Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Experimental Performance Analysis of Service Mesh Systems

Author: Richard Bieringa (2544975)

1st supervisor:	Prof. Dr. Ir. Alexandru Iosup
daily supervisor:	Ir. Erwin van Eyk
2nd reader:	Dr. Ir. Animesh Trivedi

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

September 21, 2022

"I've always believed that you should never ever give up and you should always keep fighting even when there's only a slightest chance."
by Michael Schumacher

Abstract

Service-oriented computing, an architectural approach of decomposing software into logical services, has seen an increase in popularity over the past decades. Later forms, such as the widely popular microservices-based architecture is an evolution of service-oriented computing made possible by the reductions in deployment costs and resource overhead. These architectures offer advantageous characteristics, such as an increase in flexibility and scalability. However, due to its design, it also introduces an additional layer of complexity due to the overhead in communication and the inherent reliability issues from networks. The service mesh architecture tries aims to solve these issues whilst providing a uniform way of handling security, observability, and reliability concerns without modifying application code. In this thesis, we dissect the state-of-the service mesh systems and identify three different service mesh architectures. Furthermore, we design *Mesh Bench* a benchmark used to evaluate the performance characteristics of service mesh systems. We implement a prototype of the benchmark and take an experimental approach to evaluating the most popular service mesh implementations in the form of *Istio* and *Linkerd*, and implementations using vastly different architectures in the form of *Cilium* and *Traefik mesh*. Based on the results of the performance experiments we uncover the performance overheads of service mesh systems and identify promising, future directions and evolutions of the technology.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

1

Introduction

How to design and build the IT infrastructure of the present, and of the future? A difficult, but important question in the era distinguished by globalization and digitalization. More and more of society is relying on the internet, and its wide variety of products and service offerings. From online stores, to productivity tools, streaming services to news websites, modern IT infrastructure empowers many areas and is a staple of the global economy. This was amplified and made abundantly clear in the last couple of years, where the world as a whole, moved online to prevent the further spread of COVID-19.

There are many designs and approaches that are currently in use within the field (1, 2, 3, 4). So far, there has not been a single approach, a golden architecture, that answers all the toughest challenges that we are facing today. How to minimize the end-to-end latencies, and improve reliability to improve the seamless end-user experience? How to deal with the growing threats of cybersecurity and the rise of data privacy threats? How to optimize the productivity of developers so that they can focus on business logic instead of infrastructure related concerns? These are just a few of the most pressing challenges (5), that should be considered when designing and building a modern, distributed system.

Recent trends and developments in deployment models have paved the way for the currently popular *microservices architecture*. An architectural style characterized by separating applications in many small, independent parts, that have their own responsibilities. This style of architecture is made feasible by the low cost of computational overhead caused by container technologies and emerging workload managers such as Kubernetes. With most of the organizations (71% as observed in 2021 by Statista (6)) adopting or using a microservices architecture in some for, it is important to analyse the benefits and especially the shortcomings of such an approach.

The architectural style has many benefits, such as the ability to decompose complex problems into smaller ones and allow teams to work on decoupled services. It also enables better scaling, as individual components can now be replicated instead of entire applications.

However, it also introduces a new layer of complexity, that comes in the form of service-to-service communications. Networking brings numerous challenges, it introduces additional latencies and poses security threats. The architectural style has to deal with routing and load balancing and consider the possibility of failing links in inherently unreliable networks.

For many years, companies approached this set of problems by changing the application code responsible for service-to-service communications, often in the form of client libraries that did so in a uniform manner. However, recent trends and community-wide efforts empowered the rise of the service mesh architecture. This architectural paradigm aims to solve similar challenges, without having to modify any application code or having to stick to a single programming language or framework. The architecture works by introducing a dedicated infrastructure layer, that facilitates service-to-service communications, achieved by adding network *proxies* in front of the logical services. These proxies intercept communications from and to these services, and provides a uniform approach of dealing with service-to-service traffic, enabling observability into communications, secure connections and reliability mechanisms.

Within this emerging field of service mesh systems, we focus on the performance implications of these systems. What is the computational overhead of the additional machinery introduced, and what is the impact of latency through the additional network hops? In this thesis, we present an extensive study into the performance characteristics of service mesh systems. First, we conduct a systems survey, in which we identify and analyse the state-of-the-art service mesh systems. After this, we design and introduce *Mesh Bench*, a benchmarking instrument that can evaluate the performance characteristics of service mesh systems. Finally, with a focus on reproducibility, we design, perform and analyse performance related experiments of service mesh systems.

1.1 Context

We live in the *Age of Computer Ecosystems* (7), where our society as a whole relies on the modern IT infrastructure. From governments to schools and businesses, many use the internet as a gateway to their services and offerings. The design of modern IT systems and infrastructure therefore has a large societal impact, for both the end user, the providing entities, and the every increasing number of engineers (8) that are required to develop and manage it.

Businesses that rely on IT infrastructure, can face major economic impact through slight design alterations. Amazon, for example, found that for every 100ms of additional end-to-end latency, it would lose out on roughly 1% of total sales revenue (9). And Google, indicated back in 2009 (10) that latency has a direct relation with service usage and has been actively prioritizing performance of services ever since

In this thesis, we evaluate the service mesh architecture and systems that implement this. These systems introduce an additional layer of machinery that naturally causes a performance overhead. Even though, the performance complications can have major impact, the systems in question have received lots of attention from the industry and increasing number of companies are evaluating and using such solutions in their IT infrastructure (11). With large corporations, such as Google and Microsoft backing the development of service mesh systems it is clear that this technology has lots of potential.

The service mesh landscape is a rapidly evolving field, that is in its relative infancy. There have been numerous new systems that have emerged in the last couple of years and there have been many technological advancements that pave the way for alternative approaches to the problem. It comprises a landscape where most of the systems are in active development and see frequent improvements and architectural changes (12, 13). This grants us the opportunity to evaluate the different approaches these systems take, as they evolve and explore an exciting field of bleeding-edge technologies.

Although service mesh systems have gained a lot of attention and hype from within the industry. It has received relatively little to no attention from academia as observed by our previous works in which we have conducted a topical survey of the field. In this thesis, we aim to close that gap by performing an extensive study on service mesh systems, with a focus on their performance characteristics.

Throughout this thesis, we align ourselves with the goals and visions expressed in Massivizing Computer Systems (MCS) (7). The work as presented in this thesis aims to uncover the intricacies of the computer ecosystems discussed, helping to understand them and guide future system designs. MCS envisions a domain where everything developed is tested and benchmarked in a reproducible manner. This ideology is exemplified by the experimental approach used throughout this thesis.

1.2 Problem Statement

The service mesh architecture is an emerging approach to address several challenges presented in distributed systems. As a by-product of in the increasingly popular service-oriented design, applications increasingly rely on service-to-service communications over the network. Systems that adhere to a service mesh architecture, aim to abstract away the challenges and intricacies of dealing with unreliable and dynamic networking environments, by introducing a dedicated layer of infrastructure between the applications services.

The service mesh architecture (see Figure 1.1), introduces a set of networking *proxies* to intercept all communications to and from application *services*. These proxies then forward the requests to the original intended recipients whilst dealing with the challenges of a service-oriented design. The proxies are constantly aware of the up-to-date services in a

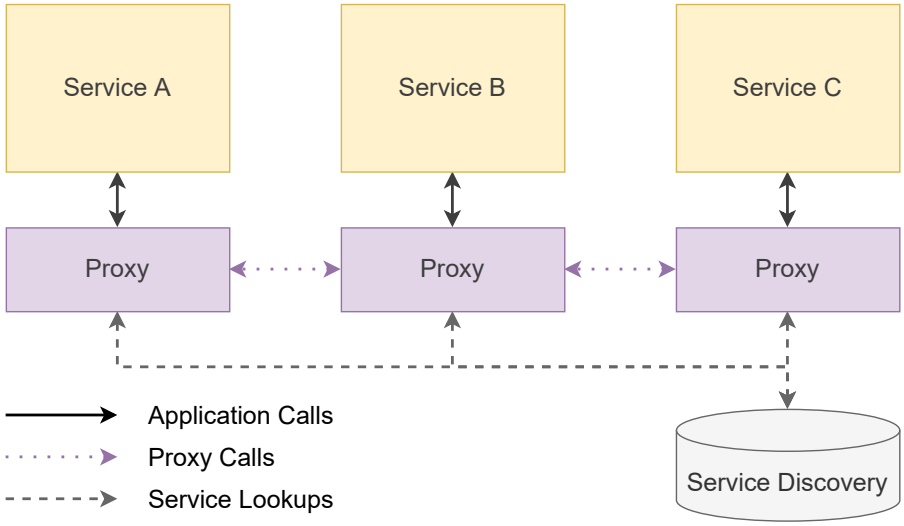


Figure 1.1: Simplistic depiction of a service mesh architecture. The core concept is that the service-to-service communications are intercepted and processed by intermediary network proxies.

dynamic system, through *service discovery* mechanisms. The proxies establish a uniform layer that can improve the infrastructure in four areas of interest; *observability*, *reliability*, *security*, and *programmability* (see background Section 2.4). The outstanding feature of a service mesh architecture is that the underlying services are not aware that their traffic is intercepted. This means that a service mesh system can be introduced without having to modify any application code.

Service mesh systems form a relatively new and exciting field, that is rapidly evolving. However, it is also a field where there is very limited output from the academic world. Specifically, there is very little known about the performance implications of these systems and the additional machinery these systems introduce. The goal of this thesis is to answer this unsolved challenge, and perform an extensive study into performance implications of service mesh systems.

Before diving into the performance implications of service mesh systems we need to establish an overview of the field by identifying current service mesh systems. After this, we have to figure out what the functional components of a service mesh are and figure out how they differ between the identified systems. During this process, we identify different implementation details and entirely different architectural approaches to the problems they aim to solve. This process allows us to see how the service mesh systems functionally differ from one another. By taking such an approach, we can create a comparison framework, which allows us to objectively compare service mesh systems and allows us to hypothesize on the performance implications.

