restrict unauthorized operations.	[Flauzac et al., 2020]
5	Automation	Integrating security practices into the CI/CD pipeline for automated checks.	[Nagpal et al., 2024]
6	Security Admission	Enforces security policies at the pod level to ensure secure configurations.	[Nordell, 2022]
7	Network Policies	Define rules to control traffic between application components.	[Budigiri et al., 2021]
8	Secret Management	Manages sensitive data like passwords and keys securely.	[Pai and Kunte, 2023]
9	Secure Communication	Ensures encrypted and authenticated communication between services.	[Zdun et al., 2023]
10	Mutual Authentication	Ensures that connections between services are encrypted and established only after both parties verify each other	[Chandramouli et al., 2020]
11	Authorization Policies	Implements access controls to ensure only legitimate interactions occur.	[Pace, 2021]
12	Traffic Management	Optimizes and secures traffic routing between services.	[Calcote and Butcher, 2019]
13	Scalability	Maintains performance and security in high-load scenarios.	[Alshuqayran et al., 2016]

Table 3.1: Requirements Table

Next, we will discuss the previous requirements.

The first requirement selected for the framework specification is **PbD**. This is because privacy is a measure that must be implemented preventively and not as a remedy. That is, in advance, rather than only offering solutions after privacy violations such as exposed information have occurred. Therefore, this must already be built into the system, being protected without the user needing to intervene, integrated seamlessly into operations without affecting performance, and the controls for these must cover your entire lifecycle retaining and properly deleting confidential data when no longer needed [Cavoukian et al., 2021].

Vulnerability analysis is another aspect to improve application security in a cloud-native environment, ensuring the code is analyzed before the deployment stage. Analysis tools allow the identification of known flaws and expose potential risks that attackers could exploit. This way, threats can be mitigated before execution, reducing the risks of compromising the application environment.

Implementing **content trust** mechanisms ensures that only trusted container images can run. The use of digital signatures protects against image manipulation, preventing the execution of maliciously modified artifacts. Therefore, this practice strengthens security by ensuring that each component used in the environment is legitimate and has not suffered from unauthorized changes during its life cycle.

While applications are running, their protection is enforced by considering security policies. Those policies avoid privilege escalation and minimize the attack surface. Custom security profiles define specific restrictions on running processes, ensuring that applications operate with only the necessary privileges. This approach of **runtime security** effectively reduces the attack surface while adhering to ZTA principles and implementing PoLP.

When integrated into the CI/CD pipeline, security practices ensure that compliance checks and security analysis are automated throughout the development cycle. This **automation** process allows for the early identification of vulnerabilities, preventing security flaws from spreading [Nagpal et al., 2024].

Security Admission defines security standards for executing pods within the Kubernetes environment, ensuring that only secure configurations are accepted. Implementing different levels of security allows the restriction of permissions granted to pods, preventing the execution of processes with unnecessary privileges respecting the PoLP and reducing the risk of attacks.

There must be control for communication between system components to be carried out securely. This is done through **network policies** that allow the definition of isolation rules, limiting access to only strictly necessary services. This prevents unauthorized lateral movements within a cluster and reduces the risk of internal attacks.

Secure management of secrets such as API keys and access credentials prevents the exposure of sensitive information in source code or logs. To this end, specific or already incorporated tools allow the **secret management** to store and distribute the information securely, ensuring that only authorized services can access them.

In a cloud-native environment, protecting transmitted data is a fundamental requirement, as **secure communication** between services can prevent attacks such as packet interception and tampering. To this end, end-to-end encryption and **mutual authentication** ensure the integrity and confidentiality of data in transit.

The implementation of **authorization policies** ensures that only legitimate interactions are allowed within the environment, so mechanisms such as RBAC and ABAC enable refined access control, ensuring that the system components operate within the stipulated permissions.

Efficient **traffic management** is necessary to optimize the distribution of requests between services and ensure availability and performance. This can be done by implementing techniques such as traffic division and load balancing, preventing resource overload, and improving framework resilience.

The framework architecture must be designed to support dynamic growth without compromising security. Therefore, **scalability** practices, such as automatic container replication and dynamic resource adjustment, ensure the infrastructure can handle increased demands safely and efficiently.

3.2 Architecture

The proposed architecture for the security and privacy framework on a cloud-native platform is structured to integrate technologies and the ability to effectively address the complex challenges posed by cloud-native platforms adopting security practices, allowing for a robust architecture. Combining components and integrations with specific technologies and support tools to guarantee security and privacy at all system layers, since best practices include considering security from the beginning, reducing costs by avoiding vulnerabilities in later stages and improving the overall quality of the software, strengthening resistance against attacks [Assal and Chiasson, 2018].

The framework incorporated advanced security practices widely adopted on cloud-native platforms. This includes component analysis to identify vulnerabilities before implementation, ensuring they comply with security standards. A mechanism implemented to ensure the integrity and authenticity of the components, avoiding alterations or prohibited access. Furthermore, through the standard security profile, it was configured to protect the operational environment against possible attacks, reinforcing defenses during their execution.

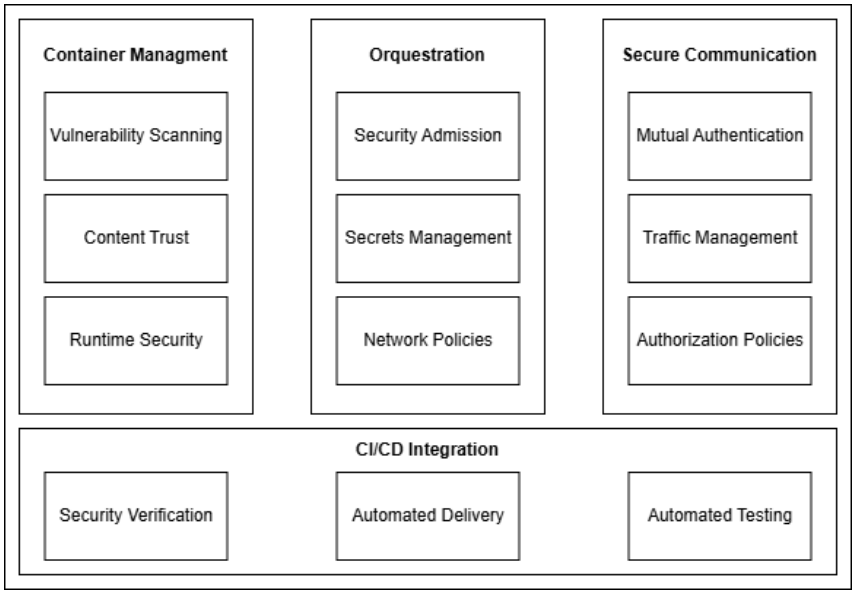

Figure 3.1: Framework's Architecture

Figure 3.1 depicts the Security and Privacy Framework architecture for Cloud-Native applications with their components. The structure is organized into four main blocks: **Container Management**, **Orchestration**, **Secure Communication** and **CI/CD Integration**, each of them associated with specific requirements.

Container Management refers to the security practices applied directly to container management, ensuring containers are secure from when they are building until they are deployed and running. The first block, named **Vulnerability Scanning**, represents the need for scanning container images before deployment to identify known vulnerabilities, preventing the execution of insecure components. Related to verifying the authenticity of container images through digital signatures is the **Content Trust** block, ensuring that only trusted versions are executed. The third **Runtime Security** block applies security policies during container execution, limiting permissions and preventing possible attacks based on privilege escalation.

Orchestration ensures policies are applied at the cluster level. It presents **Security Admission** as it controls which configurations are allowed for running pods, restricting unsafe practices. This also includes **Secret Management**, which implements secure management of the credentials and sensitive information, preventing undue exposure. Still within orchestration, intending to ensure isolation by preventing unauthorized access between pods, there are **Network Policies** that define the communication rules between the different services.

The third represents **Secure Communication**, as it deals with the mechanisms that ensure that communications between the framework components are secure, protecting the integrity and confidentiality of data. **Mutual authentication** block ensures that connections between services are encrypted and established only after

both parties verify each other using digital certificates. Providing load balancing is the **Traffic Management** for controlling and optimizing safe communication of services. Lastly, for this portion, ensuring only authenticated entities can interact between the different services is the responsible block named **Authorization Policies**.

CI/CD Integration, through the implementation of security in the development pipeline, ensures security checks and tests are part of the software life cycle. Focusing on Security Verification ensures that automatic security checks and verifications prevent vulnerable code from being promoted to other lifecycle phases. Related to continued delivery, Automated Delivery ensures that only verified versions are deployed. Finally, Automated Testing ensures that security tests are implemented automatically, preventing the introduction of vulnerabilities throughout development.

Figure 3.1 represents what the framework should encompass, ranging from security in building and running container images through secure orchestration and communication to integrating security into the continuous development process. This representation helps ensure that the framework on a cloud-native platform is resilient against threats, complies with regulations, and provides a secure and private environment.

3.3 Summary

This chapter proposed the security and privacy framework specification to protect cloud-native platforms.

The next chapter will present the validation work of the security and privacy framework to protect cloud-native platforms supported by a PoC.

Chapter 4

Validation

This chapter describes the validation work of the proposed security and privacy framework on a native cloud platform using a systematic approach PoC, which involves testing the framework against predefined requirements and scenarios. Each framework component, container management, orchestration, and secure communication, is thoroughly analyzed to verify its ability to mitigate vulnerabilities, enforce security policies, and protect sensitive data. Integration with CI/CD pipelines is also highlighted, automating and strengthening the life cycle at the end of each specific mechanism implemented. The results will be critically analyzed to confirm the practical effectiveness of the framework and identify potential areas for improvement.

4.1 Container Management

PoC is created to demonstrate how the framework specifications are applied to protect cloud-native applications using scenarios that reflect the best practices referred to in the previous list in Chapter 3.

This PoC is focused on testing a critical part of the project to verify that the framework can be implemented. It is usually applied to decide whether it is worth the financial investment to move forward with the implementation, thus reducing costs and time if it is not viable. Combining technologies such as Docker, Kubernetes, and Istio to strengthen security in cloud-native applications and the CI/CD integration with the GitHub Actions tool.

Among the list of objectives of this PoC for the container management portion are the security validation of container images before their implementation, as well as the activation of DCT to guarantee the authenticity and integrity of the pulled images and blocking calls to the system so that privilege escalation does not occur.

Several security practices exist in Docker integration, such as using official and verified images; there are also techniques for strengthening containers, such as minimizing permissions, configuring security policies, continuous monitoring to maintain a robust security system, and logging for early incident detection.

4.1.1 Vulnerability Scanning

To ensure that only images used in a cloud-native environment are free from vulnerabilities, scanning is carried out using tools such as Trivy and Grype to allow the identification of known vulnerabilities, mapping them with public databases such as CVE that categorize by severity.

To validate the security of images used in the environment, the detection of vulnerabilities before deployment, and this way, demonstrating the effectiveness of scanning tools in the CI/CD pipeline.

In the implementation of this PoC, it is necessary to have Docker installed, and the Nginx image in the latest version was used during the various validation test implementations.

Two selected tools are used for scanning the image to identify the vulnerabilities, classify them based on severity, and point out which CVE associated or vulnerable packages are in a report. Based on this, it is possible to correct critical and high vulnerabilities; the logs prove the effectiveness of the tool's scanning process.

Automating this image analysis to check images with many vulnerabilities is possible in integrating the GitHub actions CI/CD pipeline.

Image scanning tools exist to find image vulnerabilities, and the more types of risks the scanner reveals, the more effectively it is possible to intercept and react to the data. In this way, it is possible to have an advantage in positioning to stop vulnerabilities and other threats before they spread. For this practical portion dedicated to image scanning, tools like Trivy and Grype were used for the same nginx image to compare their results later in this chapter.

The command applied in the figure 4.1 is used to download the latest version of the official ngnix docker image from docker hub, a popular container image registry.

```
PS C:\Users\Yuka\Documents\Software Tese\Trivy> docker pull nginx:latest
latest: Pulling from library/nginx
e4fff0779e6d: Pull complete
2a0cb278fd9f: Pull complete
7045d6c32ae2: Pull complete
03de31afb035: Pull complete
0f17be8dcff2: Pull complete
14b7e5e8f394: Pull complete
23fa5a7b99a6: Pull complete
Digest: sha256:447a8665cc1dab95b1ca778e162215839ccbb9189104c79d7ec3a81e14577add
Status: Downloaded newer image for nginx:latest
docker.io/library/nginx:latest
```

Figure 4.1: Docker pull command

Figure 4.2 presents the previously downloaded image now available in this PoC.

```
PS C:\Users\Yuka\Documents\Software Tese\Trivy> docker images
REPOSITORY                    TAG                                       IMAGE ID       CREATED         SIZE
nginx                         latest                                    5ef79149e0ec   2 weeks ago     188MB
```

Figure 4.2: Docker image

Installation of Trivy is confirmed with the version command shown in figure 4.3.

```
PS C:\Users\Yuka\Documents\Software Tese\Trivy> .\trivy.exe --version
Version: 0.54.1
```

Figure 4.3: Trivy version

The most recent version of the image with the name 'nginx' was chosen because it presents a wide variety of vulnerabilities, categorized as 'UNKWON', 'LOW', 'MEDIUM', 'HIGH' and 'CRITICAL'. The actions taken to get the image and then analyze it were the commands ´docker pull nginx:latest' and then 'trivy image nginx:latest' as presented in the table, with the total 145 vulnerabilities separated into the existing categories as shown in the figures 4.4 and 4.5.

```
PS C:\Users\Yuka\Documents\Software Tese\Trivy> .\trivy.exe image nginx:latest
2024-08-29T10:55:36+01:00       INFO    [db] Need to update DB
2024-08-29T10:55:36+01:00       INFO    [db] Downloading DB...  repository="ghcr.
52.46 MiB / 52.46 MiB [-----------------------------------------------------------
2024-08-29T10:56:15+01:00       INFO    [vuln] Vulnerability scanning is enabled
2024-08-29T10:56:15+01:00       INFO    [secret] Secret scanning is enabled
2024-08-29T10:56:15+01:00       INFO    [secret] If your scanning is slow, please
2024-08-29T10:56:15+01:00       INFO    [secret] Please see also https://aquasecu
2024-08-29T10:56:32+01:00       INFO    Java DB Repository      repository=ghcr.i
2024-08-29T10:56:32+01:00       INFO    Downloading the Java DB...
639.00 MiB / 639.00 MiB [---------------------------------------------------------
2024-08-29T11:01:48+01:00       INFO    The Java DB is cached for 3 days. If you
2024-08-29T11:01:49+01:00       INFO    Detected OS     family="debian" version="
2024-08-29T11:01:49+01:00       INFO    [debian] Detecting vulnerabilities...   o
2024-08-29T11:01:49+01:00       INFO    Number of language-specific files       n
2024-08-29T11:01:49+01:00       WARN    Using severities from other vendors for s

nginx:latest (debian 12.6)
==========================
Total: 152 (UNKNOWN: 0, LOW: 89, MEDIUM: 44, HIGH: 16, CRITICAL: 3)
```

