INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Segurança Zero Trust para Mircroserviços em Sistemas Escaláveis

JOSÉ MIGUEL DE JESUS SILVA
Junho de 2024

POLITÉCNICO
DO PORTO

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Zero Trust Security for Microservices in Scalable Systems

José Silva

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of Computer Systems

Supervisor: Jorge Pinto Leite
Co-Supervisor: Pedro Daniel Carvalho De Sousa Rodrigues

Porto, June 28, 2024

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore the work presented in this document is original and authored by me, having not previously been used for any other end.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, June 28, 2024

Dedicatory

To the three most formidable forces in my life:

To my dear girlfriend, whose support and occasional eye-rolls have fueled both my academic pursuits and questionable sense of humor. You are the main reason of this adventure of two years, through every late night and early morning, your belief in me never wavered.

To my mother, whose persistent belief in my intelligence has kept me going, even when my eagerness to talk about topics that she doesn't understand puts her through a headache. Your faith in me is truly supernatural.

To my esteemed sister, the academic luminary whose intellectual shadow I've been lucky enough to bask in. Your wisdom is only surpassed by your ability to make me feel like the younger, less scholarly sibling. Here's to hoping this thesis makes you proud and doesn't earn me a red-inked critique from the expert.

May this paper serve as a testament to the chaos you've each brought into my life, making every moment simultaneously challenging and incredibly entertaining. Without you, my bibliography might be more extensive, but my life would certainly be less colorful.

Abstract

In the dynamic landscape of contemporary cybersecurity, the Zero Trust model has emerged as a pivotal paradigm, revolutionizing conventional notions of network trust. This thesis undertakes a comprehensive exploration of key dimensions within Zero Trust implementation, encompassing the theoretical and practical evaluation of diverse technologies, real-world performance assessments, and an in-depth analysis of resource implications, regulatory alignment, and human-centric vulnerabilities. Through a nuanced investigation, this research contributes valuable insights to the ongoing discourse on cybersecurity, offering practical guidance for organizations navigating the challenges of modern computing environments. By addressing technology selection, performance in real-world scenarios, resource considerations, regulatory compliance, and human factors, this thesis aims to empower cybersecurity practitioners with a holistic understanding of the intricate dynamics surrounding Zero Trust security.

Keywords: Zero-Trust, Service Mesh, Kubernetes, Cybersecurity

Resumo

Esta tese explora as complexidades e as praticidades da implementação dos princípios de segurança Zero Trust (ZT) em ambientes computacionais dinâmicos e escaláveis, com um foco particular em Kubernetes. À medida que as medidas de segurança tradicionais lutam para acompanhar o rápido avanço da tecnologia, a mudança para plataformas de orquestração de contentores como Kubernetes exige uma reavaliação dos modelos de segurança convencionais. Zero Trust, que opera com a premissa de que a confiança nunca deve ser assumida e a verificação contínua é essencial, oferece uma abordagem promissora para a proteção destes ambientes modernos.

O objetivo principal desta pesquisa é avaliar a aplicabilidade, a eficácia e as implicações do Zero Trust nos ecossistemas Kubernetes. Kubernetes, uma ferramenta de orquestração de código aberto altamente popular usada por 71% das empresas da Fortune 100, foi escolhida pela sua flexibilidade, escalabilidade e forte apoio comunitário. Estes atributos fazem dela uma candidata ideal para a implementação de Zero Trust, pois facilitam a monitorização contínua, a aplicação de políticas e os controlos de segurança dinâmicos necessários para uma proteção robusta.

A tese identifica desafios chave associados à implementação do Zero Trust em sistemas baseados em microserviços como Kubernetes. Estes incluem a alta demanda de recursos e a complexidade da implementação, que surgem da natureza distribuída dos microserviços. Os objetivos da pesquisa são avaliar várias tecnologias Zero Trust tanto teórica como praticamente, avaliar o desempenho em cenários reais e estudar o impacto na utilização de recursos, desempenho, custo e facilidade de implementação.

Para atingir esses objetivos, a tese adota uma abordagem abrangente que inclui uma revisão crítica da literatura existente e avaliações práticas em ambientes simulados. Esta metodologia permite uma análise detalhada do desempenho e da eficácia das práticas Zero Trust, juntamente com uma análise comparativa de diferentes tecnologias com base nos seus requisitos de recursos, custos e desafios de implementação.

A revisão da literatura é conduzida utilizando uma combinação de buscas em bases de dados e técnicas de "snowballing" em várias bases de dados acadêmicas, garantindo uma pesquisa ampla e detalhada. A estrutura da tese é desenhada para construir sistematicamente o entendimento do leitor, começando com uma introdução aos microserviços e Kubernetes, seguida de uma exploração aprofundada dos princípios Zero Trust, culminando na implementação prática e avaliação de desempenho das tecnologias Zero Trust em ambientes Kubernetes.

Na seção de avaliação de desempenho, vários cenários de teste são projetados para avaliar o impacto das implementações Zero Trust no desempenho, segurança e aspectos operacionais. Testes iniciais estabelecem métricas de desempenho de referência sem as tecnologias Zero Trust ativadas, proporcionando um ponto de referência para medir a sobrecarga introduzida pelos componentes Zero Trust. Testes subsequentes avaliam o impacto das políticas de rede,

mecanismos de descoberta de serviços, eficiência de recursos, escalabilidade e resiliência das soluções Zero Trust.

Indicadores chave de desempenho, como uso de CPU, uso de memória e latência de pedidos, são medidos para entender o consumo de recursos e a eficiência das diferentes tecnologias Zero Trust. A avaliação cobre Gestão de Identidade e Acesso (IAM) e Segurança de Rede e Aplicação de Políticas, com foco em mTLS, malhas de serviço como Istio, Linkerd e Cilium, e ferramentas de segurança em tempo de execução como Falco. Estas tecnologias são avaliadas pelo seu impacto na latência, uso de recursos e desempenho geral do sistema.

Observabilidade e auditoria também são aspectos críticos da avaliação. Prometheus é utilizado para a coleta de métricas, e Grafana para visualização, permitindo monitorização em tempo real e análise comparativa das diferentes implementações Zero Trust. Ferramentas de benchmarking desenvolvidas customizadamente simulam cargas de trabalho do mundo real para fornecer dados precisos de desempenho.

Em última análise, esta tese contribui para o discurso contínuo sobre cibersegurança com insights sobre a implementação pragmática dos princípios Zero Trust em Kubernetes. Oferece uma análise dos desafios e soluções, avalia os impactos de desempenho e propõe técnicas de otimização potenciais, bem como a proposição de uma nova abordagem à questão de configuração de *sidecars* em *Service Meshes* proporcionando assim uma base sólida para futuras pesquisas e desenvolvimento nesta área crítica da cibersegurança.

Acknowledgement

First and foremost, I would like to express my deepest gratitude to my co-supervisor, Mr. Pedro Rodrigues, whose guidance, wisdom, and occasional telepathic mind-reading abilities have been the cornerstone of this thesis. Your patience and belief in the finishing of the thesis have been nothing short of miraculous, thank you for making me walk the right path and for being pragmatic on every decision.

A special shout-out to my supervisor, Dr. Jorge Pinto Leite, for accepting this challenge and for the guidance throughout the thesis, your insights and support have been invaluable, and your help has kept me sane through the revisions.

To my amazing family, who now probably know more about my research than they ever wanted to, your unwavering support has been the bedrock of this journey. Thanks for pretending to be interested and thanks for pretending to understand what my thesis is about and for always asking, "Are you done yet?". Your pride in my achievements has been the fuel that kept me going through the toughest times.

To my incredible girlfriend, your love and support have been the light that guided me through the darkest times. Thank you for your endless patience, for understanding my late nights and stressed-out days, and for always knowing when I needed a hug or a word of encouragement. Your unwavering belief in me has been my greatest strength. You've been my partner, my confidant, and my biggest cheerleader, and I am endlessly grateful for everything you've done to help me reach this milestone.

Lastly, to my friends, who never let me forget the importance of a good work-life balance. Thank you for the support, providing much-needed distractions, and reminding me that there is indeed life beyond academia. Your collective wisdom, laughter, and the occasional dose of tough love have kept me grounded and motivated.

Contents

List of Figures

List of Tables

List of Acronyms

ABAC	Atribute Based Access Control.
CA	Certificate Authority.
CD	Continuous Delivery.
CI	Continuous Integration.
CNCF	Cloud Native Computing Foundation.
CNI	Container Network Interface.
DNS	Domain Name System.
eBPF	extended Berkeley Packet Filter.
ESO	External Secrets Operator.
GKE	Google Kubernetes Engine.
HPA	Horizontal Pod Autoscaling.
IBM	International Business Machines Corporation.
IP	Internet Protocol.
mTLS	Mutual Transport Layer Security.
netns	Network Namespace.
NGFW	Next Generation Firewalls.
OSI	Open Systems Interconnection.
PKI	Public Key Infrastructure.
PR	Pull Request.
RBAC	Role-Based Access Control.
SDN	Software Defined Networks.
SNAC	Standard Network Access Control.
SOA	Service-Oriented-Architecture.
SPIFFE	Secure Production Identity Framework for Everyone.
SPOF	Single Point of Failure.
URI	Uniform Resource Identifier.

VCS	Version Control System.
ZT	Zero Trust.

Chapter 1

Introduction

1.1 Context

As technology continues to evolve, traditional security measures are facing unprecedented challenges, particularly in the dynamic and scalable world of computing ecosystems. Container orchestration platforms, like Kubernetes, have revolutionized the way applications are developed, deployed, and managed. However, this shift in paradigm requires a radical departure from traditional security models to address inherent security implications.

The Zero Trust (ZT) concept challenges traditional trust within network architectures. It envisions a reality where trust is never assumed, and perpetual verification becomes the foundation of security. In today's complex computing environments, characterized by dynamic containerized environments and microservices architecture, the need for a security model that adapts, evolves, and secures in real-time is imperative.

Through this thesis, we will unravel the intricacies of Zero Trust, examining its applicability, effectiveness, and implications within the context of scalable systems like Kubernetes. By questioning traditional notions of trust and embracing a model that scrutinizes every interaction, this research contributes to the ongoing cybersecurity discourse, offering insights into the pragmatic implementation of Zero Trust principles to fortify digital assets in an era of perpetual change.

Kubernetes is the big focus of this thesis as it is the most popular open-source orchestration tool in the world (Susnjara 2023) and is the primary orchestration tool of 71% of the Fortune 100 companies (*Kubernetes Project Journey Report* 2023).

Kubernetes stands out as the orchestration platform of choice due to its unparalleled flexibility, scalability, and widespread adoption in the industry. Its ability to automate the deployment, scaling, and operation of application containers across clusters of hosts makes it indispensable for modern enterprises seeking to streamline their operations. Furthermore, Kubernetes offers a robust ecosystem of tools and extensions, allowing for seamless integration with various services and infrastructure components. This extensibility is crucial in implementing a Zero Trust architecture, as it necessitates continuous monitoring, policy enforcement, and security controls that can be dynamically applied across diverse and evolving environments.

Another compelling reason for choosing Kubernetes is its strong community support and continuous development, which ensures that it remains at the forefront of container orchestration technology. The vibrant open-source community around Kubernetes contributes to rapid innovation and the development of cutting-edge features that address emerging

security challenges. This collaborative environment also means that best practices and security standards are constantly evolving, providing a solid foundation for implementing Zero Trust principles. By focusing on Kubernetes, this thesis leverages a platform that is not only technologically advanced but also widely accepted and supported, making the insights and solutions derived from this research highly relevant and applicable to a broad range of organizations and industries.

1.2 Problem Statement

There has been a trend in cybersecurity whereby organizations are increasingly adopting the ZT security model to protect their microservices in scalable systems (He et al. 2022).

This approach is gaining popularity due to its effectiveness in protecting, auditing and securing distributed environments. While Zero Trust Security offers substantial advantages, such as enhanced protection against potential threats, it also presents unique challenges, such as the high use of resources, both in terms of computing capacity and implementation complexity. These challenges stem from the nature of microservices, making it complex to implement a robust Zero Trust Security framework that can effectively protect against security breaches and unauthorized access in dynamic and scalable systems such as Kubernetes.

1.3 Objectives

This thesis aims to delve into crucial aspects of Zero Trust implementation within dynamic computing environments, outlining clear objectives:

- Evaluate various Zero Trust technologies, both theoretically and practically, understand their high and low points and understand their applicability;
- Evaluate the performance of Zero Trust practices in real scenarios;
- Provide a comprehensive study on the challenges, state-of-the-art, and possible solutions or implementations of Zero Trust on Kubernetes;
- Determine the impact on the use of resources, performance, cost and ease of implementation of Zero Trust technologies.

1.4 Approach

To comprehensively address the challenges highlighted in Section 1.2, this thesis will adopt an approach aimed at thoroughly investigating and evaluating the various dimensions of Zero Trust implementation. The outlined objectives will be pursued through the following approach:

The primary objective is to conduct a comprehensive evaluation of various Zero Trust technologies, encompassing both theoretical and practical dimensions. The theoretical analysis will involve a critical review of existing literature, frameworks, models, and technologies used to implement Zero Trust concepts. At the same time, practical evaluations will include hands-on assessments of selected technologies in simulated environments. Building on this knowledge, the thesis will systematically assess the performance and effectiveness of Zero Trust practices in real-world scenarios.

To gauge the feasibility and practicality of Zero Trust implementations, the thesis will delve into the impact on resource utilization, performance, cost implications, and ease of implementation. A comparative analysis of different Zero Trust technologies will be conducted, shedding light on their varying resource requirements, costs, and potential challenges during deployment.

1.5 Literature Review Methodology

To conduct the literature review, the following methodology was employed:

- To research chapters 2 and 4, a combination of database search with snowballing was used. After defining a research plan for the theme, a search was conducted in the databases Google Scholar, ScienceDirect, ACM Library and Web of Science. Then, a snowball approach was applied to the found articles.
- In chapter 3, a snowballing strategy was used, followed by a database search with the found topics.

1.6 Thesis Structure

- Chapter 1 will provide the context of the developed work, the problem to solve, the methodology and approach to the problem, and the document structure.
- The literature review will comprise chapters 2, 3 and 4.
- Chapter 2 will serve as an introduction to microservices and Kubernetes. It will explain the internal workings and components of these technologies, and provide a technological background about the networking and vulnerabilities that are essential for the solution that will be developed in this thesis.
- Chapter 3 will introduce Zero Trust (ZT), explain its principles and architecture, and discuss its associated technologies, challenges, and considerations.
- Chapter 4 will present processes and protocols to apply ZT principles. This will be essential to build the solution as the technology referred (mTLS for example) is the foundation of many ZT products.
- Chapter 5 presents the methodology and implementation of the technologies performance benchmark, along with the results and its analysis.
- Chapter 6 contains the performance evaluation and proposition of potential solutions or optimization techniques, its development feasibility, and its definition.
- Chapter 7 contains the summary of findings, its contributions, and the future direction of Zero Trust in Kubernetes.

Chapter 2

Container Environments Scale

In the contemporary landscape of software development, the demand for scalable, resilient, and efficient systems is increasing (Waseem, Liang, and Shahin 2020). With the rapid growth of digital transformation initiatives across industries, there is a pressing need to adopt architectural paradigms that can support the dynamic and often unpredictable nature of modern applications. Container environments have emerged as a fundamental building block in achieving this scalability and efficiency, offering a robust solution for managing the complexities of today's software ecosystems (Alshuqayran, Ali, and Evans 2016).

Containers encapsulate applications and their dependencies into isolated units that can run consistently across various computing environments. This isolation ensures that applications behave the same way regardless of where they are deployed, be it on a developer's laptop, a testing environment, or a production cluster. By abstracting the underlying infrastructure, containers enable developers to focus on writing code without worrying about the idiosyncrasies of the deployment environment. This portability is a key driver behind the widespread adoption of containers in both development and production settings.

The scalability of container environments is further amplified by orchestration platforms like Kubernetes, which automate the deployment, scaling, and management of containerized applications. Kubernetes provides a powerful framework for handling the lifecycle of containers, ensuring that applications can scale up or down based on demand, recover from failures, and update seamlessly with minimal downtime. This orchestration capability is critical in environments where applications must handle variable workloads and maintain high availability and performance.

As organizations increasingly adopt microservices architectures, the role of container environments becomes even more pivotal. Microservices, characterized by small, independently deployable services that work together to form a larger application, align naturally with the capabilities of containers. Each microservice can be packaged as a container, enabling independent development, testing, and deployment. This modularity not only enhances scalability but also fosters innovation, as development teams can iterate on individual services without impacting the entire application.

In this context, container environments and microservices together represent a synergistic approach to modern application development. They address the need for agility, scalability, and reliability in software systems, making them indispensable tools in the toolkit of contemporary developers and operations teams.

2.1 Microservices

In the fast-paced world of software architecture, the rise of microservices has emerged as a transformative paradigm, reshaping how we design and build applications. Microservices, a contemporary approach to software development, revolutionizes the traditional monolithic model by breaking down applications into a collection of small, independent services. At its core, a microservice is the concept of modularity. Each service encapsulates a specific business capability, operating as a self-contained unit. This modularity not only enhances code maintainability but also allows for independent development, deployment, and scaling of services (*Microservices* 2023).

As an architecture, microservices are based on dividing the system into smaller and smaller parts to efficiently scale and deploy a system. As said in Thönes 2015, the point of a microservices architecture is that each application has a single responsibility. As written in Thönes 2015, this "might be a single responsibility in terms of a functional requirement, or it might be in terms of a nonfunctional requirement or, as we've started talking about them, cross-functional requirements". Modern users demand interactive and dynamic user experiences across various devices, prompting developers and organizations to seek frequent updates for their services. The traditional approach of relying on monolithic applications is considered inadequate, as claimed in Chen, Shanshan, and Zheng 2017 that microservices bring about several advantages, notably enhancing the ease of maintaining, reusing, scaling, ensuring availability, and facilitating automated deployment processes. This architecture contrasts with the centralized integration of Service-Oriented-Architecture (SOA), which has been criticized for its drawbacks. Some view microservices as an evolution of SOA, as a solution emerged to address its issues, such as tight dependencies, the need for redeployment of multiple components when upgrading a service, and challenges in managing complexity as claimed by Raj and Bhukya 2023. As microservice architecture becomes the dominant choice in the service-oriented software industry, companies begin the race to replace virtual machines with containers (Alndawi 2021).

As each service consists of a small, loosely coupled, and independently deployable unit, the question of where and how to deploy and orchestrate the services arises. In comparison to the preceding generation of virtualization technologies, which operate virtual machines with distinct operating systems atop a host OS, containers are self-contained software units that function on the host OS while sharing the same kernel and services, as pictured in Figure 2.1. The continuous deployment and DevOps capabilities of microservices align seamlessly with the rapid pace of modern development (S. Newman 2021), as teams can iterate, test, and deploy services independently, promoting a streamlined development lifecycle.

2.2 Kubernetes

Microservices have emerged as the standard model for building cloud-native applications due to their ease of development, deployment, debugging, scalability, and shareability. However, breaking down an application into independent microservices poses new challenges, particularly in ensuring secure communication among these services, especially when the decomposition results in a large number of services. Even basic cloud applications typically consist of several tens of microservices, while some of the largest platforms like Netflix and Uber may comprise hundreds or even thousands of microservices (Chen, Shanshan, and Zheng 2017), often running on multiple containers. To tackle these challenges, Kubernetes emerged as a container orchestration system.

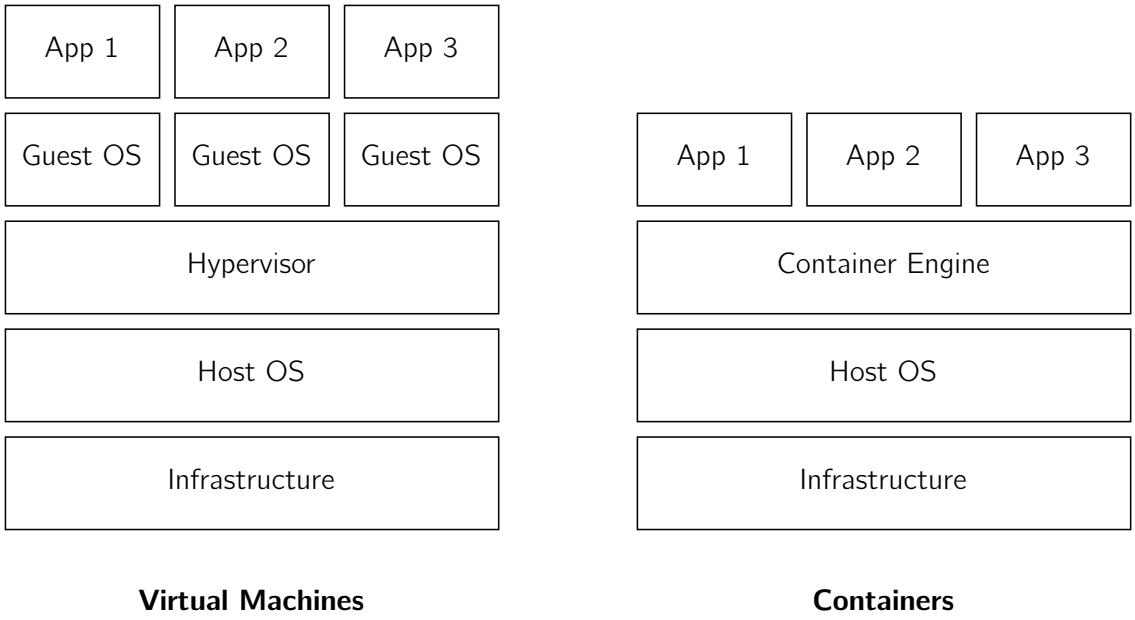

Figure 2.1: Virtual Machines vs Containers

Kubernetes is an open-source container orchestration platform that provides a framework for automating the deployment, scaling, and operations of application containers across clusters of hosts (Huang and Jumde 2020). By allowing us to define how containerized applications should run and interact with each other.

Kubernetes is supported on major container management and cloud platforms including Red Hat OpenShift, IBM Cloud, AWS, and Google Cloud.

An application operating on Kubernetes, also known as *k8s*, is deployed within a cluster, which is a group of machines, whether they are virtual or physical, designated for the execution of containerized applications.

Every cluster has a minimum of one master node and multiple worker nodes. The master nodes are responsible for overseeing the entire cluster, ensuring it remains in the desired state. They handle tasks such as scheduling the application containers on the worker nodes, which serve as the computing units.

In the depicted architecture illustrated in Figure 2.2, each master node comprises essential components that collectively form the backbone of the Kubernetes control plane. At the core is the *API Server*, tasked with orchestrating and overseeing all operations within the cluster, serving as the central point for communication and coordination. The *Scheduler* plays the role of workload management, responsible for distributing tasks among the worker nodes. Meanwhile, the *Controller* functions as responsible for the cluster's desired state, continuously working to ensure that the system aligns with the defined configuration. Finally, the master node includes *etcd*, a key-value-based database serving as the authoritative repository for all configuration information, preserving the integrity and coherence of the entire Kubernetes cluster (Minna et al. 2021).

Within the worker nodes, the *kubelet* operates on each node. It undertakes the registration of the node with the API server while also overseeing the well-being of pods generated

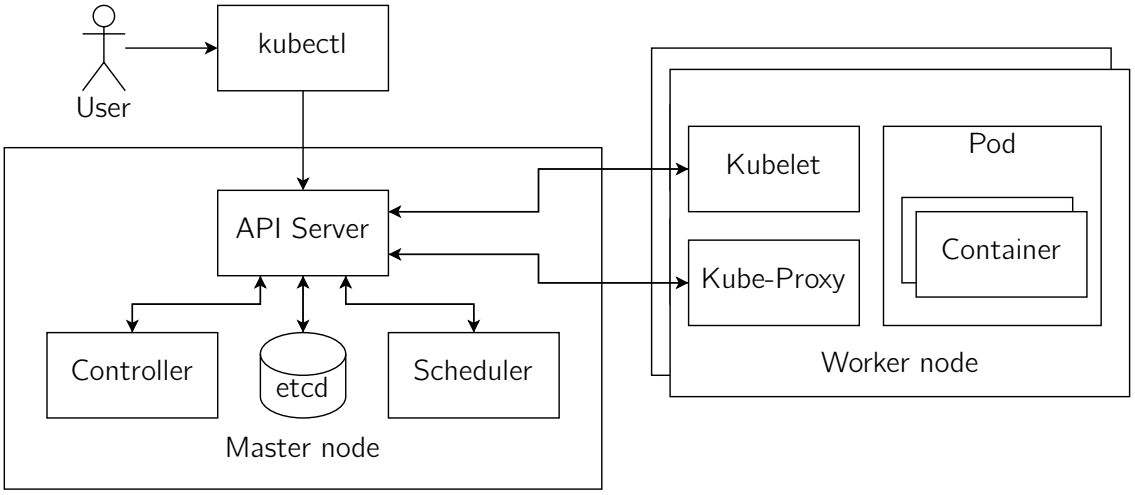

Figure 2.2: Kubernetes Architecture

Source: Bose, Rahman, and Shamim 2021

through Podspecs. Its responsibility extends to verifying the health of both pods and containers. The *Kube-Proxy* is responsible for maintaining network rules on nodes. It ensures that communication between pods and services within the cluster is appropriately forwarded (*kube-proxy* 2023). The *pod* represents the smallest deployable entity, encompassing at least one container.

2.2.1 Services

In order to understand Kubernetes Networking, we need to dive into the different types of services available in Kubernetes, as outlined in section 2.2.2. Services in Kubernetes act as a reliable endpoint for interacting with a group of pods (*Service* 2023). Since pods can change due to scaling, updates, or failures, services offer a consistent way for other components within or outside the cluster to communicate with these pods, regardless of their dynamic nature. One of the fundamental service types is the **ClusterIP** service, which provides an internal, stable IP address to expose a set of pods within the cluster. This means that other components within the cluster can access the service using this IP. Another service type is **NodePort**, which opens a specific port on all nodes in the cluster and forwards traffic to the service. This makes the service accessible externally, but it still relies on the ClusterIP for internal communication. For external access, the **LoadBalancer** service type integrates with cloud providers to provision an external load balancer, distributing traffic across the nodes. Finally, the **ExternalName** service type allows for aliasing a service to a DNS name, making it possible for pods to access external services using a custom name.