After we have laid down the groundwork to functionally compare service mesh systems,

we have to design and implement a system that can evaluate the performance of them. To properly evaluate service mesh systems and test our hypothesizes, we have to design a benchmark system that captures the performance implications of the functional components of a service mesh system. In this process, we have to clearly define the boundaries of the system and its components that we wish to evaluate. Furthermore, we have to establish a set of metrics that are relevant to service mesh systems and make sure that the benchmark system produces reproducible results. Additionally, we have to create a configurable system that can emulate a wide range of workloads. This allows us to evaluate service mesh systems in various environments.

To thoroughly understand the performance implications of service mesh systems, we have to design a set of experiments that emulate common real-world scenarios. We do this by identifying common workloads in service-oriented architectures. What are the common communication protocols in such environments, and how are they used? By designing experiments that emulate these workloads and using our benchmark system, we can objectively compare the performance implications of service mesh systems. This allows us to perform a quantitative analysis, which enables us to identify performance overheads and potential bottlenecks. Furthermore, it allows us to correlate functional differences and architectural approaches to the quantitive results, enabling an extensive performance analysis.

1.3 Main Research Questions

We decompose the challenges as described in the problem statement Section 1.2 in the following research questions.

RQ1 How to compare, and evaluate service mesh systems?

Many service mesh implementations have emerged from within the rapidly changing landscape of container and resource management technologies, before the adaptation of any standard to guide their development. It is necessary to evaluate these implementations in a systematic systems survey, where we identify and analyse the characteristics of the current iterations of service mesh systems. This allows us to create a comparison framework, that acts an overview of the state-of-the-art within the service mesh landscape.

RQ2 How to design and implement a benchmark that evaluates the performance of service mesh systems?

Based on the results of the system survey in **RQ1**, we identify the requirements for a system which that can quantitively evaluate the performance characteristics of a service mesh system. Based on these identified requirements, we should evaluate

existing instruments to see if any of these satisfy. We then have to design or extend an instrument guided by benchmarking best practices (14). The benchmarking instrument should align with the goals and visions of MCS in which we aim to produce systematic and reproducible experiments.

RQ3 What are the differences between current service mesh systems in terms of overhead, throughput, and latency?

Based on the benchmark as designed in **RQ2**, we have to conduct performance oriented experiments on different service mesh systems. We should design and perform experiments that simulate various workload patterns for such a system. The experiments should evaluate service mesh systems while they experience various levels of load in the form of throughput. The experiments should also capture the effects of various payload types and sizes. Finally, the experiments should be reproducible so that anyone can verify the results.

1.4 A Distributed Systems Approach

In this thesis, we approach the problem statement and their resulting research questions using distributed systems approach, a consolidation of best practices as taught and utilized by the members AtLarge group[1]. This approach follows the vision of MCS (7) in which it guides the thesis and the conceptual, technical and experimental work that it consist of.

First, to address **RQ1** we perform an extensive research into the state-of-the-art service mesh systems. To accomplish this, we perform a systematic systems survey. This will help us to identify various service mesh systems that exist and helps us to understand these systems in more detail. Additionally, it allows us to identify key characteristics of these systems in the form of Functional Requirements and Non-Functional Requirements. Based on these findings we can construct a framework, that represents an overview of the identified service mesh systems and allows us to compare them to one another. This part of the research helps to establish the groundwork, in which we identify the key components and architectures that influence the performance characteristics.

Secondly, we tackle **RQ2** based on the key findings from **RQ1**. With an extensive understanding of service mesh systems, we aim to design a benchmark instrument that is capable of evaluating the performance characteristics. We perform an extensive requirements analysis, in which we establish the stakeholders for such a benchmarking instrument and their respective use cases. Once these requirements are clearly defined we guide our design through established best practices that helps us to capture the most significant metrics in evaluating distributed systems. After this, we implement a prototype and validate the

[1] `https://atlarge-research.com/`

design requirements.

Ultimately, to address **RQ3** we use the instrument as devised from **RQ2**. We design experiments and workloads that evaluate the service mesh systems under various conditions. We construct an experimental environment that aims to minimize external impacts on the results of these studies and measure the variation of our experimental environment by performing various micro-benchmarks. The extensive experimental approach follows the vision of MCS and aims to construct and perform reproducible experiments that can answer the performance related impacts of service mesh systems.

1.5 Main Contributions

This thesis introduces several qualitative, quantitative and technical contributions that assist in answering the formulated research questions. Each contribution is linked to a main research question. The contributions are as follows:

1. (*Conceptual*, **RQ1**) A qualitative systems survey on the characteristics of state-of-the-art service mesh systems Chapter 3.
2. (*Conceptual*, **RQ1**) Extensive analysis on the design and architectural approach of state-of-the-art service mesh systems Chapter 3.
3. (*Conceptual*, **RQ2**) Requirements analysis and design of a service mesh benchmarking instrument Chapter 4.
4. (*Technical*, **RQ2**) Prototype implementation of the designed benchmarking instrument, *Mesh Bench* Chapter 4.
5. (*Experimental*, **RQ3**) Design and deployment of performance oriented experiments that evaluate service mesh systems under various environments Chapter 5.
6. (*Experimental*, **RQ3**) Quantitative performance results and extensive analysis of service mesh systems Chapter 5.

1.6 Reading Guide

This chapter briefly introduced the impact of system design and how current approaches evolved and why. It then discussed the recent emergence of service mesh systems and the challenges it tries to solve. However, it briefly touches upon many subjects that are part of the bigger picture that is addressed throughout the work in this thesis. It is advisable to read the contents of this thesis from the front to the back. The order of the material discussed builds on top of another and is prefaced by a background chapter that introduces

several concepts in greater detail. These concepts are used throughout the entirety of the thesis and the sections in the background chapter provide the reader with enough knowledge to understand the materials discussed.

The following chapter (Chapter 2), introduces some of these concepts in greater detail. In Section 2.1 we introduce the notion of a *container* in more detail and how it had a significant impact on cost reductions for deployments in comparison to earlier models. In Section 2.2 we discuss several evolutions in architectural design and why it gained traction. We introduce the notion of a service-oriented approach to software design and the challenges that it brings. Section 2.3 briefly introduces Kubernetes and several related concepts that return in architectural designs (in the system survey analysis Section 3.5) and experiments (in the experimental environment Section 5.1). Following that, in Section 2.4 we introduce the service mesh architecture in greater detail. It describes a general service mesh system and the challenges it tries to solve. Furthermore, it gives a comparison to related solutions that aim to solve similar challenges. Thereafter, in Section 2.5 we discuss the Cloud Native Computing Foundation (CNCF) a governing body in the field. Throughout this thesis, we use several projects maintained by the foundation and use several of their surveys, as key insights. Finally, in Section 2.6 we introduce the related work in this field.

2

Background

A decade ago, the service mesh architecture and systems implementing it did not exist, at least not in its current form. It is a by-product of the evolution in application architectures and the trends in cloud-native application design (15). The shift to service-based architectural styles became popular when the DevOps movement gained traction and became standard in the industry (16) Although this shift improved the speed and agility of software development (17) it came at the cost of additional operational complexity and overheads which service meshes try to solve.

The remainder of this chapter introduces several concepts and related topics relevant to this thesis. The topics are introduced in a specific order, which paints a general picture of the evolution and progression of technologies in the landscape. First, we introduce the concept of container technologies and elaborate on the mass adoption caused by Docker (Section 2.1). Secondly, we introduce the concept of micro-services as a natural evolution of the Service Oriented Architecture (SOA) architecture paradigm, and how it is further enabled by the adoption of container technologies (Section 2.2). Furthermore, we introduce Kubernetes, the de facto standard in container orchestration and which problems it tries to solve (Section 2.3). Afterwards, we go into the details of service mesh technology where we introduce the core characteristics that define such a system and what problems this additional layer of networking abstraction tries to solve (Section 2.4). Thereafter, we introduce the Cloud Native Computing Foundation (CNCF), a governing body in the field which is frequently mentioned throughout this thesis. Finally, in Section 2.6, we introduce the related work.

2.1 Containers

A container is a unit of executable software that is packaged with all of its dependencies[1]. By packaging everything in a single unit it makes it easy to run the application on different environments, from desktops and laptops to cloud environments. Containers isolate software from their environment and ensure that it works uniformly on all devices regardless of operating systems used or the hardware powering it.

Containers leverage a set of Linux technologies, such as *cgroups*[2] and *namespaces*[3]. The former is a mechanism to organize processes in hierarchical groups whose usage of various system resources can then be limited and monitored. The latter is a technology to wrap a system resource in an abstraction to make it appear to a process as if they had their own isolated instance of that resource. These combined with the fact that a container has a layered file system that contains the application code and operating system dependencies makes it so that it is a fast and lightweight unit of compute.

A comparison with virtual machines is commonly made because of their isolating properties and abstractions. However, a virtual machine is a software-based virtualization of an entire computer system, this includes the hardware and the entire operating system that runs on top of it. A container on the other hand is an abstraction at the application level, where it is just another process living in user space. A comparison of the different deployment models can be seen in Figure 2.1.

Docker was introduced in Santa Clara at PyCon in 2013 (18). It jump started the revolution of containers and was for many the first introduction to container technologies. It was loved by developers for its portability and speed compared to traditional virtual machine deployments that were commonly used. Due to its immense popularity and praise, which it has managed to uphold according to recent surveys as conducted by stack overflow (19), Docker became the de facto standard in container technologies. The similarly named company decided in 2015 to establish an open-source initiative that governs container related standards[4]. Along with this they donated their container image specification format[5] to this initiative as an open-source contribution. With this image specification standard in place, others could develop compatible container runtimes which were able to run the container images produced by said specification.