Figure 4.4: Trivy running and total of vulnerabilities

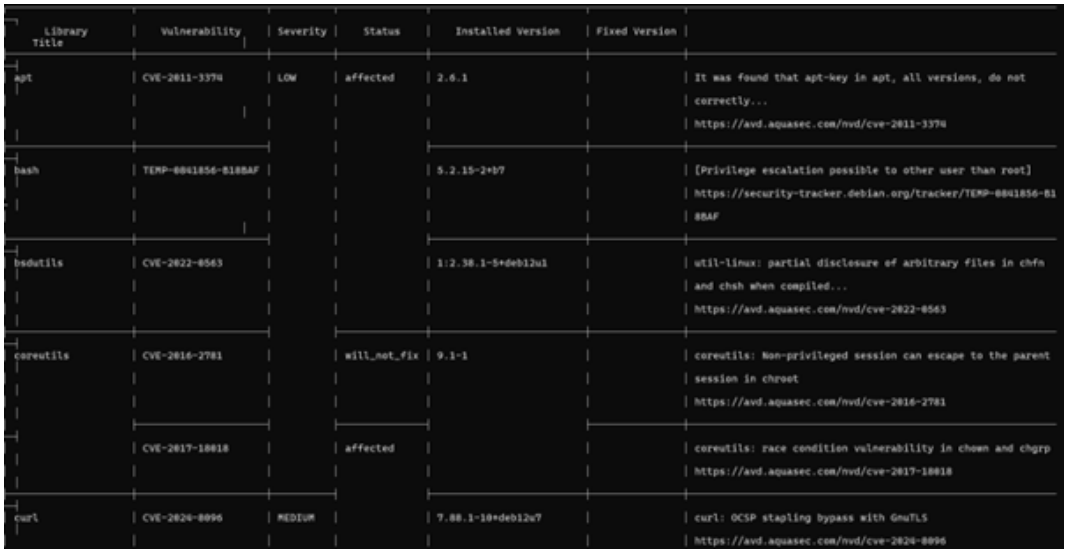

Figure 4.5: Trivy results

The other image scanning tool was Grype, and its installation process was straightforward, including the simplicity of downloading and running directly on the terminal. The figure 4.6 can verify the correct installation.

```
PS C:\Users\Yuka> grype version
Application:         grype
Version:             0.82.1
BuildDate:           2024-10-15T13:54:04Z
GitCommit:           50815e59c973cfd0c0247cbc2af00fa37f7cda5d
GitDescription:      v0.82.1
Platform:            windows/amd64
GoVersion:           go1.23.2
Compiler:            gc
Syft Version:        v1.14.1
Supported DB Schema: 5
```

Figure 4.6: Grype installed

To ensure the consistency of the results, the same image was used for analysis with the 'grype nginx:latest' command to carry out a detailed inspection of the image, identifying and classifying the detected vulnerabilities by severity levels: 'LOW', 'MEDIUM', 'HIGH', 'NEGLIGIBLE' and 'CRITICAL' as can be seen in Figure 4.7.

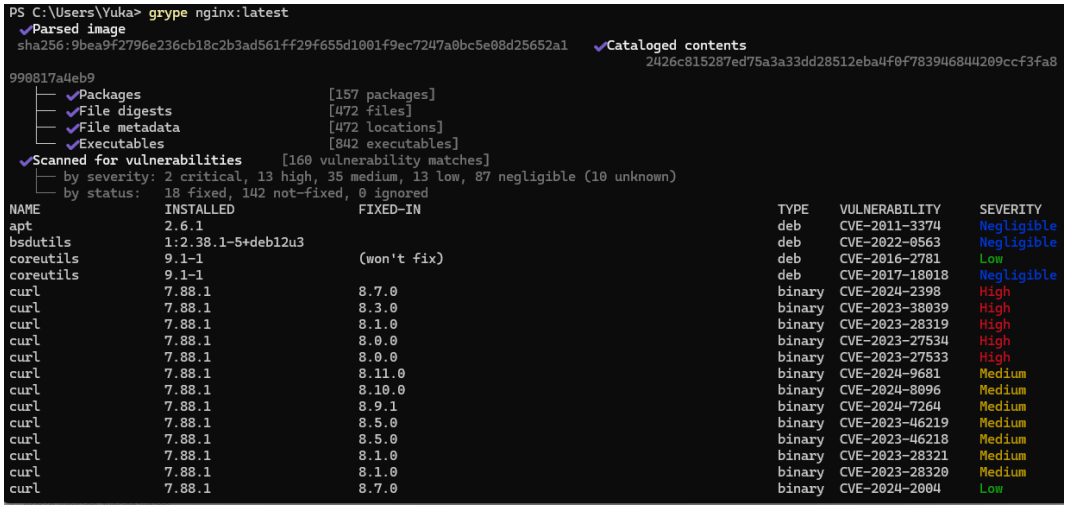

Figure 4.7: Grype scanning image results

Now that PoC of Vulnerability Scanning has been presented, the table 4.1 is composed of the results of the returned values by each of the image scanning tools, Trivy and Grype, making it easier to view and establish a comparison of the vulnerabilities detected on the nginx image in the latest version.

The severity ratings in the table 4.1 result from the vulnerability scanning by Trivy and Grype, where each tool assigns severity levels based on publicly available NVD. These categories include: 'UNKNOWN', 'LOW', 'MEDIUM', 'HIGH' and 'CRITICAL' and critical, however 'NEGLIGIBLE' is marked as null on Trivy as it is not available in their rating system.

	UNKNOWN	**LOW**	**MEDIUM**	**HIGH**	**CRITICAL**	**NEGLIGIBLE**
Trivy (145)	1	89	40	13	2	null
Grype (160)	10	13	35	13	2	87

Table 4.1: Results from the returned values by each image scanning

4.1.2 Content Trust

Beyond image scanning to find vulnerabilities, there is another way to make security more robust through DCT, which enables the integrity of the publisher of Docker images to be verified through digital signatures to verify the authenticity of the images and how they have not been tampered and come from a trusted source.

In this PoC related to the Content Trust requirement, it was possible to ensure the integrity and authenticity of the container image through digital signatures. To prevent the use of unsafe images is by activating DCT, making only safe images available for download from the repository, reassuring the integrity and authenticity of container images since this mechanism verifies that only images provided by a trusted source and that are not changed during transport or storage.

The implementation of DCT was initially done locally as it shows figure 4.8 by setting the environment variable 'docker content trust' to 1, that is, the condition true to the opposite false, represented by 0, in this way, there is a guarantee that only trusted images are pulled and pushed, enhancing security.

```
PS C:\Users\Yuka> $env:DOCKER_CONTENT_TRUST=1
```

Figure 4.8: Enable DCT

For the second step, the command shown in figure 4.9, which is used to create a new tag for the nginx image, associating it with a yukamouro25 repository on DockerHub, as this requires that images should be linked to a specific repository to ensure signature validation.

```
PS C:\Users\Yuka> docker tag nginx:latest yukamouro25/nginx:latest
```

Figure 4.9: Tagging

With Docker Content Trust enabled, pushing an image requires generating cryptographic signing keys. During this process shown in figure 4.10, docker performs actions such as checking if the image already exists if any layers are available, they end up being reused, compute SHA256 diggest to identify the image uniquely, and initiates the signing process, prompting the user to create signing keys.

Docker prompted the creation of two passphrases: a root key, a high-security key used as the primary identity for signing, and a repository key, a specific key used to sign images in a particular repository. Once the keys were successfully created, the image was signed and uploaded to Docker Hub.

```
PS C:\Users\Yuka> docker push yukamouro25/nginx:latest
The push refers to repository [docker.io/yukamouro25/nginx]
5f0272c6e96d: Mounted from library/nginx
f4f00eaedec7: Mounted from library/nginx
55e54df86207: Mounted from library/nginx
ec1a2ca4ac87: Mounted from library/nginx
8b87c0c66524: Mounted from library/nginx
72db5db515fd: Mounted from library/nginx
9853575bc4f9: Mounted from library/nginx
latest: digest: sha256:127262f8c4c716652d0e7863bba3b8c45bc9214a57d13786c854272102f7c945 size: 1778
Signing and pushing trust metadata
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with ID b9b27d7:
Repeat passphrase for new root key with ID b9b27d7:
Enter passphrase for new repository key with ID ec771cf:
Repeat passphrase for new repository key with ID ec771cf:
Finished initializing "docker.io/yukamouro25/nginx"
Successfully signed docker.io/yukamouro25/nginx:latest
```

Figure 4.10: Creating keys

Finally, to verify that the image is correctly signed, use the command on the figure 4.11 where it is possible to see all is working as intended.

```
PS C:\Users\Yuka> docker trust inspect --pretty yukamouro25/nginx:latest

Signatures for yukamouro25/nginx:latest

SIGNED TAG   DIGEST                                                             SIGNERS
latest       127262f8c4c716652d0e7863bba3b8c45bc9214a57d13786c854272102f7c945   (Repo Admin)

Administrative keys for yukamouro25/nginx:latest

  Repository Key:       ec771cfb3493ac36d969de2d637f4132b3df8d7691a3564e89bda54175c01e9d
  Root Key:     7e8fac92af04290a6d9cb7c0d6571650e50ff8091b71a9f3d65072d630bedb6c
```

Figure 4.11: Cheking keys

4.1.3 Runtime Security

System calls allow the request of services to the kernel of an OS. Some must be blocked as calls made by unauthorized users can compromise security, leading to an escalation of privileges. Therefore, in compliance with ZTA, PoLP is applied by blocking some of these syscalls. Enabling container runtime security through the use of seccomp, a profile containing numerous system calls that, when removed, are blocked. For this PoC, the chmod call was selected, which makes it impossible for users to change the permissions of a file or directory in the file system.

Each syscall allowed in a profile increases the number of possible ways to exploit vulnerabilities. Thus, by blocking this call, it is possible to reduce the attack surface of the framework, making it more secure.

The profile was defined in *JavaScript Object Notation* (JSON) format in which the chmod syscall is removed, so when an attempt is made to change the permissions of a file, this will result in an error, returning the operation not permitted message.

For container runtime security seccomp, a Linux kernel feature restricting system calls containers, was used for this project. It begins by finding a customizable default second profile and then modifying or removing specific entries by editing directly in the JSON file to turn off specific system calls.

Running the container with the customized default profile, verifying if the restriction we changed in the updated JSON file. In this case, it was achieved by removing the commonly used system call "chmod" that appears highlighted in figure 4.12 and finally testing if this operation is no longer permitted as shown in figure 4.13.

Figure 4.12: sccomp default profile

```
# touch yuka
# chmod 777 yuka
chmod: changing permissions of 'yuka': Operation not permitted
#
```

Figure 4.13: Testing chmod operation

4.1.4 Container Management Automation

The security practices, Vulnerability Scanning, Content Trust, and Runtime Security were initially performed locally and later adapted and automated in a GitHub actions workflow. This way, vulnerability checks are automated, ensuring that only safe images and their authenticity are used by activating DCT and validating security during runtime due to blocking the system call, strengthening security without compromising efficiency.

To start the GitHub actions integration process, a docker file was created present in list 4.1, a starting point for the base configurations, such as the image used to build the container. Also within this file is the EXPOSE instruction with port 80, which is not precisely an obligation but helpful information that avoids errors for future configuration.

```
1 # Use the official Nginx image as the base image
2 FROM nginx: latest
3
4 # Expose port 80
```

```
5 EXPOSE 80
```

Listing 4.1: Docker file

For the first phase dedicated to analyzing images with the ultimate objective of verifying their vulnerabilities before they go into production, it was necessary to create secret variables present in figure 4.14 for this automated process, a crucial step to guarantee the security of credentials and sensitive information since they are not exposed in the source code, only allowing access to these credentials during the execution of the workflow.

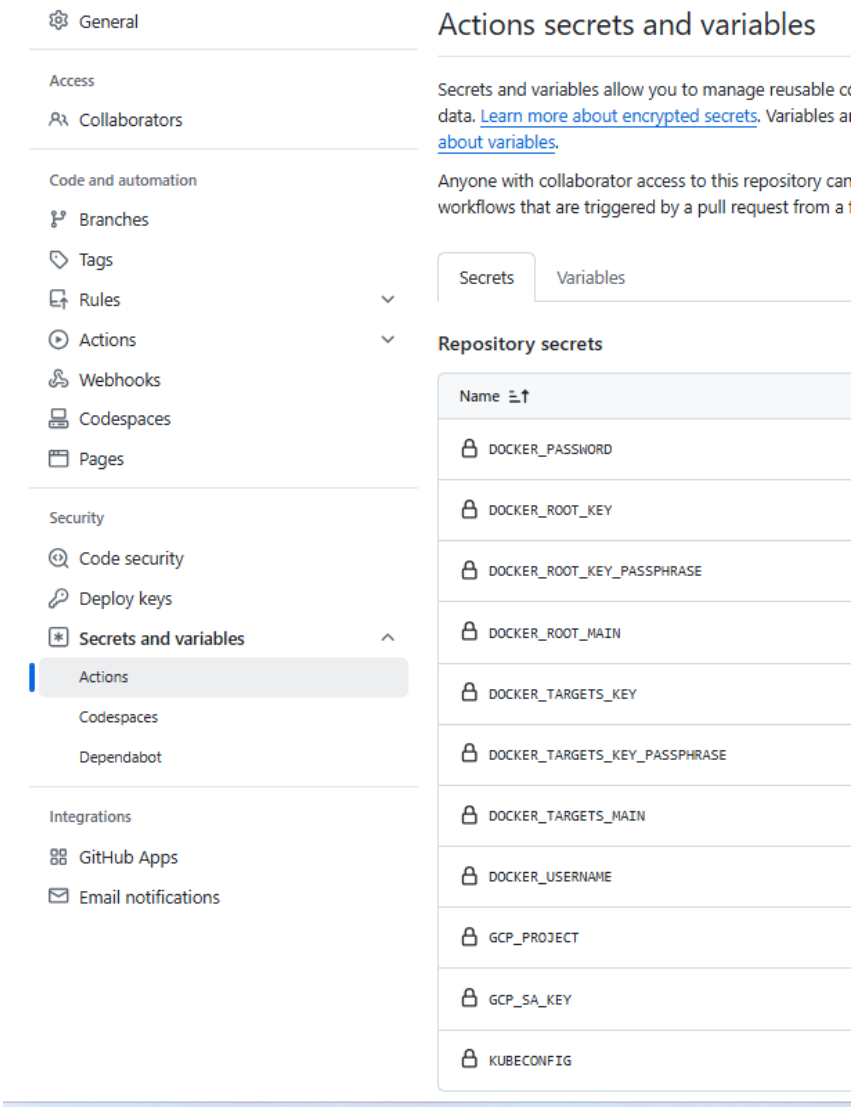

Figure 4.14: Actions secrets and variables

The listing 4.2 presents the YAML file, which contains the workflow that automates all of the processes previously mentioned, including the scanning of the image using the Trivy tool, starting with its installation, then scanning the Nginx image,

and finally by generating a report made up of a list of vulnerabilities found categorized by level of severity and also details such as the affected library or package, vulnerable and corrected version and links with more detailed information.