2.2.2 Kubernetes Networking

In Kubernetes, cluster networking refers to the networking model that allows communication between different components and pods within the cluster. There are several approaches to cluster networking in Kubernetes, with us being able to point out five distinct types of connections (*Cluster Networking* 2023):

1. **Container-to-Container** inside a single pod, where "all containers share resources and share fate" (Marmol, Jnagal, and Hockin 2015). They reside within a common network namespace, facilitating seamless communication among themselves using localhost and sharing the same port space. This co-location in a shared network namespace allows for efficient collaboration and interaction between the containers within the pod.

2. **Pod-to Pod inside the same node**, where they can communicate directly using localhost or the loopback interface. This direct interaction relies on the pod's assigned IP address, typically configured as a Virtual Ethernet (veth) pair (Figure 2.3), promoting efficient networking at the network layer. This arrangement facilitates swift and low-latency communication among pods situated within the same node.

3. When it comes to **Pod-to-Pod across different nodes in the cluster**, Kubernetes employs diverse networking solutions like Container Network Interfaces (CNIs) and software-defined networking (SDN) technologies. These solutions establish a virtual network overlay that spans the entire cluster, allowing seamless pod-to-pod communication across nodes. Noteworthy CNIs in this context include Calico[1], Flannel[2], Weave[3], and Cilium[4]. Irrespective of where a pod is located within the cluster, these networking solutions ensure continual accessibility to the pod's IP address, delivering transparent network connectivity throughout the Kubernetes cluster.

4. **Pod-to-Service** consists in using either the IP generated for the service or the DNS generated by CoreDNS. When creating a Service associated with a pod or a collection of pods, these are exposed through EndpointSlices, which serve as the definitive reference for kube-proxy, guiding the internal traffic routing process with accuracy and reliability. As described in *EndpointSlices* 2023 "EndpointSlices include references to all the Pods that match the Service selector. EndpointSlices group network endpoints together by unique combinations of protocol, port number, and Service name". When a pod communicates through a service with another pod, the service responsible for redirecting the traffic sends it to an associated EndpointSlice with matching labels. Shutdown pods and new pods are automatically removed and added from the respective EndpointSlice.

5. **External-to-Service** where External Entities communicate with the deployed applications using services. This can be achieved using a NodePort or LoadBalancer service type, using an Ingress object type backed by an Ingress Controller or implementing an API Gateway, materials referenced and explained with examples in *Services, Load Balancing, and Networking* 2023.

Applications deployed on a Kubernetes cluster should be reachable either within the cluster or externally from sources outside the cluster. This implies that, from a network standpoint, the application will have an associated Uniform Resource Identifier (URI) or Internet Protocol (IP) address. As multiple applications and their replicas can run simultaneously inside a Kubernetes cluster, to route traffic between them, Kubernetes treats each application like a Virtual Machine (Pod containing containerized application) as claimed in Huang and Jumde 2020, giving each Pod an internal IP.

[1] https://github.com/projectcalico/calico
[2] https://github.com/flannel-io/flannel
[3] https://github.com/weaveworks/weave
[4] https://github.com/cilium/cilium

The applications are given an ephemeral IP address and it is translated into an internal Domain Name System (DNS) provided by an internal service named Kube-DNS, which was replaced by CoreDNS in version 1.11 of Kubernetes because of security vulnerabilities in dnsmasq and performance issues in SkyDNS (Huang and Jumde 2020). Communication between services can either be done container to container, pod to pod, and pod to service. The latter is possible due to the capability of assigning a service to a pod (Tiwari 2023), which is capable of routing traffic into its assigned pods, assigning which is done using labels (Minna et al. 2021).

2.2.3 Kubernetes Distributed Architectures

As Kubernetes step up as the mainstream container orchestration tool, the need for geographic availability and workload redundancy emerges. Spanning across multiple clusters, data centers, and geographical locations, these workloads bring about the challenge of diverse capabilities in multi-cluster designs. The deployment of applications in such scenarios involves navigating through a mix of varied data centers, regions, and zones, adding complexity to the process. Solutions for this include projects such as the Kubernetes Federation Project[5]. This project brings the ability to move from a fragmented multi-cluster architecture, where a group of clusters works as a single system but has its separate control plane and rules, to a unified multi-cluster architecture where a single control plane manages a group of clusters with them being deployed either on different cloud providers, on-prem, in different geographic zones and/or data centers.

When using a single shared Kubernetes cluster, it's important to understand that, while cost-effective, it has drawbacks. These include a single point of failure, lack of hardware and network isolation, and limitations on the number of pods and nodes. Specifically, a Kubernetes cluster can support a maximum of 110 pods per node, 5,000 nodes, 150,000 total pods, and 300,000 total containers (The Linux Foundation 2023a).

As we can group Kubernetes multi-cluster architectures, these can vary depending on the needs and situations, with four ways to run workloads in a multi-cluster environment (Ambassador Labs 2023):

- **Single-use-cluster**, or **cluster-per-deployment** is a multi-cluster architecture where each application is deployed in its individual cluster, independent of the environment chosen for the deployment (production, staging, development, etc). This architecture, although providing hard isolation and reduced impact in case of an incident, is highly expensive compared to other options.
- In the **cluster-per-application** model, applications are deployed in separate clusters as done in cluster-per-deployment. Still, all of the different environments run in the same cluster. This model allows us to cut down on the costs that the above approach brings and has most of the advantages. It inherently brings the problem of a development workload creating incidents on the same cluster where a production workload is running.
- With the **cluster-per-environment** approach, different Kubernetes clusters are used for distinct environments, such as development, testing, staging, and production. Each environment has its dedicated cluster where multiple applications related to that environment are deployed. This model offers benefits like isolation by environment which

[5] https://github.com/kubernetes-retired/kubefed

consists of removing interference between development, testing, and production workloads, and resource allocation and configurations being tailored to the specific needs of each environment. It also allows for changes and updates to be tested in a controlled environment before being promoted to production.

- In a **replication cluster** approach, identical copies of the entire production cluster are created. This approach allows for global availability by replicating the application across multiple availability zones or data centers. User traffic is intelligently directed to the nearest or most suitable cluster, leveraging the potential for enhanced performance and reduced latency. Incorporating a health-aware global load balancer further enhances this architecture by enabling failover mechanisms. If a cluster experiences a malfunction or becomes unresponsive, user traffic seamlessly shifts to an operational cluster, ensuring continuous service availability and reliability.

Two main ways exist for configuring and operating multi-cluster architectures:

- A **controller-centered** architecture is used to manage and operate multi-cluster deployments by leveraging the Kubernetes API capabilities. This involves extending the Kubernetes control plane to handle the coordination and synchronization of resources across clusters. Projects such as Kubernetes Federation (KubeFed) fall under this category, as they provide a declarative way to specify the desired state of clusters, and controllers work to reconcile the actual state with the desired state. This approach offers a consistent and abstract way of managing clusters across different infrastructure providers.
- A **network-centered** approach focus is on the networking layer to enable communication and coordination between services running in different clusters. The use of service meshes enhances it.

2.2.4 Points of Failure

Kubernetes lays the groundwork for resilient, high-performance clusters. As we navigate the landscape of potential pitfalls, understanding these points of failure becomes paramount for ensuring the reliability and stability of a Kubernetes environment. According to the article Minna et al. 2021, several known critical points of failure can occur within Kubernetes.

Multi-tenancy

CVE-2020-8554 represents a significant security vulnerability present in all versions of Kubernetes (Mitre 2024). This vulnerability exposes a potential avenue for a man-in-the-middle (MITM) attack, particularly for an attacker with permission to create and update Services and pods within the cluster, A bad actor could exploit their permissions to manipulate the **spec.externalIPs** field of a Service, setting it to a specific external IP, potentially a well-known public IP address. This allows the attacker to intercept traffic originating from other pods or nodes in the cluster that attempt to access this external IP. Subsequently, the attacker can redirect this intercepted traffic to a malicious pod they have created, establishing a successful man-in-the-middle attack.

Furthermore, when dealing specifically with Services, attackers can initiate MITM attacks by tampering with the **status.loadBalancer.ingress.ip** field. This manipulation creates an opportunity for attackers to compromise the integrity of the cluster's communication, posing a serious threat to the security of multi-tenant Kubernetes environments.

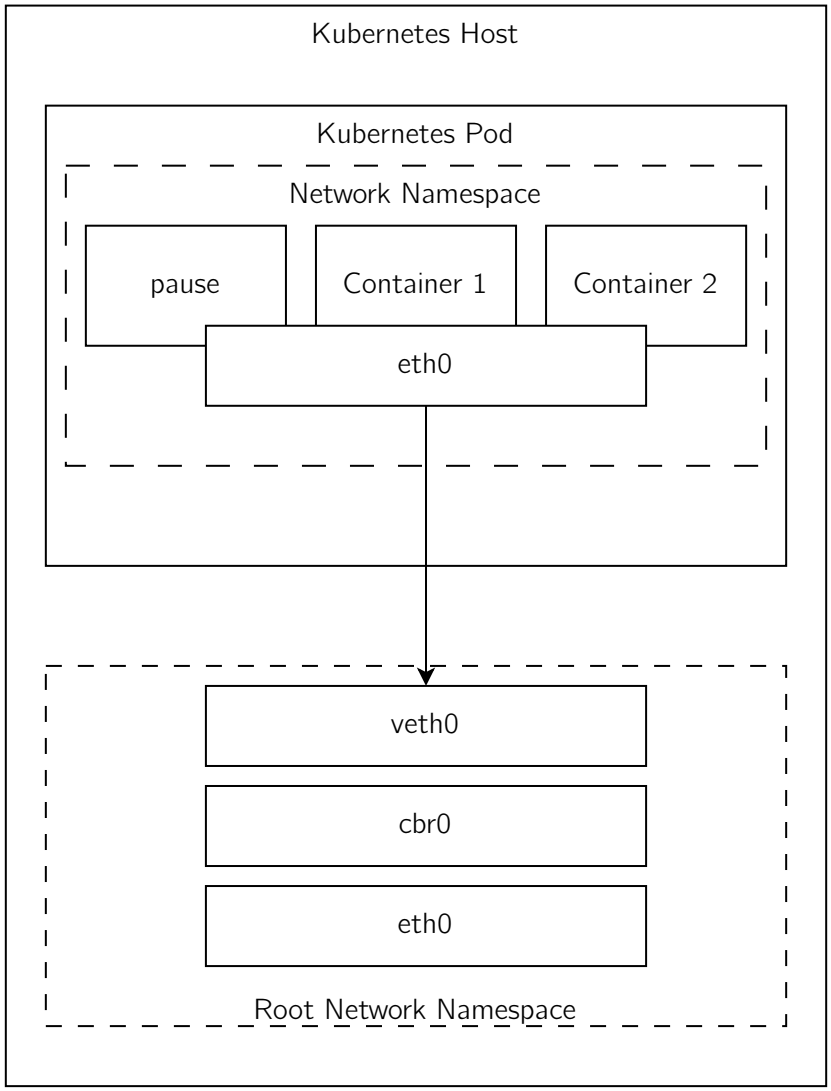

Figure 2.3: Kubernetes Namespaces Networking Architecture

As pointed out in Avrahami 2020, the vulnerability affects all Kubernetes versions and is currently unpatched. To mitigate this vulnerability it is recommended the use of a custom admission controller or a policy controller, as described in *Mitigate CVE-2020-8554 with Policy Controller* 2023.

Network Namespace and Pause Containers

A *pause container* is a container that is scheduled in every pod and serves as a placeholder, providing the necessary namespace and networking for the other containers within the same Pod. It essentially helps set up the networking namespace for the Pod and maintains the network namespace open, allowing other containers in the Pod to share the same network namespace. It becomes a security downfall as it is run as root, as escaping from the pause container Network Namespace (netns) means having access to the root netns, as shown in Figure 2.3. As summarized in Minna et al. 2021, "An attacker who is able to get on the host netns can potentially see network interfaces, routing rules, other pods netns: if the attacker has privileged access, the worker-node netns is fully compromised".

CNI Plugins

Container Network Interface (CNI) is a standard interface that allows different networking solutions to work efficiently with Kubernetes. It acts as a mediator between the container runtimes and network plugins. CNI specification defines how containers communicate with each other within a pod and with the outside world. Network plugins that implement CNI specification eliminate the need for network address translation in pod-to-pod communications. They handle tasks such as assigning IP addresses, setting up network routes, and managing network namespaces to facilitate container communication and connectivity. This information is pointed out in Kang et al. 2021.

This imposes a problem as CNI Plugins run as privileged on the worker nodes, so in case of CNI Plugin becoming compromised, the whole network is at risk (Minna et al. 2021).

Following the Open Systems Interconnection (OSI) model, we have 2 types of CNI Plugins:

- L2, such as Flannel (*flannel* 2023), which operates at the Data Link layer and is responsible for providing network connectivity based on MAC addresses, working with Ethernet frames. These may be susceptible to MAC address spoofing as attackers could forge or mimic MAC addresses to gain unauthorized access or disrupt communication.
- L3, such as Calico (*Calico* 2023), which operates at the Network layer and handles the routing.

Single Point of Failure

The setup of a Kubernetes master node by default can potentially create vulnerabilities that could affect the reliability of the entire cluster. This is due to the fact that the master node, comprising components such as the API server, controller-manager, and scheduler, acts as a Single Point of Failure (SPOF). If the node becomes inaccessible, it will disrupt management and coordination functions throughout the cluster.

As described in Section 2.2 the API server, controller-manager, and scheduler are crucial to the cluster's operation. If any of these components on the single master node fail, it can compromise the stability and functionality of the entire cluster.

Moreover, as the default configuration utilizes a single-instance etcd database, this introduces another SPOF, and if the etcd database becomes unavailable, it may result in data loss and hinder the retrieval of critical cluster information.

In the event of an outage or attack on the single master node, the cluster may still run existing workloads. However, the inability to manage or modify configurations renders the cluster unmanageable, ultimately impacting its operability.

Managed solutions such as Google Kubernetes Engine (GKE) provide the ability to replicate master nodes through different availability zones, which solves the Kubernetes SPOF problem. GKE for instance has the master node replication concept built into its architecture (Google 2023).

2.2.5 Levels of System Resilience

Resilience in the context of Kubernetes refers to the system's ability to maintain seamless operation and recover gracefully from disruptions, ensuring continuous delivery of services.

Kubernetes achieves resilience through its robust features, such as automated load balancing, self-healing capabilities, and dynamic scaling. These mechanisms contribute significantly to minimizing downtime and maximizing uptime, crucial metrics in evaluating the reliability of a system.

Ganek and Corbi 2003 refers to self-healing as the ability of a system "to recover from a failed component by first detecting and isolating the failed component, taking it off line, fixing or isolating the failed component, and reintroducing the fixed or replacement component into service without any apparent application disruption". Kubernetes achieves this important aspect of system resiliency through its native self-healing mechanisms, such as Replication Controllers that ensure a specified number of pod replicas are running at all times, health probes that check the liveness and readiness of pods, and a Horizontal Pod Autoscaling (HPA) that automatically adjusts the number of pod replicas in a deployment based on observed CPU utilization or custom metrics, adjusting the need with the demand.

Chapter 3

Zero Trust

Unlike traditional security models that rely on perimeter defenses, Zero Trust operates on the principle that no user or device, whether inside or outside the network, should be trusted by default. Every access request is thoroughly authenticated, authorized, and encrypted, minimizing the risk of unauthorized access and lateral movement within the network. This approach ensures that the security posture remains robust even if an attacker breaches the perimeter.

As discussed in Rose et al. 2020, "Zero trust security models assume that an attacker is present in the environment and that an enterprise-owned environment is no different—or no more trustworthy—than any non-enterprise-owned environment", with Zero Trust security models we assume that a hypothetical attacker has access to the infrastructure. This way it is possible to create a security model in which none of the resources, be them enterprise assets and subjects, have unverified access to each other. This can be seen as an evolution of Standard Network Access Control (SNAC) where the definition of a perimeter network is usually the "[...] most complicated, effort consuming and rapidly evolving part of the entire network" (Uctu et al. 2019; Bush and Mashatan 2022). As the source of attacks is evenly distributed between inside and outside (Bendovschi 2015), we can no longer solely rely on perimeter network protection.

The United States Government has recognized the vital importance of Zero Trust technologies and practices in improving cybersecurity. To this end, a memorandum (M-22-09) has been issued outlining a comprehensive strategy that embraces a "zero trust" model, rejecting the outdated concept of a secure network perimeter. Under this model, no user, device, or application can be automatically trusted, and every element must continuously prove its legitimacy before gaining access to any system.

This approach encompasses a range of key elements, including strengthened Identity and Access protocols, multi-factor authentication, and device controls, along with network encryption and rigorous application testing. Additionally, a government-wide data categorization strategy will be developed to enable automatic protection based on data sensitivity.

Working from detailed implementation plans, key officials will collaborate to ensure the successful execution of this zero-trust strategy. Ultimately, this approach aims to enhance the cybersecurity posture of the Federal Government significantly, safeguarding critical systems and data from cyberattacks.

To bolster cybersecurity practices, the National Institute of Standards and Technology (NIST) published a seminal whitepaper (Rose et al. 2020) that introduced the core concepts of zero-trust architecture (ZTA), highlighting its potential benefits and exploring various

deployment scenarios. Building upon this foundation, government agencies like the Cybersecurity and Infrastructure Security Agency (CISA) have released guidance documents to support the implementation of ZTA. Among these resources are maturity models designed to provide organizations with a structured roadmap for transitioning towards a full zero-trust environment.

In 2004, the *Jericho Forum*, a coalition of multinational corporations, revolutionized the concept of secure information sharing by introducing the revolutionary idea of *de-perimeterization*. This concept emphasized internal defenses and de-emphasized external boundaries, paving the way for a safer and more secure network infrastructure.

In 2009, Forrester analyst John Kindervag recognized the potential of the *de-perimeterization* model and proposed the game-changing concept of "Zero Trust" in his book "Build Security Into Your Network's DNA: The Zero Trust Network Architecture" (Kindervag 2010). This groundbreaking approach has now become a cornerstone of modern network security, providing unparalleled protection against cyber threats and data breaches.

In 2014, Google announced its efforts to move to a ZT architecture, announcing *Google BeyondCorp*, with the premise of "[...] moving to a new model that dispenses with a privileged corporate network. Instead, access depends solely on device and user credentials, regardless of a user's network location—be it an enterprise location, a home network, or a hotel or coffee shop. All access to enterprise resources is fully authenticated, fully authorized, and fully encrypted based upon device state and user credentials" (Ward and Beyer 2014).

The concept of zero trust operates under the assumption that assets or user accounts should not be inherently trusted based on their physical or network location, such as local area networks or the internet, nor based on ownership (whether they are enterprise-owned or personally owned). Authentication and authorization, for both users and devices, are distinct processes carried out before establishing a session with an enterprise resource. Zero trust has emerged in response to evolving enterprise network trends, including the prevalence of remote users and the utilization of cloud-based assets that extend beyond traditional enterprise network boundaries. Unlike traditional security models that focus on securing network segments, zero trust centers around safeguarding specific resources such as assets, services, workflows, and network accounts (Rose et al. 2020). An article by Google employees states that "[...] the perimeter is no longer just the physical location of the enterprise, and what lies inside the perimeter is no longer a blessed and safe place to host personal computing devices and enterprise applications" (Ward and Beyer 2014).

3.1 Principles of Zero Trust

The Zero Trust security framework is built on the principle of perpetual skepticism, requiring the continuous verification of users, devices, and systems. It challenges the traditional belief that everything within a network perimeter is inherently trustworthy. The framework's core principles include rigorous identity verification through **multi-factor and continuous authentication**, adhering to the principle of **least privilege access** to restrict permissions to the minimum necessary, implementing **micro-segmentation** for limited lateral movements, and maintaining **continuous monitoring** for prompt anomaly detection (Rose et al. 2020). Zero Trust assumes that a breach may occur at any time, promoting encryption for data in transit and at rest, enforcing device security standards, and emphasizing user education

to cultivate a security-aware culture. Continuous risk assessments and integration of automation and orchestration contribute to a robust security posture, enabling organizations to effectively adapt and defend against evolving cyber threats.

Identity validation plays a crucial role in security architecture by establishing secure identities for applications to transmit data. This process, akin to obtaining a passport, involves a trusted authority verifying the application workload with a series of identity proofs. The trust in identity is based on the authority and the verification process. It's imperative to have cryptographically verifiable identities, signed by a verifiable authority, as they offer higher security compared to weaker identity forms. Non-cryptographic identities, such as those based on IP addresses, are deemed less reliable and secure due to potential reallocation and staleness issues. In dynamic environments like Kubernetes, cryptographically verifiable identities provide enhanced security and reliability.

To maintain **confidentiality**, data encryption between applications is essential to thwart breaches and man-in-the-middle attacks. This requires unique encryption channels for each source-destination pair and encryption using unique keys derived from the identities of the source and destination. Customizable encryption may be necessary to meet specific security requirements.

Integrity ensures that encrypted data transmitted from the source to the destination remains unaltered by any unauthorized changes. Application owners necessitate consistent and clear policy enforcement based on application identities. Cryptographically verifiable identities with clear provenance are essential for effective access policy implementation. While standard TLS primarily verifies server identity, comprehensive policy enforcement requires reliable identities for both client and server. Cryptographic identities enable precise access control, such as allowing specific methods or requiring JWT tokens from certain issuers, ensuring secure, reliable, and auditable policy enforcement for internal applications.

3.1.1 Micro-segmentation

Micro-segmentation is a sophisticated network security technique that entails dividing a network into smaller, isolated segments to bolster security and enable better control over communication flow between distinct segments. Every segment, known as a "micro-segment," functions as an independent zone with its own unique security policies and controls (D'Silva and Ambawade 2021). This approach empowers organizations to diminish the attack surface and restrict lateral movement within the network, rendering it more difficult for unauthorized users or malicious software to move laterally if they manage to gain access to the network.

Micro-segmentation is a security technique that isolates workloads, applications, or services into distinct segments. This isolation helps in preventing unauthorized access between them. The security policies applied in micro-segmentation are at a granular level and specify what communication is allowed or denied between specific segments. This level of granularity enables fine-tuned control over network traffic, increasing the security of the system. Additionally, micro-segmentation often includes dynamic and adaptive control mechanisms, allowing security policies to adapt to changes in the network environment, such as the addition or removal of workloads. This dynamic and adaptive control further enhances the security of the system (Palo Alto Networks 2024).

Kubernetes, being a container orchestration platform, has the ability to implement micro-segmentation natively. This is made possible through the use of Network Policies, which are discussed in the Network Policies Section (ref. 4.4.2).

3.2 Zero Trust Architecture

Zero Trust Architecture follows the principle of least privilege, ensuring that users and devices are granted access only to the resources necessary for their specific roles. This is achieved through continuous monitoring and validation of user identities, device health, and contextual data such as location and time of access. The architecture is dynamic and adaptive, automatically adjusting security policies based on real-time assessments of trust levels. This adaptability is essential in today's rapidly evolving threat landscape, where static security measures may not be adequate.

3.2.1 Continuous authentication

Continuous authentication is a crucial component within the framework of Zero Trust security models. Zero Trust focuses on verifying the identity and security posture of every user and device, regardless of their location or network connection. Continuous authentication enhances this approach by constantly evaluating and re-evaluating user identities and device trust levels throughout their interactions with the system. As ZTA relies on these systems as crucial components, a low computational cost approach is necessary for their implementation (Meng et al. 2022). In highly scalable systems, the low computation approach can be achieved through the use of eBPF (ref. 4.3.1), which does not make the use of sidecars as implemented by other solutions such as Istio and Linkerd, with both this solutions being compared to the eBPF one in performance and usability.

3.2.2 Network-Centric vs. Identity-Centric

NIST Special Publication 800-207 (Rose et al. 2020) introduced two distinct methodologies for Zero-Trust implementation. The first, **Identity-centric**, utilizes identity and assigned attributes to formulate access policies for corporate resources. The second is **Network-centric**, which involves the use of intelligent switches, Next Generation Firewalls (NGFW), or Software Defined Networks (SDN) to enforce protection via network segmentation.

In the identity-centric approach, the user's established identity, along with other attributes like device posture, location, and time of access, are the primary basis for granting or denying access. This approach is all about knowing who is accessing and what they should be allowed to see and do. It reduces the attack surface by minimizing implicit trust and enables granular access control, aligning with the principle of least privilege. Additionally, this model is more adaptable to the rise of cloud-based resources and distributed workforces. However, it requires robust identity management solutions to maintain and verify user and device identities reliably, and initial setup and policy creation can be complex.

On the other hand, the network-centric approach hinges on micro-segmentation and intelligent network control. The network itself scrutinizes traffic and enforces access based on predefined rules, even if the user's identity is validated. This approach focuses on where the access is coming from and where it is trying to go. It is effective in protecting legacy systems or those where robust identity information might not be readily available and is well-suited to containing lateral movement by attackers should they gain a foothold within the network. However, it can be operationally complex to manage in a highly dynamic network environment and may not scale easily to address the flexibility demanded by cloud architectures.

3.3 Challenges and Considerations

The implementation of zero trust architectures presents numerous challenges, as highlighted in a recent survey (He et al. 2022). One major obstacle is the integration of existing technologies to meet the rigorous standards of ZTA. Key areas such as access control, identity authentication, and trust assessment are currently the subject of intense research to enhance the security and real-world applicability of ZTA solutions.

Regarding identity authentication, the survey emphasizes the superiority of multifactor authentication (MFA) over single-factor methods (He et al. 2022). MFA reduces the vulnerability to network attacks as compromised credentials alone are insufficient. Moreover, continuous authentication enhances security by continually verifying user access rights throughout sessions, mitigating mid-session threats. Prioritizing the optimization of authentication protocols to balance security with resource efficiency is identified as a top priority.

The survey advocates for risk-based access control to manage the complexities of modern security environments. This aligns with ZTA's fundamental principle of eliminating implicit trust within a network. Minimizing authorization privileges while still allowing for flexible, dynamic control is crucial. The authors anticipate a blend of access control models, such as Role-Based Access Control (RBAC) and Atribute Based Access Control (ABAC), to meet the demands of ZTA implementation.

Trust assessment is another critical component of ZTA security. The survey notes a trend toward comprehensive, fine-grained evaluations using various theories. Prioritizing the accuracy and efficiency of trust assessment, especially in resource-constrained environments, is paramount. Ultimately, ZTA will require the development of trust assessment algorithms capable of adapting to diverse network conditions and addressing privacy challenges.

In Rose et al. 2020, the author refers that trough a survey on cloud providers, the lack of available solutions result in a purchase and implementation of several different services which can very much increase the complexity of the security architecture and lead to dependencies. the same author claims that "(...) if one vital component is disrupted or unreachable, there could be a cascade of failures that impact one or multiple business processes."