[1] `https://www.docker.com/resources/what-container`
[2] `https://man7.org/linux/man-pages/man7/cgroups.7.html`
[3] `https://man7.org/linux/man-pages/man7/namespaces.7.htm`
[4] `https://opencontainers.org/`
[5] `https://github.com/opencontainers/image-spec`

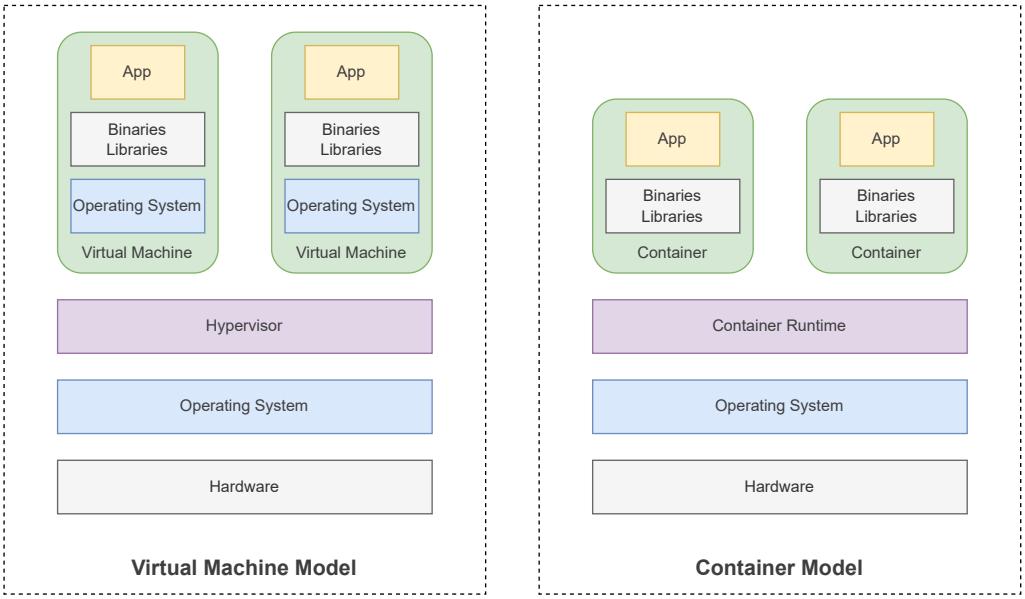


Figure 2.1: Comparison of container and virtual machine-based deployment models.

2.2 A Shift in System Design

In the previous section, we introduced containers technology and Docker (Section 2.1). These technologies reduced the cost of deployment by introducing a small, standardized unit of software. The adoption of container technology led to a shift in architectural design and led to an increase in usage of Service Oriented Architecture (SOA) design patterns. Due to the little overhead in terms of system resources caused by the container technology, it was an ideal candidate for independent self-contained components. This section will introduce the concept of a logical *service* in distributed systems and documents the general trends caused by the shift in architectural paradigms.

2.2.1 The Software Monolith

Before the adoption of SOA principles and the notion of services in general, companies, and organizations alike used to create so-called software monoliths. A monolithic approach in software design produces a self-contained software program where all of its dependencies, data access patterns and user interfacing components are combined (as depicted in Figure 2.2). This makes monolithic architectures difficult to use in distributed systems without ad-hoc solutions or frameworks (20). A software monolith can have several advantages, it produces a single binary, all code is colocated together, and it is a battle-tested architecture. However, it can also have several disadvantages, it can be hard to maintain, codebases can become gigantic over time and lead to accumulated technical debt that makes the product unmaintainable with reasonable efforts (21). This architectural pattern

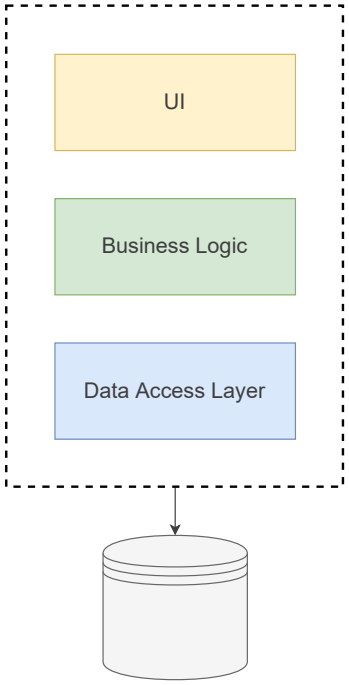


Figure 2.2: A monolithic architecture. In this form of software design, all the software is contained into a single, self-contained software program depicted by the dashed line.

also forces the developers of the application to stick to their technical architecture, such as the choice of programming language and frameworks used. Furthermore, it can suffer from dependency hell, where updating or adding libraries can break existing systems (22). Finally, the architecture does not scale very well, by scaling the application every single aspect or module has to be duplicated which is inefficient if only a subset of the application is put under load.

2.2.2 A Service-Oriented Approach

As previously stated, the monolithic architecture was not well suited for use in distributed systems. This led to a newer style of software architectures that decomposed the business logic into logical *services*, where functionalities are encapsulated and abstracted from context (23). This architectural paradigm meant that applications have to be decomposed into several self-contained units, which are then exposed via a *service interface*. The service interface utilizes common communication standards so that it can be incorporated in new applications without much hassle (24). Each service contains the code and data integrations required to execute a discrete business function. The core idea behind the SOA paradigm is that it promotes reusability and component sharing. This then translates into several benefits, such as an increase in scalability because it allows for scaling at a service level instead of having to scale an entire monolith. Furthermore, it allows software developers to

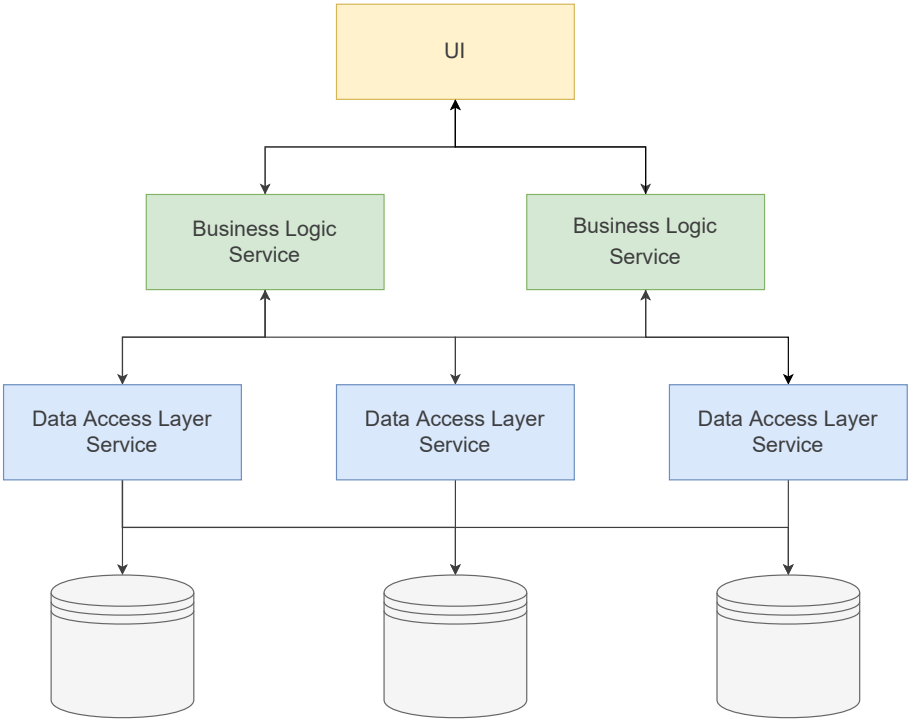


Figure 2.3: A service-oriented approach. This architectural approach allows engineers to split applications into separate business components which enables them to scale and operate individually.

use multiple technologies and frameworks, making it easier to pick the right tools for the job. This is enabled by the fact that services can exchange data over common protocols and agreed upon data representations, such as JavaScript Object Notation (JSON) for example.

2.2.3 Microservices

The *microservices* architecture (Figure 2.4) is an architectural style that emerged from the SOA. This evolvement was pushed by the DevOps movement and enabled by the decrease in deployment costs from the emerging container technology (17). The term *microservice* dates in the context of distributed systems dates at least as far back as 2013 (25). It realizes its distinct itself from the SOA paradigm by having a strong focus on its degree of independence regarding development and operation (26). The services that make up a SOA, can range from small, specialized services to enterprise-wide services, whereas a microservice consists of a highly specialized service, designed to do one thing well. This finer granularity allows for individual teams focussing on select subsets of an application and increases agility and development speed. However, this amplifies the problems that were mentioned above as it introduces more individual units in a complex system. Large

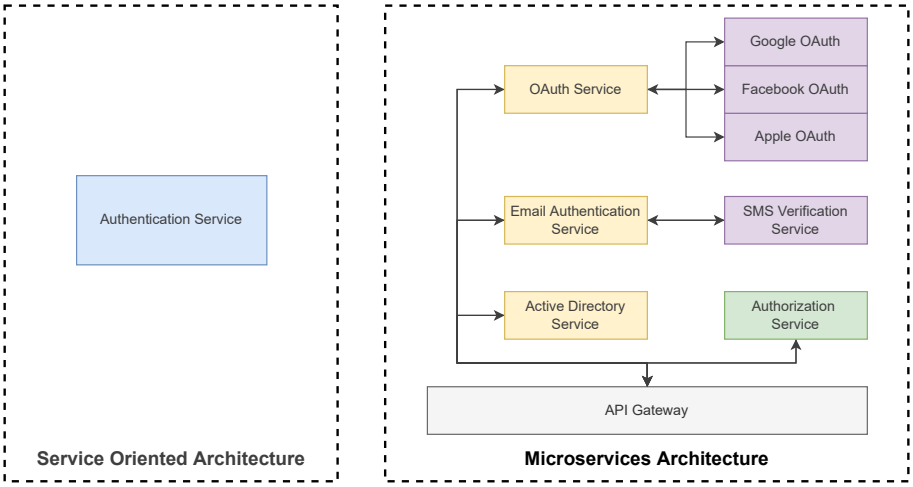


Figure 2.4: Service-Oriented architecture vs. microservices architecture both depicting an authentication service. Note: The granularity of the service definition depicts the most significant difference in the evolution of design.

companies such as Netflix that use such a microservices' architecture for example have reportedly managed to accumulate over 1000 microservices as of 2021 (3, 27).