In addition to these steps, the workflow enables DCT, sets up signing keys, and runs the seccomp profile without the chmod syscall.

```
1  name: Docker CI/CD
2
3  on:
4    push:
5      branches:
6        - main
7
8  jobs:
9    build-and-scan:
10     runs-on: ubuntu-latest
11
12     steps:
13       - name: Checkout code
14         uses: actions/checkout@v2
15
16       - name: Set up Docker Buildx
17         uses: docker/setup-buildx-action@v2
18
19       - name: Set up QEMU
20         uses: docker/setup-qemu-action@v2
21
22       - name: Login to Docker Hub
23         uses: docker/login-action@v2
24         with:
25           username: ${{ secrets.DOCKER_USERNAME }}
26           password: ${{ secrets.DOCKER_PASSWORD }}
27
28       - name: Build Docker image
29         run: |
30           docker build -t ${{ secrets.DOCKER_USERNAME }}/nginx:
                 latest .github/workflows
31
32       - name: Set up Docker Content Trust Signing Keys
33         run: |
34           mkdir -p ~/.docker/trust/private
35           echo "${{ secrets.DOCKER_ROOT_KEY }}" | base64 -d -w 0 >
                  ~/.docker/trust/private/root_key
36           echo "${{ secrets.DOCKER_TARGETS_KEY }}" | base64 -d -w
                 0 > ~/.docker/trust/private/targets_key
37           chmod 600 ~/.docker/trust/private/root_key
38           chmod 600 ~/.docker/trust/private/targets_key
39
```

```
40        - name: Install expect
41          run: sudo apt-get install -y expect
42
43        - name: Enable Docker Content Trust and Push Docker image
44          env:
45            DOCKER_CONTENT_TRUST: 1
46          run: |
47            expect -c "
48            spawn docker push ${{ secrets.DOCKER_USERNAME }}/nginx:
                  latest
49            expect {
50              \"Enter passphrase for root key with ID ${
                    DOCKER_ROOT_MAIN}:\" {
51                send \"${{ secrets.DOCKER_ROOT_KEY_PASSPHRASE }}\r\"
52                exp_continue
53              }
54              \"Enter passphrase for new root key with ID ${
                    DOCKER_ROOT_MAIN}:\" {
55                send \"${{ secrets.DOCKER_ROOT_KEY_PASSPHRASE }}\r\"
56                exp_continue
57              }
58              \"Enter passphrase for new repository key with ID ${
                    DOCKER_TARGETS_MAIN}:\" {
59                send \"${{ secrets.DOCKER_TARGETS_KEY_PASSPHRASE }}\
                      r\"
60                exp_continue
61              }
62              eof
63            }
64            interact
65            "
66
67        - name: Install Trivy
68          run: |
69            curl -sfL https://raw.githubusercontent.com/aquasecurity
                  /trivy/main/contrib/install.sh | sh -s -- -b /usr/
                  local/bin
70
71        - name: Scan Docker image with Trivy
72          run: |
73            trivy image ${{ secrets.DOCKER_USERNAME }}/nginx:latest
74
75        - name: Run Docker with Custom Seccomp Profile
76          run: |
77            docker run -d --name test-container --security-opt
                  seccomp=.github/workflows/default-github.json ${{
                  secrets.DOCKER_USERNAME }}/nginx:latest
78
79        - name: Test chmod Command Inside the Container
```

```
80          run: |
81            # Create a file named yuka
82            docker exec test-container touch yuka
83            # Attempt to change its permissions and capture the
                  output
84            docker exec test-container chmod 777 yuka || echo "chmod
                   failed as expected."
```

Listing 4.2: Docker Code on Github actions

After running the code, it is possible to get the same scan as the one done locally, as shown in figure 4.15.

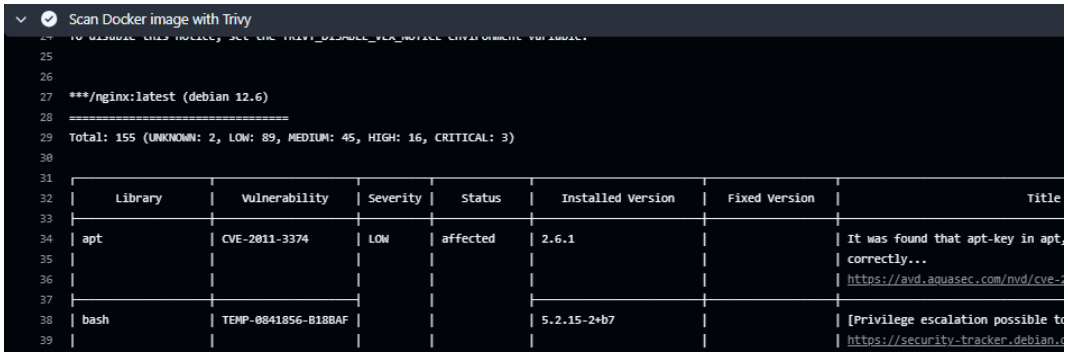

Figure 4.15: Trivy scan on GitHub Actions

The DCT stage focused on compliance with security policies through signature keys, which are essential to guarantee the integrity and authenticity of Docker images. There are two types of keys, the root key being the main trusted one, responsible for creating and signing other keys, and the target key for signing the docker images that will be published in the repository. The main one can be generated manually by the user or automatically if the image is signed for the first time and stored locally, preferably offline, to increase its security. The target key is created automatically when DCT is activated and stored with the main one. These need to be exported for use in CI/CD securely; for this, they are converted to base64 and saved in the secrets component of GitHub Actions precisely with the respective passphrases that the user defines to protect each key, thus giving extra protection.

The keys are decrypted and stored in the appropriate directory during the workflow configuration in the YAML file. Docker uses these keys and signs through DCT, ensuring they have not been changed. Finally, compliance with security standards such as access control, secure dependencies, and correct configurations is checked. The pipeline provides immediate feedback highlighting any failures that may have occurred.

The last step in the flow, which ensures runtime security container, uses the Linux kernel's seccomp security feature to restrict the set of system calls that a

process can perform with the ultimate goal of reducing the attack surface, leaving only the necessary ones, thus protecting the system against exploits that try to abuse these calls. For this process, a default JSON file available locally in the docker hub was used and manually changed to prevent a system call. In this case, "chmod" was a form of validation test in which this becomes a disallowed operation with the message present in figure 4.16.

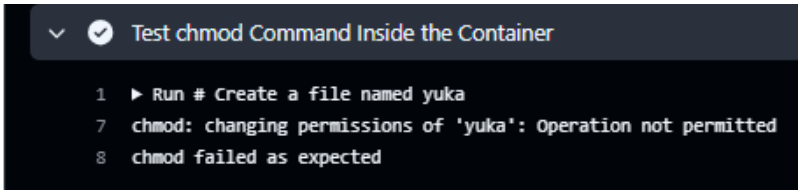

Figure 4.16: Test "chmod" comand

4.2 Orchestration

Having now secured the images in docker, it is possible in Kubernetes through the security mechanism to configure policies at the cluster level, the basic structure that organizes and manages distributed computing resources. Pods are the basic execution unit of Kubernetes representing one or more applications in containers and share the same namespace, a logical division within the cluster to isolate resources.

This PoC emphasizes the integration of Kubernetes to manage containerized applications, focusing on key areas of security and automation. Divided into Security Admission, Secret Management, and Network Policies.

4.2.1 Security Admission

PSA is configured to apply policies at the cluster level, ensuring that pods meet minimum requirements before their creation, used in labeled namespaces with different security levels: privileged, baseline, and restricted.

To apply this, namespaces were created, each with a different security level, starting with the privileged, allowing most permissions used in trusted workloads, then the baseline that will be the intermediate applying moderate security measures, and finally, the most restricted of all for control of sensitive information, the restricted.

Once the configurations of each are applied through a JSON file, with specifications that meet or violate security standards in each namespace, they are tested in several ways, both for and against, ensuring that there is a planned failure.

Figure 4.17 aims to classify pods based on the different required levels of security, which are privileged, baseline, and restricted. Each represents a different level of security, reflecting the variation in trust and access control for various workloads.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl create namespace privileged-ns
namespace/privileged-ns created

C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl create namespace baseline-ns
namespace/baseline-ns created

C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl create namespace restricted-ns
namespace/restricted-ns created
```

Figure 4.17: Creating namespaces for PSA

The following figure 4.18 shows that each namespace was labeled according to the PSA level of protection intended to be enforced. Starting with privileged, which allows the most permissions, is excellent for trusted workloads requiring extensive access. Then, the baseline, which consists of the namespace that enforces the basic security requirements, is suitable for standard applications and the one that applies the strictest security control, restricted and perfect for highly sensitive or untrusted workloads.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl label namespace privileged-ns pod-security.kubernetes.io/enforce=privileged
namespace/privileged-ns labeled

C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl label namespace baseline-ns pod-security.kubernetes.io/enforce=baseline
namespace/baseline-ns labeled

C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl label namespace restricted-ns pod-security.kubernetes.io/enforce=restricted
namespace/restricted-ns labeled
```

Figure 4.18: Labeling namespaces for PSA

After creating the namespaces in the Kubernetes cluster, they were configured by assigning security labels using the PSA mechanism. The JSON file presented in 4.3 describes a pod configured with elevated permissions (privileged: true), allowing unrestricted access to the system.

```
1  {
2    "apiVersion": "v1",
3    "kind": "Pod",
4    "metadata": {
5      "name": "test-privileged",
6      "namespace":  "restricted-ns"
7    },
8    "spec": {
9      "containers": [
10       {
11         "name": "nginx",
12         "image": "nginx",
```

```
13            "security context": {
14              "privileged": true
15            }
16          }
17        ]
18    }
19 }
```

Listing 4.3: JSON file for privileged pod

Two commands were run to create pods in different security contexts in figure 4.19. The first command applied a JSON manifest (privileged-pod.json) to create a pod in the privileged-ns namespace, as this contained security settings that defined the pod as privileged, allowing the container to have elevated permissions on the system. Because the privileged-ns namespace was previously configured with the pod-security.kubernetes.io/enforce=privileged label, Kubernetes did not prevent the creation of this pod, allowing it to run without additional security restrictions.

Unlike the first command, the second one did not use a predefined manifest but created a pod directly through the *Command-Line Interface* (CLI) using the nginx image. This pod was designed with namespace baseline-ns, which was labeled with pod-security.kubernetes.io/enforce=baseline. Since the test-baseline pod was created without any additional security configurations, it could start correctly since it did not violate the rules enforced by the baseline-ns namespace.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl apply -f privileged-pod.json --namespace=privileged-ns
pod/test-privileged created

C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl run test-baseline --image=nginx --namespace=baseline-ns
pod/test-baseline created
```

Figure 4.19: Testing the creation of privileged and baseline pods

Finally, in figure 4.20, to show that the PSA was working correctly in the opposite way, a pod with an elevated privilege was asked to try to be deployed in the restricted namespace. The error in the screenshot occurs because the PSA mechanism is enforcing restricted security policies in the restricted-ns namespace. This means that Kubernetes applies the highest level of security restrictions, preventing the creation of pods that do not meet strict security requirements.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl run test-restricted --image=nginx --namespace=restricted-ns
Error from server (Forbidden): pods "test-restricted" is forbidden: violates PodSecurity "restricted:latest": allowPrivi
legeEscalation != false (container "test-restricted" must set securityContext.allowPrivilegeEscalation=false), unrestric
ted capabilities (container "test-restricted" must set securityContext.capabilities.drop=["ALL"]), runAsNonRoot != true
(pod or container "test-restricted" must set securityContext.runAsNonRoot=true), seccompProfile (pod or container "test-
restricted" must set securityContext.seccompProfile.type to "RuntimeDefault" or "Localhost")
```

Figure 4.20: Privileged Pod in the Restricted Namespace Error

4.2.2 Secrets Management

Within Kubernetes, it also can securely manage sensitive data such as passwords, API keys, and certificates in a cluster. Ensuring that information is stored, transmitted, and accessed securely, reducing potential exposure to data leaks.

To do this PoC, information encoded in Base64 is created in a YAML file, a method of transforming binary data into a textual format using exclusively ASCII characters.

Kubernetes Secrets Management offers a mechanism that injects sensitive data, such as passwords, API keys, and certificates used by applications to run in a cluster securely and controlled, reducing the risks of exposing sensitive data through configuring files or environment variables. To do this, it was created a Secret YAML file 4.4 defining the secrets that need to be injected into a pod. In this case, the file contained the username and password encoded in base64, ensuring the sensitive information is not stored in plain text.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: my-secret
  namespace: default
type: Opaque
data:
  username: eXVrYW1vdXJvMjU=
  password: eXVrYW1vdXJvMjU=
```

Listing 4.4: Secret yaml file

After creating this file, it is applied to the Kubernetes cluster using the kubeclt command present in figure 4.21, thus allowing it to be saved securely within the cluster and available for use in the pods.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl apply -f secret.yaml
secret/my-secret created
```

Figure 4.21: Applying secrets YAML file created

The secrets were listed using the following command to check whether this was applied correctly, showing that my-secret is available in the namespace.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl get secrets
NAME        TYPE     DATA   AGE
my-secret   Opaque   2      2m21s
```

Figure 4.22: Secrets are applied

Now, it is possible to inject secrets into a pod by referencing the secret in the pod specification, allowing the sensitive data to be mounted as environmental variables or files inside the container. Accessing the pod shell and checking the environment variables or files where the secrets are expected is necessary to ensure it works accordingly. With this step, it's possible to confirm that sensitive data is only available inside where it is designated in the container in figure 4.23.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl exec -it secret-pod -- /bin/bash
root@secret-pod:/# env
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_SERVICE_PORT=443
HOSTNAME=secret-pod
PWD=/
PKG_RELEASE=1~bookworm
HOME=/root
USERNAME=yukamouro25
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
DYNPKG_RELEASE=2~bookworm
PASSWORD=yukamouro25
NJS_VERSION=0.8.5
TERM=xterm
SHLVL=1
KUBERNETES_PORT_443_TCP_PROTO=tcp
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
KUBERNETES_SERVICE_HOST=10.96.0.1
KUBERNETES_PORT=tcp://10.96.0.1:443
KUBERNETES_PORT_443_TCP_PORT=443
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
NGINX_VERSION=1.27.1
NJS_RELEASE=1~bookworm
_=/usr/bin/env
```

Figure 4.23: Pod with secret injected

The pod in Figure 4.24 is set to use a specific service account (secrets-access-sa) with access to secret permissions. This configuration allows only authorized pods to have access to particular secrets.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl describe pod secret-pod
Name:             secret-pod
Namespace:        default
Priority:         0
Service Account:  secret-access-sa
Node:             minikube/192.168.49.2
Start Time:       Sat, 07 Sep 2024 17:20:21 +0100
Labels:           app=secret-app
Annotations:      <none>
Status:           Running
IP:               10.244.0.22
```

Figure 4.24: Pod using service account

In Figure 4.25, and as with all previous configurations, it is possible to validate that they are correctly applied.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl exec -it secret-pod -- /bin/sh
# echo $USERNAME
echo $PASSWORD
yukamouro25
# yukamouro25
```