Zero Trust, while a powerful security paradigm, presents a unique set of hurdles within the dynamic realm of Kubernetes. The sheer complexity of Kubernetes environments, where pods and workloads are ephemeral, makes enforcing strict access controls difficult. Maintaining granular visibility into every interaction between microservices adds a layer of operational overhead. In order to overcome this difficulties, Service Meshes were implemented. Altough being a powerful technology, it comes with its own set of challenges (Li et al. 2019):

1. At the data plane level, the proxy component plays a crucial role in a service mesh by being deployed alongside service instances. It is responsible for intercepting and mediating traffic between microservices, so it needs to be lightweight and optimized for high performance. This requirement for high performance also applies to control plane components, especially those involved in data aggregation.

2. In a rapidly evolving tech landscape, characterized by the migration towards cloud-native environments across diverse cloud providers and on-premises infrastructures, the adaptability of a service mesh has emerged as a pivotal requirement. Ensuring configurability and extensibility is essential for its capability to seamlessly integrate with and accommodate a variety of cloud-native orchestration platforms.

3. It is crucial for a service mesh to exhibit high availability in order to mitigate the potential for biased decision-making due to data or component unavailability. Achieving fault tolerance is essential in this regard.

3.4 Service Mesh

A service mesh is an advanced infrastructure layer designed to handle secure, fast, and reliable communication between microservices in a distributed application architecture. It abstracts the complex logic of service-to-service communication, including load balancing, service discovery, and failure recovery, away from the application code, allowing developers to focus on business logic rather than infrastructural concerns. By providing a dedicated layer for managing service interactions, a service mesh can significantly enhance observability, offering granular insights into traffic patterns and service performance, which are crucial for maintaining robust and scalable microservice ecosystems.

Li et al. 2019 referred to services meshes as "a dedicated infrastructure layer for handling service-to-service communication. It's responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware".

A service mesh is composed by a data plane and a control plane (Figure 3.1). The data plane is comprised by proxies with filter traffic and apply network policies and rules, which in when deployed in kubernetes are usually as sidecars (Zhu et al. 2023) making use of technologies such as Envoy[1]. With its access to every network packet, is primarily responsible for tasks such as service discovery, health checks, routing, load balancing, authentication, authorization, and monitoring. Meanwhile, the control plane, comprised by a single or multiple controllers, handles the management and configuration of proxies for directing traffic. Furthermore, it sets up various components to enforce policies and gather telemetry data. Functioning as the central intelligence of a service mesh, the control plane does not need direct visibility into network traffic (Li et al. 2019).

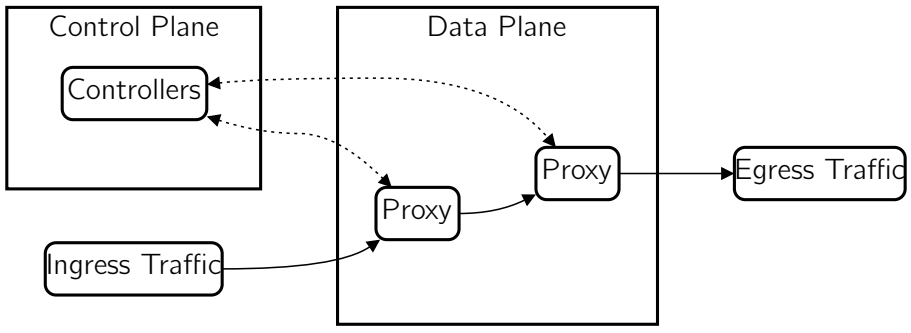

Figure 3.1: Service Mesh Architecture

As microservices architectures become increasingly prevalent, service meshes are becoming indispensable for ensuring essential inter-service communication feature such as:

- **Enhanced Observability:** Service meshes provide detailed insights into service interactions, including metrics, logs, and traces. This observability helps in identifying

[1] https://www.envoyproxy.io/

performance bottlenecks, understanding traffic patterns, and diagnosing issues efficiently.

- **Improved Security:** Service meshes offer robust security features such as mutual TLS (mTLS) for encrypting communications between services, fine-grained access control policies, and automatic certificate management, ensuring secure service-to-service communication.
- **Traffic Management:** Advanced routing capabilities, such as traffic splitting, load balancing, and fault injection, allow for precise control over how traffic is directed between services. This aids in canary releases, A/B testing, and gradual rollouts of new features.
- **Resilience and Fault Tolerance:** Service meshes provide built-in mechanisms like retries, timeouts, and circuit breakers, enhancing the resilience of the application by automatically handling transient failures and preventing cascading failures.
- **Decoupling Service Logic:** By externalizing service communication logic to the mesh, developers can focus on business logic without worrying about implementing retry mechanisms, load balancing, or other infrastructural concerns in their code.

When implementing a service mesh, it's important to consider the additional complexity it brings to the system. Setting up and managing a service mesh requires a deep understanding of its components and configuration, which can be challenging for teams without prior experience.

Another aspect to keep in mind is the increase in resource overhead. Sidecar proxies consume extra CPU and memory resources, which can lead to higher infrastructure costs and potentially impact the performance of the microservices they are associated with.

Additionally, the introduction of sidecar proxies and extra network hops in service meshes can cause latency in inter-service communication. For applications sensitive to latency, this additional overhead requires careful consideration and mitigation strategies.

Chapter 4

Kubernetes Security

Securing a Kubernetes cluster is crucial to ensure the confidentiality, integrity, and availability of your applications and data. Kubernetes protection strategies involve various measures, including node hardening, network security, access controls, and monitoring.

4.1 Rootless Images

Traditionally running containers require elevated privileges as containers share the host's kernel. Rootless containers consist of containers without superuser access.

Rootless containers allow users to run without the need for superuser access, achieved by using namespaces, user namespaces (Linux Foundation 2023) and seccomp (*A seccomp overview* 2023), introduced in 2002, 2008 and 2005 to the Linux Kernel, respectively.

Kubernetes dictates that a resource's visibility is limited to its respective namespace. User namespaces leverage user mapping to establish a connection between the user and group IDs in the container and those in the host namespace, thereby ensuring secure isolation. This crucial feature bolsters security measures by restricting the impact of potential vulnerabilities within a container, even in the event of a compromised process.

The shift to rootless containers represents a significant advancement in container security. By eliminating the need for superuser privileges, rootless containers reduce the attack surface and limit the potential damage that could be inflicted if a container is compromised. This security enhancement is achieved through the use of user namespaces, which map container user IDs to non-privileged user IDs on the host system. This mapping ensures that even if an attacker gains control of a rootless container, they do not have elevated privileges on the host, thereby containing the impact of the security breach.

4.2 Image Patching

Image patching involves updating the software and dependencies within an image to address security vulnerabilities, and bugs, or to incorporate new features. As new vulnerabilities are discovered, patching images is essential to be able to ensure the security and stability of containers.

Image patching is a cornerstone of maintaining a secure and resilient Kubernetes environment. It involves regularly updating container images to address newly discovered security vulnerabilities, apply bug fixes, and integrate enhancements or new features. The dynamic

nature of software development means that vulnerabilities are constantly identified and disclosed. Therefore, continuous image patching is necessary to protect against potential exploits that could compromise containers, and by extension, the entire Kubernetes cluster. Effective image patching not only mitigates security risks but also ensures that applications run smoothly with the latest improvements and stability enhancements. A comprehensive image patching strategy should include the use of automated tools and processes to streamline updates.

4.2.1 Signing Images

Signing images allows us to ensure that no bad actor can introduce risk to the entire software supply chain by deploying compromised images to Kubernetes clusters.

In Kubernetes, a common approach is to use a combination of tools such as Sigstore (*sigstore* 2023) and Kyverno (*Kyverno* 2023). In this approach, Sigstore delivers a way to sign and verify container images and/or artifacts, providing key pairs to do so. Kyverno on the other hand runs as a dynamic admission controller in Kubernetes, validating the image singing, as only signed and verified images can be deployed to the clusters.

4.2.2 Immutable Images

Containers have wide-ranging execution privileges in their designated environments. If a bad actor gains unauthorized access, the container's capabilities can extend to tasks such as creating files, downloading scripts, and modifying applications. To prevent such post-exploitation actions, Kubernetes implements restrictions on a container's file system.

Although this confinement helps boost security, it can also negatively affect the normal functioning of legitimate container applications, causing crashes or unusual behavior. To avoid unintended consequences on valid applications, Kubernetes administrators can opt for a strategy of adding supplementary read/write file systems.

To implement this, Kubernetes comes with a default field `securityContext` which allow us to block writes to the filesystem with `readOnlyRootFilesystem: true`. We can then mount a specific volume with a writeable location if needed (NSA 2022).

4.3 Runtime Analysis

Runtime analysis in Kubernetes involves monitoring and analyzing the performance, health, and behavior of applications and containers running in a Kubernetes cluster during their execution. This is crucial for ensuring the reliability, scalability, and optimal performance of the applications.

The goal of Runtime Analysis is to identify and address vulnerabilities throughout the container lifecycle. According to Pothula, K. Kumar, and S. Kumar 2019, this involves gathering real-time data on user content within the container environment, as well as information from external sources such as firewalls, IDS/IPS, WAF, anti-malware, DLP, and logs. By analyzing this data, we can pinpoint security weaknesses.

In order to put this into practice, we need to log and monitor systems and send the data to a central monitoring center for analysis of container behavior. Pothula, K. Kumar, and S. Kumar 2019 recommends using Machine Learning to detect patterns and abnormal behavior

using large volumes of monitoring data. Establishing a reference baseline allows us to quickly flag containers that may be compromised.

This level of auditing can be achieved by using a SIEM such as Wazuh[1]. Tools like this enable security analytics, intrusion detection, and vulnerability detection in real-time (Wazuh 2023).

4.3.1 extended Berkeley Packet Filter (eBPF)

Traditionally, the operating system kernel has been a crucial domain for implementing observability, security, and networking features, given its privileged oversight of the entire system. However, the evolution of the kernel has been challenging due to its central role and stringent requirements for stability and security. This has historically limited the rate of innovation at the operating system level compared to external functionalities (*What is eBPF?* 2023).

"eBPF is a revolutionary kernel technology that allows developers to write custom code that can be loaded into de kernel dynamically, changing the way the kernel behaves [...]" enabling developers to deliver highly performant networking, observability and security tools (Rice 2023).

In Liu et al. 2020 the authors claim that a "[...] non-intrusive collection of user application L7/L4 layer network protocol interaction information based on eBPF, data collection of more than 10M throughputs per second can be achieved without modifying any kernel and application code, while the impact on the system application is less than 1%", with the possibility of using Machine Learning to "analyze and diagnose application network performance and problems, analyze network performance bottlenecks and locate specific instance information for different applications, and realize protocol-independent network performance problem location and analysis" (Pothula, K. Kumar, and S. Kumar 2019).

The use of extended Berkeley Packet Filter (eBPF) has gained significant attention in the computer science community. It has been suggested that eBPF can solve the kernel's feature creeping problem (*Feature creep* 2023). This is because eBPF enables developers to create isolated features that reside in the userspace and interact with the kernel, thereby keeping the kernel's fast-path fast. Although eBPF may initially add complexity to the kernel's code, the benefits of isolation are expected to compensate in the long term.

As noted in Sadiq et al. 2023, "eBPF provides a range of functionalities such as packet monitoring, tracing new processes, and detecting any event generated by the computer." This capability enables deep monitoring and boundary assertion, which has significant implications for implementing Zero Trust Architectures.

eBPF problems start with the fact that it interacts with the Linux kernel, meaning security liability if one of the eBPF services get compromised. In the 2021 hacker convention named DEFCON, an eBPF based rootkit with "command and control with remote and persistent access, data theft and exfiltration techniques, Runtime Application Self-Protection evasion techniques, and finally two original container breakout techniques" (DEFCONConference 2021) was presented, demonstrating the vulnerability of eBPF.

[1] https://wazuh.com/

4.4 Networking

As discussed in Section 2.2.2, Kubernetes Networking entails a complex task as it supports different levels of abstraction, ranging from pod-to-pod, pod-to-service, to container-to-container communication.

As network traffic flows from endpoint to endpoint within a Kubernetes cluster, it may include confidential and sensitive data that requires protection during transit, to prevent Man-In-The-Middle attacks. To address this concern, the implementation of Mutual Transport Layer Security (mTLS) has emerged.

4.4.1 mTLS

mTLS is a security protocol that enables mutual authentication between network entities involved in a connection. In the context of TLS, which is widely used for encrypting communications over the Internet, mTLS extends beyond traditional one-way authentication by requiring both the client and the server to present valid TLS certificates and authenticate using public/private key pairs. (*What is mTLS?* 2023).

In a standard TLS connection, the server provides its TLS certificate to prove its identity, and the client verifies this certificate before establishing an encrypted communication channel. However, in mTLS, the client also possesses a TLS certificate, and the server, in addition to presenting its certificate, authenticates the client's certificate. This two-way authentication ensures that both parties are who they claim to be, thereby enhancing the security of the communication.

4.4.2 Network Policies

As namespaces reveal insufficient for network isolation (Budigiri et al. 2021), network policies rise as a way to "provide the guardrails needed to restrict traffic between pods (in and/or across namespaces) as well as between pods and external networks, by explicitly specifying allowed and denied connections" (Budigiri et al. 2021).

Network policies are a way to define how pods are allowed to communicate with each other and other network endpoints. They provide a declarative way to specify the rules that determine which network traffic is allowed to and from a set of pods and are useful for controlling and securing the communication between different components in a Kubernetes cluster.

It is worth noting that within Kubernetes, the ingress and egress traffic can be restricted, given that all pods are unrestricted for egress by default (Cilium 2024d). Additionally, a study (Budigiri et al. 2021) demonstrated that the use of CNI plugins, such as Calico and Cillium, to enforce network policies in Kubernetes does not impose significant performance overheads. These findings are important for organizations that rely on Kubernetes to manage their containerized workloads, as they highlight the feasibility of implementing network policies without compromising the overall performance of the system.

As Network Policies natively have their limitations (Cilium 2024d), with these including the inability to force internal cluster traffic through a common gateway, handle TLS-related tasks, implement node-specific policies, target services by name (though targeting by labels is possible), create or manage "Policy requests" fulfilled by a third party, apply default policies to all namespaces or pods, conduct advanced policy querying and reachability analysis, log

network security events, explicitly deny policies (NetworkPolicies are a *deny by default* policy with only allow rules), and prevent loopback or incoming host traffic. Workarounds may involve using third-party solutions like as service mesh such as Istio[2] to enable Layer 7 network capabilities.

4.5 RBAC

One way of hardening a cluster is controlling the access to the API server. Changing a manifest in a cluster, for updating a deployment container image, for example, we need to be able to either:

- Run a kubectl patch command as referred in The Linux Foundation 2023b to update the object while it's running.
- Apply a new Deployment, Statefulset, or Daemonset to the cluster with a new image reference to update the current one.

Both of these methods require cluster access. Those who can run software on the deployment can harness computing resources, which may result in service disruptions and unauthorized access to data, as demonstrated in the well-known Tesla Cryptojacking incident (L. H. Newman 2023).

Implementing an RBAC is essential for securing a cluster, as RBAC " [...] is the most important authorization method for both developers and admins in Kubernetes" (Rice and Hausenblas 2018), as it manages and restricts access to cluster resources.

According to the official Kubernetes documentation, the RBAC framework relies on roles and cluster roles. Roles determine permissions within a specific namespace, providing access to resources such as services and pods (*Authorization Overview* 2023). Meanwhile, cluster roles extend these permissions globally throughout the entire cluster. To associate roles with users, groups, or service accounts, role bindings, and cluster role bindings are utilized, either within a namespace or on a cluster-wide scale. In RBAC, users, groups, and service accounts are considered entities, with service accounts facilitating pod interactions with the Kubernetes API. The framework operates on a set of verbs (such as get, list, and watch) and resources (such as pods and services), enabling the creation of specific permissions.

To ensure a reliable RBAC system, it is essential to adhere to best practices, including implementing the least privilege principle, allotting only the necessary permissions for each task, regularly testing and auditing RBAC policies, and utilizing namespaces to differentiate between various users, teams, and projects. In their *Tooling and Good Practices* section, Rice and Hausenblas 2018 compiled a list of tools for auditing RBAC policies.

4.6 GitOps

As mentioned in the RBAC section (see 4.5), updating an image in a cluster requires access. In 2017, Alexis Richardson, CEO of WeaveWorks, introduced GitOps (*The History of GitOps* 2023, *What's in a name?* 2023), which Beetz and Harrer 2022 describes as a model for operating cloud-native applications using Git as a single source of truth. This means that environments are operated solely through the contents of a Git repository, including "[...] creating, changing, and destroying environments". "GitOps increases application reliability

[2] https://istio.io/

because Git enables commits to be easily reverted and environment changes to be rolled back. [...] Git facilitates stronger security since it supports proving authorship and change origins by digitally signing commits". Storing all application information on Git provides not only a central location for the entire application flow but a rollback and sense of ownership since "[...] it supports proving authorship and change origins by digitally signing commits" (Beetz and Harrer 2022).

In a blog post by the GitOps introducing company (*Delivering Quality at Speed With GitOps* 2023), four principles of GitOps were introduced:

1. **The entire system is described declaratively.** With a declarative approach, the user defines the intended state of their system and the GitOps tools handle the rest, ensuring that the system reaches that desired state. This differs from an imperative approach, which involves providing specific instructions or procedures for making alterations.
2. **The canonical desired system state is versioned in Git.** All configurations, whether they pertain to infrastructure or applications, are securely archived in a Git repository. This version control mechanism facilitates the retention of a comprehensive history of modifications, rendering it feasible to discern, reverse, and scrutinize any alterations made to the system.
3. **Approved changes to the desired state are automatically applied to the system.** This process is typically facilitated by a continuous reconciliation loop that monitors the Git repository for changes and ensures that the actual state of the system aligns with the approved or desired state stored in the repository.
4. **Software agents ensure correctness and alert on divergence.** These agents are responsible for continuously monitoring the Git repository, detecting changes, and automatically applying those changes to the target environment.

A recently published survey by the Cloud Native Computing Foundation (CNCF)[3] in Cloud Native Computing Foundation (CNCF) 2023 reported that 31% of the respondents started using GitOps in their cloud environments during the past year and 60% have been using it for more than a year. The remaining are either evaluating them or don't plan to use them at all.

According to sources (*Pros and Cons of GitOps* 2023), one potential obstacle to implementing GitOps practices and values is the potential increase in complexity. The core concept of expressing the desired state declaratively within version control leaves little room for manual adjustments to the infrastructure, which can require a significant cultural shift. As stated in *Pros and Cons of GitOps* 2023, "GitOps represents a process change that requires a lot of discipline from all the participants and commitment to switching their approach".

GitOps can be divided into two different approaches for managing changes and updates in a system, particularly in the context of Continuous Integration (CI) and Continuous Delivery (CD): push-based and pull-based.

[3]The Cloud Native Computing Foundation (CNCF) is an open-source software foundation that promotes the development and adoption of cloud-native computing. It hosts critical projects like Kubernetes, Prometheus, and Envoy and focuses on creating scalable, resilient applications, fostering innovation, and setting industry standards for cloud-native technologies.

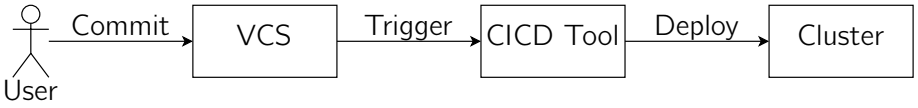

Figure 4.1: Simplified Push-Based GitOps Approach

4.6.1 Push-Based

In a push-based deployment model, illustrated in Figure 4.1 changes are actively pushed to the target system from a centralized source or by an external entity. Limoncelli 2018 describes a GitOps push-based as the following process:

1. Managing configuration settings in a Version Control System (VCS), typically Git. The configuration file, written in a declarative language, enables idempotent updates. External contributors, like customers, can propose changes via Pull Request (PR), facilitated by user-friendly documentation.
2. CI automatically runs tests on configuration files during both the submission and post-approval stages.
3. Human approval of a PR triggers the CI/CD process, where some initially manual tests are gradually automated.
4. Following PR approval and successful automated testing, a CD system takes charge, deploying the changes into the production environment.

The source of changes initiates the deployment process and pushes the changes to the target environment. This introduces the problems of allowing cluster access to the CI/CD tools and the idempotency of the processes applied to the cluster, which Ramadoni, Utami, and Fatta 2021 describes as "[...] security issues of a person's ability to directly access and change clusters and the ineffective rollback process in the application deployment process to an application platform". For CI/CD tools to access the cluster, it must be opened to run kubectl commands. The same doesn't apply to the pull-based approach (see 4.6.2) but doesn't solve either the security or the idempotency problem.

4.6.2 Pull-Based

In a pull-based deployment model, the target system actively retrieves or pulls changes from a centralized VCS. As illustrated in Figure 4.2, the desired state of the system is declared in a VCS repository, and the target system continuously checks for changes in the repository. When changes are detected, the system automatically pulls and applies those changes to converge to the desired state.

This approach is notoriously famous in Kubernetes because of the use of the built-in Kubernetes component *controller*, described in 2.2. This allows for the implementation of orchestration tools that make use of it. Attardi et al. 2018 refers to it as a tool that "analyzes the model, compares it with the current state of the system being deployed, determines the resources that need to be provisioned, generates a sequence of executable steps needed to bring the deployment in line with the model, and coordinates the execution of those steps until the system reaches a configuration that satisfies all constraints".

By transitioning from a traditional imperative model to a declarative model, organizations can achieve greater consistency and flexibility when modeling infrastructure, as well as the

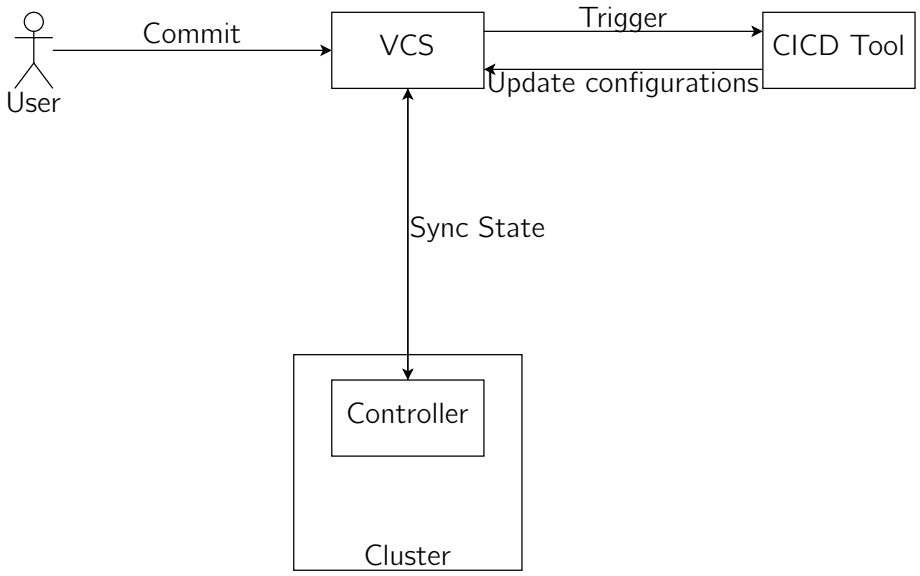

Figure 4.2: Simplified Pull-Based GitOps Approach

intricate connections between infrastructure, network, and application elements. The automation of translating these models into actual infrastructure and cloud-specific deployment operations falls on the orchestrator, freeing system administrators from the complexities of these tasks (Attardi et al. 2018). This paradigm shift can help streamline deployment processes and improve consistency.

According to a study by Ramadoni, Utami, and Fatta 2021, using a GitOps pull-based approach can solve the configuration drift problem caused by the push approach. With the controller checking the version control system for the desired state and applying it, there is no discrepancy between the desired and applied configuration. Additionally, this approach addresses security concerns such as direct cluster access by primarily managing access controls through the VCS repository and letting the orchestrator manage the appliance. According to the micro survey conducted by CNCF (Cloud Native Computing Foundation (CNCF) 2023), the primary security-based drivers for adopting GitOps pull-based are the reasons above stated. Specifically, 69% of the respondents cited the replacement of manual processes as the main reason, while 62% identified the removal of direct cluster access as their top priority. These two reasons make up the top two of the "What do you believe are/will be the biggest security benefits of GitOps", as shown in the results taken from the survey presented in Figure 4.3.

4.6.3 Tools

With the rise in popularity of GitOps, several tools such as ArgoCD and Flux have risen as well to fulfill market needs. Such tools use Kubernetes native controller to verify and restore the desired state. According to the study conducted by the CNCF in Cloud Native Computing Foundation (CNCF) 2023, both of these tools were the most used GitOps projects.

According to the official project documentation (*Architectural Overview - Argo CD - Declarative GitOps CD for Kubernetes* 2023), ArgoCD can manage multiple clusters simultaneously, allowing for the implementation of distributed architectures as described in Section 2.2.3.

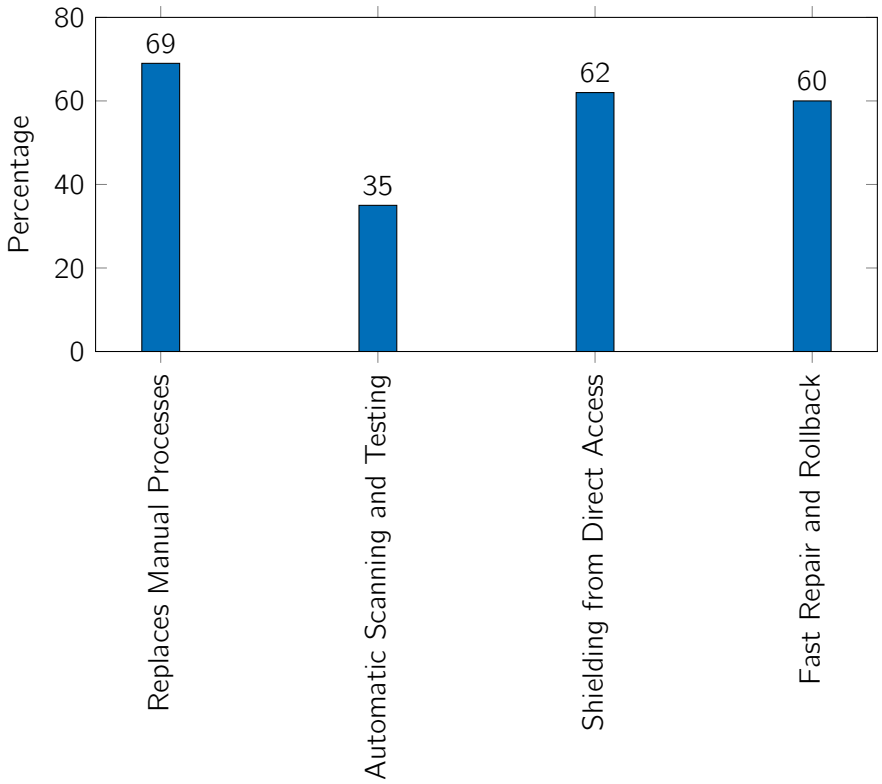

Figure 4.3: Security Benefits of GitOps

Source: Cloud Native Computing Foundation (CNCF) 2023

ArgoCD presents a centralized interface and API that provides users with a seamless view and management of applications across all connected clusters. It allows for cluster-specific configurations, and access control through RBAC, and ensures consistent deployment processes while adapting to each cluster's unique requirements.