2.2.4 A New Set of Challenges

However, by splitting the application up into several components, we introduce additional complexities. First off, the coarse granularity of the service-oriented design means that testing and validating every combination and condition may be very complex or even impossible (28). Furthermore, the loosely coupled services might be an architect's dream; however, it introduces additional complexities for a software developer (29). Additionally, the service-to-service communications can now introduce failures, especially if they are carried out over unreliable networks such as the internet. Finally, the interoperability of services introduces additional challenges. How and where can we reach the services? Which version of the service is running and is it still compatible with everything? If there are multiple instances of this service, which one should be targetted to prevent overloading and ensure equal load?

One approach used to solve these problems was through software libraries to handle all service-to-service communications (Figure 2.5) in a uniform way (30). Companies like Google, Netflix, and Twitter developed custom software libraries for this, such as *Stubby* (31), *Hystrix* (32) and *Finagle* (33) respectively. These libraries would perform load balancing, implement retry mechanisms, routing, and telemetry. A downside of this approach was that this meant that the libraries were usually written in a single programming language, locking the developers in, or resulting in having to support multiple libraries.

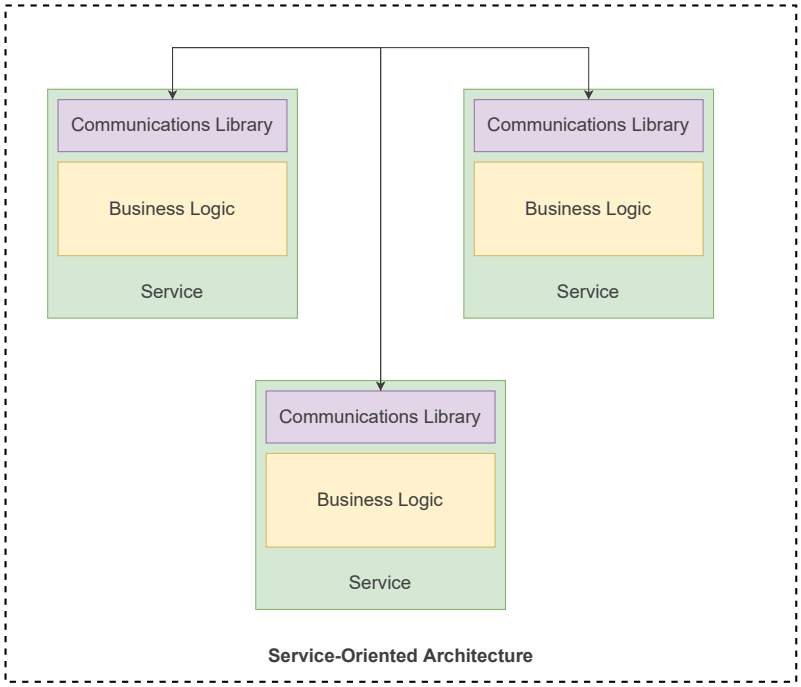


Figure 2.5: Software library approach for solving the challenges introduces by a service-oriented design. In this design, the software library implements a uniform client and server API, designed to handle fault tolerance, load balancing and latency optimizations to construct high-concurrency services.

Furthermore, it resulted in a scenario where updating the library meant that every service implementing it also required an update.

2.3 Kubernetes

In the previous sections we discussed the changes in deployment models (Section 2.1) and how this caused a shift in design based on decoupling business logic (Section 2.2). In this section, we introduce Kubernetes, a container-centric resource manager, that is the de facto standard within the field. With over 96% of organizations using or evaluating Kubernetes according to a recent survey (11) conducted by the the Cloud Native Computing Foundation (CNCF), the industry-leading foundation for container technologies, it is clear that the technology had a large impact on the industry and technological developments around it. Throughout the remainder of the work presented in this thesis we use Kubernetes, as most of the systems we examine and evaluate, rely or build on top of Kubernetes. We also briefly introduce some relevant Kubernetes related concepts, which are used throughout this thesis, such as the architectural diagrams discussed in Section 3.5.2.

The Kubernetes project was initially conceived by Google and started in June 2014 and

launched in early 2015 at OSCON (34). It was designed to be a container-centric resource manager, and its design was heavily influenced by the internal tooling used within Google (35). With the container as unit of compute, Kubernetes provides a streamlined approach to declaratively manage workloads. From *service discovery* and *load balancing* to *storage orchestration*, Kubernetes aims to solve the many challenges that appear when managing large amounts of containers in production environments. Based on concepts of *Control Theory* (36, 37), the system uses a declarative approach which tries to keep the ecosystem in a desired state at any given time.

Kubernetes consists of a set of components[1] that have to be installed on several *nodes*[2] (physical or virtual machines) to form a compute *cluster*. The core idea of Kubernetes is that is controlled through a centralized *kube-apiserver*, which controls all components and acts as an interface for operators to control the cluster and manage workloads running on it.

2.3.1 The Kubernetes Networking Model

Kubernetes relies on a specific networking model, which makes it easier for the *cluster operator*[3] to manage and direct network traffic within the cluster. The networking model required essentially comes down to the following rules (38).

1. All containers can communicate with all other containers without *NAT*
2. All nodes can communicate with all containers (and vice-versa) without *NAT*
3. The IP that a container sees itself as is the same IP that others see it as

Kubernetes does not come with a solution that implements this networking model, however, it requires it for operation. There are many ways to implement the Kubernetes networking model, third-party solutions (often referred to as a *Container Networking Interface*) such as *Calico* or *Flannel*. Another option commonly used to make use of *Virtual Private Cloud* solutions offered by various cloud providers such as Amazon Web Services or Google Cloud Platform, which allow the user to emulate this flat networking model without having to install third-party software solutions.

2.3.2 Pod

Although Kubernetes is a container-centric resource manager, it does not use the container abstraction to manage the workloads within the ecosystem. Instead, it uses an abstraction

[1] `https://kubernetes.io/docs/concepts/overview/components/`

[2] `https://kubernetes.io/docs/concepts/architecture/nodes/`

[3] The term cluster operator is used throughout this thesis to indicate the person operating a Kubernetes cluster, i.e. deploying workloads, managing networking and security etc.

on top of containers, namely the Pod. Pods are the smallest deployable units of computing that you can create and manage in Kubernetes[1].

It consists of one or more containers that emulate a logical *host*, much like a virtual machine would. This means that each Pod has its own isolated *networking stack*[2] through *namespaces* (Section 2.1), and that storage volumes are shared between containers inside a Pod. Networking between containers within the same Pod can be achieved through the *loopback device* and reached through *localhost* and there cannot be multiple processes that listen on the same port, much like a regular host. This is important to note, as this concept and network isolation paradigm is used by the most common service mesh architecture discussed in this thesis (see Section 3.5.2.1).

2.3.3 Service

The final Kubernetes concept we introduce is the notion of a *service*[3]. In Kubernetes the service abstraction provides a mechanism to expose an application as a network service. Briefly explained, it provides a stable IP address that is used to reach a set of Pods allowing the cluster operator to expose those applications and provide a stable interface to reach them.

Although we do use the Kubernetes service abstraction in this thesis during our experimental evaluation Chapter 5, it can cause some confusion since it conflicts with the definition of a logical service as explained in Section 2.2. Therefore, it is important to note that the term service refers to a *logical service* in the context of this thesis and not to the Kubernetes concept with a similar name, unless specifically stated otherwise.

2.4 Service Mesh

In the previous sections we discussed the evolution in deployment models (Section 2.1) and how this caused a shift in system design, based on decoupling business logic into logical services (Section 2.2). We then went on to briefly introduce Kubernetes and several related concepts (Section 2.2). In this section, we build on top of the topics discussed in the background chapter and introduce the *service mesh architecture*, which is at the core of the work presented in this thesis. It is an architectural style, that aims to solve the communicative challenges caused by service-oriented design.

A service mesh is a dedicated layer of networking infrastructure that sits between logical software services and aims to solve some communicative challenges introduces by a SOA.

[1] `https://kubernetes.io/docs/concepts/workloads/pods/`

[2] The networking stack, in the context of this thesis, refers to the Linux networking stack and consists of all the layers a packet has to traverse on a Linux-based system.