Figure 4.25: Secrets accessible from inside the pod

Despite showing that this secrets management configuration was well applied, the opposite was also demonstrated, creating an unauthorized pod to test access control. Figure 4.26 represents the inside of the unauthorized pod, and it is impossible to see either the username or the password.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl exec -it unauthorized-pod -- /bin/sh -c 'env'
KUBERNETES_SERVICE_PORT=443
KUBERNETES_PORT=tcp://10.96.0.1:443
HOSTNAME=unauthorized-pod
HOME=/root
PKG_RELEASE=1~bookworm
DYNPKG_RELEASE=2~bookworm
TERM=xterm
KUBERNETES_PORT_443_TCP_ADDR=10.96.0.1
NGINX_VERSION=1.27.1
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
KUBERNETES_PORT_443_TCP_PORT=443
NJS_VERSION=0.8.5
KUBERNETES_PORT_443_TCP_PROTO=tcp
NJS_RELEASE=1~bookworm
KUBERNETES_SERVICE_PORT_HTTPS=443
KUBERNETES_PORT_443_TCP=tcp://10.96.0.1:443
KUBERNETES_SERVICE_HOST=10.96.0.1
PWD=/
```

Figure 4.26: Unauthorized pod

This PoC focused on Secret Management ensured that only confidential information, in this case, the password, be visible only to those pods that have permission to do so.

4.2.3 Network Policies

The Calico tool implemented fine-grained network policies to control traffic between pods, services, and external endpoints, a powerful network security solution for Kubernetes. The installation of the calico began by applying the official calico manifest provided by Project Calico. Figure 4.27 shows that several components were successfully created to manage networking policies in the cluster after using the manifest.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl apply -f https://docs.projectcalico.org/manifests/calico.yaml
poddisruptionbudget.policy/calico-kube-controllers created
serviceaccount/calico-kube-controllers created
serviceaccount/calico-node created
configmap/calico-config created
customresourcedefinition.apiextensions.k8s.io/bgpconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/bgppeers.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/blockaffinities.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/caliconodestatuses.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/clusterinformations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/felixconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/globalnetworksets.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/hostendpoints.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamblocks.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamconfigs.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipamhandles.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ippools.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/ipreservations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/kubecontrollersconfigurations.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networkpolicies.crd.projectcalico.org created
customresourcedefinition.apiextensions.k8s.io/networksets.crd.projectcalico.org created
clusterrole.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrole.rbac.authorization.k8s.io/calico-node created
clusterrolebinding.rbac.authorization.k8s.io/calico-kube-controllers created
clusterrolebinding.rbac.authorization.k8s.io/calico-node created
daemonset.apps/calico-node created
deployment.apps/calico-kube-controllers created
```

Figure 4.27: Setting up Calico

To ensure that the Calico was running correctly after installation, a command was used to check the status of the Calico pods. From figure 4.28, it was possible to confirm that it was fully operational.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl get pods -n kube-system -l k8s-app=calico-node
NAME                READY   STATUS    RESTARTS   AGE
calico-node-jzlg4   1/1     Running   0          7m16s
```

Figure 4.28: Calico running

The next step is shown in 4.5 a YAML file that defines a network policy called deny-all in the default namespace; this is because it is essential to ensure that no ingress traffic or egress is allowed for pods within the namespace where the policy is applied.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: deny-all
  namespace: default
spec:
  podSelector: {}
  policyTypes:
    - Ingress
    - Egress
```

Listing 4.5: Kubernetes NetworkPolicy - Deny All Traffic

Figure 4.29 shows the command that lists all network policies applied in the default namespace, and the output confirms the presence of the deny-all policy.

This means that no incoming or outgoing traffic is allowed for pods with the default namespace, as this policy isolates network communication.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl get networkpolicies -n default
NAME       POD-SELECTOR   AGE
deny-all   <none>         106s
```

Figure 4.29: Policy deny-all applied

In the next step, a network policy named allow-ingress defines rules for pods receiving traffic from others. It then allows for a reduction in the attack surface and an intended communication flow, showing that the traffic is well-defined and controlled and preventing unauthorized pods from accessing sensitive services. Listing 4.6 shows the new policy allowing pods with the role frontend to receive traffic from only pods labeled with the role backend.

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-ingress
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: frontend
  ingress:
    - from:
        - podSelector:
            matchLabels:
              role: backend
```

Listing 4.6: Kubernetes NetworkPolicy - Allow Ingress from Backend to Frontend

Two pods were labeled to reflect both roles to test the applied configuration. secret-pod is the role frontend, and test-privileged is the role backend, as shown in Figure 4.30.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl get pods -n default --show-labels
NAME              READY   STATUS    RESTARTS      AGE     LABELS
secret-pod        1/1     Running   0             41h     role=frontend
test-privileged   1/1     Running   1 (42h ago)   2d14h   role=backend
```

Figure 4.30: Frontend and backend roles labelled to test the previous configuration

To prove this is all working accordingly, figure 4.31 represents that the secret pod can communicate with the test-privileged which is the backend pod because there is an *HyperText Markup Language* (HTML) response.

```
C:\Users\Yuka\Documents\Software Tese\Kubernetes>kubectl exec -it secret-pod -- curl 10.244.0.19:80
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
html { color-scheme: light dark; }
body { width: 35em; margin: 0 auto;
font-family: Tahoma, Verdana, Arial, sans-serif; }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

Figure 4.31: Secret-pod communicating with test-privileged pod

4.2.4 Orchestration Automation

This PoC aims to demonstrate the feasibility of an automated pipeline for CI/CD on Kubernetes using GitHub Actions and Google Kubernetes Engine (GKE). The flow allows every new change to be automatically built, tested, and deployed to the Kubernetes cluster, ensuring efficient, continuous delivery.

Offered by Google Cloud, GKE was chosen due to its role in automating deployment for this PoC for scaling and managing containerized applications. Given that Kubernetes already contains these features, the reason for using GKE is the benefit from the advanced cluster management features that the Google Cloud Platform provides. A cluster represents the basis of GKE and comprises at least one master cluster and multiple worker machines, the nodes. These master machines and nodes run on the Kubernetes cluster orchestration system, ensuring robust performance and security perfect for CI/CD pipelines.

Figure 4.32 shows the Kubernetes cluster running on GKE that was initially set up to support CI/CD integration. This process involves configuring node pools, consisting of a group of VMs instances that can be tailored to different workloads according to the needs as well as the definition of network policies such as RBAC and, crucially, leveraging the built-in features of GKE to provide a solid foundation for automated deployments.

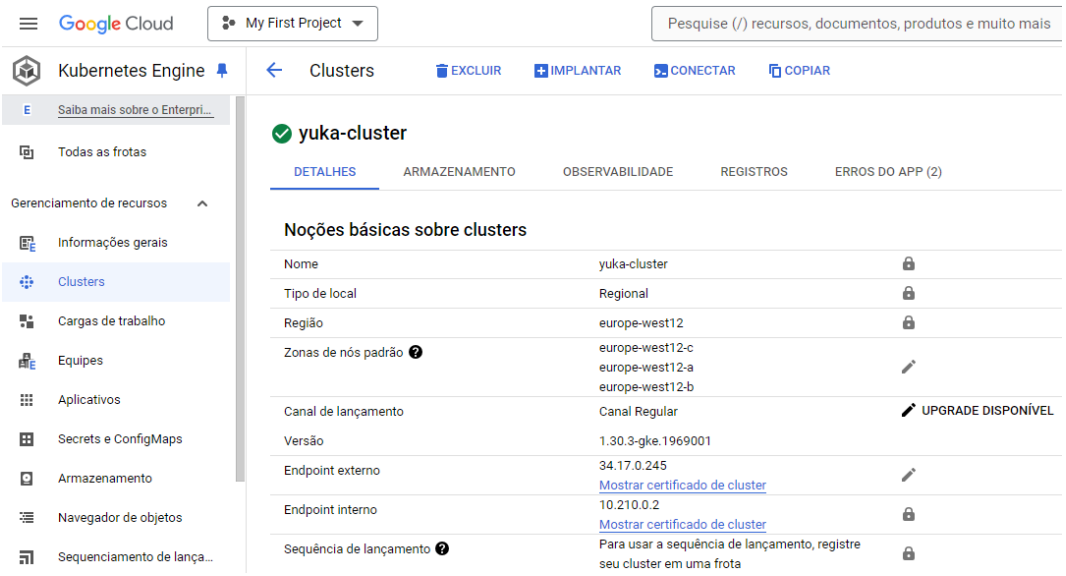

Figure 4.32: Cluster running on GKE

To allow the use of the auto-scaling feature given by GKE, which then will enable workloads to be dynamically balanced according to resource consumption, this cluster called "yuka-cluster" was set up to allow horizontal scaling, also called scaling out, which, instead of upgrading the server, like vertical scaling or scaling in, adds in more machines (nodes or instances) to the infrastructure if needed.

To connect in the local machine to the GKE cluster, running on GKE using the command: gcloud container clusters get-credentials <cluster-name> –region <region-name>2 –project <project-id>

Both deployment and service configuration YAML files used locally in Kubernetes were leveraged to enable automation in Github actions. Responsible for defining how the applications are deployed and managed within the Kubernetes cluster is the deployment.yaml in listing 4.7, specifying the number of pods running at all times and restarting or replacing it in case any pod crashes or fails to avoid downtime.

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: my-app
5    labels:
6      app: my-app
7  spec:
8    replicas: 2
9    selector:
10     matchLabels:
11       app: my-app
12   template:
```

```
13      metadata:
14        labels:
15          app: my-app
16      spec:
17        containers:
18        - name: nginx
19          image: yukamouro25/nginx:latest
20          ports:
21          - containerPort: 80
22          env:
23          - name: DB_PASSWORD
24            valueFrom:
25              secretKeyRef:
26                name: db-secret    # Name of the secret
27                key: db-password  # The key from the secret.yaml
```

Listing 4.7: Deployment yaml

To expose the deployed application to the network, the service.yaml in listing 4.8 specifies a LoadBalancer service; this way, the Nginx application is available to external users via public IP, and in contrast to the pods, ephemerality services provide a stable endpoint.

```
1       apiVersion: v1
2       kind: Service
3       metadata:
4         name: my-app-service
5       spec:
6         selector:
7           app: my-app
8         ports:
9           - protocol: TCP
10            port: 80
11            targetPort: 80
12        type: LoadBalancer
```

Listing 4.8: Service yaml

Through GitHub Actions, after applying both the YAML files in the CI/CD pipeline, the result shown in figure 4.33 says "unchanged" for both because Kubernetes shows the current state of the resources, so as there are no changes since they already were applied previously, it confirms that the existing configurations in the cluster was maintained consistently.

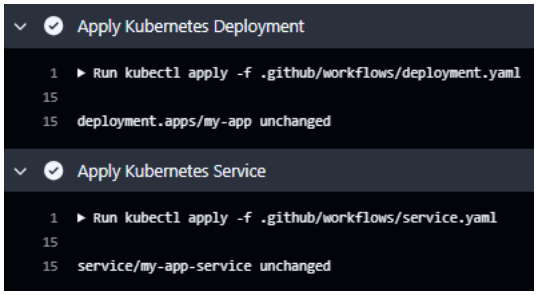

Figure 4.33: Applying deployment.yaml and service.yaml on GitHub Actions

The output available in figure 4.34 shows the result of the configuration present in service.yaml, in which it is possible to see that the service was assigned to an external IP (34.17.68.240).

```
PS C:\Users\Yuka\Documents\Software Tese\Kubernetes\Github actions> kubectl get services
NAME              TYPE           CLUSTER-IP      EXTERNAL-IP    PORT(S)        AGE
kubernetes        ClusterIP      34.118.224.1    <none>         443/TCP        175m
my-app-service    LoadBalancer   34.118.228.30   34.17.68.240   80:32260/TCP   3m15s
```

Figure 4.34: Extenal IP

The IP presented in the last figure is the one used to access the nginx Welcome page presented in figure 4.35.

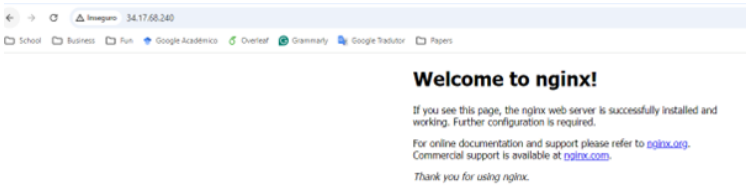

Figure 4.35: Welcome nginx page

To securely connect Github actions to the Kubernetes cluster, a private key necessary for authentication and access to the service account was generated, with an expiration date ending in the year 9999 to allow long-term access unless manually revoked.

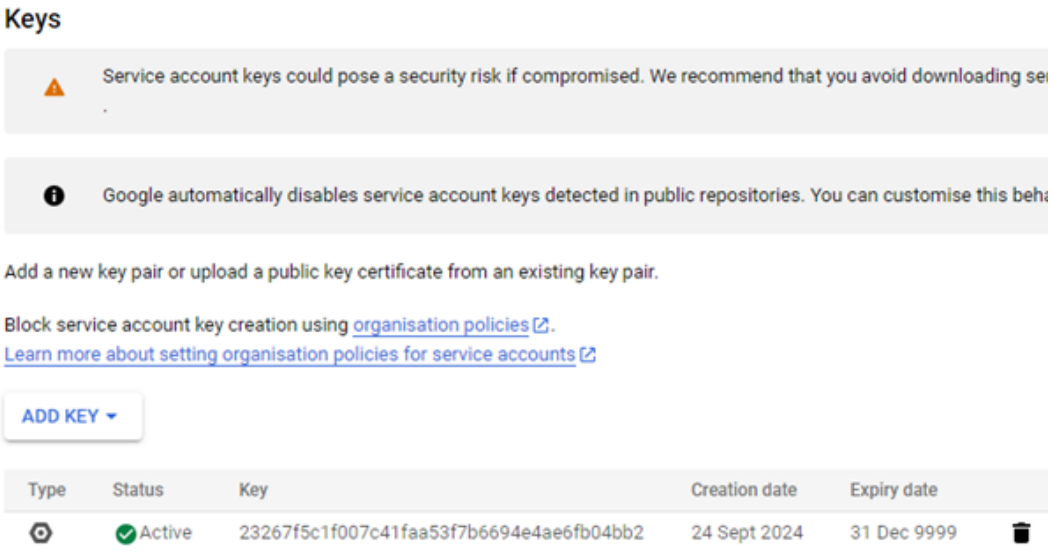

Figure 4.36: Key

The process within GitHub actions follows a similar logic to that carried out locally. Both configuration files and secrets are applied manually. The flow shown in image 3.42 regarding GitHub actions is now automated, starting with the configuration, which includes authentication on Google Cloud and ends with the application of secrets to the Kubernetes cluster. Image 3.43 then confirms that the secret was applied correctly, indicated by the command "kubectl get secrets" as "db-secret" is an item in the list, thus ensuring the management of sensitive information.

Figure 4.37: GitHub actions flow

```
PS C:\Users\Yuka\Documents\Software Tese\Kubernetes\Github actions> kubectl get secrets
NAME        TYPE     DATA   AGE
db-secret   Opaque   1      7m
```

Figure 4.38: Secret applied

To repeat the initial local process now in an automated way on GitHub actions, the policy denies communication between all pods in the cluster. After applying this configuration, which blocks communication between the tested pods locally and within the GKE environment, figure 4.39 proves that it is used correctly, as it is impossible to ping between the pods, resulting in the message "Operation not permitted."

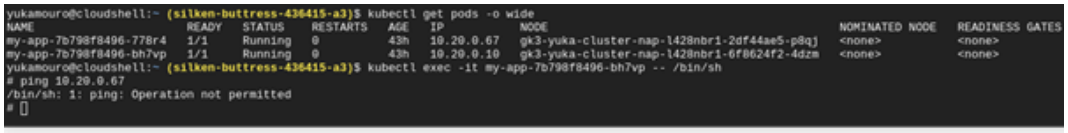

Figure 4.39: Operation not permitted

```yaml
name: Apply Kubernetes Secrets and Deploy

on:
  push:
    branches:
      - main

jobs:
  apply-secrets:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout the code
      - name: Checkout code
        uses: actions/checkout@v2

      # Step 2: Authenticate to Google Cloud
      - name: Authenticate to Google Cloud
        uses: google-github-actions/auth@v1
        with:
          credentials_json: ${{ secrets.GCP_SA_KEY }}  # Use the
              stored service account key JSON

      # Step 3: Set up Google Cloud SDK
      - name: Set up Google Cloud SDK
        uses: google-github-actions/setup-gcloud@v1
        with:
          project_id: ${{ secrets.GCP_PROJECT }}

      # Step 4: Install GKE gcloud auth plugin
      - name: Install GKE gcloud auth plugin
        run: |
          gcloud components install gke-gcloud-auth-plugin
```

```
33
34        # Step 5: Check active account (to verify authentication)
35        - name: Check active account
36          run: gcloud auth list
37
38        # Step 6: Set up kubectl
39        - name: Set up kubectl
40          uses: azure/setup-kubectl@v3
41          with:
42            version: 'latest'
43
44        # Step 7: Authenticate kubectl with GKE
45        - name: Authenticate kubectl with GKE
46          run: |
47            gcloud container clusters get-credentials yuka-cluster
                  --region europe-west12 --project ${{ secrets.
                  GCP_PROJECT }}
48
49        # Step 8: Apply Kubernetes Secret
50        - name: Apply Kubernetes Secret
51          run: |
52            kubectl apply -f .github/workflows/secret.yaml
53
54        # Step 9: Apply Kubernetes Deployment
55        - name: Apply Kubernetes Deployment
56          run: |
57            kubectl apply -f .github/workflows/deployment.yaml
58
59        # Step 10: Apply Kubernetes Service
60        - name: Apply Kubernetes Service
61          run: |
62            kubectl apply -f .github/workflows/service.yaml
63
64        # Step 11: Apply Kubernetes Network Policy
65        - name: Apply Kubernetes Network Policy
66          run: |
67            kubectl apply -f .github/workflows/network-policy.yaml
```

Listing 4.9: Kubernetes Code on Github actions

4.3 Secure Communication

This PoC is dedicated to integrating secure communication with a service mesh architecture through the Istio platform divided into three parts: mutual authentication, traffic management, and authorization policies. All these parts will initially be tested and validated the configurations locally to make it easier to identify problems

as it is a controlled environment. Once everything is working as expected, it will be deployed in the cloud.

4.3.1 Mutual Authentication

Istio uses sidecar proxies (Envoy) to manage communication between services. For the automatic injection of these proxies into Kubernetes pods, the Istio injection label must be applied to the namespace, as seen in figure 4.40. This way, it is guaranteed that any pods deployed with the default namespace can automatically have an action sidecar injected.