4.7 Secret Management

The global digital landscape is witnessing a significant expansion, with millions of cryptographic keys emerging annually. This proliferation is not confined to public arenas like code-sharing platforms; rather, it extends into more secluded realms such as private repositories and corporate IT assets. International Business Machines Corporation (IBM) reported in IBM 2023 "Use of stolen or compromised credentials remains the most common cause of a data breach. Stolen or compromised credentials were the primary attack vector in 19% of breaches in the 2022 study and also the top attack vector in the 2021 study, having caused 20% of breaches. Breaches caused by stolen or compromised credentials had an average cost of USD 4.50 million". From their 2023 report, stolen credentials lost its *most common cause of a data breach* first place to phishing which accounts for 16% of the occurrences. Although stolen credentials came second with a drop to 15% compared to 2022, the average cost for the same primary attack vector upped to USD 4.62 million (IBM 2023).

Secrets are a way to securely manage sensitive information, such as passwords, API keys, and other confidential data, separately from the application code and configuration. These

cryptographic artifacts are essential for ensuring the security and privacy of communication within Kubernetes clusters. If secrets are accidentally disclosed or compromised, it can pose a critical threat to the security of the entire containerized infrastructure.

To solve the problem of exposing sensitive information, Kubernetes provides a resource called *Secrets* (*Secrets* 2023) to handle these sensitive pieces of information securely. The official documentation for Kubernetes tells us that as "Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods".

Secrets in Kubernetes consist of base64-encoded strings stored in the etcd (see 2.2). By default, this data is stored unencrypted so anyone with access to the API Server can read and write the contents of the secrets.

Official documentation (*Secrets* 2023) recommends taking the following measures to securely use secrets:

1. Use encryption at rest for secrets.
2. RBAC for secrets.
3. Restrict container access to secrets.
4. Consider the use of External Secret Providers.

4.7.1 Encryption At Rest

As outlined in the Secret Management section (4.7), secrets are currently stored unencrypted in etcd. However, Kubernetes offers a straightforward solution to encrypt secrets using a dedicated key (*Encrypting Confidential Data at Rest* 2023). While this method allows for encryption and decryption with a known key, it is not entirely secure, as keys can be leaked.

It is important to note that updating a key is a multi-step process that requires restarting the API-Server, which may result in downtime.

4.7.2 External Secret Provider

Kubernetes has native support for storing secrets, but it is also possible to store secrets externally using what is commonly referred to as *vaults*.

These repositories can range from simple key-value stores to more complex relational or non-relational databases, offering varying degrees of complexity (Idreos and Callaghan 2020).

One noteworthy feature of these platforms is their ability to handle a diverse range of secret formats securely. Whether you're dealing with basic key-value pairs or opting for more scalable database solutions, these systems are adept at encrypting and storing secrets.

One objective of these solutions is to reduce the problem of "secret sprawl." This happens when sensitive information like passwords, API keys, or cryptographic keys are spread across different systems, applications, and repositories in an uncontrolled and disorganized way. This occurs when these credentials are managed and stored in an ad-hoc manner without a centralized and structured approach to secret management. Examples of this include database credentials hardcoded code repositories and credentials in plaintext in configmaps.

Vault's creator, Hashicorp, has identified three major challenges associated with secret sprawl in their publication, HashiCorp 2023:

1. When secrets are dispersed across various repositories, it becomes difficult to keep track of which credentials are where leading to a level of confusion and uncertainty.
2. Traditional tools, like Wikis, Dropbox, and VCS, are not designed for secure secret management. As a result, they lack robust access controls and audit logs, making it challenging to monitor who accesses what secrets.
3. Dispersed secrets make it challenging to identify the source of leaked credentials. Without detailed access logs, it's difficult to determine whether a breach was caused by an insider. Remediation can also be complicated, particularly when credentials are hardcoded in source code, requiring intricate steps like code changes, recompilation, and redeployment.

Kicked off by GoDaddy, External Secrets Operator (ESO)[4] is a CNCF Sandbox level accepted project, which "[...] reads information from a third-party service like AWS Secrets Manager and automatically injects the values as Kubernetes Secrets" (*external-secrets* 2023). This operator enables users to securely store secrets in a centralized location and access them in a controlled manner from applications.

ESO introduces two new CustomResources, SecretStore, and ExternalSecret, which allow for the declarative specification of the Secret Storage location and the respective wanted secrets. In doing so, we can have it inject these secrets into the cluster as needed, as illustrated in Figure 4.4. It is worth noting that ESO is designed to guarantee that the secret values remain in sync with the external API, thus ensuring the accuracy and consistency of the secrets across the system.

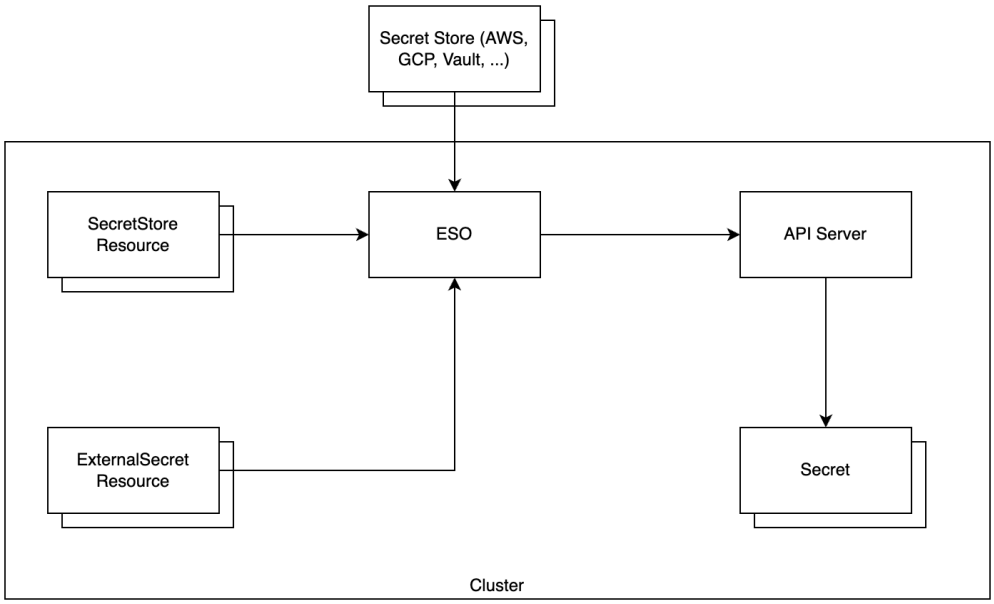

Figure 4.4: ESO Architecture

[4] https://external-secrets.io/

While this is a solid approach to managing secrets in a scalable environment, it does come with some security risks that could be exploited by malicious individuals. In their official documentation, ESO thoroughly analyzes the threat model of their architecture and identifies seven types of potential threats. As illustrated in Figure 4.4 and on their official threat modeling page, allowing operators access to the *kube-apiserver* could create a potential vulnerability, as it grants them permission to read and write secret resources across all namespaces within a cluster (*Threat Model* 2023).

For solving this centralized problem of having a single instance managing the cluster secrets coming from external secret stores, products such as *Vault* introduced the *sidecar pattern* into secret injection.

Sidecars in Kubernetes are an additional container deployed alongside a primary container within the same pod. The primary container is the main application or service, while the sidecar provides supporting functionalities, such as logging, monitoring, or data syncing. Both containers in the pod share the same network namespace and can communicate with each other through localhost.

By utilizing the power of Vault, a shared volume can be mounted within a pod through a sidecar implementation, typically established at `/vault/secrets`, allowing for secrets to be shared with other containers residing in the pod. Within this process illustrated in Figure 4.5, Vault generates two containers - one init and one sidecar. The init container is responsible for preloading the volume with the necessary secrets, while the sidecar oversees authentication and synchronization of the secrets. Additionally, the sidecar container is responsible for maintaining secure communication with Vault, facilitated through TLS certificates (Hashicorp 2022).

Although other methods of authentication can be configured, authentication with Vault instance is done using a service account attached to the Pod.

4.7.3 PKI

Public Key Infrastructure (PKI) is a foundational framework designed to manage cryptographic keys and digital certificates in a manner that ensures the security and authenticity of digital communication. The primary components of PKI include public keys, which are openly shared for encryption and signature verification, private keys, which are securely retained and used for decryption and signature creation, and Certificate Authority (CA), which validates identities and generates certificates.

A pivotal element of PKI is the digital certificate, a document that binds a public key to an individual or entity, signed by a trusted CA, allowing for creating secure communications with identity validation.

Many secrets management platforms incorporate external identity management with Public Key Infrastructure (PKI) to authenticate recipients of secrets and enable the transmission of PKI-encrypted payloads. Within this operational framework, the platform identifies the intended recipient of a particular secret and encrypts the content using their public key. This ensures that only the designated recipient can decrypt the secret, using their private key (Securosis 2018).

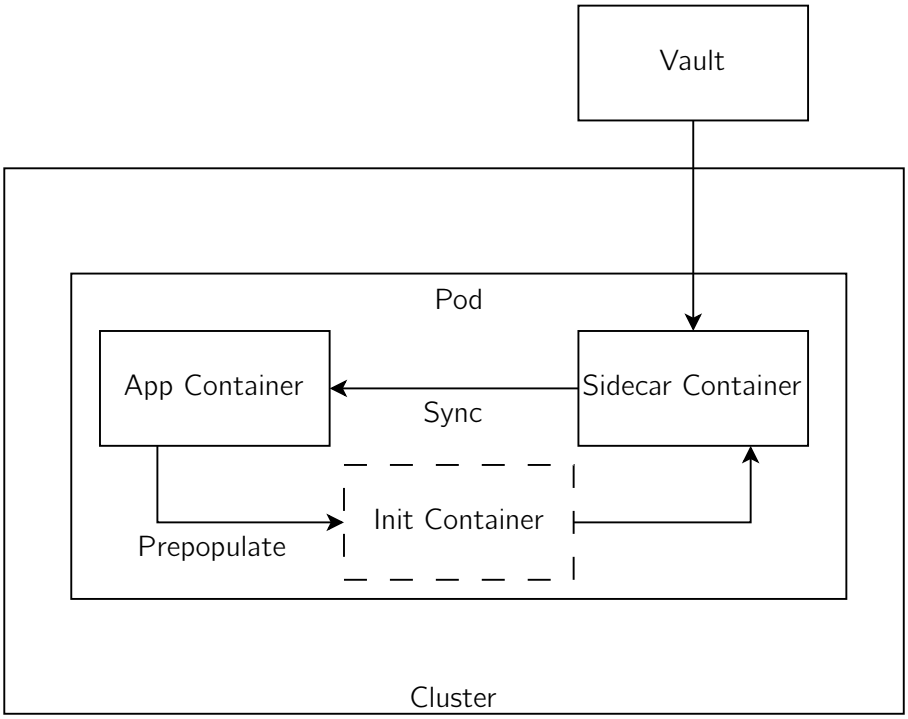

Figure 4.5: Sidecar Pattern Using Vault

4.8 Zero Trust in Kubernetes

Zero trust is a security paradigm that assumes no implicit trust is granted to systems or users based solely on their network location. This approach mandates strict verification of every request as though it originates from an open network. In the dynamic and distributed environments of Kubernetes, zero trust becomes essential to safeguard microservices and data.

Today's cloud-native applications are often distributed across multiple Kubernetes clusters. These applications are updated frequently and can dynamically scale based on user demand, which allows them to achieve resource efficiency without relying on co-location or the underlying infrastructure. However, this distribution increases the attack surface and the risk of security breaches due to the presence of multiple entry points. It is crucial to establish access policies and ensure secure communications among these distributed applications to prevent data loss, theft, forgery, and mishandling, thereby reducing significant business risks.

The implementation of Zero Trust (ZT) at the infrastructure level enables developers to direct their focus towards the development of application code, reducing the need to extensively consider the security aspects of deployment. This freedom grants developers the opportunity to invest their time in constructing environment-agnostic applications.

In a 2016 publication (Kindervag 2016), Kindervag emphasized the importance of internal network security, stating, "We've been so worried about the perimeter, we've forgotten about the malicious user on our internal network." It was noted that while companies have traditionally focused their security controls on the perimeter, it is crucial to extend these controls to encompass the internal network as well. Kindervag further highlighted the need for appropriate controls to be deployed across the entire network, with a specific emphasis on logging all network traffic. As referred in section 3.1, this approach calls for the need the

implementation of a controller or supervisor to enforce encryption, logging, and observability between workloads, authenticating and authorizing communications. The protocol to use is mTLS, as described in section 4.4.1. Two approaches to workload security in Kubernetes have been suggested: sidecar and sidecarless.

4.8.1 Sidecar Approach

Pods in Kubernetes are the smallest manageable units of computing, capable of supporting one or more containers. This allows for the co-scheduling of multiple containers that can access the same network interface. An emerging practice involves the use of dedicated helper containers, known as sidecars, to handle auxiliary tasks such as logging, configuration, and observability. Sidecars are tightly integrated with the primary application container and share the same lifecycle.

The sidecar approach entails deploying a proxy alongside each microservice within a Kubernetes pod, responsible for managing security-related tasks such as encryption, authentication, and authorization of network traffic. This model is commonly associated with service meshes like Istio, Linkerd, and Consul Connect, which provide a comprehensive framework for implementing zero-trust security by managing communication between services consistently and securely.

While it is feasible to integrate sidecar functions within the main container, there are distinct advantages to using separate containers. This approach allows for granular security control by assigning each microservice its own dedicated sidecar, enabling fine-tuned management of traffic policies and security configurations. Furthermore, it isolates security functions from the application code, streamlining the application development process. Additionally, the sidecar approach offers flexibility, as sidecars can be independently configured to accommodate customized security policies for each microservice. Moreover, sidecars facilitate comprehensive logging and monitoring of network traffic, aiding in the identification and analysis of potentially malicious activities. This separation also allows for adjustments to the containers' control groups, ensuring that CPU resources are prioritized for the main container.

However, there are challenges associated with this approach. Deploying a sidecar for each microservice may lead to increased resource consumption, resulting in higher operational costs. A study presented at KubeCon 2024 (Sun 2024) highlighted a substantial 70% increase in costs when using sidecars, impacting companies with up to a 30% rise in expenses (Figure 4.1). Managing numerous sidecars may introduce operational complexities, particularly in large-scale deployments. Furthermore, the addition of an extra network hop by sidecars can cause heightened latency in communication between services.

Table 4.1: Resource Utilization and Cost Comparison

Configuration	Reserved vCPU	CPU Utilization (%)	Reserved Memory (GB)	Memory Utilization (%)	Nodes	Cost / Month
No Mesh	132	32%	17	17%	17	$3326.45
Sidecars	226	35%	29	55%	29	$5674.53
Ambient (L4 Only)	132	40%	17	19%	17	$3326.45
Ambient + Waypoint Proxy	142	39%	18	22%	18	$3522.12

Source: Sun 2024

4.8.2 Istio

Istio is an open-source service mesh designed to enhance distributed applications by enabling secure, efficient, and transparent service-to-service communication. It provides features such as automatic load balancing for different types of traffic, fine-grained traffic control, and a flexible policy layer for access control and rate limiting. Additionally, Istio facilitates automatic metrics, logs, and traces for all traffic within a cluster. Its control plane is built to be extensible and runs on Kubernetes, allowing seamless integration with other clusters, VMs, and external endpoints (Istio.io 2024).

Istio Architecture (Figure 4.6) comprises two main components: the data plane, which includes Envoy proxies deployed alongside each service in the cluster, and the control plane which manages and configures the proxies to direct traffic. The proxies intercept all network traffic for the service, implementing features such as mTLS and traffic management, while also sending logs and other telemetry data to the control plane.

Pilot communicates with the Envoy proxy using the Envoy API and is responsible for managing traffic, handling routing, and performing service inspection. Citadel plays a crucial role in enabling secure inter-service communication by managing user authentication and handling certificates and credentials. Lastly, Galley is responsible for managing configurations, processing them, and distributing them to Envoy proxies.

4.8.3 Linkerd

Similar to Istio, Linkerd is an open-source service mesh comprised by two main components, control plane and data plane, as illustrated in Figure 4.7.

The control plane, running in a dedicated Kubernetes namespace (default: linkerd), consists of several key components. The destination service aids data plane proxies by providing necessary information for service discovery, policy enforcement, and service profiling, which influences metrics, retries, and timeouts. The identity service acts as a TLS Certificate Authority, managing certificate signing requests (CSRs) from proxies and issuing certificates that enable mutual TLS (mTLS) for secure proxy-to-proxy connections. Additionally, the proxy injector functions as a Kubernetes admission controller, handling webhook requests when pods are created, and inspecting for specific annotations to inject the necessary Linkerd proxy and initialization containers (Linkerd 2024a).

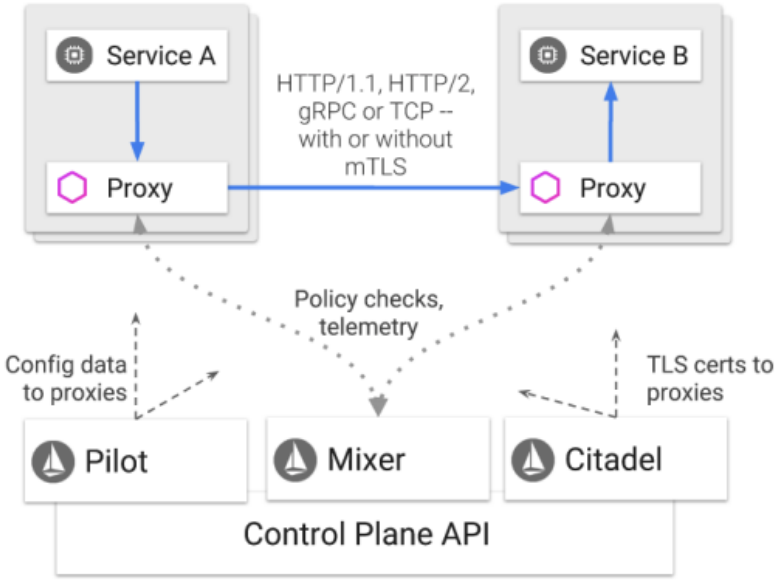

Figure 4.6: Istio Architecture

Source: Cilium 2024c

The data plane consists of ultralight micro-proxies, written in Rust, deployed as sidecar containers alongside application pods. These proxies use iptables rules to transparently intercept TCP traffic to and from pods, handling protocols like HTTP, HTTP/2, TCP, and WebSocket. The Linkerd2-proxy, designed specifically for the service mesh use case, features transparent proxying with zero configuration, automatic Prometheus metrics export, latency-aware layer-7 load balancing, layer-4 load balancing for non-HTTP traffic, automatic TLS encryption, and an on-demand diagnostic tap API.

While Envoy's flexibility and general-purpose design contribute to its popularity, they also introduce a level of complexity that conflicts with Linkerd's design goals (Linkerd 2024b).

In terms of resource consumption, Linkerd2-proxy is optimized to be highly efficient, a critical consideration given the nature of sidecar proxies in a service mesh. One of this thesis's focal points is to benchmark and compare these technologies, as optimization is crucial for maintaining the performance and cost-effectiveness of the service mesh as it grows.

Security is another pivotal factor in this architectural choice. Linkerd2-proxy is built using Rust, a programming language that offers inherent memory safety advantages over C++, which is used in Envoy. This choice significantly reduces the reliance on manual security practices and results in fewer vulnerabilities, aligning with Linkerd's emphasis on robust security measures. By minimizing human error and enhancing code safety, Rust allows Linkerd2-proxy to deliver a more secure data plane.

4.8.4 Sidecarless Approach

In contrast to the sidecar approach, the sidecarless model simplifies the deployment of microservices by embedding auxiliary tasks directly within the primary application container

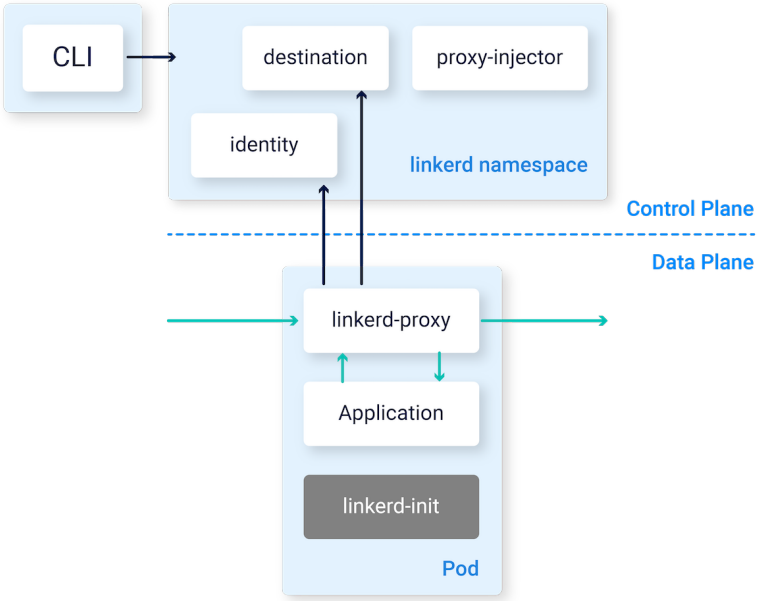

Figure 4.7: Linkerd Architecture

Source: Linkerd 2024a

or by utilizing shared infrastructure components provided by the Kubernetes ecosystem. This approach eliminates the need for additional containers within a pod, thereby reducing resource overhead and operational complexity.

One significant advantage of the sidecarless approach is its efficiency in resource utilization. Without the need for multiple containers per pod, it minimizes CPU and memory consumption, leading to potentially lower operational costs. This approach can be particularly beneficial in high-density environments where optimizing resource allocation is critical. Additionally, the sidecarless model can reduce the complexity of managing and orchestrating numerous sidecar containers, streamlining the deployment and scaling processes.

This approach is commonly used by eBPF solutions in a per-node approach using a Daemonset rather than a per-pod as the case of in sidecar approach.

4.8.5 Cilium

Not only a service mesh, Cilium is one of the most advanced and powerful CNI plugins for Kubernetes. It establishes a virtual Ethernet device for each Pod and designates one side of the link as the default interface in the Pod's network namespace. Subsequently, Cilium attaches extended eBPF programs to the ingress traffic control hooks of these virtual Ethernet devices to intercept all incoming packets from the Pod. The intercepted packets are processed before reaching the network stack and IPTables, resulting in reduced latency by up to around 30% as demonstrated in the conference paper Budigiri et al. 2021.

Cilium has two core components: an agent running as a Daemonset and an operator. The agent daemons subscribe to events from the Kubernetes API and are responsible for managing containers' networking and eBPF programs. The operator handles all management operations that need to be executed once for the entire cluster, rather than once for each Node (Cilium 2024b).

A CNI and eBPF, Cilium provides CRDs for implementing Network Policies. Although it has been demonstrated to have better performance than the native kube-proxy, those network rules operate at Layer 4 of the OSI model, and for implementing HTTP ruling it is needed Layer 7 proxying as done by the solutions previously referred, Istio and Linkerd. To implement L7 features such as mutual authentication, Cilium proxies the traffic through an Envoy instance present in each of the Cilium Agent pods from the Daemonset making all the L7 traffic policies depend on the availability of the specific node pod (Cilium 2024d).

To address the challenge of identity verification in Kubernetes, Cilium incorporates Secure Production Identity Framework for Everyone (SPIFFE), allowing dor trustworthy identity issuance, identity attestation and dynamic scalable environments (Cilium 2024a).

In a SPIRE setup integrated with Cilium, obtaining a SPIFFE identity involves a well-defined process centered around the interaction between workloads, SPIRE agents, and the SPIRE server. The central components of this system include the SPIRE server, which serves as the root of trust for the domain, and the SPIRE agent, which operates on each node. The SPIRE agent first acquires its identity from the SPIRE server and then handles identity requests from workloads running on its node.

To manage mTLS implementation, when a workload starts up, it connects to the local SPIRE agent using the SPIFFE workload API and provides its identity details. The SPIRE agent then validates the workload's identity by ensuring that the workload is running on the correct node and that its labels match the expected set. After validation, the SPIRE agent attests the workload's identity request to the SPIRE server. Upon verifying the request, the SPIRE server issues a SPIFFE Verified Identity Document (SVID) back to the SPIRE agent. This document includes a TLS key pair in the X.509 format. The SPIRE agent then delivers the SVID to the workload, completing the identity issuance process. This process is illustrated in Figure 4.8.

In environments where Cilium is used, Cilium agents acquire a common SPIFFE identity. These agents are capable of requesting identities on behalf of other workloads, thereby streamlining the identity management process. This integration allows for efficient and secure handling of identity requests, ensuring that workloads can reliably obtain and use SPIFFE identities within the SPIRE system.