[3] `https://kubernetes.io/docs/concepts/services-networking/service/`

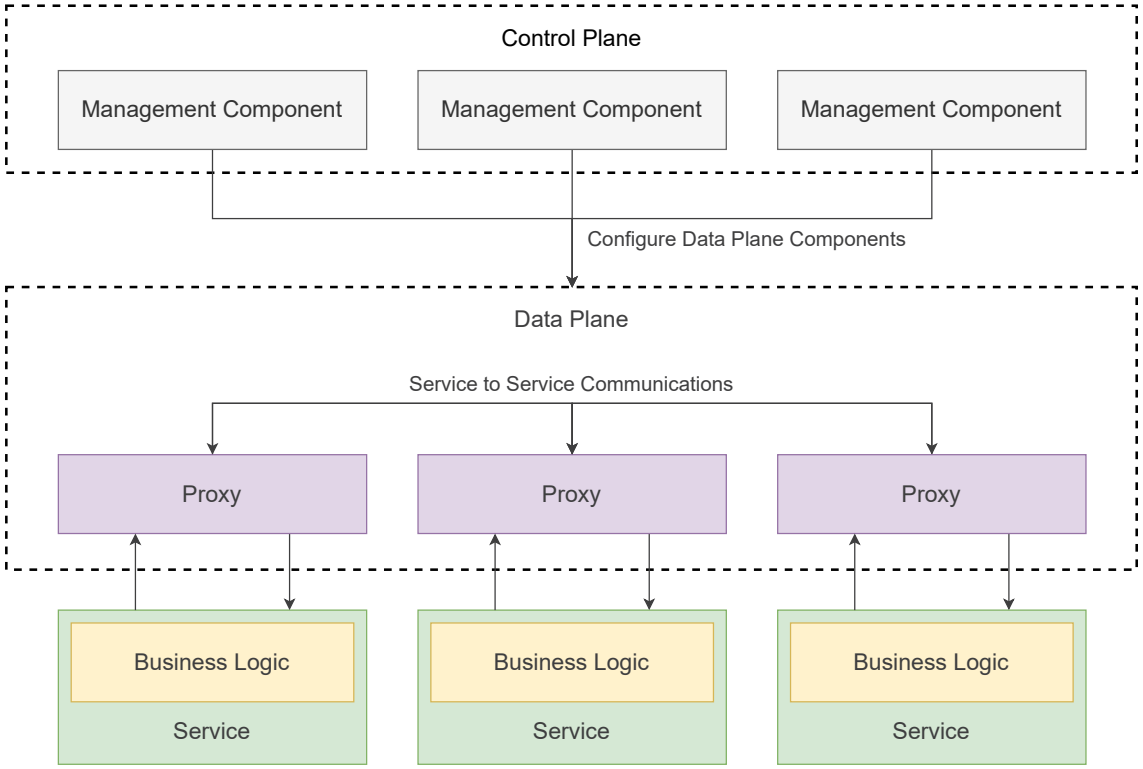


Figure 2.6: Service mesh architecture.

The terminology was introduced in the last decade (39), and several systems adhering to this architecture have seen the light of day since. The premise of introducing additional observability in the system, increasing the reliability of it while also adding security best practices to little or no additional lines of application code made this technology a hot topic in the field. However, this technology does not come in the form of a zero cost offering. The service mesh architecture introduces additional machinery to achieve its goals and is depicted in Figure 2.6. The components that make up a service mesh can be divided into two categories. A set of management components that constitute the *Control Plane* and a set of components that handle the network traffic referred to as the *Data Plane*.

2.4.1 Data Plane

The data plane is what actually defines the dedicated layer of networking infrastructure, it is what creates the mesh. It is created by placing *proxies* in front of the logical services. The goal of these proxies is to intercept all traffic to and from a given service. All service-to-service communications, and direct requests to a service go through these proxies. Notably, these proxies are *Layer 7-aware TCP proxies* like many popular general purpose proxies

such as *HAProxy*[1] or *NGINX*[2]. Although the choice of proxy is an implementation detail, most service mesh use a proxy that has a focus on high performance and a feature set that aligns with the many benefits a service mesh can provide (as discussed in Section 2.4.3).

By introducing these proxies into the system, an increase in system resources is to be expected. More importantly, the introduction of a network proxy leads to an increase in the amount of network hops in the *data path*[3] for each packet. The downside of this is that this therefore leads to an increase in request latency and can cause for scalability concerns. In this thesis, we take an experimental approach to quantify the performance overheads caused by a service mesh system. Therefore, we will focus on the impact of the data plane components.

2.4.2 Control Plane

A second set of components makes up the control plane of a service mesh system. This set of components is used to manage and facilitate the functionalities of a service mesh. It provides coordination and configuration for the data plane proxies and performs functions such as service discovery and metric aggregation. Another component often found in the control plane of service mesh is a *Certificate Authority*, which issues TLS certificates to the data plane proxies enabling them to authenticate each other and use encrypted communications.

Compared to the data plane components, the control plane components have a lesser impact on the performance of systems. First, it does not affect the network traffic in the data path directly, and only has an impact on system resources. Furthermore, this impact is mostly static, as there often is just one instance of each component which provides a sharp contrast to the most common data plane design which results in a proxy per individual service (Section 3.5.2.1).

2.4.3 Benefits of a Service Mesh

To understand the benefits of a service mesh system, we first need to understand which challenges such a system is trying to solve. As previously discussed in Section 2.2, we expanded on the benefits of a service-oriented approach, but also briefly explained some downsides of such an architecture. In short, it adds another layer of complexity into a system that comes in the form of communicative overhead. Services have to deal with the inherent by-products of communications and networking. Network connections can fail,

[1] `http://www.haproxy.org/`

[2] `https://www.nginx.com/`

[3] The term data path in this context, and throughout the rest of this thesis, refers to the machinery (both software and hardware) a network packet has to pass through to reach its destination.

services can crash or experience downtime, services now have to deal with load balancing and service discovery just to name a few of those challenges.

The service mesh architecture and the systems that implement this try to address these challenges in a uniform fashion. It provides a way to implement distributed systems best practices without having to modify the business logic that makes up a logical service. The advantages of a service mesh architecture can be categorized into four groups.

1. **Observability** The first set of features and advantages related to a service mesh system deals with the observability of communications. The data plane of a service mesh can log the details of network request in a uniform manner. Furthermore, it captures metrics such as request latencies, request payload sizes, request volumes, and status codes to name a few. Another desirable aspect is that it allows operators to view and create service topology maps, depicting the relations between services or inspect and trace individual requests throughout a system.

2. **Reliability** Every engineer and user wants a reliable system; however, there are many challenges that one has to solve to make a system reliable. Network connections can fail, services can crash or experience downtime or a new software update can cause compatibility issues. A service mesh tries to solve some of these problems by introducing a set of reliability features. Some of these include the ability to automate and configure request retries and timeouts. Another feature that helps in this area is the ability to perform traffic splitting and shifting, often used for *canary deployment*[1].

3. **Security** A major advantage of using a service mesh system is related to the security related features that it provides. One of the major challenges of modern IT systems is related to security and data privacy. How can we make sure that data cannot be observed by bad actors and limit or control the access within a system? Due to the design of the data plane all communications go through a network proxy (see Section 2.4.1). This enables a service mesh to automatically encrypt all data transfers in service-to-service communications through modern encryption standards. Furthermore, it enables the operator to configure access control for said services with finer granularity.

4. **Programmability** A final category of features that a service mesh provides is related to the programmability of such systems. The first set of programmability features is related to the type of proxy used in the data plane. Due to the layer 7

[1] A canary deployment is a software rollout strategy that is based on staged releases. The core idea is that the software is only available to a particular subset of users, which is slowly incremented over time. This practice serves as an early warning and can minimize the impact of failed deployments.

aware network proxies that it uses, it can make decisions based on application level protocol details. This allows an operator to configure networking, routing and load balancing on application level details such as *HTTP* headers, paths or *Kafka* topics. The second set of features that enable an additional layer of programmability is due to the extensibility features often found within the proxies used in service mesh systems. Many of the popular data plane proxies such as *Envoy* allow the user to extend the default feature set[1].

2.4.4 Alternatives to the Service Mesh Architecture

We discussed the set of challenges that a service mesh tries to solve, however, this set of challenges related to communications have existed long before the notion of a service mesh even existed. How did we solve these challenges before, and what are some alternative solutions?

One approach would be to manually configure and handle all the aspects and features that a service mesh architecture implements. This would be the best performing option, requires no additional machinery to implement and allows the software engineers to use any language or framework, that best suit their needs to implement various logical services. The downside of this approach is that it will most likely lack in uniformity, and requires a significant effort from the software engineers to implement in a microservices architecture, as it would mean that engineers would have to manually manage and implement the intricacies of networking best practices and repeat the process for every single service.

Another approach used was to utilize another design pattern that implements an *Enterprise Service Bus*, an open standard, message-based, distributed integration infrastructure that provides routing, invocation and mediation services to facilitate the interactions of disparate distributed applications and services in a secure and reliable manner (40). This dedicated piece of infrastructure combines Message-Oriented Middleware, web services, transformation, and routing intelligence as a backbone for Service-Oriented Architecture. The most significant difference with the service mesh architecture, is that the services within such an Enterprise Service Bus architecture relied on the infrastructure and middlewares of it (41). In a service mesh architecture, the services are not aware of any topology changes or that their networking traffic is being intercepted at all. This is a key difference and important to note, as it provides a clear separation of concerns from the Enterprise Service Bus architecture frequently used in the 90s.

Another common alternative was already briefly touched upon in Section 2.2.4, in which we introduced the software library-based approach that did not rely on additional machinery to be implemented. The software libraries are used to implement a uniform client and server interface. With a focus on load balancing mechanisms, fault tolerance with retry and

[1] `https://www.envoyproxy.io/docs/envoy/latest/extending/extending`

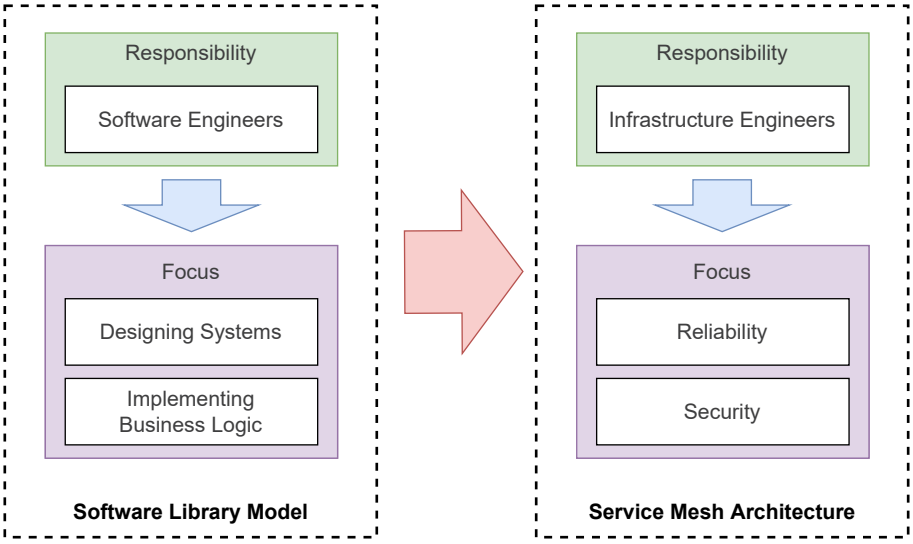


Figure 2.7: Service mesh responsibility model. Compares the shift in responsibilities compares to the software library approach and shows that it can provide a clear separation of concerns for the stakeholders involved. It allows the stakeholders involved to focus and work on the objectives that align with their roles.

timeout feature sets the software library implements many of the features and mechanisms present in a service mesh system. This approach can be the most performant option, as it does not introduce additional network hops and complex systems. However, a clear downside of such an approach is that this would mean that software engineers would be limited to use a single programming language to implement all business logic, or maintain multiple software libraries to support different technology stacks. Furthermore, an update to such a library would mean that all the services running that library would require an update as well.