```
PS C:\Users\Yuka> kubectl label namespace default istio-injection=enabled
namespace/default labeled
```

Figure 4.40: Istio injection enabled

Figure 4.41 verifies the configuration of mutual TLS, and a sample application named httpbin was deployed. Additionally, a sleep pod was deployed as a test client due to its simplicity of permanent execution without leaving automatically, allowing essential commands to test rapid validation.

```
PS C:\Users\Yuka> kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.13/samples/httpbin/httpbin.ya
ml
serviceaccount/httpbin created
service/httpbin created
deployment.apps/httpbin created
PS C:\Users\Yuka> kubectl apply -f https://raw.githubusercontent.com/istio/istio/release-1.13/samples/sleep/sleep.yaml
serviceaccount/sleep created
service/sleep created
deployment.apps/sleep created
```

Figure 4.41: Sample application httpbin deployed and sleep pod as test client

To reinforce the mutual TLS between services in the mesh, the PeerAuthentication policy was applied, which guarantees that all communications between services are encrypted. Creating an mtls-policy YAML activates the mutual TLS in 'STRICT' mode, thus ensuring that any communication is authenticated and encrypted. The figure 4.42 then shows this policy being applied.

```
PS C:\Users\Yuka\Documents\Software Tese\Istio> kubectl apply -f mtls-policy.yaml
peerauthentication.security.istio.io/default created
```

Figure 4.42: PeerAuthentication policy applied

To verify that the mutual TLS is indeed working as intended, a test call was made from the sleep pod to the httpbin service. In figure 4.43, the return of HTTP 200 status confirms that communication between the two services occurred successfully, thus validating that Istio is guaranteeing mutual TLS between the services.

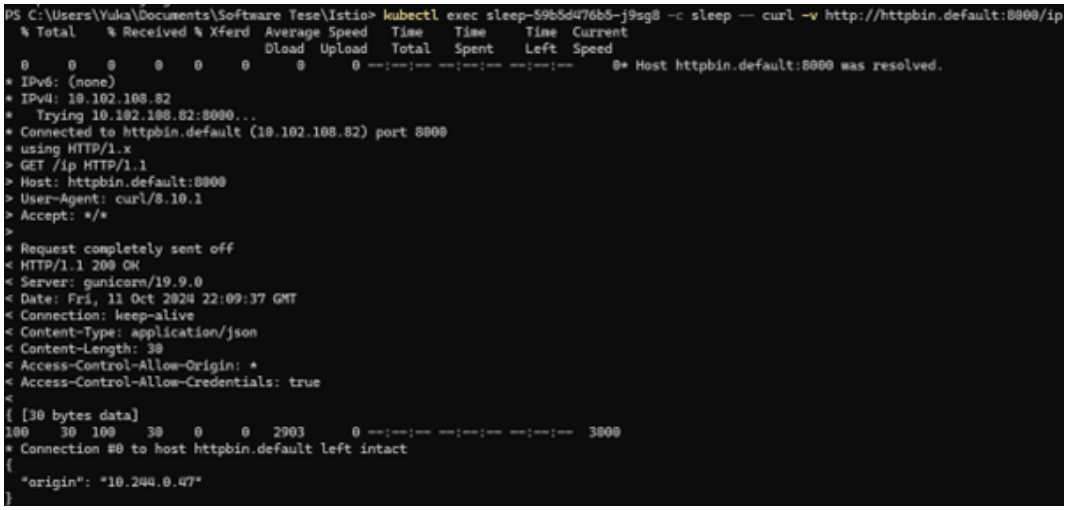

Figure 4.43: Request completely sent off

The DestinationRule in Istio is a configuration that defines traffic policies between specific services within the service mesh. This allows the activation or deactivation of the encryption and authentication of the mutual TLS with the configuration of ISTIO MUTUAL. We can then prevent attacks such as MITM. Figure 4.44 output confirmed that it applies to all services in the default namespace. of the encryption and authentication of the mutual TLS with the configuration of ISTIO MUTUAL.

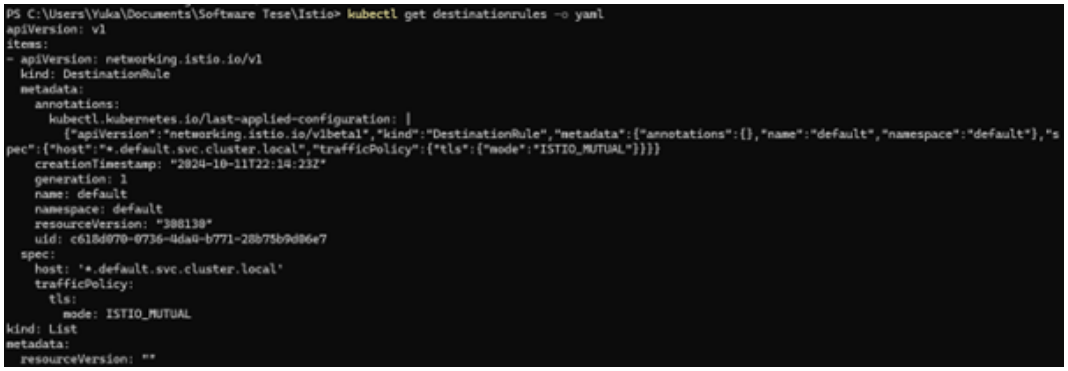

Figure 4.44: DestinationRule applied

4.3.2 Traffic Management

From a security point of view, traffic management aims to ensure that only necessary and authorized communications occur within the cluster. Thus, through network policies that define explicit rules for traffic, it is possible to obtain granular access control, preventing internal attacks, segmenting, and isolating the environment so that malicious traffic in one section does not affect another.

The first step in implementing this portion of the PoC was to create an Ingress Gateway, figure 4.45 that allows the internal services of the service mesh to be exposed to external traffic.

Figure 4.45: Ingress gateway created

The listing 4.10 shows the gateway-htttpbin YAMLthat creates a gateway and a virtual service for the httpbin service.

```yaml
apiVersion: networking.istio.io/v1
kind: Gateway
metadata:
  name: httpbin-gateway
spec:
  selector:
    istio: ingressgateway
  servers:
    - port:
        number: 80
        name: http
        protocol: HTTP
      hosts:
        - "*"
---
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
    - "*"
  gateways:
    - httpbin-gateway
  http:
    - match:
        - uri:
            exact: /status
      route:
        - destination:
            host: httpbin
            port:
              number: 8000
```

Listing 4.10: gateway-httpbin-yaml

For this ingress gateway to work on Minikube, a tunnel must be created to receive external traffic. In Figure 4.46, this terminal must remain open where the command was executed for the tunnel to be active and the gateway to function correctly.

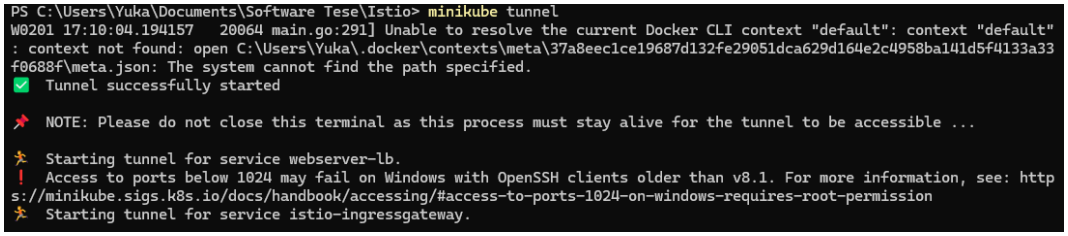

Figure 4.46: Minikube tunnel

As the name suggests, traffic splitting divides the traffic, for this practical example, between two versions (v1) and (v2) of the httpbin service. This technique causes part of the traffic to be directed to the new version (v2) while another part remains on (v1). This approach is helpful for canary releases, in which software developers deploy a new feature or version to a small portion of users, thus increasing the chance of impacting fewer users if it is compromised.

To do so, a new file YAML is created with the name traffic-splitting, which defines, as shown in 4.11, a destination rule and a virtual service. The Destination Rule specifies two subsets (v1) and (v2), while the virtual service defines the percentage of traffic division for each; indicate in just one, in this case, 50, since the remainder will be directed to the other version.

```yaml
apiVersion: networking.istio.io/v1
kind: DestinationRule
metadata:
  name: httpbin
spec:
  host: httpbin
  subsets:
    - name: v1
      labels:
        version: v1
    - name: v2
      labels:
        version: v2
---
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: httpbin
spec:
  hosts:
    - "*"
```

```
  gateways:
    - httpbin-gateway
  http:
    - route:
        - destination:
            host: httpbin
            subset: v1
          weight: 50
        - destination:
            host: httpbin
            subset: v2
          weight: 50
```

Listing 4.11: Traffic splitting YAML

Figure 4.47 shows this traffic splitting rule YAML being applied with both the destination rule created and the virtual service configured.

```
PS C:\Users\Yuka\Documents\Software Tese\Istio> kubectl apply -f traffic-splitting.yaml
destinationrule.networking.istio.io/httpbin created
virtualservice.networking.istio.io/httpbin configured
```

Figure 4.47: Traffic splitting rule applied

4.3.3 Authorization Policies

Controlling access to specific services within the service mesh more granularly than traffic splitting based on user or service identity criteria, thus promoting security and compliance. For this, an authorization policy can be used to deny and access a specific service based on origin criteria. In the case of the httpbin service, a policy was created as seen in 4.12 in which access is exclusive to the sleep service in the default namespace.

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: httpbin-policy
  namespace: default
spec:
  action: ALLOW
  rules:
    - from:
        - source:
            principals: ["cluster.local/ns/default/sa/sleep"]
```

Listing 4.12: Authorization policy YAML

Figure 4.48 then shows the command that creates the authorization policy in the Istio service mesh, configured according to the instructions provided in this YAML file.

```
PS C:\Users\Yuka\Documents\Software Tese\Istio> kubectl apply -f auth-policy.yaml
authorizationpolicy.security.istio.io/httpbin-policy created
```

Figure 4.48: Authorization policy created

To test that this configuration specifies that it uses the service account identity as the criteria for authorization, in this case, only the sleep service account is allowed to access the htttpbin service, ensuring that all other services cannot establish this communication. The following commands were used with the respective results shown in Figure 4.49, where the first shows output that the httpbin service responded with the address "origin": "10.244.0.47", indicating that the sleep pod was able to access the httpbin service, confirming the correct application of the authorization policy and the following command that shows the response of the headers sent back by httpbin including Accept, Host and User-Agent, confirming that the sleep pod was also able to access the headers endpoint.

```
PS C:\Users\Yuka\Documents\Software Tese\Istio> kubectl exec $(kubectl get pod -l app=sleep -o jsonpath='{.items[0].metadata.name}') -c sleep -- cur
l http://httpbin.default:8000/ip
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    30  100    30    0     0   1199      0 --:--:-- --:--:-- --:--:--  1250
{
  "origin": "10.244.0.47"
}
PS C:\Users\Yuka\Documents\Software Tese\Istio> kubectl exec $(kubectl get pod -l app=sleep -o jsonpath='{.items[0].metadata.name}') -c sleep -- cur
l http://httpbin.default:8000/headers
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   114  100   114    0     0   4837      0 --:--:-- --:--:-- --:--:--  4956
{
  "headers": {
    "Accept": "*/*",
    "Host": "httpbin.default:8000",
    "User-Agent": "curl/8.10.1"
  }
}
```

Figure 4.49: Testing authorization policy

The definition of the deny-all authorization policy in a YAML is to block all incoming requests to services within a namespace. It is done by configuring this policy without specifying any rule, leaving the spec empty shown in listing 4.13, so Istio automatically denies all attempts to access services in the namespace where this policy is then applied.

```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: deny-all
  namespace: default
spec: {}
```

Listing 4.13: Deny all YAML

A denial pattern is implemented in Figure 4.50 where all traffic is blocked for security reasons until contrary permissions are imposed, following a ZTA.

```
PS C:\Users\Yuka\Documents\Software Tese\Istio> kubectl apply -f deny-all-policy.yaml
authorizationpolicy.security.istio.io/deny-all created
```

Figure 4.50: Deny all YAML applied

It was also created an YAML file, represented in listing 4.14 named allow-httpbin-sleep that allowed policy for specific traffic, which only authorizes the sleep service to access the httpbin service.

```yaml
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: allow-httpbin-sleep
  namespace: default
spec:
  action: ALLOW
  rules:
    - from:
        - source:
            principals: ["cluster.local/ns/default/sa/sleep"]
```

Listing 4.14: Allow httpbin service YAML

And finally, figure 4.51 shows this being applied.

```
PS C:\Users\Yuka\Documents\Software Tese\Istio> kubectl apply -f allow-httpbin-sleep.yaml
authorizationpolicy.security.istio.io/allow-httpbin-sleep created
```

Figure 4.51: Allow httpbin service applied

4.3.4 Secure Communication Automation

In Kubernetes environments with Istio, integrating CI/CD is essential as it allows configuration updates and security policies to be automated and consistent. With GKE as cluster infrastructure, it provides a robust and efficient pipeline. In figure 4.52, all the configuration files are necessary to implement the workflow.