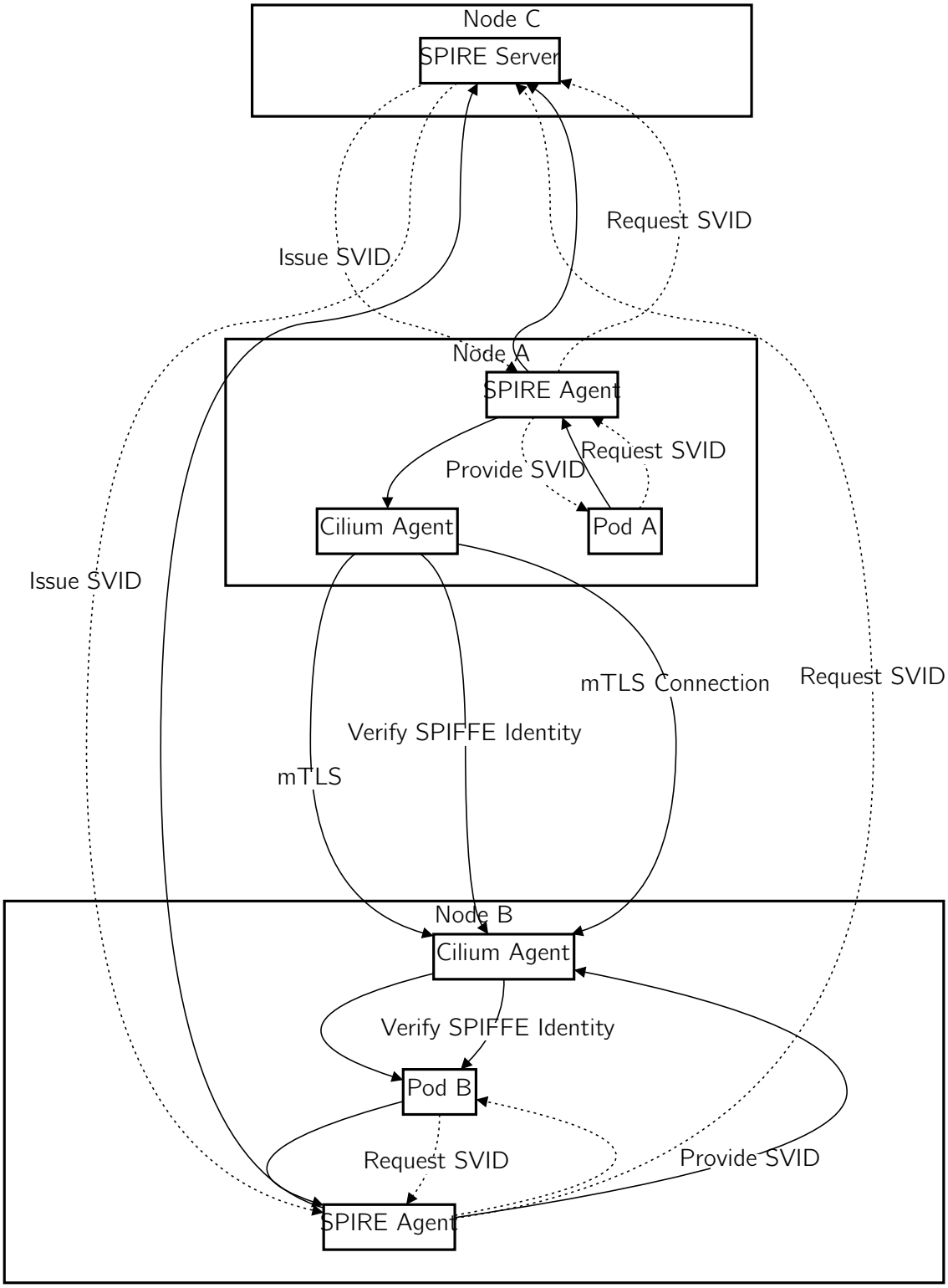

Figure 4.8: mTLS in Cilium using SPIFFE and SPIRE Server

Chapter 5

Performance Evaluation and Analysis

5.1 Testing Scenarios

The performance evaluation of Zero Trust implementations in Kubernetes will encompass a range of carefully designed scenarios to assess their impact on various performance, security, and operational aspects.

To ensure accurate comparisons, initial tests will establish baseline performance metrics without Zero Trust technologies enabled. These tests will use various types of workloads (request/response) under different load conditions (low, medium, high). The data obtained will provide a reference point to measure the overhead introduced by Zero Trust components. From here, evaluations of different components to gather insight of the metrics will be made.

5.1.1 Network Policy and Security

A series of tests will evaluate the impact of network policies with increasing levels of complexity. Starting with basic allow/deny rules, gradually introducing policies that cover multiple namespaces, fine-grained controls (HTTP method restrictions, header-based filtering), and complex traffic patterns. These tests will highlight how policy granularity affects performance and resource consumption.

Scenarios will contrast traffic flows with and without authentication and authorization enforcement. These tests will examine the overhead of identity verification, token validation, and complex policy evaluation at different layers.

The evaluation of service mesh efficiency, network security, and policy enforcement is crucial for understanding the performance of service meshes such as Istio, Linkerd and Cilium in modern cloud-native environments. In this regard, it is necessary to assess the resource consumption of control plane components and sidecar proxies of Istio, and to analyze the overhead of mTLS encryption and policy enforcement. Similarly, the resource efficiency of Linkerd compared to Istio, with a focus on lower footprint, and the performance impact of its mTLS and traffic control features must be measured. Cilium's network-layer approach to policy enforcement and potential service mesh functionalities should also be analyzed, along with its impact on latency compared to Istio/Linkerd and its resource efficiency due to the extended extended Berkeley Packet Filter (eBPF) usage. Such an evaluation can provide important insights into the effectiveness of service meshes in ensuring network security and policy enforcement in modern cloud-native environments. The Key Performance Indicators will be CPU Usage, Memory usage and Request Latency.

5.1.2 Service Discovery

Central to the dynamic nature of Kubernetes, tests will be conducted focusing on service discovery mechanisms. These will involve scenarios such as deploying new services, scaling services up/down, and simulating service failures. Metrics will center on discovery latency, endpoint accuracy, convergence time after changes, and the overall resource usage of service discovery components.

5.1.3 Resource Efficiency

A crucial aspect of the evaluation will be measuring the resource consumption of the various Zero Trust technologies. Tests will specifically track the overhead incurred by control plane components, such as policy engines and certificate authorities. Additionally, the impact of sidecar proxies (if used by technologies like service meshes) on individual pod resource usage (CPU and memory) will be carefully monitored. These metrics will also be measured as the system scales the number of nodes and pods in the Kubernetes Cluster helping, to determine the trade-offs between security benefits and the potential resource footprint of Zero Trust solutions within Kubernetes.

5.1.4 Scalability and Resilience

To assess behavior under real-world conditions, testing the impact of scaling both the number of nodes in the Kubernetes cluster and the number of deployed pods will be done. Observations will focus on how well Zero Trust technologies maintain performance and security posture under increased load. Additionally, failure scenarios will be simulated to observe the resilience of ZT solutions and their ability to maintain a secure state when components malfunction.

5.1.5 Observability and Auditing

For this evaluation, Prometheus will act as the core metrics collection tool due to its popularity, open-source nature, and widespread adoption within Kubernetes environments. To gather comprehensive data, metrics will be collected from several sources. The Zero Trust technologies themselves often provide Prometheus-compatible endpoints, exposing metrics related to latency and resource consumption. When the technology of the service being deployed doesn't export metrics or is not discovered natively by Prometheus, custom metrics will be coded into the service, and service monitors will be created. Additionally, Kubernetes-level metrics will be captured via the Kubernetes metrics API and exporters like node-exporter and kube-state-metrics. These will offer insights into cluster-wide resource utilization and pod-level metrics.

Grafana will serve as the visualization platform, enabling the creation of tailored dashboards for monitoring and analysis. These will include real-time monitoring dashboards to display crucial metrics like request latency, error rates, throughput, and resource usage of Zero Trust components. This real-time visibility will be essential for identifying any performance bottlenecks that may arise. Comparative analysis dashboards will be vital for visualizing performance differentials between the various Zero Trust implementations as test scenarios increase in complexity (policy granularity, varying authentication/authorization requirements). Security-focused dashboards will highlight policy enforcement statistics, successful

vs. blocked attack attempts, and audit logs, offering insights into the effectiveness of the different ZT approaches.

5.1.6 Specific Zero Trust Technologies and Evaluation

The projects selected for the work were based on their maturity level in CNCF and their suitability for the evaluation conducted. The evaluation covered two main areas:

- Identity and Access Management (IAM).
- Network Security and Policy Enforcement.

For IAM, the focus is on mTLS. The aim is to compare the authentication latency and overhead of the mTLS implementation on different technologies with native Kubernetes authentication, measuring the impact on request latency due to key validation.

In the area of Network Security and Policy Enforcement, the evaluation covered Service Meshes (Istio, Linkerd, Cilium) and Gatekeeper[1] with Open Policy Agent (OPA). With Istio, the aim was to evaluate the resource consumption of control plane components and sidecar proxies and to analyze the overhead for mTLS encryption and policy enforcement. With Linkerd, the goal was to assess its resource efficiency compared to Istio, focusing on a lower footprint, and to measure the performance impact of its mTLS and traffic control features. Cilium was evaluated for its network-layer approach to policy enforcement and potential service mesh functionalities, its impact on latency compared to Istio/Linkerd, and its resource efficiency due to eBPF usage. Calico was tested for the overhead of its network policy enforcement compared to native Kubernetes network policies. Finally, Gatekeeper was assessed for the impact of integrating it for policy enforcement on authorization latency and control plane resource usage.

For runtime security the evaluated tool will be Falco[2], measuring the resource usage of *modern-era* runtime security trough the use of eBPF technology and its impact on the overall system usage.

5.1.7 Benchmarking Tools

After conducting a thorough evaluation of various testing tools, a lack of consensus on the standard benchmarking criteria for these technologies was discovered. Many testing tools rely on synthetic tests created with testing suites that do not accurately simulate real-world workload resource usage, as shown in a similar study published on the official Linkerd blog (William Morgan 2024).

In response to this challenge, a Python application that can send requests to a specific endpoint and measure the response latency was developed. The application then exports this information as a Prometheus metric. The design of the application allows for an incremental increase in the number of requests sent per second, enabling us to monitor the evolution of resource usage and latency as the tests progress.

[1] https://open-policy-agent.github.io/gatekeeper
[2] https://falco.org/

5.1.8 Methodology

In the earlier sections of section 5.1, it was specified that we will conduct benchmarks using a custom-developed tool to measure latency and resource usage. The initial test will entail sending requests from point A to point B. Figure 5.1 depicts the data flow between points A and B in a sidecar approach to a ZT service mesh implementation. In contrast, Figure 5.2 visualizes the same process without sidecars, utilizing Cilium in this specific scenario.

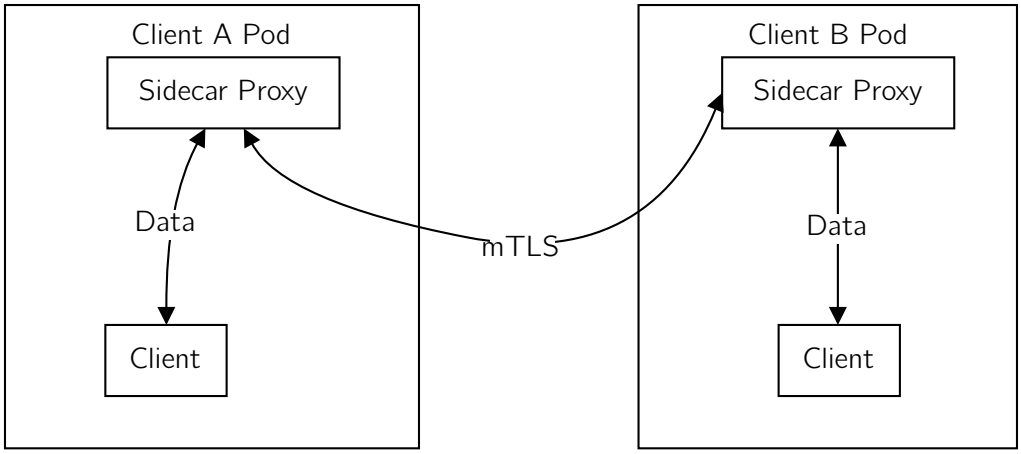

Figure 5.1: mTLS A to B using sidecars

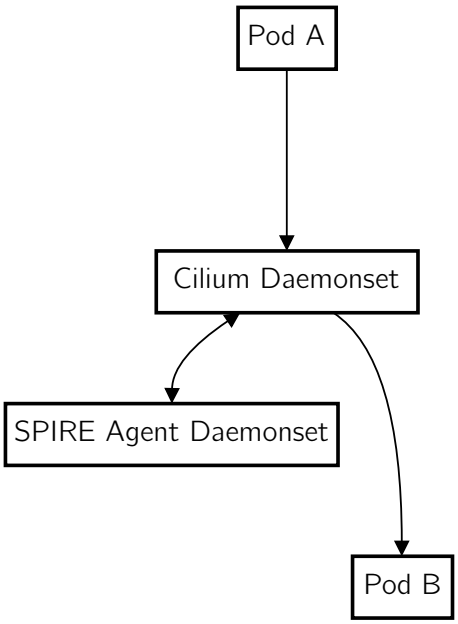

Figure 5.2: mTLS A to B to C without using sidecars (Cilium)

Given that certain technologies make use of sidecars, we will assess resource usage on a per-service basis, as each service must host a sidecar. Therefore, another test will entail initiating requests from point A to point B and then rerouting them to C. Figure 5.3 represents the data flow from points A, B, and C in a sidecar-based ZT service mesh implementation. In contrast, Figure 5.4 illustrates the same process without sidecars, using Cilium in this particular scenario.

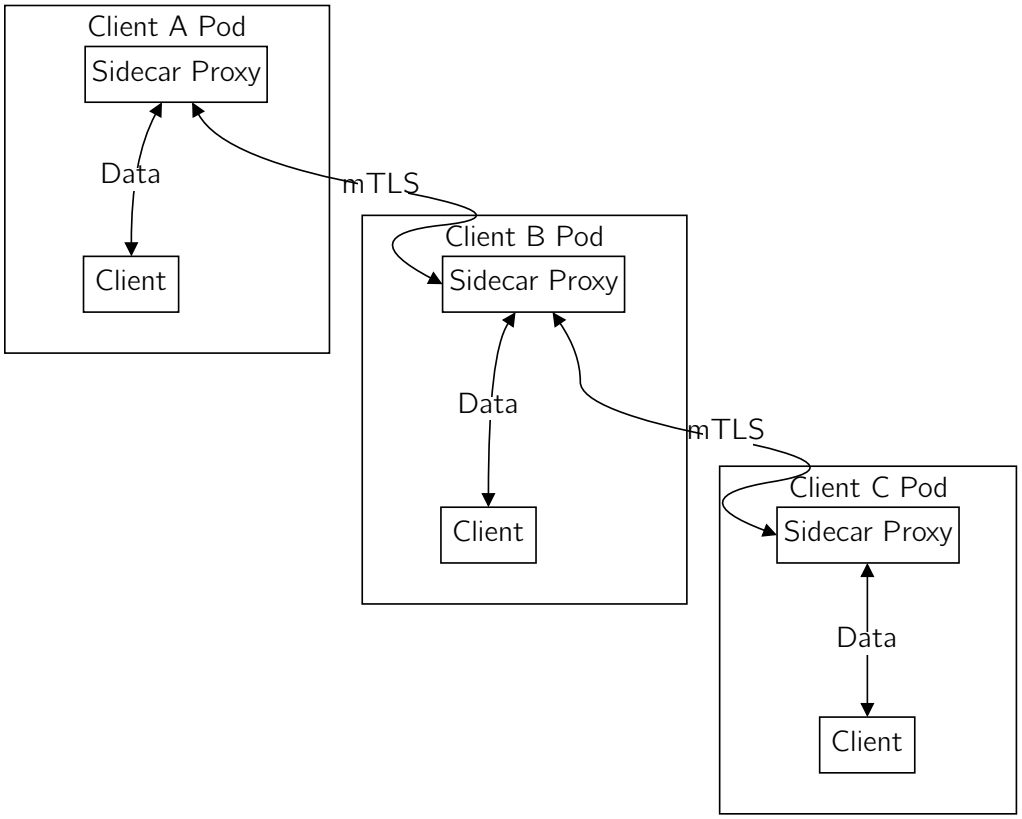

Figure 5.3: mTLS A to B to C using sidecars

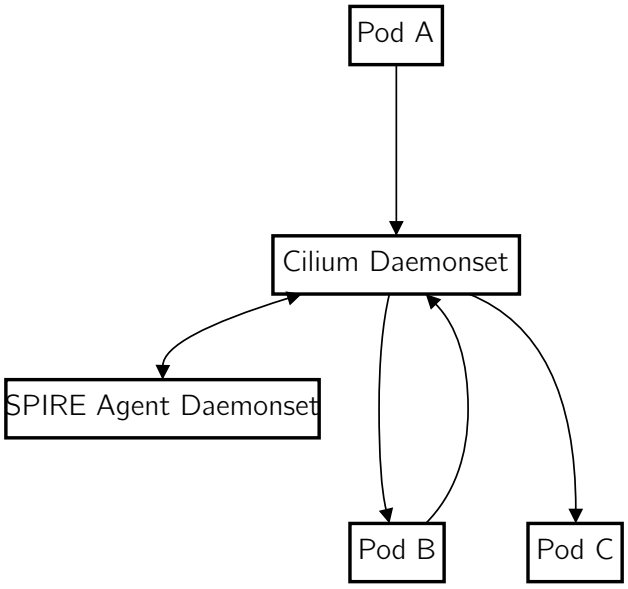

Figure 5.4: mTLS A to B to C without using sidecars (Cilium)

This benchmarking methodology allows us to compare sidecar and sidecarless implementations of service meshes in Kubernetes.

For Gatekeeper benchmarking we will be assigning all the Pod Security Policies from the tool's official page and will be measuring the impact on resource usage as policy breaking pods are

scheduled into the cluster. Firstly we will implement basic policies such as requiring specific labels on pods or restricting certain container images. Then we will gradually introduce more complex policies that involve multiple conditions and constraints, measuring the resource usage of Gatekeeper components as well as Pod creation time for each of the policies.

For benchmarking Falco resource usage we will be only using the official stable policies following the rest of the thesis on benchmarking only production-ready and production-approved tools and configurations. With this, a pod that violates policies will be deployed and the resource usage of Falco will be measured.

5.2 Environment Setup

The environment setup is a critical component of any performance evaluation, particularly when testing Zero Trust implementations in Kubernetes. A well-configured environment ensures that the tests are reliable, reproducible, and accurately reflect real-world conditions. This section outlines the significance of having the right environment setup and provides a framework for configuring the testing infrastructure on Google Kubernetes Engine (GKE) clusters.

Consistency and Reproducibility

Consistency in the environment setup is crucial for obtaining reproducible results. Variations in hardware, software configurations, or network conditions can introduce inconsistencies, making it difficult to attribute performance changes to the Zero Trust technologies being evaluated. By standardizing the environment, we ensure that any observed performance differences are due to the technologies under test rather than external factors.

Realistic Testing Conditions

A properly configured environment simulates real-world conditions, providing insights into how Zero Trust implementations will perform in production. This includes setting up representative workloads, as referred in 5.1.7, realistic network configurations, and appropriate security policies. Such a setup helps in identifying potential bottlenecks and performance impacts that might not be apparent in a simplified testing environment.

Comprehensive Monitoring and Observability

Effective performance testing requires detailed monitoring and observability. The environment setup includes Prometheus and Grafana to collect and visualize metrics, trace requests, and monitor the health of the system. This comprehensive visibility is essential for diagnosing performance issues and understanding the impact of Zero Trust components on system behavior.

Automation and Efficiency

Automating the environment setup using Terraform ensures efficiency and consistency. Automation reduces the risk of human error, speeds up the deployment process, and allows for quick reconfiguration and scaling as needed. This is particularly important when running extensive performance tests that require frequent setup and teardown of the environment.

5.2.1 Compliance

Setting up the environment with appropriate security configurations is vital, especially when evaluating security-centric technologies like Zero Trust. The setup Kubernetes cluster is done according to the **CIS Google Kubernetes Engine (GKE) Benchmark v1.5.0**. To audit, it was used a combination of reports from kube-bench[3] and manual testing.

The official Kubernetes CIS Kubernetes Benchmark was not implemented as this benchmark contains recommendations and controls which the end-user is not able to view or modify in GKE, as Google uses a shared responsibility model where the control plane, including the control plane VMs, API server, etcd, kube-controller-manager, kube-scheduler and nodes operating systems are managed by Google itself (Google 2024).

A secure testing environment not only protects the integrity of the tests but also provides a realistic assessment of the security posture of the Zero Trust solutions.

5.2.2 Hardware and Software Configuration

For the purpose of performance testing, various nodepools were employed to evaluate different service mesh implementations, separated by Taints and NodeSelectors (The Linux Foundation 2024). Each nodepool consisted of a single e2-highcpu-32 virtual machine from the Google Cloud Platform, featuring 32 virtual CPUs and 32 gigabytes of RAM. Further detailed specifications of these virtual machines can be found in the official Google Cloud Platform documentation[4].

The provisioned clusters were running on version 1.27.7, which is the latest stable Kubernetes version provided by GKE that is included in the CIS Google Kubernetes Engine (GKE) Benchmark v1.5.0. The container runtime used was containerd, which is under active development and comes as the default in GKE managed clusters. As shown in Espe et al. 2020, containerd has been shown to provide the best performance among the available choices.

5.3 Results Analysis

This section aims to conduct a thorough analysis of the data and processes pertinent to our study. This analysis will involve both quantitative and qualitative approaches, enabling us to identify areas for improvement and establish the groundwork for targeted solutions.

[3] https://github.com/aquasecurity/kube-bench
[4] https://cloud.google.com/compute/docs/general-purpose-machines

5.3.1 Service Mesh A-B Benchmark

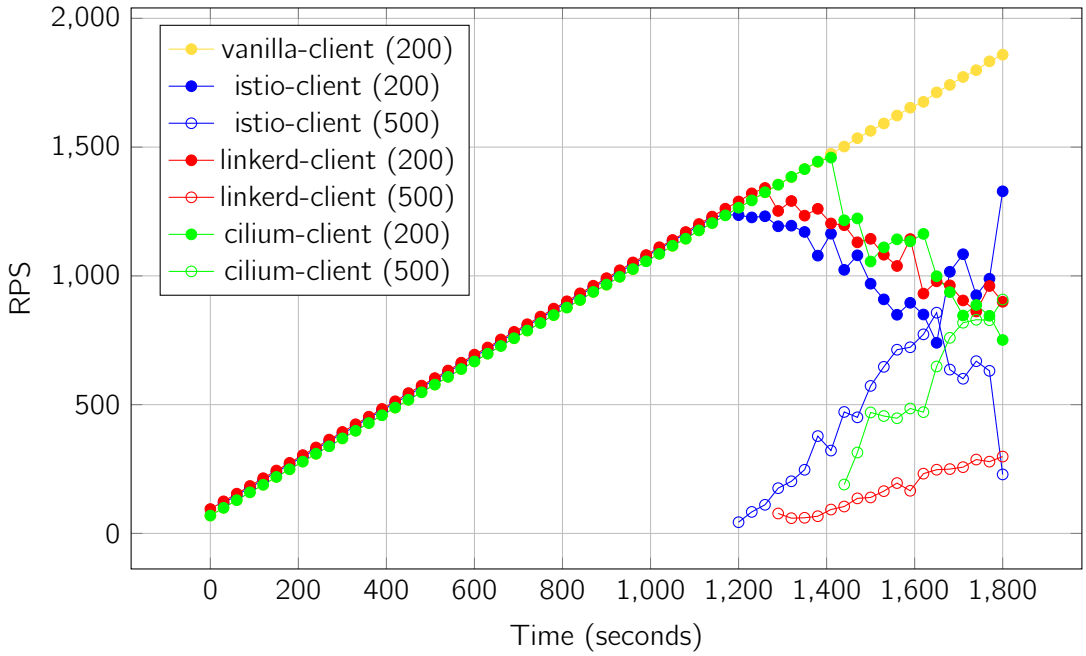

Figure 5.5: Requests per second (A–B)

The Figure 5.5 represents the Requests per Second of the different service meshes over a 30-minute interval. This figure enables us to assess the capacity for processing requests of each technology. The figure is relative to the benchmark from point A to point B as described in the methodology section (5.1.8).

Through the analysis of Figure 5.5 and Table 5.1 Istio sustained an average of 815.33 successful requests per second throughout the 30-minute test duration, achieving a maximum throughput of 1328.33 requests per second. However, HTTP 500 errors began to appear at the 20-minute mark (1200 seconds). Linkerd maintained an average of 828.45 successful requests per second over the same duration, with a peak throughput of 1341.43 requests per second and HTTP 500 errors starting at the 1290-second mark. Cilium showed a maximum throughput of 1459.37 requests per second and an average of 825.52 successful requests per second, with HTTP 500 errors occurring at the 1440-second mark. The vanilla client got a 100% success rate across the entire interval, with no drop in performance.

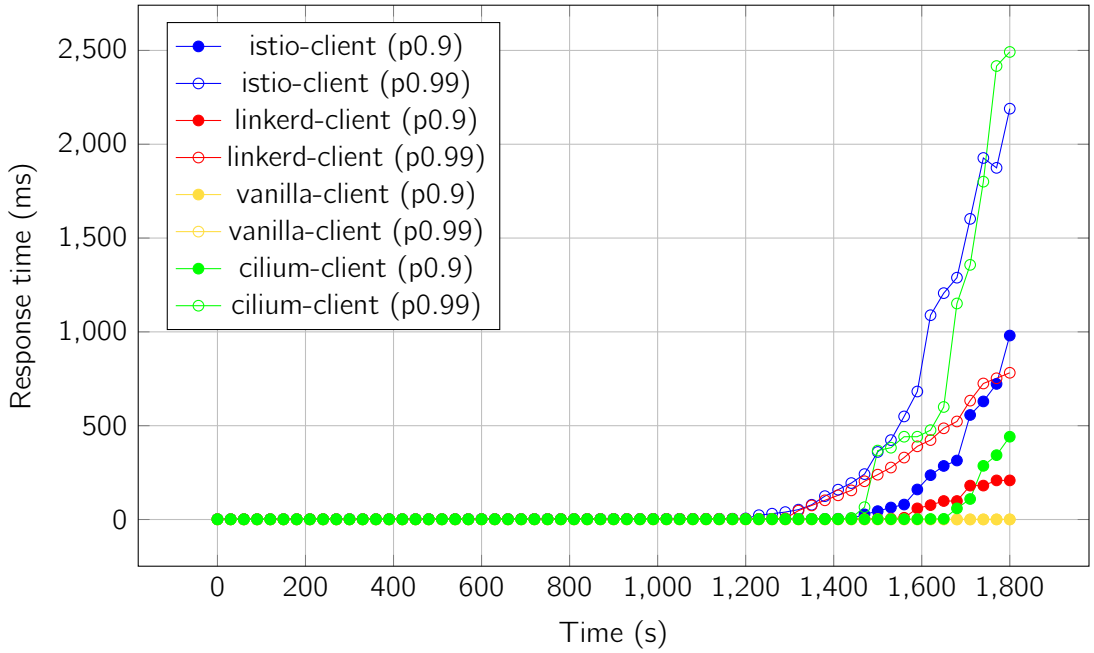

Figure 5.6: Response times during 30 minutes (both percentiles) (A–B)

Through the analysis of the data presented in Figures 5.6, 5.7 and 5.8 and Table 5.1, we can see that Istio exhibited a maximum response time of 2189.55 ms and an average response time of 233.56 ms, with the 99th percentile response time averaging 2.72 ms during the initial 20 minutes. Linkerd demonstrated a maximum response time of 782.03 ms and an average response time of 104.44 ms, with the 99th percentile response time averaging 2.07 ms during the first 20 minutes. Cilium showed a maximum response time of 2491.84 ms and an average response time of 198.07 ms, with the 99th percentile response time averaging 1.54 ms during the initial 20 minutes. In both percentiles (Figures 5.8 and 5.7) Cilium shows the highest average latency, followed by Cilium then Linkerd.