In Figure 2.7 we compare the most common alternative to the service mesh architecture and shows the shift in responsibility that it caused. By using a service mesh architecture, the infrastructure engineers maintain the service mesh system. The features that a service mesh provides align with the goals and objectives that such a stakeholder has, such as maintaining a reliable platform and having a security-first mindset. For software engineers this meant that they do not have to maintain, update and implement these libraries and can focus on system design and implementing business logic instead. This provides a clear separation of concerns and allows the stakeholders to focus more on the goals and objectives aligned with their respective roles.

2.5 Cloud Native Computing Foundation

In this section, we briefly introduce the Cloud Native Computing Foundation (CNCF), an entity that is often mentioned throughout this thesis as they are a governing body in the field.

The CNCF is part of the *Linux Foundation*[1] and was founded in 2015 at the same time the first version of Kubernetes was introduced. The goal of this foundation is to be an open-source, vendor-neutral hub of *cloud native computing* (42). Where cloud native technologies are defined as technologies that "...empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach".

The CNCF governs many open-source software projects, Kubernetes is one of such projects as it was donated to the foundation upon release. There are currently 1119 projects governed by the foundation at the time of writing, which together, form the *CNCF landscape*. This landscape has an estimated market cap of $20.3 trillion and a total funding of $51.6 billion[2]. Projects are categorized in three levels of maturity[3], ranging from (in ascending order) *Sandbox*, to *Incubated* to *Graduated*.

Throughout the rest of this thesis, we use several projects that are governed by the CNCF, as many of the service mesh systems are part of the landscape. In addition to that, we use Kubernetes as the resource manager to construct our experimental environment (see Section 5.1.4.1) and use various other projects in our implementation of *Mesh Bench* (see Section 4.5). In addition to the projects, we use a lot of the data the foundation gathers through their annual surveys, which provide key insights into the field of cloud native, and serverless technologies.

2.6 Related Work

In this section, we discuss the identified related work in the field. The core work in this thesis is divided into three primary chapters, the systems survey (Chapter 3), the benchmark design (Chapter 4), and the experimental evaluation (Chapter 5). For each of these chapters we introduce a section with related work from both academia and the industry.

[1] The Linux Foundation is a non-profit technology consortium that aims to standardize Linux, support its growth, and promote its commercial adoption.

[2] `https://landscape.cncf.io/`

[3] `https://www.cncf.io/projects/`

2.6.1 System Survey

Before conducting a system survey of the field of service mesh systems, we identified earlier studies that aimed to synthesize the field. In previous works we have conducted we conducted a literature survey on the Kubernetes ecosystem. During this survey, we uncovered the field of service mesh systems and identified the fact that little academic output was generated with a focus on this field.

During the system survey, we identified the works of Li et al. (43), which have conducted a literature review service mesh systems back in 2019. In this work, the authors evaluated four different service mesh systems and provided a high-level, architectural overview. In their evaluation of service mesh systems they compared the systems primarily based on non-functional attributes such as the availability of source code and the activity of the projects themselves. With a focus on system goals and high-level concepts this work established the only secondary study we have uncovered on this topic.

2.6.2 Benchmark Tools

Before we started the work in this thesis, we searched and evaluated existing tools in the field which could perform performance related experiments on service mesh systems. During this, we identified two tools.

The first tool that we have identified is created by *Kinvolk*[1]. This benchmark tool[2] consists of several shell scripts which conduct experiments to evaluate and compare two popular service mesh systems in the form of *istio* and *linkerd*. The benchmark installs one of the previously mentioned service mesh systems and performs load tests on a predetermined, sample application.

The second tool that we have identified is *Meshery*[3]. Meshery is a service mesh *management plane*, which can install and manage various service mesh systems. It is an extensive platform that allows users to interact with service mesh systems on various supported platforms such as Kubernetes and Docker. Meshery supports many features and is currently under active development. First, it can manage the lifecycle of service mesh systems, meaning that it can install, remove or update them. Secondly, it can install applications on a Kubernetes cluster, as well as adapt commonly used patterns. Furthermore, it can perform simple load tests and report results in an open standard[4] which they are maintaining. Additionally, they can perform conformity tests to check if a given service mesh system adheres to certain Kubernetes specific standards[5].

[1] `https://kinvolk.io/`
[2] `https://github.com/kinvolk/service-mesh-benchmark`
[3] `https://meshery.io/`
[4] `https://smp-spec.io/`
[5] `https://smi-spec.io/`

2.6.3 Experimental Evaluation

The goal of this thesis is to address the lack of performance related studies to service mesh systems. However, we did manage to identify related performance studies on these systems. We identified two performance oriented studies that used the benchmark tool by Kinvolk. The first study was from the authors of the tool themselves (44). The second study was performed by the creators of linkerd (45). Both of these studies compare istio and linkerd, two of the most popular service mesh systems.

3

Systems Survey and Analysis of State-of-the-Art

Following a general introduction to concepts relevant to this thesis in Chapter 2, in this chapter, we present a literature and systems survey on the state-of-the-art service mesh systems. The goal of this chapter is to provide an answer to **RQ1**, *How to compare, and evaluate service mesh systems?* To identify the existing systems and objectively compare these systems, we first conduct a systematic survey. By using a systematic process, the results as presented in this chapter can be reproduced by anyone who follows the steps as presented in our methodology. The survey is conducted in two phases. The goal of the first phase is to identify existing systems in the service mesh landscape. After this, we survey the literature on these individual systems. The data obtained through this survey is then synthesized and used to create a framework which compares the properties and characteristics of these state-of-the-art service mesh systems. Finally, we conclude this chapter by providing an extensive analysis on the obtained qualitative results, and produce an answer to the research question.

The remainder of this chapter is structured as follows. First, in Section 3.1 we discuss the related work and identify the need for a systematic systems survey. Next, in Section 3.2 we define the goals which we aim to achieve with the survey. After this, in Section 3.3 we present and explain the methodology used to conduct this survey. Furthermore, in Section 3.4 we present and discuss the obtained results. Subsequently, in Section 3.5, we analyse the data in detail and introduce a comparison framework for state-of-the-art service mesh systems. Finally, in Section 3.6, we summarize our work and establish an answer to research question **RQ1**.

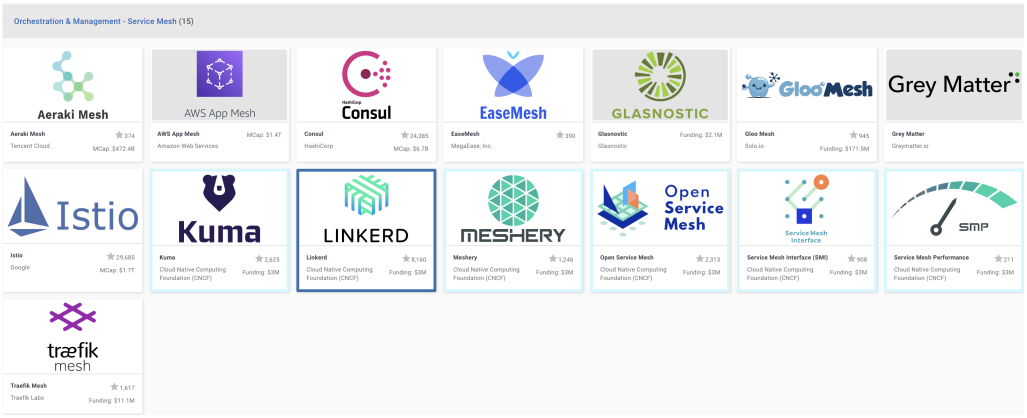


Figure 3.1: CNCF landscape of service mesh technologies as of March 2022.

3.1 Motivation

To justify a formal systematic survey we first evaluate the existing related work in the field. In Section 2.6.1, we discuss the singular secondary study that we identified before executing this system survey. We concluded, similarly to the authors of the paper, that there is a lack of academic literature in this field. Even though the work from Li et al. (43) was concluded in 2019, the situation still prevails. This provides a sharp contrast with the amount of attention the field has received from the industry as it contains an abundance of blog posts and talks. The author of one of the most popular service mesh systems even went as far to call it the "the world's most over-hyped technology" (39). Aside from the attention it has been getting, it has also been shown that service mesh systems are becoming increasingly popular as indicated in surveys conducted by Red Hat (46) and the CNCF (47).

In addition to the attention of the industry, the field has seen many developments. New systems have emerged, and existing systems have evolved. With very recent developments (12, 13) (less than two months old at the time of writing) paving the way for fundamentally different approaches, the field is as exciting as it gets.

This all results in a justification for conducting the survey, and that the timing of it manages to capture an interesting, evolutionary shift within the field.

3.2 Survey Objectives

Before conducting a systematic systems survey, we first establish a set of objectives which we will try to achieve in this work. This allows us to structure the research and adjust its methodologies based on the objectives. The following list represents the formulated objectives.