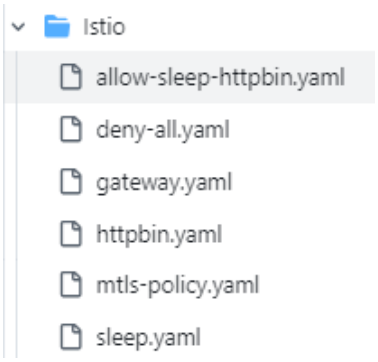

Figure 4.52: Configuration files used

The CI/CD workflow in the previous figure is dedicated to Istio in GKE using GitHub Actions and allows the automatic implementation of security policies and essential services, thus bringing advantages to the service mesh. This process is protected by adopting an agile and secure *Development and Operations* (DevOps) culture since any change in the code is quickly tested and applied.

The successfully executed workflow presented in figure 4.53 has the following main steps: initialization and preparation, job configuration, code checkout, and authentication on Google Cloud. The configuration of the GKE environment, deploying services and policies such as httpbin, sleep, and mutual TLS, thus promoting secure communication as in the steps previously carried out locally. And finally, cleaning and finishing the work.

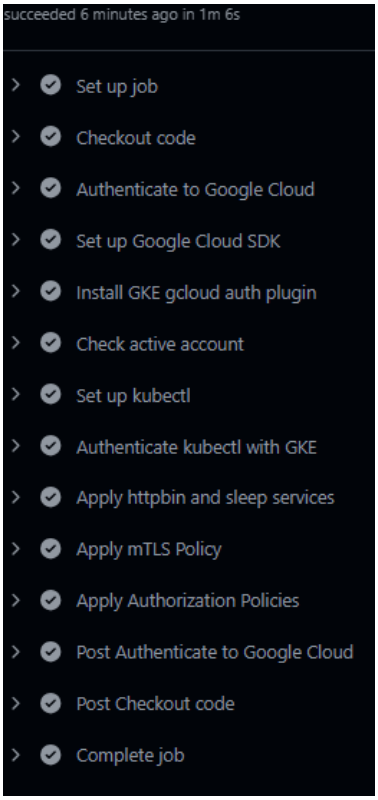

Figure 4.53: Workflow Istio

The PoC demonstrates the implementation of the CI/CD workflow present in list 4.15 to configure Istio in a GKE cluster through GitHub actions where the objective is to ensure the automated implementation of security policies and communication between services. The workflow presented in 4.15 is divided into several steps, which are commented, starting with step 1, which consists of cloning the repository, that is, obtaining the most recent code from the GitHub repository. Step 2 authenticates to Google Cloud using the key stored in GitHub Secrets, allowing the pipeline to have permission to access GKE. Then, in step 3, the GKE configuration is done by installing the *Software Development Kit* (SDK) on Google Cloud, where the GCP project where Kubernetes is located is defined. Step 4 consists of installing the plugin required for authentication in GKE, and step 5 verifies whether this was successful. The kubectl configuration is done in step 6, where the latest version will be installed, which is necessary for managing the Kubernetes cluster. Step 7 gets the credentials for the GKE cluster so that kubctl can interact with it. In step 8, the httpbin and sleep services are implemented and used for communication and security testing within Istio, as was done locally. Step 9 applies the mutual TLS policy, ensuring all communications are encrypted and authenticated. Finally, in the 10th and final step, authorization policies are defined to control which services can communicate with each other within Istio.

```yaml
name: Apply Istio Configuration

on:
  push:
    branches:
      - main

jobs:
  apply-secrets:
    runs-on: ubuntu-latest

    steps:
      # Step 1: Checkout the code
      - name: Checkout code
        uses: actions/checkout@v2

      # Step 2: Authenticate to Google Cloud
      - name: Authenticate to Google Cloud
        uses: google-github-actions/auth@v1
        with:
          credentials_json: ${{ secrets.GCP_SA_KEY }}  # Use the
              stored service account key JSON

      # Step 3: Set up Google Cloud SDK
      - name: Set up Google Cloud SDK
        uses: google-github-actions/setup-gcloud@v1
        with:
          project_id: ${{ secrets.GCP_PROJECT }}

      # Step 4: Install GKE gcloud auth plugin
      - name: Install GKE gcloud auth plugin
        run: |
          gcloud components install gke-gcloud-auth-plugin

      # Step 5: Check active account (to verify authentication)
      - name: Check active account
        run: gcloud auth list

      # Step 6: Set up kubectl
      - name: Set up kubectl
        uses: azure/setup-kubectl@v3
        with:
          version: 'latest'

      # Step 7: Authenticate kubectl with GKE
      - name: Authenticate kubectl with GKE
        run: |
```

```
47            gcloud container clusters get-credentials yuka-cluster
                  --region europe-west12 --project ${{ secrets.
                  GCP_PROJECT }}
48
49        # Step 8: Apply httpbin and sleep services
50        - name: Apply httpbin and sleep services
51          run: |
52            kubectl apply -f .github/workflows/Istio/httpbin.yaml
53            kubectl apply -f .github/workflows/Istio/sleep.yaml
54
55        # Step 9: Apply mTLS Policy
56        - name: Apply mTLS Policy
57          run: |
58            kubectl apply -f .github/workflows/Istio/mtls-policy.
                  yaml
59
60        # Step 10: Apply Authorization Policies
61        - name: Apply Authorization Policies
62          run: |
63            kubectl apply -f .github/workflows/Istio/allow-sleep-
                  httpbin.yaml
```

Listing 4.15: Istio Code on Github actions

4.4 Discussion

The security and privacy framework cloud-native applications was validated through PoCs from the previous section. This section discusses the outcome of the implemented security measures, evaluating their effectiveness and comparing them with existing solutions and good practices.

In the initial phase, two image scanning tools, Trivy and Grype, are incorporated into the framework to report vulnerabilities in the latest version of the nginx image. This image was chosen because it contains several types of severity levels, thus obtaining a report composed of all kinds of results.

In the state-of-the-art chapter, in the section on related work, examples of image-scanning tools were found. However, to have a more coherent discussion, it was decided to use the same image to obtain results from two different tools. As expected, these results were not the same due to several factors, one of which is the database each uses and how they present the results.

Although similar, these do not have the same severity level parameters, so the results may not match exactly. Both use NVD; however, Trivy adopts advisories from OSs and community repositories, while Grype uses advisories from GitHub and the Anchore Vulnerability Database. Another difference is how each analyzes

dependencies, with Trivy being more direct and faster. At the same time, Grype is more detailed in specific packages and also uses SBOM generated by tools like Syft.

Another factor, although unlikely, is that the database update times are different; that is, if a newly added vulnerability is present, it may be detected by one of the tools and not the other. However, to have more consistent results, it is always recommended that both databases be updated.

The table 4.2 summarizes which factors differentiate one image scanning tool from another, helping to reflect why the results presented were different values.

Factor	**Trivy**	**Grype**
Vulnerability Sources	NVD, GitHub Security Advisories, OS vendor advisories, community repositories	NVD, GitHub Advisories, Anchore DB, OS vendor advisories
Detection Method	Analyzes OS packages and application dependencies	Analyzes OS packages and can use SBOMs
Update Frequency	Regularly updates, but may vary	May update at different times
Default Configuration	Scans for vulnerabilities, misconfigurations, and secrets	Focuses only on vulnerabilities
Filters and Parameters	Allows filtering by severity and type of vulnerability	Allows filtering by available patches and severity

Table 4.2: Comparison between Trivy and Grype

Although different results were obtained, both tools were able to fulfil their purpose of scanning for vulnerabilities so that they can be corrected before the images are deployed, thus complying with the security requirement, Vulnerability Scanning, detailed in the framework specification chapter.

Since Grype uses more sources, it can easily identify a larger number of vulnerabilities, but this can lead to over-reporting, false positives, when a vulnerability is referenced but not applicable or even the existence of duplicate records.

To have a greater balance in terms of accuracy and efficiency in vulnerability scanning, best practices suggest combining multiple tools and customizing their output to reduce the number of false positives while ensuring good security coverage.

Ensuring security in cloud-native applications requires a proactive approach to vulnerability management. Thus automating scans in CI/CD pipelines provides an efficient mechanism to identify security risks at the early stage of the software development cycle.

In the future, implementing continuous monitoring would be a crucial strategy for enhancing the framework's resilience. By detecting vulnerabilities in real-time, proactive measures can be taken more swiftly, significantly reducing the attack surface and strengthening overall security.

When it comes to PoC focused on content trust, this is a fundamental security mechanism as it ensures both the integrity and authenticity of containerized images. This provides a layer of verification that helps prevent the deployment of images that could later compromise security through the use of malicious images. DCT was then used to ensure that only verified images were signed so they could be pulled or executed, thus increasing the security of the cloud-native environment.

To implement this PoC it was necessary to activate DCT through an environment variable to ensure that only signed images can be executed. Followed by the key signing process, the root key and the repository key and associating them with the repository.

The benefits of this, such as image integrity and authenticity, lead to attack mitigation, preventing the execution of compromised images, and in turn reduces the likelihood of attacks based on manipulated images, since these must adhere to the required good security practices.

One of the challenges is centred on the complexity of key management since their storage and maintenance require rigorous practices. Furthermore, when this is implemented in the CI/CD pipeline, there is no guarantee that the images can be changed after they have been signed.

Possible improvements would be the integration of security at runtime, thus linking to the next PoC and also the implementation of additional validations in the pipeline to avoid post-signature modifications.

Unlike the previous PoCs that focus on static security, this one is focused on runtime security because there is a need for continuous security against attacks after the containers are deployed. Thus, this PoC to decrease the attack surface makes use of a default seccomp profile that is applied to restrict dangerous system calls. Thus complying with ZTA, since it prevents the execution of commands that could lead to an elevation of privileges or the improper modification of permissions. For example, it was chosen to block the chmod call since it prevents changing the permissions on files. In the end, an attempt is made to make this call, which is immediately blocked, followed by the configured error message "operation not permitted" to inform the user that he cannot invoke this call.

In related work, it was selected a paper that talked about a new approach instead of the one that was applied, in which only one call or several are blocked individually by removing them from the profile. This suggests a study of the call sequences that attackers typically try before exploiting a system. However, it was concluded that it

was better to opt for the traditional approach as this could erroneously block calls that were not intended to corrupt the system but a mere coincidence.

Moving now to security and privacy in container orchestration, the security admission PoC has come to reinforce security through the implementation of admission policies preventing the execution of insecure workloads. These policies are made at the cluster level, requiring pods to meet the minimum requirements before being created.

To illustrate, PSA was applied to namespaces with different security levels: privileged, baseline and restricted. Pods were created and tested in different namespaces to verify that policies were correctly applied, a privileged pod was purposefully deployed in a restricted namespace, resulting in an error due to PSA being applied.

Blocking the execution of pods that do not meet the minimum security requirements leads to a reduction in the attack surface since it is no longer possible to deploy pods with excessive permissions or vulnerable configurations.

The restrictions imposed despite the centralized application of the rules, there is a lower probability of manual errors and adequate configurations, as this offers ease in security management. But it can also lead to the possibility of preventing the execution of legitimate applications, leading to unnecessary blocks.

The implementation of PSA leads to greater control and security in the Kubernetes environment, although there are future improvements such as the integration of tools such as Gatekeeper that allows the definition of more flexible policies and also an implementation of continuous auditing to verify whether these rules are being respected in the long term.

In the previous PoC on content trust, one of the challenges was the management of the root and repository keys, and the importance of their security to prevent information leaks. Thus, Kubernetes comes with Secret Management built-in, which allows the storage of data at rest in a secure manner.

Through Secret Management PoC it is possible to mitigate the risk of incorrect storage of credentials, which can lead to the exposure of passwords in source code or environment variables. Starting with the creation of a secret object that allows storing sensitive information and avoiding its direct exposure, the values were encoded in base 64, ensuring that they do not remain in plain text in the configuration files. The secrets were created using a YAML, where each credential has a name and a specific namespace. After its definition, it is applied to the Kubernetes cluster to guarantee the security of the use of this secret; the credentials are injected directly into the pods instead of being stored in the source code. Furthermore, to prevent unauthorized access, service accounts were created with specific permissions, ensuring only that pods with appropriate permissions could access these secrets. This ensures more refined access that prevents unauthorized applications from exposing or using sensitive credentials.

In related work, GKE presented several advantages over other solutions such as AKS and EKS. Among these advantages, this was the first managed service in Kubernetes, uses Cloud KMS to encrypt secrets, allowing users to manage keys centrally and with greater control, implements multiple layers of security, combining hardware and software to protect stored secrets, easy integration with CI/CD pipelines, facilitating automatic rotation of secrets and ensuring that credentials are managed in a secure and auditable manner.

In addition to secrets management, a PoC was also created dedicated to defining network policies in Kubernetes with the role of promoting security in communication between pods. Through these it was possible to limit network traffic, consequently leading to a reduction in the attack surface and preventing lateral movement within the cluster.

The related work compared the performance of two popular tools, Calico and Cilium, both of which allow the implementation and enforcement of Network Policies within Kubernetes clusters, ensuring traffic isolation between pods and services. Additionally, both make use of eBPF to improve security performance in packet processing, reducing dependence on iptables rules and optimizing traffic filtering.

However, it was concluded that if the focus is on performance in communication between nodes, Calico is the best choice, as it avoids tunnelling overhead and offers a more efficient routing model. So, for this PoC the calico tool was chosen to apply traffic control rules, such as the deny-all policy so that no unwanted traffic would be allowed by default. After applying this restrictive policy, a policy was added that only allowed traffic between specific pods in which only pods with the backend role could communicate with the frontend pods, reinforcing the applied policy.

It was possible to apply PoLP when limiting traffic since it ensures that each pod only has the access necessary for its operation. There was an improvement in security in environments that share the same infrastructure and also a notable reduction in the attack surface as it prevents attackers from compromising a pod and being able to move laterally through the cluster.

This PoC starts by activating secure traffic control, for this, the automatic injection of the envoy proxy was configured in all pods with the default namespace, ensuring that all will automatically have a proxy sidecar, which allows the application of security policies. To reinforce this security, a PeerAuthentication policy was created that forces the use of mutual TLS, so that all communications are mandatory authenticated and encrypted, blocking external services from intercepting the data. To validate this configuration, two pods were created and the HTTP 200 response confirmed that mutual authentication was working. In addition, a DestinationRule was configured to define how secure traffic would be handled, in this case, which would reinforce that all connections between services in the cluster must use mutual TLS.

The use of mutual TLS prevents attackers from intercepting and modifying traffic between services, promoting strong authentication since all services within the service mesh must prove their identity, reducing the risk of malicious connections occurring. The transmitted data is encrypted, thus ensuring privacy. Due to the centralized configuration via Istio, it becomes easily deployable and scalable without the need to change the service code.

After ensuring that communication between services is done securely and authenticated with mutual TLS, it was necessary to complement it with another Istio component, authorization policies, since these determine which identities are allowed to access certain services. The combination of both is crucial since an attacker could obtain a valid certificate for the mesh and still access a service improperly, exploiting the open permissions.

Although Kubernetes offers basic traffic routing features, it does not have enough granularity to have more advanced control of traffic within the cluster. For this purpose, PoC was implemented to make a traffic division. This aims to control the entry and exit of cluster traffic, directing it intelligently and allowing performance and security optimization through load balancing. To this end, an ingress gateway was created that allows traffic entering the cluster to be routed as defined in the virtual service rules, in this case, 50One of the benefits of using traffic splitting is that if one of the services fails, it can automatically direct traffic to more stable versions, reducing downtime and failover. Thus, it is confirmed as analyzed in one of the related works that the implementation of techniques such as traffic division is essential to promote high availability in cloud-native environments.