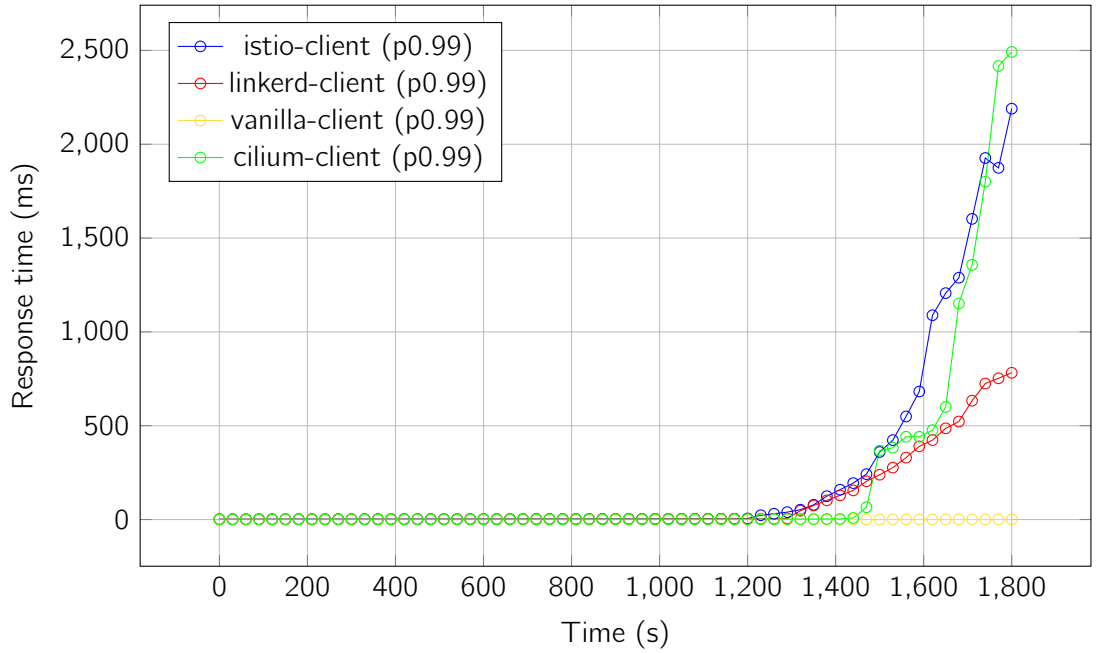

Figure 5.7: Response times during 30 minutes (99th percentile) (A–B)

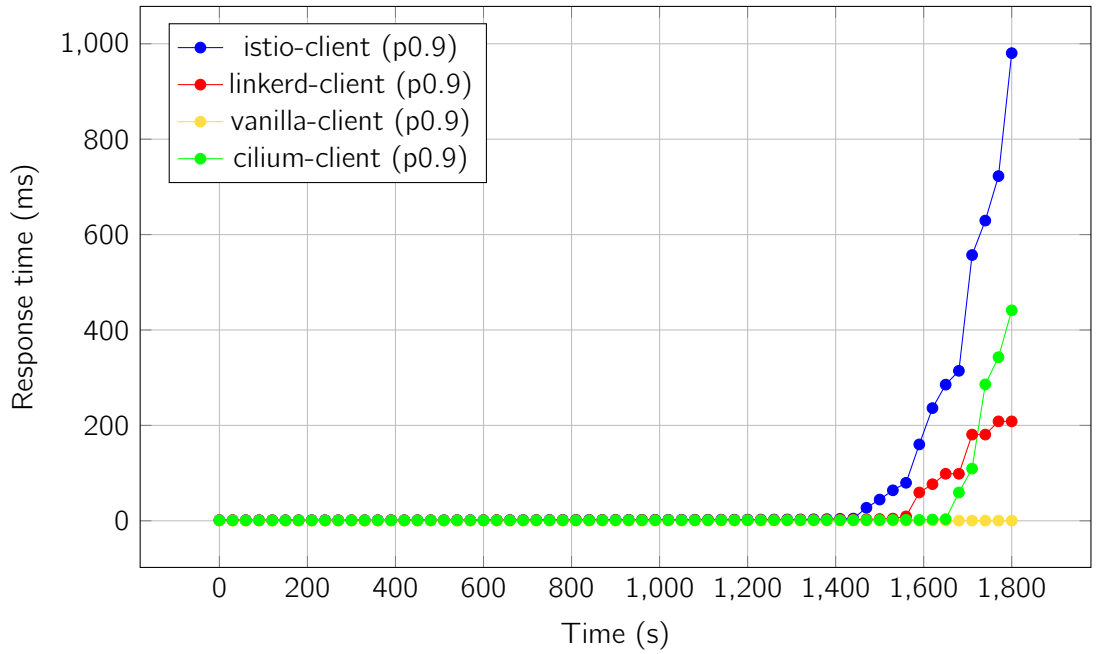

Figure 5.8: Response times during 30 minutes (90th percentile) (A–B)

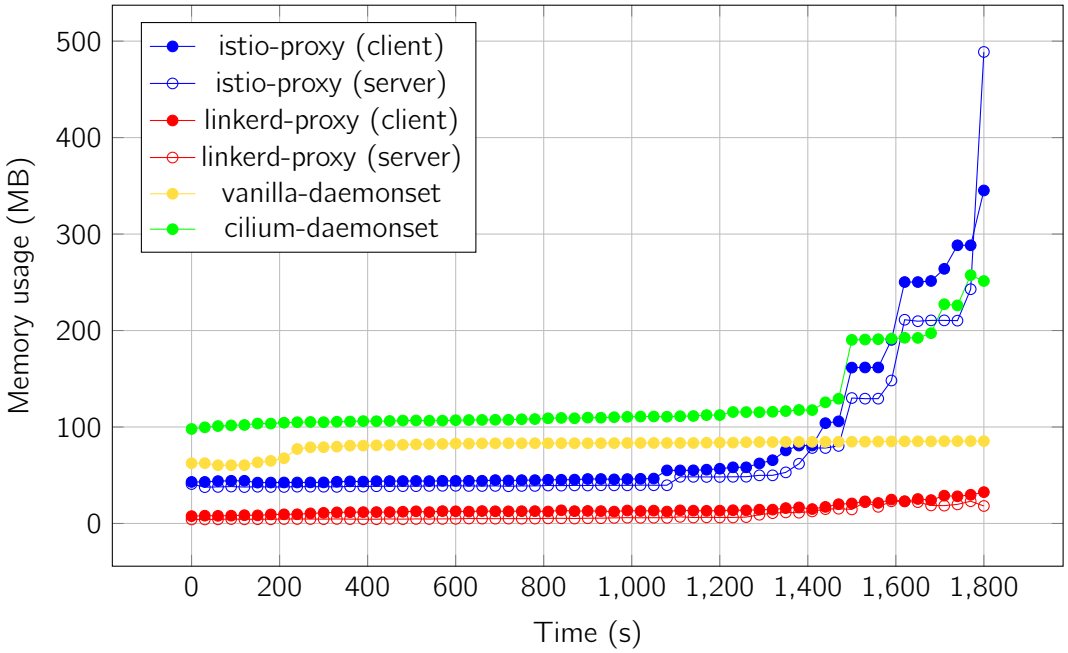

Figure 5.9: Memory usage of the proxies in MB (A–B)

Analyzing Figure 5.9 and Table 5.1, the Istio client sidecar exhibited an average memory consumption of 84.82 MB, peaking at 345.2 MB, while the server sidecar had an average memory utilization of 73.72 MB, reaching a peak of 488.78 MB. Combined, the Istio sidecars had an average total memory usage of 158.54 MB. Linkerd's client sidecar consumed an average of 14.74 MB of memory, with a maximum of 32.48 MB, and the server sidecar had an average memory footprint of 8.59 MB, peaking at 23.05 MB. Together, Linkerd's sidecars averaged a total memory usage of 23.33 MB. Cilium, using a daemonset approach, recorded an average memory consumption of 127.23 MB, with a maximum of 257.40 MB.

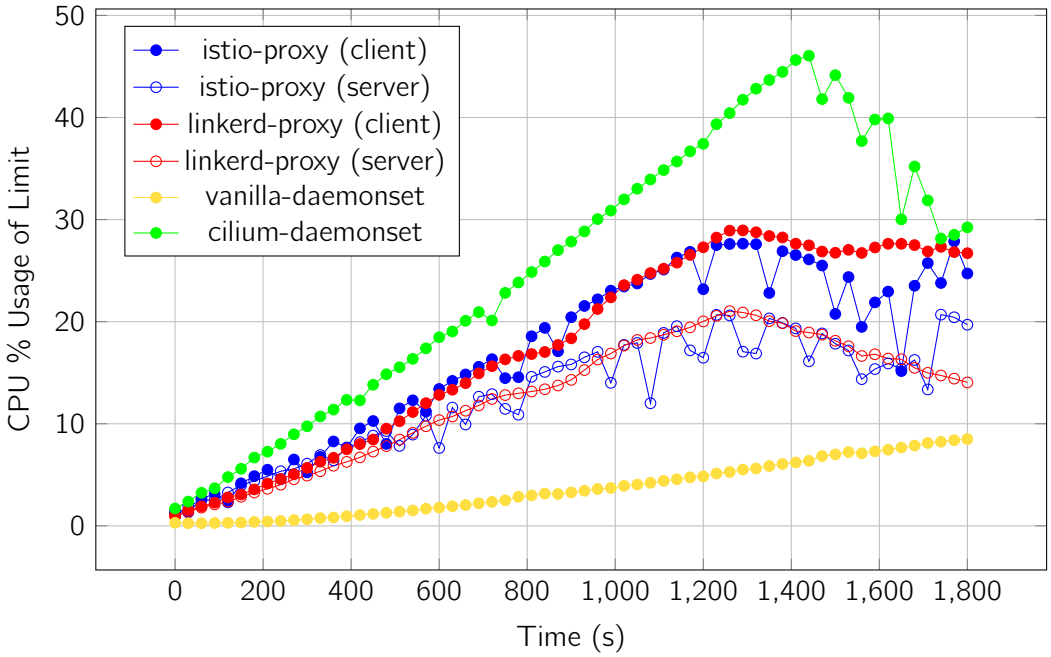

Figure 5.10: CPU usage of the proxies in % (A–B)

The Istio client sidecar had an average CPU utilization of 16.96%, peaking at 27.69%, while the server sidecar demonstrated slightly lower average CPU usage at 12.76%, with a maximum of 20.69%. Combined, Istio's sidecars presented an average total CPU usage of 29.72%. Linkerd's client sidecar showed an average CPU usage of 17.73%, reaching a maximum of 28.94%, and the server sidecar had an average CPU utilization of 12.63%, with a maximum of 21.03%. Together, Linkerd's sidecars averaged a total CPU usage of 30.36%. Cilium's daemonset exhibited an average CPU utilization of 25.30%, peaking at 46.05% (Figure 5.10).

Table 5.1: Zero Trust Components Metrics During 30 minutes A-B Testing

Metric	Istio	Linkerd	Cilium	Vanilla
Client Sidecar				
Avg CPU Usage (%)	16.96	17.73	N/A	N/A
Max CPU Usage (%)	27.87	28.94	N/A	N/A
Avg Memory Usage (MB)	84.82	14.74	N/A	N/A
Max Memory Usage (MB)	345.2	32.48	N/A	N/A
Server Sidecar				
Avg CPU Usage (%)	12.76	12.63	N/A	N/A
Max CPU Usage (%)	20.69	21.03	N/A	N/A
Avg Memory Usage (MB)	73.72	8.59	N/A	N/A
Max Memory Usage (MB)	488.78	23.05	N/A	N/A
DaemonSet				
Avg CPU Usage (%)	N/A	N/A	25.30	3.65
Max CPU Usage (%)	N/A	N/A	46.05	8.50
Avg Memory Usage (MB)	N/A	N/A	127.23	80.55
Max Memory Usage (MB)	N/A	N/A	257.4	85.37
Total Resource Usage				
Avg Total CPU Usage (%)	29.72	30.36	25.30	3.65
Avg Total Memory Usage (MB)	158.54	23.33	127.23	80.55
Request Handling				
Max Successful Requests per Second	1328.33	1341.43	1459.37	1858.83
Avg Successful Requests per Second	815.11	828.45	825.52	965.82
Request Success Rate (http200)	83.68%	94.20%	86.85%	100%
Max Response Time (ms, 99th percentile)	2189.55	782.03	2491.84	0.64
Avg Response Time (ms, 99th percentile)	233.56	104.44	198.07	0.50
Max Response Time (ms, 90th percentile)	980.34	208.4	441.2	0.36
Avg Response Time (ms, 90th percentile)	68.76	19.67	21.35	0.31
Failure Timestamp (s)	1200	1290	1440	N/A

The analysis of the Table 5.1 comparing Istio, Linkerd, and Cilium in the A to B benchmark reveals distinct differences in performance and resource usage among these service mesh technologies. Istio shows the slowest performance in request handling metrics, with the lowest throughput and the highest response times, indicating it is less efficient under load compared to the other technologies. Despite this, Istio consumes the most resources, particularly in terms of memory usage on both the client and server sides. This high resource consumption does not translate into better performance, showing Envoy's drawback.

Linkerd and Cilium offer a balance between resource usage and performance. Linkerd uses the least memory overall, making it more resource-efficient compared to Istio and Cilium due to the usage of Linkerd2 proxy. However, in terms of request handling, both Linkerd and Cilium do not outperform the best performer in the original dataset. Linkerd shows slightly better performance than Cilium in some metrics.

It is possible to associate the difference in memory consumption with the proxy implemented by each service. As Istio consists in the implementation of Envoy Proxy written in C++ to handle the service mesh features (Istio 2024), Linkerd makes use of an *in-house* purpose-built proxy for the Linkerd service mesh, written in Rust.

In summary, Istio consumes the most resources but performs the slowest in request handling. Linkerd and Cilium provide a middle ground, being more efficient in resource usage than Istio but not reaching the top performance levels observed in the dataset.

5.3.2 Service Mesh A-B-C Benchmark

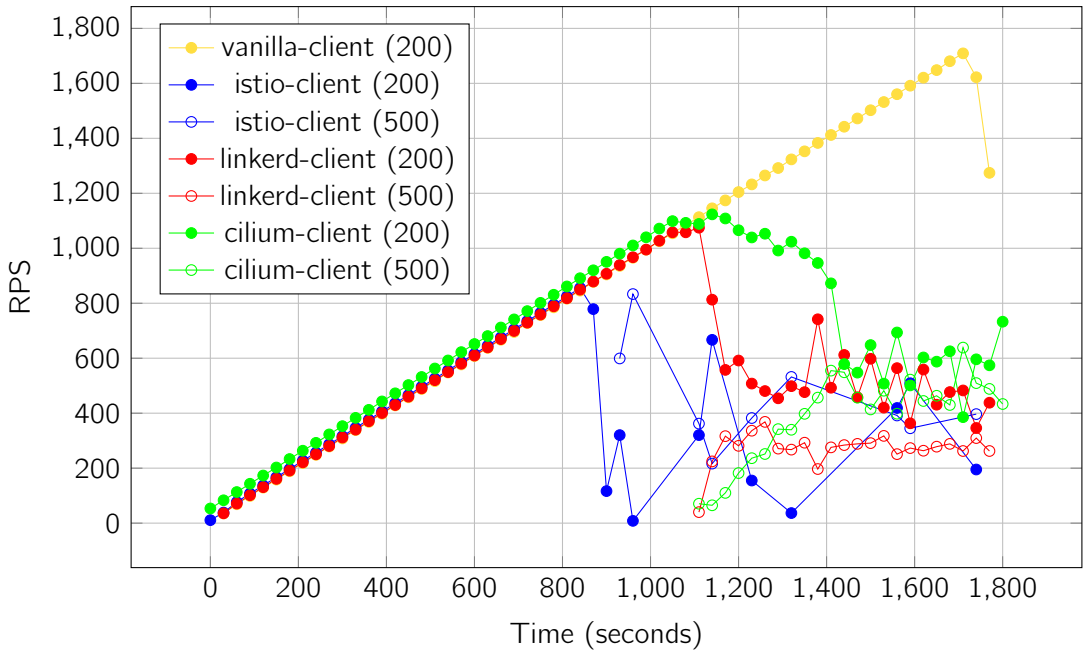

Figure 5.11: Requests per second (A–B–C)

When considering the maximum successful requests per second based on the results presented in Figure 5.11 and Table 5.2, Cilium leads with 1123.37 RPS, followed closely by Linkerd with 1074.77 RPS, while Istio trails behind at 855.6 RPS. The average successful requests per second also shows Cilium and Linkerd ahead with 667.76 RPS and 554.67 RPS respectively, compared to Istio's 403.99 RPS. The request success rate (http200) also reflects this trend, with Cilium achieving the highest rate at 81.52%, Linkerd at 83.99%, and Istio significantly lower at 79.93%. Additionally, using the failure timestamps to indicate the resilience of each system under load, Cilium and Linkerd show longer endurance before failure, with failure timestamps at 1110 seconds each, whereas Istio fails sooner at 930 seconds. This indicates that both Linkerd and Cilium can handle a higher load of requests more efficiently and sustain their performance for a longer duration compared to Istio.

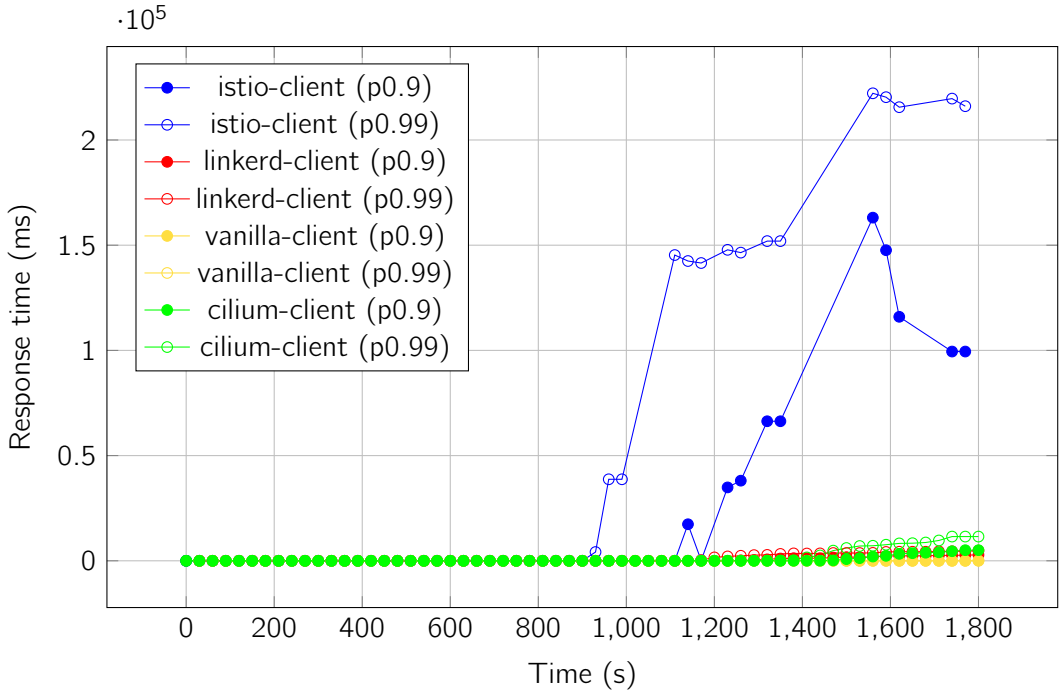

Figure 5.12: Response times during 30 minutes (both percentiles) (A–B–C)

Latency measurements (Figure 5.12 and Table 5.2) reveal that Istio consistently exhibits the highest response times, particularly at the 99th and 90th percentiles. At the 99th percentile, Istio recorded a maximum response time of 47894.61 ms, while Linkerd and Cilium demonstrated much lower response times of 4805.62 ms and 11517.69 ms respectively. This trend is also reflected in the average response times at the 99th percentile, with Istio at 18453.85 ms compared to Linkerd's 549.81 ms and Cilium's 1759.51 ms. These 99th percentile values highlight that Istio has substantial delays under high load, representing the worst-case performance for 99% of the requests.

Similarly, at the 90th percentile, which represents the maximum response time for 90% of the requests, Istio exhibits the highest latency at 163092.62 ms, in contrast to Linkerd's 2809.73 ms and Cilium's 5045.47 ms. The average response times at the 90th percentile further underscore Istio's higher latency, indicating that it may be less suitable for scenarios requiring low-latency performance. These percentile metrics are essential in understanding the upper limits of response times experienced by most users and provide valuable insights into system performance under stress.

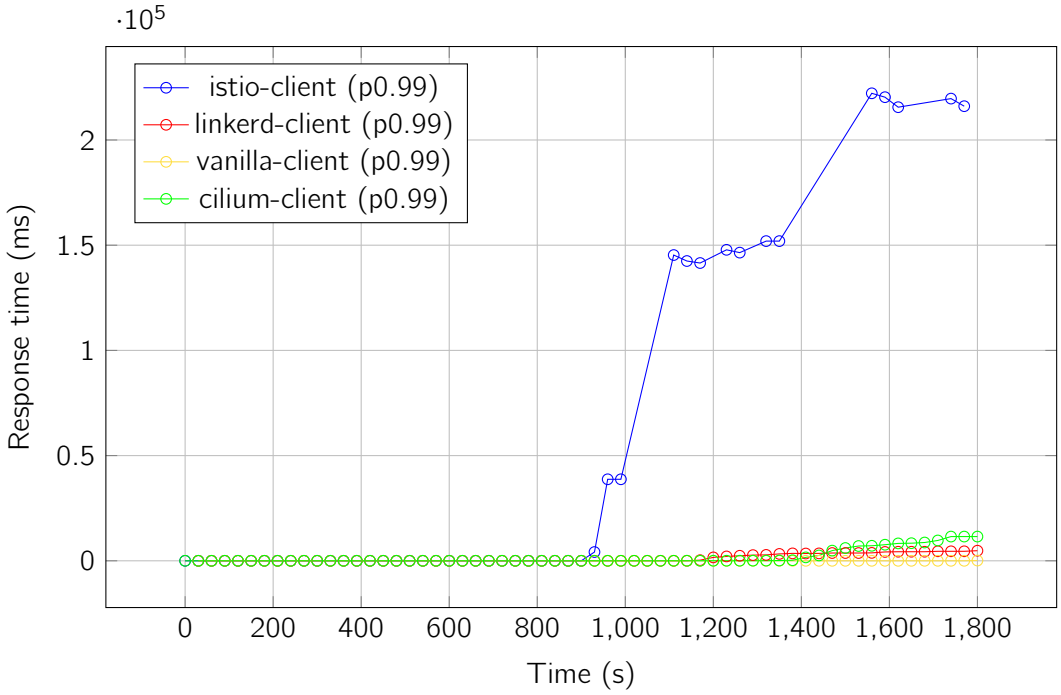

Figure 5.13: Response times during 30 minutes (99th percentile) (A–B–C)

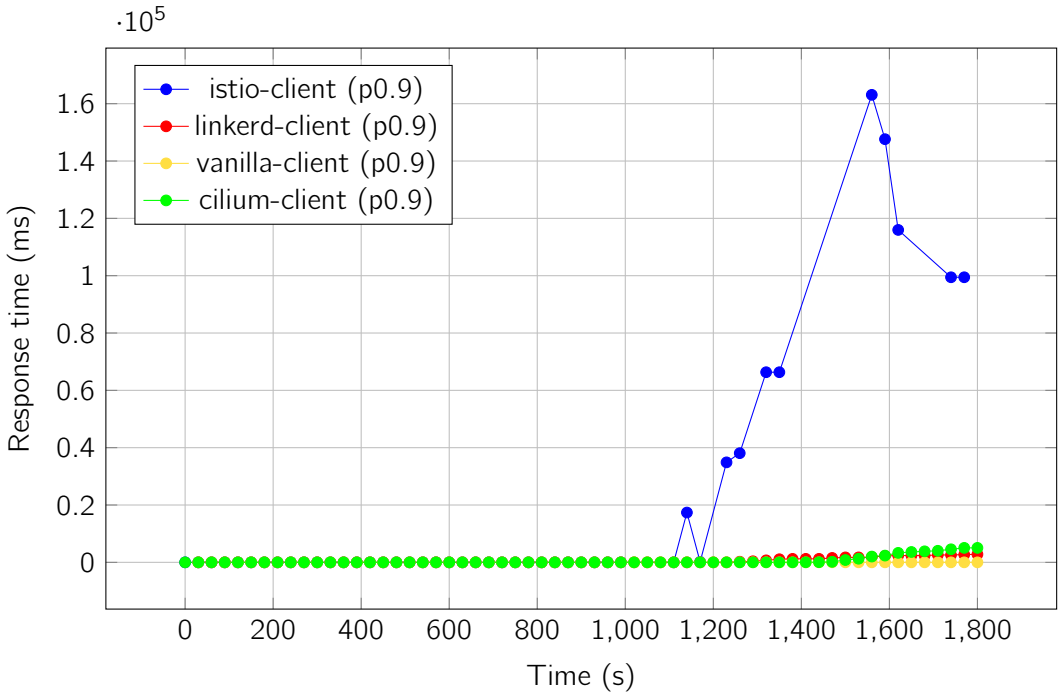

Figure 5.14: Response times during 30 minutes (99th percentile) (A–B–C)

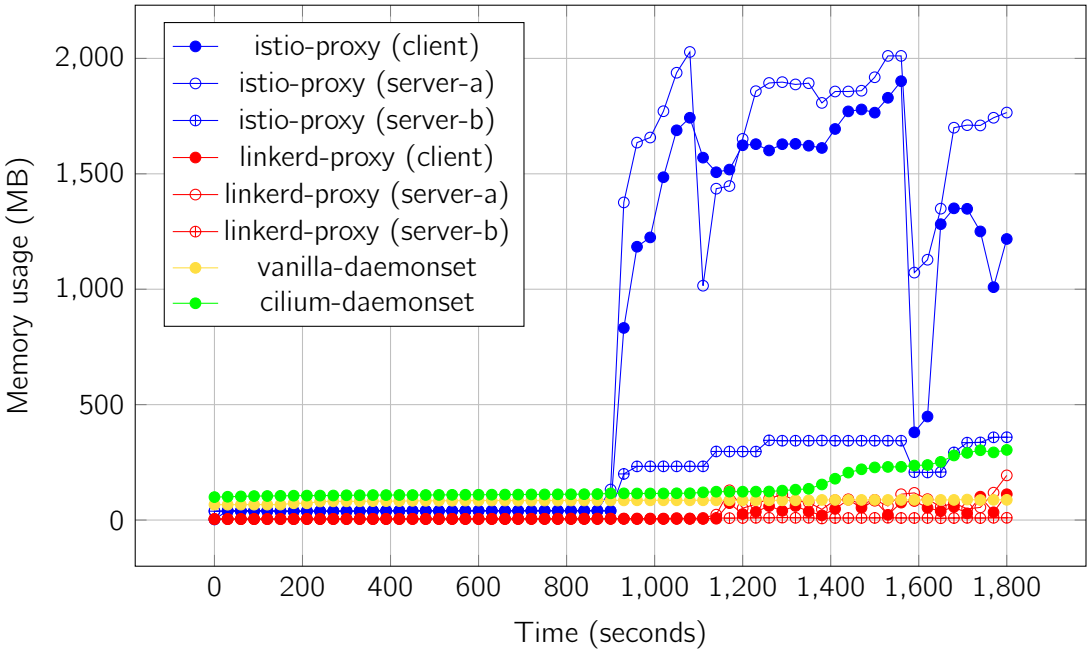

Figure 5.15: Memory usage of the proxies in MB in 30 minutes (A–B–C)

In terms of memory usage (Figure 5.12 and Table 5.2), Istio shows significantly higher consumption. The client sidecar of Istio uses an average of 726.52 MB and peaks at 2027.94 MB, while Linkerd uses an average of 23.14 MB, peaking at 111.8 MB. For server sidecar A, Istio again uses more memory with an average of 856.00 MB and a peak of 2044.71 MB, whereas Linkerd uses 35.35 MB on average, peaking at 193.5 MB. In server sidecar B, Istio's average memory usage is 165.21 MB, peaking at 358.86 MB, while Linkerd uses an average of 6.25 MB, peaking at 10.55 MB. The DaemonSet memory usage for Cilium is 144.54 MB on average, peaking at 303.83 MB. All these components added up show a big difference in resource usage from Istio compared to the rest of the technologies tested, which shows the inefficiency of the Envoy proxy usage compared to its usage in Cilium, where it is used as a centralized proxy.