O1 Bridge the gap between academia and industry.

In the previous section (Section 3.1), we identified that there is a large gap between interest from academia and the industry. We identified that there is a lack of formal publications in the field, whereas preliminary exploration showed that this is in sharp contrast to the interest as expressed from the industry. This leads to this first objective, which is to close this gap and provide formal research in this emerging field.

O2 Gain insight into the current state-of-the-art service mesh systems.

The second objective of this survey is to gain a clear understanding of the field and the current state-of-the-art service mesh systems. We want to identify the systems that currently exist and identify their defining properties and characteristics.

O3 Obtain knowledge about relevant performance related intricacies.

Finally, we want to identify and analyse the properties and components that capture the performance implications of these systems. By examining these systems in detail, and by taking a closer look at the design of these systems, we can hypothesize on the performance complications of design decisions. This is important for the later parts of our work so that we can analyse the experimental results and relate back to architectural differences uncovered.

The objectives and their respective order in which they are presented are closely tied to the general flow of research conducted within this thesis. We first established the gap between industry and academia in our previous work and formulated this thesis around the idea to close this **O1**. Furthermore, we apply best practices by conducting a survey of the field. This helps us to understand the field and allows us to examine the current state-of-the-art systems in detail **O2**. The data synthesis of this survey allows us to establish a framework to compare the existing and future service mesh systems which might emerge. The results of this will be used to provide the answer to the research question **RQ1** during our analysis. Additionally, the survey and resulting framework allows us to determine relevant metrics and components to measure **O3**. This provides a stepping stone towards the following chapter, in which we design an instrument to compare the performance characteristics of these systems (Chapter 4).

3.3 Methodology

The following section covers the methodology that is used throughout the systematic systems review. The idea of the systematic approach is to make the results as presented in this chapter reproducible. This is done by extensively detailing the methodology used and

explaining the decisions we had to take. We present the guidelines, data sources and steps taken to reach the dataset used in the data synthesis.

The remainder of this section is structured as follows. First, in Section 3.3.1 we identify and discuss methods and techniques commonly used in systems surveys. Secondly, in Section 3.3.2, we combine the methods and techniques identified, and present the strategy that we used to approach this survey. Subsequently, we introduce the research protocol that we established to conduct the survey.

3.3.1 Defining a Strategy

There are many methods and techniques out there to find, select or synthesize the data for a survey. The methodology and technique used throughout such a survey ultimately dictates the form a survey takes on and influences the results it produces. It is important to identify, compare and choose such methods and techniques that fit the needs of our system-focussed survey.

In this section, we will present the identified methods that we can use for a survey. For each method, we discuss their advantages and disadvantages and whether we will use them in some form within the survey.

3.3.1.1 Unguided Techniques

Unguided search is the first technique that we have identified. This technique is commonly used to produce an initial dataset of relevant literature on the topic. This process requires the researcher to traverse various digital libraries equipped with relevant keywords to produce the dataset. Another commonly used technique that we have identified is *backward snowballing*. This is another unguided method often used as a complementary technique to expand the dataset and is achieved by searching for related papers in the reference section. By combining these techniques, a researcher can create a literature review based on the generated dataset. However, both of the techniques are unguided, and therefore the process is not reproducible. Using such unguided techniques can result in entirely different datasets if the process is to be repeated by another researcher, and thus does not produce much scientific value.

3.3.1.2 Systematic Approach

To prevent any form of bias to have an effect on the process and results of a survey, we have to establish and utilize a systematic approach. We have identified a state-of-the-art methodology which aims to create a systematic process which will be used to guide this survey. The works of Kitchenham and Charters (48) introduce a battle-tested and auditable set of guidelines for literature reviews aimed at software engineering researchers

with the goal of promoting reproducibility. The introduced *Systematic Literature Review* methodology consists of three steps, planning the review, conducting the review and reporting the review. We decided to use these guidelines as a baseline for the survey, as it aligns with our goals and environment.

In addition to the guidelines introduced by the Systematic Literature Review methodology, we utilize the guidelines as introduced in the works of Petersen et al. (49, 50). The authors introduce a set of guidelines and techniques which can be used to conduct a mapping study on a given topic. Among the guidelines and techniques, the authors introduce a method to perform comparative analysis. These guidelines and methods align with our needs and goals to compare service mesh systems in the field as stated by the formulated research question **RQ1**.

3.3.1.3 Inclusion of Grey Literature

One of the drawbacks of the Systematic Literature Review method is that it only concerns Primary Literature, which consists of published peer-reviewed academic articles. In our previous work, in which we have conducted a systematic literature review on the Kubernetes ecosystem, we came to the conclusion that this field lacked published and peer-reviewed articles. Additionally, previous secondary studies on the field came to the same conclusion (43). The lack of published (formal) literature in the field makes it impossible for us to conduct the research to an extent which satisfies our goals. This led us to explore additional forms of literature. Grey Literature is a form of literature that does not belong in the formal category, and can be in the form of blog posts, videos or white papers. Exploratory research has shown that inclusion of Grey Literature can be beneficial to our research, as most of the literature is presented in such form by the industry.

To maintain a systematic approach with the inclusion of Grey Literature, we had to define or find additional guidelines that supported this requirement. We have identified the state-of-the-art method for conducting a survey which enabled the inclusion of Grey Literature. The Multivocal Literature Review methodology as introduced by Garousi et al. (51) adapts and extends the Systematic Literature Review method from Kitchenham et al. so that it can include both forms of literature.

Additionally, the authors have created a checklist which can be used to verify the need for inclusion of Grey Literature within a survey. If one or more of the questions in the checklist is answered with a "yes", it indicates that Grey Literature can be beneficial to the survey. We answered the questions in the provided checklist and the results of this can be seen in Table 3.1. These results reaffirmed our requirement for Grey Literature throughout the survey and made us adapt the guidelines as described by the methodology.

Question	Answer
Is the subject "complex" and not solvable by considering only the formal literature?	Yes
Is there a lack of volume or quality of evidence, or a lack of consensus of outcome measurement in the formal literature?	Yes
Is the contextual information important to the subject under study?	Yes
Is it the goal to validate or corroborate scientific outcomes with practical experiences?	Yes
Is it the goal to challenge assumptions or falsify results from practice using academic research or vice versa?	No
Would a synthesis of insights and evidence from the industrial and academic community be useful to one or even both communities?	Yes
Is there a large volume of practitioner sources indicating high practitioner interest in a topic?	Yes

Table 3.1: Questionnaire from the works of Garoussi et al. (51), which helps to decide whether to include Grey Literature in a literature survey.

3.3.2 Our Approach to a Systematic Survey

In the previous section (Section 3.3.1), we identified several methodologies and guidelines which we can use to conduct a survey. We explained why we have chosen some of these guidelines for our systems survey and how they could apply to our research goals. The resulting strategy can be seen in more detail in the form of a flow chart in Figure 3.2. This figure depicts two of the three stages in from the Systematic Literature Review methodology, namely the planning phase and the conducting phase. It also lists all the deliverables of this systems survey (D1 - D5).

Up to this point, we have discussed two steps of the planning phase. This first step of the planning phase P1 , consists of establishing a need for the survey, which we did in Section 3.1. Next, we established our objectives **O1** - **O3** and research question **RQ1** in Section 3.2 which relates to process P2 .

3.3.3 Review Protocol

The review protocol is a fundamental aspect of the Systematic Literature Review methodology, it specifies the methods used to conduct the systems review in this thesis. A pre-defined protocol is required to reduce the possibility of any bias. Establishing the review protocol is part of our approach as specified in Figure 3.2 and depicts the process P3 .

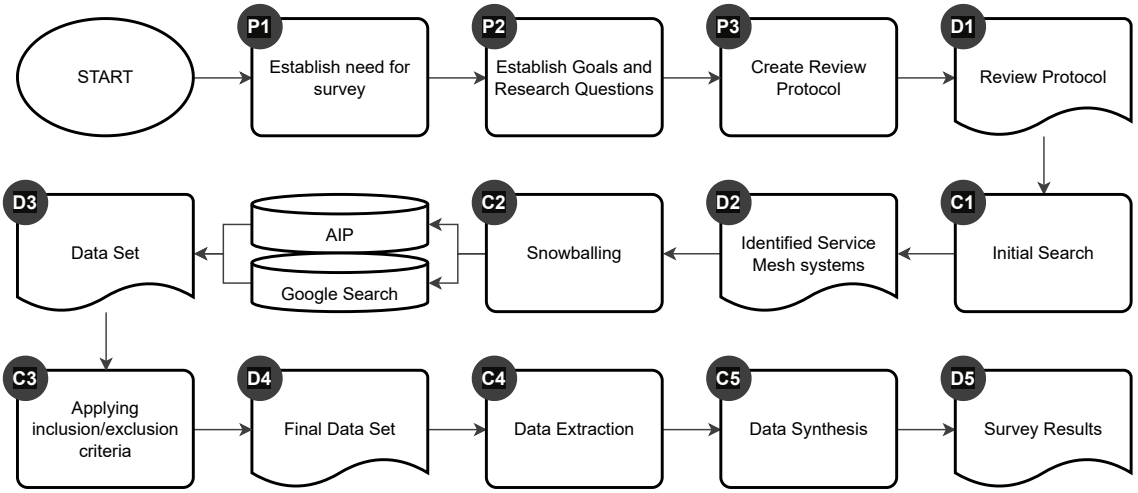


Figure 3.2: System survey approach.

In the remainder of this section, we introduce the components that establish the research protocol. In Section 3.3.3.1, we introduce the search strategy which is used to search for the relevant literature. In Section 3.3.3.2, we establish the selection criteria which are used to filter the dataset which we obtained through the research strategy. Finally, in Section 3.3.3.3, we describe the methods used to extract and synthesize the data.