Unlike network policies that control who can connect to whom, authorization policies control what each service can do after establishing the connection. These can be applied at multiple levels: mesh-wide, that is, globally to the mesh, namespace or even workload.

For the PoC regarding access control, an allow policy was applied, which allows only the sleep service to communicate with the httpbin service within the default namespace, thus prohibiting other unauthorized access. The deny-all policy was used to completely block, preventing all access from being allowed by default. Thus this model follows a ZTA, where no communication should be allowed without explicit rules.

The highlighted benefits are the reduction of the attack surface, greater control over identities, and even more granular security due to the different levels at which it can be applied. However, one suggestion for improvement was integration with monitoring tools such as Prometheus or Grafana that can track authorized and blocked requests, to see possible attack attempts.

Each of the applied solutions played a crucial role in building a robust framework, aligned with the best security, privacy and resilience practices for cloud-native

environments. The PoCs carried out were fundamental to validate the effectiveness of the adopted strategies, demonstrating in practice how each technique contributes to mitigating risks and strengthening the security of the environment. Furthermore, comparison with related studies reinforced that the choices made for each technique were not arbitrary, but rather based on technical analyses and practical evidence.

Security in cloud-native environments cannot rely on a single isolated technique, and this study has shown that the real value lies in the strategic combination of multiple approaches. The resulting framework represents a significant advance in protection in cloud-native environments by providing privacy, resilience, and compliance with security standards.

4.5 Summary

This chapter described the validation work of the proposed security and privacy framework through the use of a PoC to cloud-native environments, following the requirements and architecture outlined, Moreover, its results were discussed and compared to the state-of-the-art.

The next chapter will present the conclusion that summarizes the objectives achieved and explores future improvements to ensure the framework remains adaptive and resilient.

Chapter 5

Conclusion and Future Work

This work aimed to contribute to improving the security of Cloud Native applications and the construction of safer, private, and scalable cloud environments. To achieve this, a security and privacy framework for cloud-native platforms was specified, departing from a review of the state-of-the-art, identifying the key challenges and best practices in securing cloud-native applications as requirements.

A PoC was used to demonstrate how to follow a set of good practices from the proposed framework to improve the security of cloud-native applications. The proposed framework adopts security practices along cloud-native lifetime, from container security and orchestration to observability, showing that these layers working together can reduce vulnerabilities and enforce compliance. Adopting security in CI/CD alongside *Development, Security, and Operations* (DevSecOps) practices contribute to reducing misconfiguration risks and vulnerabilities in production environments. The results demonstrated how those security measures increase the protection of cloud-native applications. The integration of scanning tools helped identify and mitigate container image security issues even before the deployment stage. Furthermore, enabling DCT provided authenticity. Using security policies in Kubernetes, such as PSA, the management of secrets, policies, and networks contributes to improving the protection of clusters. Finally, observability was achieved by adopting a service mesh to enhance secure communication between microservices through mutual TLS and authorization policies.

In the future, this work can follow different paths. It can be extended by considering cloud-native secret management tools, continuous monitoring components

to detect threats, and evaluating the impact on performance. The results denote limitations, such as increased computational cost in implementing real-time threat detection systems in high-demand scenarios where resource optimization is key. To reduce the likelihood of errors in human activities while ensuring high availability and security, it is suggested that cloud-native secret management tools be adopted to contribute to streamlining encryption processes. The complexity of key management since encryption and access to these credentials in a dynamic environment such as cloud-native adds overhead that requires careful access control. *Artificial Intelligence* (AI) concepts can also help improve policy security by dynamically adjusting network permissions based on real-time threat and anomalous detection. Moreover, to reduce the risk of attacks or unauthorized activities, adopting a ZTA mindset is suggested by integrating runtime security monitoring components for threat detection and incident response. Tools can be used to analyze system calls and improve visibility. Finally, the impact on performance (e.g., latency and availability) on monitoring and enforcing security policies comes at a cost because the requirements of computational resources are aspects that should be considered in the future.

References

[Adinegoro et al., 2022] Adinegoro, F., Rahmania, C., Zaini, I. N., Negara, R. M., and Hertiana, S. N. (2022). Latency and ram usage comparison of advanced and lightweight service mesh. In *2022 5th International Seminar on Research of Information Technology and Intelligent Systems (ISRITI)*, pages 369–372. [Cited on page 25]

[Ahamed et al., 2021] Ahamed, W. S. S., Zavarsky, P., and Swar, B. (2021). Security audit of docker container images in cloud architecture. In *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*, pages 202–207. IEEE. [Cited on page 28]

[Al Jawarneh et al., 2019] Al Jawarneh, I. M., Bellavista, P., Bosi, F., Foschini, L., Martuscelli, G., Montanari, R., and Palopoli, A. (2019). Container orchestration engines: A thorough functional and performance comparison. In *ICC 2019-2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE. [Cited on page 4]

[Ali and Meghanathan, 2011] Ali, I. and Meghanathan, N. (2011). Virtual machines and networks-installation, performance study, advantages and virtualization options. *arXiv preprint arXiv:1105.0061*. [Cited on page 2]

[Ali, 2023] Ali, S. A. (2023). Securing cloud-native applications: Addressing security challenges in containerization and microservices architectures. *International Journal of Machine Intelligence for Smart Applications*, 13(10):1–15. [Cited on page 19]

[Alshuqayran et al., 2016] Alshuqayran, N., Ali, N., and Evans, R. (2016). A systematic mapping study in microservice architecture. In *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*, pages 44–51. IEEE. [Cited on pages 2 and 33]

[Ambala, 2024] Ambala, A. (2024). Exploring the dynamics of software bill of materials (sboms) and security integration in open source projects. [Cited on page 15]

[Andersson and Hysing Berg, 2022] Andersson, M. and Hysing Berg, R. (2022). Docker container images: Concerns about available container image scanning tools and image security. [Cited on page 13]

[Arango et al., 2017] Arango, C., Dernat, R., and Sanabria, J. (2017). Performance evaluation of container-based virtualization for high performance computing environments. *arXiv preprint arXiv:1709.10140*. [Cited on page 12]

[Assal and Chiasson, 2018] Assal, H. and Chiasson, S. (2018). Security in the software development lifecycle. In *Fourteenth Symposium on Usable Privacy and Security (SOUPS 2018)*, pages 281–296, Baltimore, MD. USENIX Association. [Cited on page 35]

[Babu et al., 2014] Babu, S. A., Hareesh, M. J., Martin, J. P., Cherian, S., and Sastri, Y. (2014). System performance evaluation of para virtualization, container virtualization, and full virtualization using xen, openvz, and xenserver. In *2014 Fourth International Conference on Advances in Computing and Communications*, pages 247–250. [Cited on page 12]

[Balla et al., 2020] Balla, D., Simon, C., and Maliosz, M. (2020). Adaptive scaling of kubernetes pods. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–5. [Cited on page 21]

[Benzel et al., 2005] Benzel, T. V., Irvine, C. E., Levin, T. E., Nguyen, T. D., Clark, P. C., and Bhaskare, G. (2005). Design principles for security. Technical report, Monterey, California. Naval Postgraduate School. [Cited on page 32]

[Bhatia, 2024] Bhatia, P. (2024). Automated vulnerability scanning with trivy. [Cited on page 14]

[Boettiger, 2015] Boettiger, C. (2015). An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1):71–79. [Cited on page 12]

[Boles et al., 2024] Boles, B., O'Donoghue, E., Muneza, A. R. M., Perkins, G., Izurieta, C., and Reinhold, A. M. (2024). Deciphering discrepancies: A comparative analysis of docker image security. In *24th IEEE International Conference on Source Code Analysis and Manipulation*. [Cited on page 13]

[Budigiri, 2023] Budigiri, G. (2023). Secure and scalable policy management in cloud native networking. In *Proceedings of the 24th International Middleware Conference: Demos, Posters and Doctoral Symposium*, pages 11–14. [Cited on page 23]

[Budigiri et al., 2021] Budigiri, G., Baumann, C., Mühlberg, J. T., Truyen, E., and Joosen, W. (2021). Network policies in kubernetes: Performance evaluation and

security analysis. In *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, pages 407–412. IEEE. [Cited on pages 28, 29, and 33]

[Calcote and Butcher, 2019] Calcote, L. and Butcher, Z. (2019). *Istio: Up and running: Using a service mesh to connect, secure, control, and observe*. O'Reilly Media. [Cited on pages 24, 25, and 33]

[Cavoukian et al., 2021] Cavoukian, A. et al. (2021). Privacy by design: The seven foundational principles. *IAPP Resource Center*. [Cited on pages 33 and 34]

[Cha and Kim, 2021] Cha, D. and Kim, Y. (2021). Service mesh based distributed tracing system. In *2021 International Conference on Information and Communication Technology Convergence (ICTC)*, pages 1464–1466. [Cited on page 4]

[Chandramouli et al., 2020] Chandramouli, R., Butcher, Z., et al. (2020). Building secure microservices-based applications using service-mesh architecture. *NIST Special Publication*, 800:204A. [Cited on pages 28, 30, and 33]

[Chaudhuri, 2016] Chaudhuri, A. (2016). Internet of things data protection and privacy in the era of the general data protection regulation. *Journal of Data Protection & Privacy*, 1(1):64–75. [Cited on page 5]

[Chen, 2018] Chen, L. (2018). Microservices: architecting for continuous delivery and devops. In *2018 IEEE International conference on software architecture (ICSA)*, pages 39–397. IEEE. [Cited on page 18]

[Chintale and Kethireddy, 2024] Chintale, P. and Kethireddy, R. R. (2024). Shift-left security integration: Automating vulnerability detection in container images. [Cited on page 13]

[Combe et al., 2016] Combe, T., Martin, A., and Di Pietro, R. (2016). To docker or not to docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62. [Cited on page 16]

[Cultrera, 2022] Cultrera, F. M. (2022). A performance analysis of mesh models for cloud-based workflows. [Cited on page 25]

[da Silva et al., 2018] da Silva, V. G., Kirikova, M., and Alksnis, G. (2018). Containers for virtualization: An overview. *Applied Computer Systems*, 23(1):21–27. [Cited on page 12]

[Dahlmanns et al., 2023] Dahlmanns, M., Sander, C., Decker, R., and Wehrle, K. (2023). Secrets revealed in container images: an internet-wide study on occurrence and impact. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 797–811. [Cited on page 15]

[Daniels, 2009] Daniels, J. (2009). Server virtualization architecture and implementation. *XRDS: Crossroads, The ACM Magazine for Students*, 16(1):8–12. [Cited on page 2]

[De Benedictis and Lioy, 2019] De Benedictis, M. and Lioy, A. (2019). Integrity verification of docker containers for a lightweight cloud environment. *Future generation computer systems*, 97:236–246. [Cited on page 16]

[De Sousa et al., 2019] De Sousa, N. F. S., Perez, D. A. L., Rosa, R. V., Santos, M. A., and Rothenberg, C. E. (2019). Network service orchestration: A survey. *Computer Communications*, 142:69–94. [Cited on page 4]

[Dhinakaran et al., 2024] Dhinakaran, D., Sankar, S., Selvaraj, D., and Raja, S. E. (2024). Privacy-preserving data in iot-based cloud systems: A comprehensive survey with ai integration. *arXiv preprint arXiv:2401.00794*. [Cited on pages 19 and 20]

[Dikhanbayeva et al., 2020] Dikhanbayeva, D., Shaikholla, S., Suleiman, Z., and Turkyilmaz, A. (2020). Assessment of industry 4.0 maturity models by design principles. *Sustainability*, 12(23):9927. [Cited on page 6]

[Disterer, 2013] Disterer, G. (2013). Iso/iec 27000, 27001 and 27002 for information security management. *Journal of Information Security*, 4(2). [Cited on page 27]

[Dragoni et al., 2017] Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216. [Cited on page 18]

[Elamin and Kierkels, 2021] Elamin, M. and Kierkels, R. (2021). Analysis of a rarely implemented security feature: signing container images with a notary server. [Cited on page 16]

[Fagarasan et al., 2021] Fagarasan, C., Popa, O., Pisla, A., and Cristea, C. (2021). Agile, waterfall and iterative approach in information technology projects. In *IOP Conference Series: Materials Science and Engineering*, volume 1169, page 012025. IOP Publishing. [Cited on page 7]

[Fevereiro, 2023] Fevereiro, D. D. M. (2023). Smart orchestration on cloud-native environments. Master's thesis. [Cited on page 21]

[Flauzac et al., 2020] Flauzac, O., Mauhourat, F., and Nolot, F. (2020). A review of native container security for running applications. *Procedia Computer Science*, 175:157–164. [Cited on page 33]

[Fong and Steinder, 2008] Fong, L. and Steinder, M. (2008). Duality of virtualization: simplification and complexity. *ACM SIGOPS Operating Systems Review*, 42(1):96–97. [Cited on page 4]

[Force and Initiative, 2013] Force, J. T. and Initiative, T. (2013). Security and privacy controls for federal information systems and organizations. *NIST Special Publication*, 800(53):8–13. [Cited on page 27]

[Fu et al., 2019] Fu, Y., Zhang, S., Terrero, J., Mao, Y., Liu, G., Li, S., and Tao, D. (2019). Progress-based container scheduling for short-lived applications in a kubernetes cluster. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 278–287. IEEE. [Cited on page 21]

[Haddad et al., 2023] Haddad, A., Aaraj, N., Nakov, P., and Mare, S. F. (2023). Automated mapping of cve vulnerability records to mitre cwe weaknesses. *arXiv preprint arXiv:2304.11130*. [Cited on page 13]

[Hahn et al., 2020] Hahn, D. A., Davidson, D., and Bardas, A. G. (2020). Security issues and challenges in service meshes–an extended study. *arXiv preprint arXiv:2010.11079*. [Cited on page 25]

[Hannousse, 2021] Hannousse, A. (2021). Searching relevant papers for software engineering secondary studies: Semantic scholar coverage and identification role. *IET Software*, 15(1):126–146. [Cited on page 6]

[Holm et al., 2011] Holm, H., Sommestad, T., Almroth, J., and Persson, M. (2011). A quantitative evaluation of vulnerability scanning. *Information Management & Computer Security*, 19(4):231–247. [Cited on page 33]

[Honkaranta et al., 2021] Honkaranta, A., Leppänen, T., and Costin, A. (2021). Towards practical cybersecurity mapping of stride and cwe—a multi-perspective approach. In *2021 29th Conference of Open Innovations Association (FRUCT)*, pages 150–159. IEEE. [Cited on page 13]

[Ibrahim et al., 2016] Ibrahim, A. S., Hamlyn-Harris, J., and Grundy, J. (2016). Emerging security challenges of cloud virtual infrastructure. *arXiv preprint arXiv:1612.09059*. [Cited on page 5]

[Islam and Hasan, 2017] Islam, T. and Hasan, M. S. (2017). A performance comparison of load balancing algorithms for cloud computing. In *2017 International Conference on the Frontiers and Advances in Data Science (FADS)*, pages 130–135. IEEE. [Cited on page 3]

[Jaisinghani, 2022] Jaisinghani, G. (2022). Vulnerability management in the age of containers–a review. *International Journal of Information Security (IJIS)*, 1(01). [Cited on page 13]