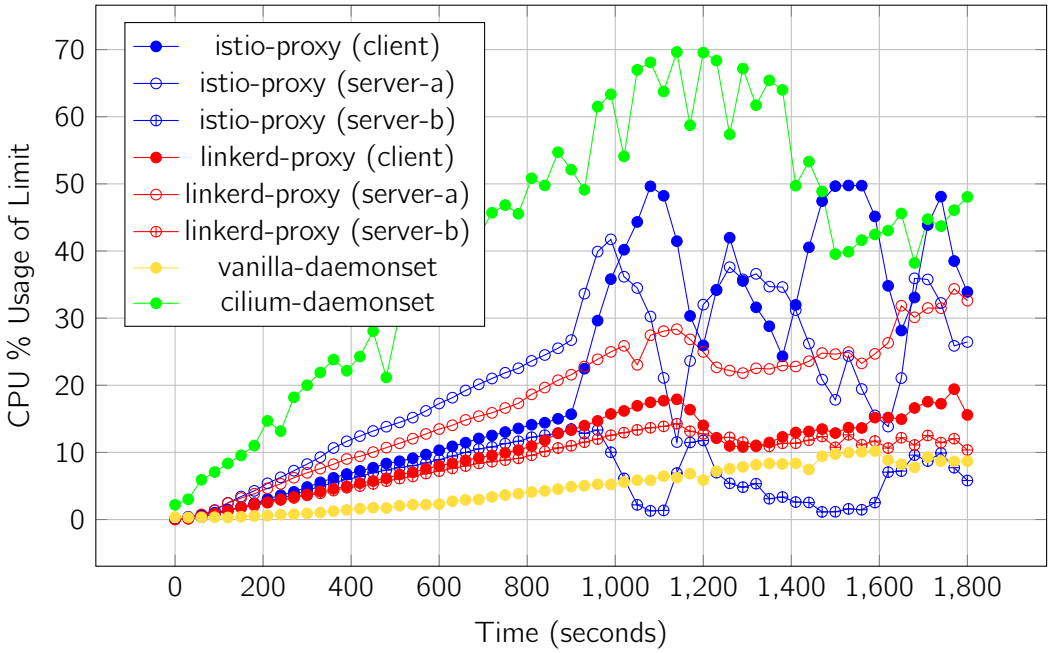

Figure 5.16: CPU usage of the proxies in % in 30 minutes (A–B–C)

Examining CPU usage, Istio also appears to use more CPU resources compared to Linkerd. For the client sidecar, Istio has an average CPU usage of 22.92% and a maximum of 49.76%, whereas Linkerd uses 10.43% on average, with a maximum of 19.42%. In server sidecar A, Istio uses 20.68% on average, with a peak of 41.73%, while Linkerd uses 17.90% on average, peaking at 34.36%. Server sidecar B shows Istio with an average CPU usage of 6.25%, peaking at 13.48%, while Linkerd uses 8.65% on average, with a peak of 14.27%. Cilium's DaemonSet shows an average CPU usage of 40.76%. Overall, Istio generally consumes more CPU than Linkerd, and Cilium's DaemonSet has significant CPU usage as it implements the same proxy as Istio.

Table 5.2: Zero Trust Components Metrics During 30 minutes A-B-C Testing

Metric	Istio	Linkerd	Cilium	Vanilla
Client Sidecar				
Avg CPU Usage (%)	22.92	10.43	N/A	N/A
Max CPU Usage (%)	49.76	19.42	N/A	N/A
Avg Memory Usage (MB)	726.52	23.14	N/A	N/A
Max Memory Usage (MB)	2027.94	111.8	N/A	N/A
Server Sidecar A				
Avg CPU Usage (%)	20.68	17.90	N/A	N/A
Max CPU Usage (%)	41.73	34.36	N/A	N/A
Avg Memory Usage (MB)	856.00	35.35	N/A	N/A
Max Memory Usage (MB)	2044.71	193.5	N/A	N/A
Server Sidecar B				
Avg CPU Usage (%)	6.25	8.65	N/A	N/A
Max CPU Usage (%)	13.48	14.27	N/A	N/A
Avg Memory Usage (MB)	165.21	6.25	N/A	N/A
Max Memory Usage (MB)	358.86	10.55	N/A	N/A
DaemonSet				
Avg CPU Usage (%)	N/A	N/A	40.76	4.78
Max CPU Usage (%)	N/A	N/A	69.65	10.21
Avg Memory Usage (MB)	N/A	N/A	144.54	82.86
Max Memory Usage (MB)	N/A	N/A	303.83	88.69
Total Resource Usage				
Avg Total CPU Usage (%)	49.85	36.98	40.76	4.78
Avg Total Memory Usage (MB)	1747.73	64.74	144.54	82.86
Request Handling				
Max Successful Requests per Second	855.6	1074.77	1123.37	1800.0
Avg Successful Requests per Second	403.99	554.67	667.76	905.61
Request Success Rate (http200)	79.93%	83.99%	81.52%	100%
Max Response Time (ms, 99th percentile)	47894.61	4805.62	11517.69	122.13
Avg Response Time (ms, 99th percentile)	18453.85	549.81	1759.51	3.19
Max Response Time (ms, 90th percentile)	163092.62	2809.73	5045.47	0.97
Avg Response Time (ms, 90th percentile)	549.81	549.81	594.04	0.79
Failure Timestamp (s)	930	1110	1110	N/A

In summary, analyzing Table 5.2, Cilium shows the best performance in terms of request handling (RPS) and latency, indicating it can handle higher traffic with lower delay. With Istio being the highest consumer in memory and CPU, it is limiting factor in resource-constrained environments, as a sidecar needs to be running for each pod to obtain the full benefit of the service mesh.

Based on the results we discussed earlier, it seems that the difference in memory consumption may be linked to the specific proxy used by each service. While Istio uses the Envoy Proxy

to manage its service mesh features (Istio Architecture), Linkerd uses its own custom-built proxy for its service mesh.

The differences in memory consumption among these technologies reflect their architectural approaches. Istio and Linkerd utilize sidecars extensively, while Cilium adopts a per-node controller model, centralizing traffic and encryption management. These distinctions influence their resource consumption profiles and operational behaviors under varying loads and stress conditions.

5.3.3 Network Policy Layers Comparison

Table 5.3: Cilium Network Policy performance in different layers

Resource Usage (Daemonset)	**L3**	**L4**	**L7**	**L7 + mTLS**
Avg CPU Usage (%)	4.25	4.23	40.49	40.76
Max CPU Usage (%)	10.6	10.2	67.51	69.65
Avg Memory Usage (MB)	95.83	96.16	144.70	144.54
Max Memory Usage (MB)	100.73	101.83	304.85	303.83
Request Handling				
Max Successful Requests per Second	1759.0	1775.9	1225.3	1123.37
Avg Successful Requests per Second	931.54	920.85	670.69	667.76
Request Success Rate (http200)	98.24%	96.8%	83.44%	81.52%
Max Response Time (ms, 99th percentile)	321.02	10859.46	11489.41	11517.69
Avg Response Time (ms, 99th percentile)	9.13	190.35	2008.13	1759.51
Max Response Time (ms, 90th percentile)	0.88	0.87	5792.51	5045.47
Avg Response Time (ms, 90th percentile)	0.65	0.66	674.08	594.04
Failure Timestamp (s)	1710	1650	1140	1110

The Table 5.3 compares the performance of Cilium network policies across different layers: L3, L4, L7, and L7 with mTLS, focusing on resource usage and request handling. As discussed previously, L7 policies are implement trough the use of an envoy proxy in Cilium's Daemonset Pod. These tests were conducted using the A-B-C approach in order to get a better view on the performance differences as it adds an extra connection compared to the A-B approach.

In terms of resource usage, L3 and L4 layers exhibit significantly lower average and maximum CPU usage, with both averaging around 4.25% and peaking at about 10.6%. In contrast, the L7 and L7 with mTLS layers show much higher CPU demands, averaging around 40.5% and peaking at over 67%. Memory usage follows a similar pattern. L3 and L4 use around 95-96 MB on average, with maximum usage just above 100 MB. The L7 layers, however, use considerably more memory, averaging around 144 MB and peaking at over 300 MB.

When examining request handling capabilities, L3 and L4 layers outperform the higher layers in terms of the number of successful requests per second. Both L3 and L4 can handle over 1750 requests per second at their peak, while L7 and L7 with mTLS manage fewer, around 1225 and 1123 requests respectively. The average successful requests per second are also higher for L3 and L4, with values just above 920, compared to around 670 for the L7 layers. Additionally, the request success rate is highest for L3 at 98.24%, followed closely by L4

at 96.8%. This rate drops significantly for L7 and L7 with mTLS, falling to 83.44% and 81.52%.

Response times further highlight the performance differences. L3 has the best response times, with a maximum of 321 ms and an average of 9.13 ms for the 99th percentile. L4 shows higher response times, with a maximum of 10859 ms and an average of 190 ms. L7 and L7 with mTLS have the highest response times, with maximum values over 11400 ms and averages around 2008 and 1759 ms, respectively. Similar trends are observed in the 90th percentile response times.

Lastly, the timestamp indicating the onset of failures occurs later for L3 and L4, at 1710 and 1650 seconds, respectively, compared to 1140 and 1110 seconds for L7 and L7 with mTLS. This detailed comparison highlights the balance between the enhanced security capabilities of higher-layer policies and their effect on performance. L3 and L4 layers offer quicker processing with fewer security features compared to the more resource-intensive L7 and L7 with mTLS layers.

5.3.4 Gatekeeper and Falco

The findings from this testing (Table 5.4) indicate that OPA Gatekeeper integrates seamlessly with existing PSPs without imposing noticeable performance overhead. This compatibility is pivotal for organizations migrating from PSPs to Gatekeeper, ensuring a smooth transition while maintaining robust policy enforcement standards. OPA Gatekeeper, leveraging the Open Policy Agent (OPA), introduces a flexible, policy-driven approach to Kubernetes cluster management, expanding upon the capabilities originally offered by PSPs.

Table 5.4: Comparison of memory usage, CPU usage, and pod creation time with and without policies for Gatekeeper components

Policies	**Memory (MB)**	**CPU (Cores)**	**Pod creation time (ms)**
No policies	44.73	0.14	384
All 15 official policies	44.67	0.09	401

The implementation was seamless with relative ease as the language used to write the policies, Rego, revealed clear and concise for it.

The consistent performance observed suggests that OPA Gatekeeper's architecture, which involves evaluating policies through OPA's rego language against Kubernetes resources, is designed to operate efficiently even under varying cluster sizes and complexities. This design not only supports scalable deployments but also ensures that policy evaluations do not adversely impact cluster performance, even when enforcing multiple policies across extensive deployments and diverse environments. The findings from this benchmark emphasize OPA Gatekeeper as a robust and performant solution for Kubernetes policy enforcement, offering enhanced flexibility and control over cluster security.

The same can be said for Falco and Falco Sidekick, which based on the analysis conducted (Tables 5.5 and 5.6), was observed that both did not experience a significant increase in resource usage during testing phases. Falco, responsible for real-time security monitoring by tracking system calls and container activities, operated without imposing noticeable additional demands on CPU or memory resources. Similarly, Falco Sidekick, which manages alert notifications and external integrations, also maintained stable resource utilization levels.

Table 5.5: Falco: Memory and CPU Usage at Different Violation Rates

Violation/second	Memory (MB)	CPU (Cores)
0	39.41	0.597
50	40.12	0.588
100	38.12	0.601
300	41.72	0.571
500	40.55	0.594

Table 5.6: Falco Sidekick: Memory and CPU Usage at Different Violation Rates

Violations/second	Memory (Falco Sidekick) (MB)	CPU (Falco Sidekick) (Cores)
0	18.66	0.061
50	20.7	0.065
100	18.21	0.070
300	18.35	0.093
500	18.52	0.133

Chapter 6

Optimizing Zero Trust Implementations

6.1 Solution Strategies

In the course of evaluating Zero Trust technologies, we observed significant variations in resource utilization, throughput, and latency among Istio, Linkerd, and Cilium. Each technology demonstrated unique strengths and weaknesses, influenced by their architectural designs and operational mechanisms. This chapter delves into these performance characteristics, proposing potential optimization techniques to enhance the efficiency and effectiveness of Zero Trust implementations.

The majority of these optimization solutions are derived from the official documentation of each technology. Upon analysis, it's evident that the primary bottleneck lies in the sidecar technology, particularly with Envoy, which exhibits a significantly larger footprint compared to its competitors. Since deployment occurs on a per-service basis with sidecars, the key optimization strategy is to exclusively utilize sidecars for Layer 7 dependent functionalities, while delegating network policy implementation for technologies like eBPF (such as Cilium) to operate at Layer 4. According to the findings in Budigiri et al. 2021, this approach could yield improvements of up to 30% in comparison to Kubernetes-native policies.

Istio Optimization Strategies

Istio, known for its wide range of features, has been found to consume significant system resources, particularly in terms of CPU and memory usage. The simultaneous operation of client and server sidecars has been identified as a major factor contributing to this increased overhead. While these components provide strong security features, their impact on system performance has been evident. To address these challenges, several optimization strategies can be suggested.

Firstly, adjusting the resource requests and limits for sidecars can effectively prevent excessive resource allocation. By setting appropriate CPU and memory limits, a balance can be achieved between performance and resource efficiency.

Additionally, implementing lazy loading for security policies is a viable option for reducing initial overhead. This involves loading policies only when needed, rather than during startup, thereby reducing the initial memory footprint and improving startup times.

Optimizing the configuration of Envoy proxies is another way to enhance system performance. This includes fine-tuning settings such as connection pooling and enabling protocols like HTTP/2 or gRPC, which can improve throughput and reduce latency.

Furthermore, the implementation of adaptive rate-limiting mechanisms is crucial for effectively managing network traffic. This approach helps prevent system overloads and ensures stable performance under varying traffic conditions.

Linkerd Optimization Strategies

Linkerd has demonstrated impressive CPU and memory metrics while maintaining a relatively low resource footprint. To further enhance its performance, the following strategies can be considered:

One approach involves streamlining sidecar operations, achieved by reducing the frequency of telemetry data collection and utilizing efficient serialization formats like protobuf to minimize resource usage.

Another strategy is to introduce advanced caching mechanisms for frequently accessed policies and configurations, which can reduce response times and alleviate the computational burden on sidecars.

Employing sophisticated load balancing algorithms, such as consistent hashing or least-request, can improve request distribution and enhance overall system responsiveness.

An additional optimization involves fine-tuning connection reuse settings, including keep-alive intervals and idle connection timeouts, to decrease the overhead associated with establishing new connections and improve latency.

Cilium Optimization Strategies

Cilium's per-node controller model demonstrates distinct performance characteristics, exhibiting higher CPU utilization but moderate memory consumption. To enhance Cilium's performance, the following techniques can be considered:

Firstly, optimizing BPF programs involves fine-tuning eBPF programs for specific workloads to improve packet processing efficiency. This includes simplifying eBPF logic and optimizing data structures within the programs.

Secondly, implementing Distributed Traffic Management can shard traffic across multiple nodes, thereby reducing the load on individual controllers and enhancing scalability.

Additionally, implementing Selective Encryption, based on traffic sensitivity and compliance requirements, can reduce the computational overhead associated with encrypting all traffic indiscriminately.

Finally, Resource Allocation Tuning should be considered by adjusting resource allocation for Cilium agents based on node capacity and expected traffic patterns to optimize performance and resource utilization under varying conditions.

6.2 Defining a Theoretical Solution

Given the observed performance characteristics and the need for optimized Zero Trust implementations within dynamic computing environments, a balanced solution that leverages the strengths of different technologies while addressing their weaknesses would be ideal.

As Gatekeeper and Falco did not present increases in resource usage, they will not be included in the definition of the final solution.

Although sidecars have significant advantages over refactoring applications, they also have notable limitations. One of the primary issues is their invasiveness. Since sidecars are part of the pod, any update or malfunction necessitates a pod restart, which can be disruptive for workloads. Another significant drawback is the potential for underutilization of resources. The requests and limits of the pods need to be adjusted pre-deployment on a per-pod basis. This adjustment can lead to substantial underutilization of resources within the cluster. This issue was reflected during benchmarking, as there was no insight pre-deployment about how much resources the sidecars would consume. Accurate measurement of resource use in real-world scenarios requires rigorous testing, a process that can be particularly challenging for small companies with limited resources.

Moreover, there is a dependence on the technology itself. Although all the technologies benchmarked were open-source, the level of support varied significantly. Linkerd2 Proxy, the proxy for the Linkerd service mesh, is maintained by a small team, which makes support potentially limited and less robust. In contrast, Envoy is implemented and supported by individual contributors from leading global companies such as Google, Amazon, and Netflix, ensuring a more extensive and reliable support network.

From the data obtained through benchmarking the service meshes, the performance can vary heavily depending on the technology used. One takeaway we can get from the results is that layer 7 network policy implementation is not yet in a *must use for any case* state, as the resource overhead that it implies on the system is still too high. In their 2023 Hype Cycle Report(Gartner 2024), Gartner put Cybersecurity Mesh Architecture in the *Innovation Trigger* phase, predicting developments in the area.

To go through that problem, this thesis proposes the implementation of a configurable data plane for a *per block* basis instead of a *per service* as is implemented on the current state of the technologies. Through the work of Budigiri et al. 2021 and Sedghpour and Townend 2022, it is possible to refer to eBPF as the future, as Liu et al. 2020 suggests a "nonintrusive collection of user application L7/L4 layer network protocol interaction information based on eBPF, data collection of more than 10M throughputs per second can be achieved without modifying any kernel and application code, while the impact on the system application is less than 1%", we propose a combination of eBPF with sidecars to implement a service mesh. The solution will be using an eBPF powered Control Plane as implemented in Cilium, in order to obtain the performance from such technology. Like Cilium, the solution will implement Custom Resource Definitions to extend the vanilla capabilities of Kubernetes Network Policies.

In this proposition the data plane would be composed of proxies that would be responsible for a group of services, being able to scale accordingly to the needs and resource usage of the pods in said group. With this approach, we would be able to tailor the proxy usage to the service needs and implement layer 7 relevant configurations. The configuration would

be as present in Figure 6.1. This would be majorly an adaptation of the mTLS solution that Cilium provides using an Envoy proxy in each one of the Daemonset pods.

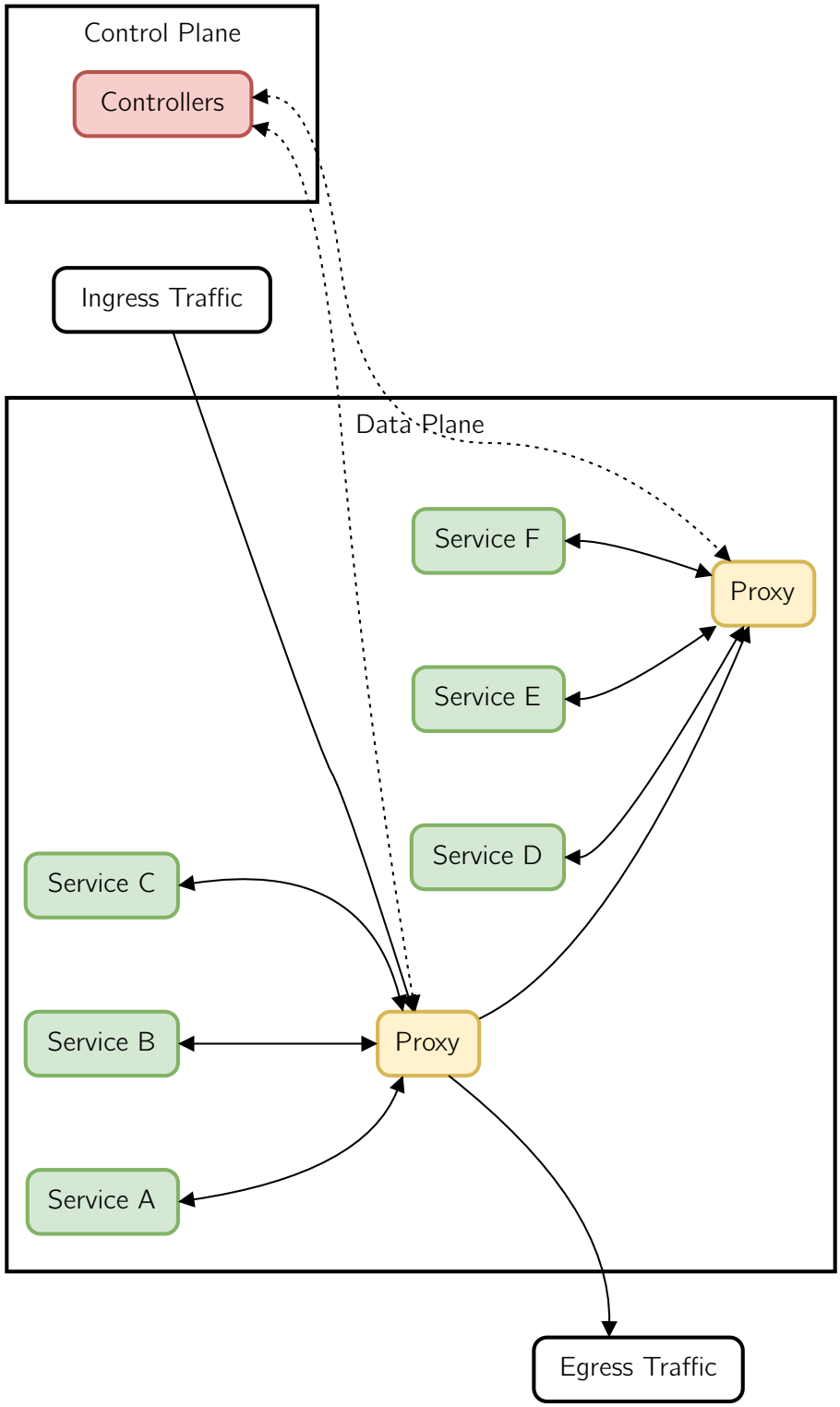

Figure 6.1: Service Mesh Architecture Proposal

The main feature, the data plane must be able to scale horizontally the proxies according

to the traffic from the selected services. The controller must be able to determine, given a set of pods, how granular should the proxy usage be, as more usage on a pod should imply its own sidecar.

6.3 Feasibility Assessment

The proposed approach of implementing a configurable data plane on a per-block basis using a combination of eBPF with sidecars presents an innovative direction for enhancing service mesh capabilities within dynamic computing environments. However, several feasibility considerations must be addressed before committing to its development and deployment. Implementing a novel approach like the one proposed requires adequate time for the research, development, and testing phases. Time constraints arose due to project deadlines and other concurrent priorities. Allocating resources for the development and testing of a new data plane solution is beyond the budgetary limits as it requires a specialized skillset which takes time to gather and requires heavy testing infrastructure which would require additional financial outlay.

Given these constraints, the feasibility of developing the proposed new approach is limited at this time. While the potential benefits in terms of performance, scalability, and security enhancements are clear, the significant technical, resource, and operational challenges make it difficult to pursue this solution within the current framework. Collaboration with industry experts or leveraging open-source community support could mitigate some of these challenges, potentially paving the way for future advancements in Zero Trust implementations within dynamic computing environments.