[Javed and Toor, 2021] Javed, O. and Toor, S. (2021). An evaluation of container security vulnerability detection tools. In *Proceedings of the 2021 5th International Conference on Cloud and Big Data Computing*, pages 95–101. [Cited on page 28]

[Kalaiselvi et al., 2023] Kalaiselvi, R., Ravisankar, S., M, V., and Ravindran, D. (2023). Enhancing the container image scanning tool - grype. In *2023 2nd International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*, pages 1–6. [Cited on page 14]

[Kosińska et al., 2023] Kosińska, J., Baliś, B., Konieczny, M., Malawski, M., and Zieliński, S. (2023). Toward the observability of cloud-native applications: The overview of the state-of-the-art. *IEEE Access*, 11:73036–73052. [Cited on page 4]

[Kostova et al., 2020] Kostova, B., Gürses, S., and Troncoso, C. (2020). Privacy engineering meets software engineering. *On the Challenges of Engineering Privacy By Design*. [Cited on page 20]

[Kubernetes, 2019] Kubernetes, T. (2019). Kubernetes. *Kubernetes. Retrieved May*, 24:2019. [Cited on page 21]

[Kuppusamy et al., 2018] Kuppusamy, T. K., DeLong, L. A., and Cappos, J. (2018). Uptane: Security and customizability of software updates for vehicles. *IEEE vehicular technology magazine*, 13(1):66–73. [Cited on page 16]

[Kurtzer et al., 2017] Kurtzer, G. M., Sochat, V., and Bauer, M. W. (2017). Singularity: Scientific containers for mobility of compute. *PloS one*, 12(5):e0177459. [Cited on page 12]

[Kutsa, 2025] Kutsa, D. (2025). Building resilient microservices with kubernetes and istio. [Cited on pages 28 and 30]

[Kwon and Lee, 2020] Kwon, S. and Lee, J.-H. (2020). Divds: Docker image vulnerability diagnostic system. *IEEE access*, 8:42666–42673. [Cited on page 13]

[Larrucea et al., 2018] Larrucea, X., Santamaria, I., Colomo-Palacios, R., and Ebert, C. (2018). Microservices. *IEEE Software*, 35(3):96–100. [Cited on page 2]

[Larsson et al., 2020] Larsson, L., Tärneberg, W., Klein, C., Elmroth, E., and Kihl, M. (2020). Impact of etcd deployment on kubernetes, istio, and application performance. *Software: Practice and experience*, 50(10):1986–2007. [Cited on page 25]

[Li and Palanisamy, 2018] Li, C. and Palanisamy, B. (2018). Privacy in internet of things: From principles to technologies. *IEEE Internet of Things Journal*, 6(1):488–505. [Cited on page 20]

[Lin et al., 2020] Lin, C., Nadi, S., and Khazaei, H. (2020). A large-scale data set and an empirical study of docker images hosted on docker hub. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 371–381. [Cited on page 4]

[Linthicum, 2017] Linthicum, D. S. (2017). Cloud-native applications and cloud migration: The good, the bad, and the points between. *IEEE Cloud Computing*, 4(5):12–14. [Cited on page 4]

[Lopes et al., 2020] Lopes, N., Martins, R., Correia, M. E., Serrano, S., and Nunes, F. (2020). Container hardening through automated seccomp profiling. In *Proceedings of the 2020 6th International Workshop on Container Technologies and Container Clouds*, pages 31–36. [Cited on pages 17 and 18]

[Luksa, 2017] Luksa, M. (2017). *Kubernetes in action*. Simon and Schuster. [Cited on page 21]

[Malhotra et al., 2014] Malhotra, L., Agarwal, D., Jaiswal, A., et al. (2014). Virtualization in cloud computing. *J. Inform. Tech. Softw. Eng*, 4(2):1–3. [Cited on page 3]

[Medel et al., 2018] Medel, V., Tolosana-Calasanz, R., Bañares, J. Á., Arronategui, U., and Rana, O. F. (2018). Characterising resource management performance in kubernetes. *Computers & Electrical Engineering*, 68:286–297. [Cited on page 22]

[Mell et al., 2011] Mell, P., Grance, T., et al. (2011). The nist definition of cloud computing. [Cited on page 3]

[Monteiro et al., 2018] Monteiro, D., Gadelha, R., Maia, P. H. M., Rocha, L. S., and Mendonça, N. C. (2018). Beethoven: an event-driven lightweight platform for microservice orchestration. In *Software Architecture: 12th European Conference on Software Architecture, ECSA 2018, Madrid, Spain, September 24–28, 2018, Proceedings 12*, pages 191–199. Springer. [Cited on page 19]

[Moriconi et al., 2023] Moriconi, F., Neergaard, A. I., Georget, L., Aubertin, S., and Francillon, A. (2023). Reflections on trusting docker: Invisible malware in continuous integration systems. In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 219–227. IEEE. [Cited on page 16]

[Myrbakken and Colomo-Palacios, 2017] Myrbakken, H. and Colomo-Palacios, R. (2017). Devsecops: a multivocal literature review. In *Software Process Improvement and Capability Determination: 17th International Conference, SPICE 2017, Palma de Mallorca, Spain, October 4–5, 2017, Proceedings*, pages 17–29. Springer. [Cited on page 21]

[Nagpal et al., 2024] Nagpal, A., Pothineni, B., Parthi, A. G., Maruthavanan, D., Banarse, A. R., kumar Veerapaneni, P., Sankiti, S. R., and Jayaram, V. (2024). Framework for automating compliance verification in ci/cd pipelines. *Journal ID*, 9471:1297. [Cited on pages 33 and 34]

[Nascimento et al., 2024] Nascimento, B., Santos, R., Henriques, J., Bernardo, M. V., and Caldeira, F. (2024). Availability, scalability, and security in the migration from container-based to cloud-native applications. *Computers*, 13(8):192. [Cited on page 18]

[Neves, 2024] Neves, S. S. (2024). Mesh microservices on kubernetes clusters. Master's thesis. [Cited on page 25]

[Nevola, 2023] Nevola, F. (2023). *Securing communication between microservices in a multi-cloud scenario using Istio service mesh.* PhD thesis, Politecnico di Torino. [Cited on page 24]

[Nordell, 2022] Nordell, F. (2022). A systematic evaluation of cves and mitigation strategies for a kubernetes stack. [Cited on pages 22, 28, 29, and 33]

[O'Donoghue et al., 2024] O'Donoghue, E., Reinhold, A. M., and Izurieta, C. (2024). Assessing security risks of software supply chains using software bill of materials. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering-Companion (SANER-C)*, pages 134–140. IEEE. [Cited on page 14]

[of the European Union, 2022] of the European Union, C. (2022). Directive (eu) 2022/2555 of the european parliament and of the council of 14 december 2022 on measures for a high common level of cybersecurity across the union, amending regulation (eu) no 910/2014 and repealing directive (eu) 2016/1148 (nis2 directive). [Cited on page 27]

[Pace, 2021] Pace, M. (2021). *Zero Trust networks with Istio.* PhD thesis, Politecnico di Torino. [Cited on pages 24, 26, and 33]

[Pai and Kunte, 2023] Pai, S. and Kunte, S. R. (2023). Secret management in managed kubernetes services. *International Journal of Case Studies in Business, IT and Education (IJCSBE)*, 7(2):130–140. [Cited on pages 28, 29, and 33]

[Patil and Modi, 2019] Patil, R. and Modi, C. (2019). An exhaustive survey on security concerns and solutions at different components of virtualization. *ACM Computing Surveys (CSUR)*, 52(1):1–38. [Cited on page 3]

[Perrey and Lycett, 2003] Perrey, R. and Lycett, M. (2003). Service-oriented architecture. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.*, pages 116–119. [Cited on page 2]

[Potdar et al., 2020] Potdar, A. M., Narayan, D., Kengond, S., and Mulla, M. M. (2020). Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171:1419–1428. [Cited on page 12]

[Prinsloo et al., 2019] Prinsloo, J., Sinha, S., and Von Solms, B. (2019). A review of industry 4.0 manufacturing process security risks. *Applied Sciences*, 9(23):5105. [Cited on page 15]

[Queiroz et al., 2023] Queiroz, R., Cruz, T., Mendes, J., Sousa, P., and Simões, P. (2023). Container-based virtualization for real-time industrial systems—a systematic review. *ACM Comput. Surv.*, 56(3). [Cited on page 11]

[Rad et al., 2017] Rad, B. B., Bhatti, H. J., and Ahmadi, M. (2017). An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228. [Cited on page 12]

[Radack, 2012] Radack, S. (2012). Cloud computing: A review of features, benefits, and risks, and recommendations for secure, efficient implementations. [Cited on page 3]

[Rahaman et al., 2023] Rahaman, M. S., Islam, A., Cerny, T., and Hutton, S. (2023). Static-analysis-based solutions to security challenges in cloud-native systems: Systematic mapping study. *Sensors*, 23(4):1755. [Cited on page 5]

[Rashid and Chaturvedi, 2019] Rashid, A. and Chaturvedi, A. (2019). Cloud computing characteristics and services: a brief review. *International Journal of Computer Sciences and Engineering*, 7(2):421–426. [Cited on page 3]

[Riegel, 2024] Riegel, B. J. (2024). Leveraging ebpf in orchestrated edge infrastructures. [Cited on page 24]

[Rose et al., 2015] Rose, K., Eldridge, S., and Chapin, L. (2015). The internet of things: An overview. *The internet society (ISOC)*, 80:1–50. [Cited on page 5]

[Ross et al., 2019] Ross, R., Pillitteri, V., Dempsey, K., Riddle, M., and Guissanie, G. (2019). Protecting controlled unclassified information in nonfederal systems and organizations. Technical report, National Institute of Standards and Technology. [Cited on page 26]

[Sahoo et al., 2010] Sahoo, J., Mohapatra, S., and Lath, R. (2010). Virtualization: A survey on concepts, taxonomy and associated security issues. In *2010 Second International Conference on Computer and Network Technology*, pages 222–226. [Cited on page 2]

[Schneider, 2003] Schneider, F. (2003). Least privilege and more [computer security]. *IEEE Security Privacy*, 1(5):55–59. [Cited on page 20]

[Schneider, 2000] Schneider, F. B. (2000). Enforceable security policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50. [Cited on page 3]

[Scholz et al., 2018] Scholz, D., Raumer, D., Emmerich, P., Kurtz, A., Lesiak, K., and Carle, G. (2018). Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 1, pages 209–217. IEEE. [Cited on page 23]

[Seo et al., 2014] Seo, K.-T., Hwang, H.-S., Moon, I.-Y., Kwon, O.-Y., and Kim, B.-J. (2014). Performance comparison analysis of linux container and virtual machine for building cloud. *Advanced Science and Technology Letters*, 66(105-111):2. [Cited on page 12]

[Shamim et al., 2020] Shamim, M. S. I., Bhuiyan, F. A., and Rahman, A. (2020). Xi commandments of kubernetes security: A systematization of knowledge related to kubernetes security practices. *2020 IEEE Secure Development (SecDev)*, pages 58–64. [Cited on page 22]

[Singh et al., 2024] Singh, A. P., Kuzminykh, I., and Ghita, B. (2024). Industry perception of security challenges with identity access management solutions. In *2024 IEEE International Black Sea Conference on Communications and Networking (BlackSeaCom)*, pages 312–315. [Cited on page 18]

[Singh et al., 2022] Singh, J., Patel, C., and Chaudhary, N. K. (2022). Resilient risk-based adaptive authentication and authorization (rad-aa) framework. In *International Conference on Information Security, Privacy and Digital Forensics*, pages 371–385. Springer. [Cited on page 18]

[Sobieraj and Kotyński, 2024] Sobieraj, M. and Kotyński, D. (2024). Docker performance evaluation across operating systems. *Applied Sciences*, 14(15):6672. [Cited on page 3]

[Song et al., 2023] Song, S., Suneja, S., Le, M. V., and Tak, B. (2023). Sequence-based system call filtering for enhanced container security, is it beneficial? In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, pages 278–280. [Cited on pages 18, 28, and 29]

[Souppaya et al., 2017] Souppaya, M., Morello, J., and Scarfone, K. (2017). Application container security guide. Technical report, National Institute of Standards and Technology. [Cited on page 19]

[Theodoropoulos et al., 2023] Theodoropoulos, T., Rosa, L., Benzaid, C., Gray, P., Marin, E., Makris, A., Cordeiro, L., Diego, F., Sorokin, P., Girolamo, M. D., et al.

(2023). Security in cloud-native services: A survey. *Journal of Cybersecurity and Privacy*, 3(4):758–793. [Cited on pages 18 and 21]

[Tolone et al., 2005] Tolone, W., Ahn, G.-J., Pai, T., and Hong, S.-P. (2005). Access control in collaborative systems. *ACM Computing Surveys (CSUR)*, 37(1):29–41. [Cited on page 17]

[Ubale Swapnaja et al., 2014] Ubale Swapnaja, A., Modani Dattatray, G., and Apte Sulabha, S. (2014). Analysis of dac mac rbac access control based models for security. *International Journal of Computer Applications*, 104(5):6–13. [Cited on page 17]

[Ugale and Potgantwar, 2023] Ugale, S. and Potgantwar, A. (2023). Container security in cloud environments: A comprehensive analysis and future directions for devsecops. *Engineering Proceedings*, 59(1):57. [Cited on page 14]

[Union, 2016] Union, E. (2016). General data protection regulation. `http://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679`,. visited on 2021-09-21. [Cited on page 26]

[van der Slik et al., 2021] van der Slik, M., Wiersma, F., and PwC, W. O. (2021). Validating the replacement filtering features of popular alternative admission controllers for pod security policies. [Cited on page 22]

[Villamizar et al., 2015] Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., and Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590. [Cited on page 1]

[Voigt and Von dem Bussche, 2017] Voigt, P. and Von dem Bussche, A. (2017). The eu general data protection regulation (gdpr). *A Practical Guide, 1st Ed., Cham: Springer International Publishing*, 10:3152676. [Cited on pages 5 and 19]

[Wilken and Eulisse, 2024] Wilken, T. and Eulisse, G. (2024). Managing software build infrastructure at alice using hashicorp nomad. In *EPJ Web of Conferences*, volume 295, page 05013. EDP Sciences. [Cited on page 25]

[Xia et al., 2023] Xia, B., Bi, T., Xing, Z., Lu, Q., and Zhu, L. (2023). An empirical study on software bill of materials: Where we stand and the road ahead. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 2630–2642. IEEE. [Cited on page 14]

[Xu et al., 2017] Xu, Q., Jin, C., Rasid, M. F. B. M., Veeravalli, B., and Aung, K. M. M. (2017). Decentralized content trust for docker images. In *IoTBDS*, pages 431–437. [Cited on pages 16 and 33]

[Zarić,] Zarić, M. Inheritance of seccomp profiles in distributed clouds. [Cited on page 17]

[Zdun et al., 2023] Zdun, U., Queval, P.-J., Simhandl, G., Scandariato, R., Chakravarty, S., Jelic, M., and Jovanovic, A. (2023). Microservice security metrics for secure communication, identity management, and observability. *ACM transactions on software engineering and methodology*, 32(1):1–34. [Cited on page 33]

[Zhou et al., 2023] Zhou, J., Li, X., Wang, Q., Qin, X., Miao, W., and Tian, J. (2023). Balancing load: An adaptive traffic management scheme for microservices. In *2022 IEEE 28th International Conference on Parallel and Distributed Systems (ICPADS)*, pages 641–648. [Cited on pages 28 and 30]