Chapter 7

Conclusion

This chapter provides an overview of the thesis as a whole, providing clear insight into the findings, a summary of the contributions that this dissertation provided, the limitations on the writing of the thesis and on the implementation as well as a definition of the future directions which this thesis may perhaps take us.

7.1 Summary of Findings

The thesis delves into the application of Zero Trust (ZT) principles in Kubernetes environments, highlighting the significant resource demands and complexities associated with distributed microservices systems. Performance tests demonstrate that activating ZT components such as mTLS and service meshes (Istio, Linkerd, Cilium) leads to increased CPU and memory usage, resulting in higher latencies. Although these implementations enhance security through Identity and Access Management (IAM) and Network Security policies, they also result in additional computational costs.

Balancing security and performance remains a significant challenge. The proposed solution integrates eBPF and sidecars to establish a customizable data plane, addressing resource overhead and scalability issues. By leveraging the efficiency of eBPF and Envoy proxies, this approach aims to optimize resource utilization and enhance the implementation of layer 7 network policies, laying the groundwork for future research in securing Kubernetes environments.

With this thesis, we believe that we have made valuable contributions to the field, highlighting the practicality and challenges of implementing Zero Trust (ZT) principles in Kubernetes environments. Through our state-of-the-art research and architectural proposal, we have taken a step forward in the implementation of Zero Trust in Kubernetes.

7.2 Contributions

This thesis made the following contributions:

1. **State-of-the-Art:** This thesis provides a comprehensive review and critical analysis of existing Zero Trust (ZT) security models, specifically within the context of Kubernetes environments. By synthesizing current research and technologies, it highlights the strengths and limitations of contemporary ZT implementations and positions our work within the broader cybersecurity landscape. The detailed examination of sidecars, service meshes, and their respective impacts on system performance and security adds depth to the understanding of ZT principles in dynamic computing environments.

2. **Performance Analysis:** Through rigorous benchmarking and performance testing, this research offers valuable insights into the resource demands and operational impacts of various ZT technologies, such as mTLS, Istio, Linkerd, and Cilium. The analysis reveals the significant overheads associated with these technologies, particularly in terms of CPU and memory usage and increased latencies. These findings provide a critical reference point for practitioners and researchers aiming to implement ZT in Kubernetes, highlighting the trade-offs between enhanced security and system performance. This thesis also provides clear insight on the system usage of Falco and Gatekeeper tools which revealed to have a low footprint and not performance increase during testing.
3. **Architecture Proposal:** The thesis proposes a novel architecture that combines eBPF with sidecars to create a configurable data plane for Kubernetes environments. This innovative solution addresses the limitations of traditional sidecar implementations, such as resource underutilization and the invasiveness of updates. By leveraging eBPF's efficiency and the flexibility of Envoy proxies, the proposed architecture optimizes resource usage and enhances the scalability of ZT implementations. This contribution offers a practical framework for deploying ZT principles in Kubernetes, paving the way for future advancements in container security.

7.3 Limitations

The proposed approach of implementing a configurable data plane using eBPF with sidecars presents promising advancements but faces several feasibility challenges. Developing and testing this novel approach requires significant time, which was limited by project deadlines and concurrent priorities. Additionally, the specialized skill set and robust testing infrastructure needed exceed the budgetary limits of this project. Integrating eBPF with sidecars and ensuring seamless performance and scalability introduces complex technical challenges. Given these constraints, the feasibility of fully developing this solution is limited at this time. However, collaboration with industry experts and leveraging open-source community support could help address these challenges in the future.

7.4 Future Directions

This thesis opens several avenues for future research, potentially leading to a PhD project. Future work could focus on extended research and development to overcome technical challenges, integrating eBPF and sidecars, and refining the data plane architecture. Collaborative efforts with industry experts and the open-source community would provide additional support and resources. Research should also aim to optimize resource utilization, reduce overhead, and enhance the efficiency of layer 7 network policies. Additionally, deploying the solution in diverse environments to evaluate performance under varying loads would provide valuable insights. Furthermore, developing advanced IAM and network security policies leveraging eBPF and sidecars would enhance the overall security posture of Kubernetes environments.

While this master's thesis lays the groundwork for an innovative approach to Zero Trust in Kubernetes, significant work remains to realize its full potential. The proposed direction promises substantial advancements in container security and performance, making it a compelling subject for further research in a PhD project.

Bibliography

A seccomp overview (2023). url: https://lwn.net/Articles/656307/ (visited on 12/26/2023).

Alndawi, Tara (2021). *Replacing Virtual Machines and Hypervisors with Container Solutions*. eng. url: https://urn.kb.se/resolve?urn=urn:nbn:se:mdh:diva-54607 (visited on 11/14/2023).

Alshuqayran, Nuha, Nour Ali, and Roger Evans (Nov. 2016). "A Systematic Mapping Study in Microservice Architecture". In: *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51. doi: 10.1109/SOCA.2016.15. url: https://ieeexplore.ieee.org/abstract/document/7796008/authors#authors (visited on 06/27/2024).

Ambassador Labs (2023). *Understanding Multiple Kubernetes Clusters*. en. url: https://www.getambassador.io/resources/multi-cluster-kubernetes (visited on 12/03/2023).

Architectural Overview - Argo CD - Declarative GitOps CD for Kubernetes (2023). url: https://argo-cd.readthedocs.io/en/stable/operator-manual/architecture/ (visited on 12/22/2023).

Attardi, Giuseppe et al. (May 2018). "Declarative Modeling for Deploying a Container Platform". en. In: *2018 32nd International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. Krakow: IEEE, pp. 386–389. isbn: 978-1-5386-5395-1. doi: 10.1109/WAINA.2018.00116. url: https://ieeexplore.ieee.org/document/8418101/ (visited on 12/19/2023).

Authorization Overview (2023). en. Section: docs. url: https://kubernetes.io/docs/reference/access-authn-authz/authorization/ (visited on 12/19/2023).

Avrahami, Yuval (Dec. 2020). *Protecting Against an Unfixed Kubernetes Man-in-the-Middle Vulnerability (CVE-2020-8554)*. en-US. url: https://unit42.paloaltonetworks.com/cve-2020-8554/ (visited on 12/11/2023).

Beetz, Florian and Simon Harrer (July 2022). "GitOps: The Evolution of DevOps?" en. In: *IEEE Software* 39.4, pp. 70–75. issn: 0740-7459, 1937-4194. doi: 10.1109/MS.2021.3119106. url: https://ieeexplore.ieee.org/document/9565152/ (visited on 12/19/2023).

Bendovschi, Andreea (Jan. 2015). "Cyber-Attacks – Trends, Patterns and Security Countermeasures". In: *Procedia Economics and Finance*. 7th INTERNATIONAL CONFERENCE ON FINANCIAL CRIMINOLOGY 2015, 7th ICFC 2015, 13-14 April 2015,Wadham College, Oxford University, United Kingdom 28, pp. 24–31. issn: 2212-5671. doi: 10.1016/S2212-5671(15)01077-1. url: https://www.sciencedirect.com/science/article/pii/S2212567115010771 (visited on 01/04/2024).

Bose, Dibyendu Brinto, Akond Rahman, and Shazibul Islam Shamim (June 2021). "'Under-reported' Security Defects in Kubernetes Manifests". In: *2021 IEEE/ACM 2nd International Workshop on Engineering and Cybersecurity of Critical Systems (EnCyCriS)*, pp. 9–12. doi: 10.1109/EnCyCriS52570.2021.00009. url: https://ieeexplore.ieee.org/document/9476056 (visited on 01/04/2024).

Budigiri, Gerald et al. (June 2021). "Network Policies in Kubernetes: Performance Evaluation and Security Analysis". en. In: *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*. Porto, Portugal: IEEE, pp. 407–412. isbn: 978-1-66541-526-2. doi: 10.1109/EuCNC/6GSummit51104.2021.9482526. url: https://ieeexplore.ieee.org/document/9482526/ (visited on 12/30/2023).

Bush, Matthew and Atefeh Mashatan (Aug. 2022). "From Zero to One Hundred: Demystifying zero trust and its implications on enterprise people, process, and technology". en. In: *Queue* 20.4, pp. 80–106. issn: 1542-7730, 1542-7749. doi: 10.1145/3561799. url: https://dl.acm.org/doi/10.1145/3561799 (visited on 01/04/2024).

Calico (Nov. 2023). original-date: 2016-07-21T15:45:54Z. url: https://github.com/projectcalico/calico (visited on 11/27/2023).

Chen, Rui, Li Shanshan, and Li Zheng (Dec. 2017). "From Monolith to Microservices: A Dataflow-Driven Approach". en. In: *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*. Nanjing: IEEE, pp. 466–475. isbn: 978-1-5386-3681-7. doi: 10.1109/APSEC.2017.53. url: http://ieeexplore.ieee.org/document/8305969/ (visited on 11/14/2023).

Cilium (June 2024a). *Mutual Authentication (Beta) — Cilium 1.16.0-dev documentation*. url: https://docs.cilium.io/en/latest/network/servicemesh/mutual-authentication/mutual-authentication/ (visited on 06/19/2024).

– (2024b). *Component Overview — Cilium 1.13.17 documentation*. url: https://docs.cilium.io/en/v1.13/overview/component-overview/ (visited on 06/19/2024).

– (2024c). *Istio 1.0: How Cilium enhances Istio with socket-aware BPF programs*. url: https://cilium.io/blog/2018/08/07/istio-10-cilium/ (visited on 06/19/2024).

– (2024d). *Network Policy*. url: https://docs.cilium.io/en/v1.13/security/policy/ (visited on 06/19/2024).

Cloud Native Computing Foundation (CNCF) (2023). *GitOps Microsurvey*. en. Tech. rep. CNCF. url: https://www.cncf.io/wp-content/uploads/2023/11/CNCF_GitOps-Microsurvey_Final.pdf.

Cluster Networking (2023). en. Section: docs. url: https://kubernetes.io/docs/concepts/cluster-administration/networking/ (visited on 11/27/2023).

D'Silva, Daniel and Dayanand D. Ambawade (Apr. 2021). "Building A Zero Trust Architecture Using Kubernetes". en. In: *2021 6th International Conference for Convergence in Technology (I2CT)*. Maharashtra, India: IEEE, pp. 1–8. isbn: 978-1-72818-876-8. doi: 10.1109/I2CT51068.2021.9418203. url: https://ieeexplore.ieee.org/document/9418203/ (visited on 01/04/2024).

DEFCONConference (Aug. 2021). *DEF CON 29 - Guillaume Fournier, Sylvain Afchain, Sylvain Baubeau - eBPF, I thought we were friends!* url: https://www.youtube.com/watch?v=5zixNDolLrg (visited on 12/30/2023).

Delivering Quality at Speed With GitOps (2023). en. url: https://www.weave.works/blog/delivering-quality-at-speed-with-gitops (visited on 12/21/2023).

Encrypting Confidential Data at Rest (2023). en. Section: docs. url: https://kubernetes.io/docs/tasks/administer-cluster/encrypt-data/ (visited on 12/23/2023).

EndpointSlices (2023). en. Section: docs. url: https://kubernetes.io/docs/concepts/services-networking/endpoint-slices/ (visited on 12/03/2023).

Espe, Lennart et al. (2020). "Performance Evaluation of Container Runtimes:" en. In: *Proceedings of the 10th International Conference on Cloud Computing and Services Science*. Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, pp. 273–281. isbn: 978-989-758-424-4. doi: 10.5220/0009340402730281. url: http://www.

scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0009340402730281 (visited on 05/18/2024).

external-secrets (Dec. 2023). en-US. url: https://www.cncf.io/projects/external-secrets/ (visited on 12/23/2023).

Feature creep (Aug. 2023). en. Page Version ID: 1172494095. url: https://en.wikipedia.org/w/index.php?title=Feature_creep&oldid=1172494095 (visited on 12/27/2023).

flannel (Nov. 2023). original-date: 2014-07-10T17:45:29Z. url: https://github.com/flannel-io/flannel (visited on 11/27/2023).

Ganek, A. G. and T. A. Corbi (2003). "The dawning of the autonomic computing era". en. In: *IBM Systems Journal* 42.1, pp. 5–18. issn: 0018-8670. doi: 10.1147/sj.421.0005. url: http://ieeexplore.ieee.org/document/5386835/ (visited on 12/13/2023).

Gartner (2024). *Gartner Hype Cycle 2023*. en. url: https://www.gartner.com/en/articles/what-s-new-in-the-2023-gartner-hype-cycle-for-emerging-technologies (visited on 06/25/2024).

Google (2023). *Regional clusters*. en. url: https://cloud.google.com/kubernetes-engine/docs/concepts/regional-clusters (visited on 12/11/2023).

– (2024). *CIS Benchmarks*. en. url: https://cloud.google.com/kubernetes-engine/docs/concepts/cis-benchmarks (visited on 06/24/2024).

HashiCorp (2023). *How do you centrally manage and secure secret credentials, keys, and passwords?* en. url: https://www.hashicorp.com/resources/what-is-secret-sprawl-why-is-it-harmful (visited on 12/26/2023).

Hashicorp (2022). *Agent Sidecar Injector Overview*. en. url: https://developer.hashicorp.com/vault/docs/platform/k8s/injector (visited on 12/26/2023).

He, Yuanhang et al. (June 2022). "A Survey on Zero Trust Architecture: Challenges and Future Trends". en. In: *Wireless Communications and Mobile Computing* 2022. Ed. by Yan Huo, pp. 1–13. issn: 1530-8677, 1530-8669. doi: 10.1155/2022/6476274. url: https://www.hindawi.com/journals/wcmc/2022/6476274/ (visited on 12/22/2023).

Huang, Kaizhe and Pranjal Jumde (July 2020). *Learn Kubernetes Security: Securely orchestrate, scale, and manage your microservices in Kubernetes deployments*. en. Google-Books-ID: fmDwDwAAQBAJ. Packt Publishing Ltd. isbn: 978-1-83921-218-5.

IBM (2023). *Cost of a Data Breach Report*. en. Tech. rep., p. 78.

Idreos, Stratos and Mark Callaghan (June 2020). "Key-Value Storage Engines". en. In: *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. Portland OR USA: ACM, pp. 2667–2672. isbn: 978-1-4503-6735-6. doi: 10.1145/3318464.3383133. url: https://dl.acm.org/doi/10.1145/3318464.3383133 (visited on 12/23/2023).

Istio, 3 Minute (2024). *Architecture*. en. url: https://istio.io/latest/docs/ops/deployment/architecture/ (visited on 06/13/2024).

Istio.io (2024). *The Istio service mesh*. en. url: https://istio.io/latest/about/service-mesh/ (visited on 06/19/2024).

Kang, Zhuangwei et al. (Oct. 2021). "A Comprehensive Performance Evaluation of Different Kubernetes CNI Plugins for Edge-based and Containerized Publish/Subscribe Applications". en. In: *2021 IEEE International Conference on Cloud Engineering (IC2E)*. San Francisco, CA, USA: IEEE, pp. 31–42. isbn: 978-1-66544-970-0. doi: 10.1109/IC2E52221.2021.00017. url: https://ieeexplore.ieee.org/document/9610274/ (visited on 12/11/2023).

Kindervag, John (2010). "Build security into your network's dna: The zero trust network architecture". In: *Forrester Research Inc* 27. url: https://www.actiac.org/system/files/Forrester_zero_trust_DNA.pdf (visited on 01/04/2024).

Kindervag, John (2016). "No More Chewy Centers: The Zero Trust Model Of Information Security". en. In.

kube-proxy (2023). en. Section: docs. url: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/ (visited on 11/18/2023).

Kubernetes Project Journey Report (June 2023). en-US. url: https://www.cncf.io/reports/kubernetes-project-journey-report/ (visited on 01/06/2024).

Kyverno (2023). en. url: https://kyverno.io/ (visited on 12/27/2023).

Li, Wubin et al. (Apr. 2019). "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. ISSN: 2642-6587, pp. 122–1225. doi: 10.1109/SOSE.2019.00026. url: https://ieeexplore.ieee.org/document/8705911 (visited on 06/25/2024).

Limoncelli, Thomas A. (June 2018). "GitOps: A Path to More Self-service IT: IaC + PR = GitOps". en. In: *Queue* 16.3, pp. 13–26. issn: 1542-7730, 1542-7749. doi: 10.1145/3236386.3237207. url: https://dl.acm.org/doi/10.1145/3236386.3237207 (visited on 12/19/2023).

Linkerd (2024a). *Architecture*. en. url: https://linkerd.io/2.15/reference/architecture/ (visited on 06/19/2024).

– (2024b). *Why Linkerd doesn't use Envoy*. en. url: https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/ (visited on 06/19/2024).

Linux Foundation (2023). *namespaces - Linux manual page*. url: https://man7.org/linux/man-pages/man7/namespaces.7.html (visited on 12/26/2023).

Liu, Chang et al. (Dec. 2020). "A protocol-independent container network observability analysis system based on eBPF". en. In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. Hong Kong: IEEE, pp. 697–702. isbn: 978-1-72819-074-7. doi: 10.1109/ICPADS51040.2020.00099. url: https://ieeexplore.ieee.org/document/9359159/ (visited on 12/27/2023).

Marmol, Victor, Rohit Jnagal, and Tim Hockin (2015). "Networking in Containers and Container Clusters". en. In.

Meng, Lei et al. (Aug. 2022). "A continuous authentication protocol without trust authority for zero trust architecture". en. In: *China Communications* 19.8, pp. 198–213. issn: 1673-5447. doi: 10.23919/JCC.2022.08.015. url: https://ieeexplore.ieee.org/document/9861234/ (visited on 01/04/2024).

Microservices (2023). url: https://martinfowler.com/articles/microservices.html (visited on 11/14/2023).

Minna, Francesco et al. (Sept. 2021). "Understanding the Security Implications of Kubernetes Networking". In: *IEEE Security & Privacy* 19.5. Conference Name: IEEE Security & Privacy, pp. 46–56. issn: 1558-4046. doi: 10.1109/MSEC.2021.3094726. url: https://ieeexplore.ieee.org/document/9497237?denied= (visited on 11/18/2023).

Mitigate CVE-2020-8554 with Policy Controller (2023). en-US. url: https://cloud.google.com/blog/products/application-development/protecting-your-kubernetes-deployments-policy-controller (visited on 12/11/2023).

Mitre (2024). *CVE-2020-8554*. url: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8554 (visited on 06/15/2024).

Newman, Lily Hay (2023). "Hackers Enlisted Tesla's Cloud to Mine Cryptocurrency". en-US. In: *Wired* (). Section: tags. issn: 1059-1028. url: https://www.wired.com/story/cryptojacking-tesla-amazon-cloud/ (visited on 12/19/2023).

Newman, Sam (July 2021). *Building Microservices*. en. Google-Books-ID: ZvM5EAAAQBAJ. "O'Reilly Media, Inc." isbn: 978-1-4920-3397-4.

NSA (Aug. 2022). *Kubernetes Hardening Guide*.

Palo Alto Networks (2024). *What Is Microsegmentation?* en-US. url: https://www.paloaltonetworks.com/cyberpedia/what-is-microsegmentation (visited on 01/04/2024).

Pothula, Dharmanandana Reddy, Krishna Kumar, and Sanil Kumar (Oct. 2019). "Run Time Container Security Hardening Using A Proposed Model Of Security Control Map". en. In: *2019 Global Conference for Advancement in Technology (GCAT)*. BANGALURU, India: IEEE, pp. 1–6. isbn: 978-1-72813-694-3. doi: 10.1109/GCAT47503.2019.8978433. url: https://ieeexplore.ieee.org/document/8978433/ (visited on 12/27/2023).

Pros and Cons of GitOps (2023). *Pros and Cons of GitOps: Why More Companies Should Use GitOps? | Maxima Consulting*. en. url: https://www.maximaconsulting.com/newsroom/gitops-pros-cons (visited on 12/22/2023).

Raj, Vinay and Hanumanthu Bhukya (July 2023). "Assessing the Impact of Migration from SOA to Microservices Architecture". en. In: *SN Computer Science* 4.5, p. 577. issn: 2661-8907. doi: 10.1007/s42979-023-01971-2. url: https://doi.org/10.1007/s42979-023-01971-2 (visited on 11/14/2023).

Ramadoni, Ashudi, Ema Utami, and Hanif Al Fatta (Aug. 2021). "Analysis on the Use of Declarative and Pull-based Deployment Models on GitOps Using Argo CD". en. In: *2021 4th International Conference on Information and Communications Technology (ICOIACT)*. Yogyakarta, Indonesia: IEEE, pp. 186–191. isbn: 978-1-66543-394-5. doi: 10.1109/ICOIACT53268.2021.9563984. url: https://ieeexplore.ieee.org/document/9563984/ (visited on 12/19/2023).

Rice, Liz (Mar. 2023). *Learning EBPF*. en. "O'Reilly Media, Inc." isbn: 978-1-09-813509-6.

Rice, Liz and Michael Hausenblas (Nov. 2018). *Kubernetes Security*. en-US. isbn: 978-1-4920-3906-8. url: https://kubernetes-security.info/ (visited on 12/19/2023).

Rose, Scott et al. (Aug. 2020). *Zero Trust Architecture*. en. Tech. rep. National Institute of Standards and Technology. doi: 10.6028/NIST.SP.800-207. url: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-207.pdf (visited on 10/31/2023).

Sadiq, Amin et al. (Jan. 2023). "Detection of Denial of Service Attack in Cloud Based Kubernetes Using eBPF". en. In: *Applied Sciences* 13.8. Number: 8 Publisher: Multidisciplinary Digital Publishing Institute, p. 4700. issn: 2076-3417. doi: 10.3390/app13084700. url: https://www.mdpi.com/2076-3417/13/8/4700 (visited on 12/30/2023).

Secrets (2023). en. Section: docs. url: https://kubernetes.io/docs/concepts/configuration/secret/ (visited on 12/22/2023).

Securosis (Jan. 2018). *Securosis: Understanding and Selecting a Secrets Management Platform*.

Sedghpour, Mohammad Reza Saleh and Paul Townend (Aug. 2022). "Service Mesh and eBPF-Powered Microservices: A Survey and Future Directions". In: *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. ISSN: 2642-6587, pp. 176–184. doi: 10.1109/SOSE55356.2022.00027. url: https://ieeexplore.ieee.org/document/9912629 (visited on 06/25/2024).

Service (2023). en. Section: docs. url: https://kubernetes.io/docs/concepts/services-networking/service/ (visited on 12/12/2023).

Services, Load Balancing, and Networking (2023). en. url: https://kubernetes.io/docs/concepts/services-networking/ (visited on 12/03/2023).

sigstore (2023). en. url: https://www.sigstore.dev/undefined/ (visited on 12/27/2023).

Sun, Liu (Mar. 2024). *Future of Service Mesh is Sidecar-less with Istio Ambient Mesh*. en-US. url: https://kccnceu2024.sched.com/event/1aQgX/future-of-service-

mesh-is-sidecar-less-with-istio-ambient-mesh-project-lightning-talk (visited on 06/18/2024).

Susnjara, Stephanie (Nov. 2023). *Top 6 Kubernetes use cases*. en-US. url: https://www.ibm.com/blog/kubernetes-use-cases/www.ibm.com/blog/kubernetes-use-cases (visited on 01/03/2024).

The History of GitOps (2023). en. url: https://www.weave.worksurl! (visited on 12/19/2023).

The Linux Foundation (2024). *Assigning Pods to Nodes*. en. Section: docs. url: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/ (visited on 05/18/2024).

– (2023a). *Considerations for large clusters*. en. Section: docs. url: https://kubernetes.io/docs/setup/best-practices/cluster-large/ (visited on 12/03/2023).

– (2023b). *Update API Objects in Place Using kubectl patch*. en. Section: docs. url: https://kubernetes.io/docs/tasks/manage-kubernetes-objects/update-api-object-kubectl-patch/ (visited on 12/19/2023).

Thönes, Johannes (Jan. 2015). "Microservices". en. In: *IEEE Software* 32.1, pp. 116–116. issn: 0740-7459, 1937-4194. doi: 10.1109/MS.2015.11. url: https://ieeexplore.ieee.org/document/7030212/ (visited on 11/13/2023).

Threat Model (2023). url: https://external-secrets.io/latest/guides/threat-model/ (visited on 12/24/2023).

Tiwari, Himanshu (2023). "Demystifying Kubernetes Service Types: ClusterIP, NodePort , LoadBalancer & ExternalName". en. In: 4.

Uctu, Goksel et al. (Oct. 2019). "Perimeter Network Security Solutions: A Survey". en. In: *2019 3rd International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*. Ankara, Turkey: IEEE, pp. 1–6. isbn: 978-1-72813-789-6. doi: 10.1109/ISMSIT.2019.8932821. url: https://ieeexplore.ieee.org/document/8932821/ (visited on 11/04/2023).

Ward, Rory and Betsy Beyer (2014). "A New Approach to Enterprise Security". en. In: 39.6.

Waseem, Muhammad, Peng Liang, and Mojtaba Shahin (Dec. 2020). "A Systematic Mapping Study on Microservices Architecture in DevOps". In: *Journal of Systems and Software* 170, p. 110798. issn: 0164-1212. doi: 10.1016/j.jss.2020.110798. url: https://www.sciencedirect.com/science/article/pii/S0164121220302053 (visited on 11/14/2023).

Wazuh (2023). *Overview*. en-US. url: https://wazuh.com/platform/overview/ (visited on 12/27/2023).

What is eBPF? (2023). *What is eBPF? An Introduction and Deep Dive into the eBPF Technology*. en. url: https://ebpf.io/what-is-ebpf/ (visited on 12/27/2023).

What is mTLS? (2023). *What is mTLS? | Mutual TLS*. en-us. url: https://www.cloudflare.com/learning/access-management/what-is-mutual-tls/ (visited on 12/30/2023).

What's in a name? (2023). *What's in a name? Moving GitOps beyond buzzword*. en. url: https://github.com/readme/featured/defining-gitops (visited on 12/19/2023).

William Morgan (2024). *Benchmarking Linkerd and Istio: 2021 Redux*. en. url: https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/ (visited on 06/16/2024).

Zhu, Xiangfeng et al. (Oct. 2023). "Dissecting Overheads of Service Mesh Sidecars". en. In: *Proceedings of the 2023 ACM Symposium on Cloud Computing*. Santa Cruz CA USA: ACM, pp. 142–157. isbn: 9798400703874. doi: 10.1145/3620678.3624652. url: https://dl.acm.org/doi/10.1145/3620678.3624652 (visited on 06/25/2024).