3.3.3.1 Search Strategy

To create a reproducible data set, we have to establish a search strategy which we use to obtain literature on the topic. The search strategy differs based on the type of literature, where different requirements apply for Primary Literature and Grey Literature.

The first component that makes up the search strategy is related to the data sources. The Atlarge team[1] has identified several digital libraries which are well established and which contain the leading publications in the field of distributed systems. Additionally, years of experience in the field led to the creation of a list of established and well-respected conferences. This consolidated knowledge led to the development of a software instrument Article Information Parser (AIP). This instrument is used to generate the initial dataset of Primary Literature for this survey and query it. To establish a dataset of Grey Literature in this survey, we use Google Search to identify relevant literature.

The second component of the search strategy is related to the search technique. Since the goal of this survey to compare state-of-the-art service mesh systems, we first have to identify said systems. To achieve that, we first conducted exploratory research in the field. To reduce bias in this process, we first used the single identified secondary study (43) to obtain an initial set of service mesh systems. After this, we identified the service mesh systems

[1] `https://atlarge-research.com/`

	Query	Search Strategy
Q1	Service Mesh	Initial Search
Q2	Service Mesh Implementation	Initial Search
Q3	*Identified Service Mesh System*	Snowballing

Table 3.2: Search queries established in the review protocol.

that were part of official CNCF projects[1], as they have proven to be an authoritative entity in the field as discussed in Section 2.5. This list was then complemented by results obtained from Grey Literature using generic search queries.

The final component of the search strategy is related to the search queries that we use to establish the resulting dataset. To find and identify the current state-of-the-art service mesh systems, we used generic terms as described above. After this, we utilized a snowballing method in which we based the search queries on the identified systems. The final list of search queries used can be seen in Table 3.2. Queries **Q1** and **Q2** represent the queries used to obtain and expand the list of identified service mesh systems, while **Q3** indicates the system-specific queries that were used.

3.3.3.2 Selection Criteria

Since we have decided to also include Grey Literature in our survey, we decided to establish two sets of selection criteria since they are vastly different in terms of content and medium. For both types of literature, we establish a checklist. In this checklist, we define the guidelines that we use to either include or exclude a data point. Guidelines prefixed with a checkmark (✓) symbol indicate a reason to include a publication in the research, whereas the cross (✗) symbol indicates a reason for exclusion.

The following checklist (Section 3.3.3.2) contains a set of selection criteria that applies to the Primary Literature that is obtained through the search process. For this, we apply a set of generic criteria in order to

In addition to the search queries, we used a filter to filter out any results before 2014 since that the initial release date of Kubernetes (34). This limits the results based on the scope of this research, which provides a focus on service mesh systems in the Kubernetes ecosystem.

(✓) Publications that are written in English.

(✓) Publications introducing novel concepts.

[1] `https://landscape.cncf.io/card-mode?category=service-mesh&grouping=category`

(✓) Publications that focus on architectural problems and solutions.

(✓) Publications performing performance analysis studies.

(✗) Publications before 2014.

(✗) Publications that do not have service mesh systems as the primary subject in the research.

(✗) Secondary or tertiary studies regarding service mesh systems.

(✗) Publications that are not full-text (e.g., small snippets or as part of a demo).

The following checklist (Section 3.3.3.2) contains a set of selection criteria that applies to the Grey Literature that is obtained through the search process. For this form of literature we had to establish a set of rigorous guidelines since it can otherwise result in numerous data entries or lead to a biased set of results.

(✓) Publications that are written in English.

(✓) Publications introducing novel concepts.

(✓) Publications originating from well established and known organizations (e.g. CNCF, Google, Red Hat, etc.).

(✓) Publications that supply official vendor or platform documentation.

(✓) Publications in the form of blog posts from organizations or authorities in the field.

(✓) Publications in the form of videos from talks or conferences related to the field.

(✓) Source code repositories of open-source service mesh systems (e.g., a GitHub repository).

(✗) Publications in which we cannot validate the authority of an author (e.g., anonymous posts, or posts by someone not established in the field).

(✗) Publications which do not provide any novelty for the survey (e.g., tutorials on the technologies).

(✗) Publications that are not full-text (e.g., small snippets, parts of a demo or short social media excerpts such as tweets).

3.3.3.3 Data Extraction

Once we have established a final dataset by filtering the dataset based on the selection criteria, we can start the data synthesis process. To achieve this, we first have to establish a set of data extraction guidelines. These guidelines have to be in line with the goals which of this survey. Since we want to compare state-of-the-art service mesh systems, we have to identify the key characteristics and properties of such systems. Through the preliminary research conducted on the field, we understand the fundamentals of a service mesh system and have identified the challenges it tries to solve. With this in mind, we try to identify and synthesize the data based on key areas.

First off, we try to compare these systems based on their architectural approach. With this take, we can identify and compare service mesh systems based on their design decisions, such as their control and data plane components. Secondly, we take a functional approach to synthesize these systems. By identifying sets of functionality, we can evaluate and compare these systems on domain-specific functionalities such as their observability, security, or resiliency characteristics. Furthermore, we take a meta analytical approach to these systems. With this, we try to establish certain aspects such as the number of users the system has, the development activity on the project, whether a project is open-source or not and the maturity of the system.

3.4 Results

In this section, we present the results from the systems survey as deliverable D5 . First, we present the identified service mesh systems in Section 3.4.1. Furthermore, in Section 3.4.2 we introduce functional and non-functional requirements which we use to differentiate service mesh systems. Finally, in Section 3.5.1 we present a comparison framework for state-of-the-art service mesh systems, in which we compare the identified systems.

3.4.1 State-of-the-art Service Mesh Systems

During the initial phase of the systems survey, we set out to identify existing service mesh systems with our initial search strategy (**Q1** & **Q2** in Table 3.2). During this phase, we identified 14 different service mesh systems, which can be seen in Table 3.3, which represents deliverable D2 . By identifying these systems, we could run targeted queries on these them, which enabled us to dive deeper into their individual architectures.

However, some systems in the list will be excluded from the rest of the work presented in this thesis. In particular, *Alibaba Cloud Service Mesh*, *Aspen Mesh*, *Citrix ADC*, and *Linkerd* are excluded. The first two systems are managed, closed-source service offerings of the popular *Istio* service mesh. The third system, has the option to utilize *Istio* to construct

ID	Service Mesh	Vendor Page
SM1	AWS App Meshh	https://aws.amazon.com/app-mesh/
SM2	Cilium	https://cilium.io/
SM3	Consul	https://www.consul.io/
SM4	Ease Mesh	https://megaease.com/easemesh/
SM5	Istio	https://istio.io/
SM6	Kuma	https://kuma.io/
SM7	Linkerd2	https://linkerd.io/
SM8	Nginx Service Mesh	https://www.nginx.com/products/nginx-service-mesh
SM9	Open Service Mesh	https://openservicemesh.io/
SM10	Traefik Mesh	https://traefik.io/traefik-mesh/
-	Alibaba Cloud Service Mesh	https://www.alibabacloud.com/product/servicemesh
-	Aspen Mesh	https://aspenmesh.io/
-	Citrix ADC	https://www.citrix.com/nl-nl/products/citrix-adc/
-	Linkerd	https://linkerd.io/

Table 3.3: Identified service mesh systems and their respective vendor pages. The service mesh systems with an ID as indicated in the table, will be used and referred to throughout the rest of the system survey.

a service mesh. We excluded these systems, as they provide no unique capabilities, are presented in the form of black box software and additionally has a financial cost attached to it. Finally, the last of the excluded systems mentioned, is superseded by *Linkerd2* and is discontinued.

3.4.2 Identified Requirements

To compare existing service mesh systems, we have to introduce a set of requirements to compare them on. We do this in the form of *Functional Requirements* and *Non-Functional Requirements*. The former set of requirements describe the functions and characteristics of a system, such as the features and capabilities a specific system has. The latter, however, take a more general approach to describing the systems on its behaviour.

3.4.2.1 Functional Requirements

In this section, we introduce a list of six *Functional Requirements* (**FR1** - **FR4**). We derived these requirements from the core functionalities of service mesh systems. For each of the defined requirements, we provide a brief description of what the requirement entails and how we extract the information from the identified systems to measure this.

FR1 Observability Capabilities Indicates whether the system supports a wide vari-

ety of observability capabilities. Observability characteristics range from capturing metrics, to monitoring and tracing. To determine this, we have established a list of capabilities on which we compare these systems.

FR2 Security Capabilities Indicates whether the system supports common security capabilities. To determine this, we first established a list of common security features, on which we compare the individual systems.

FR3 Resilience Capabilities Indicates whether the system supports a wide variety of reliability characteristics. Service mesh systems can improve the reliability of service-to-service communications in a system. To determine the capabilities of an individual system, we established a list of common reliability characteristics and compared the individual systems to these capabilities.

FR4 Support for Multiple Deployment Models Indicates whether the system supports multiple deployment models. The most common scenario is to have a single cluster and a single service mesh. However, situations might exist where a topology consists of multiple clusters or multiple service meshes to provide for extra isolation, for example. We compare the identified systems to check for support on this characteristic.

3.4.2.2 Non-Functional Requirements

In this section, we introduce a list of six *Non-Functional Requirements* (**NFR1** - **NFR5**). We established this list to provide for a general overview of the service mesh systems. For each of the defined requirements, we provide a brief description of what the requirement entails and how we extract the information from the identified systems to measure this.

NFR1 Application Protocol Support Service proxies can be aware of application level protocols which enables features such as making routing based on application-level details (such as HTTP headers, or Apache Kafka topics) and rich metrics collection. This requirement indicates whether a service mesh system supports a wide variety of application level protocols. To determine this, we established a list of common service-to-service protocols and also identified additional application level protocol support for each of the identified systems. The requirement is only fully satisfied if, in addition to the commonly listed protocols, it supports additional application level protocols.

NFR2 Open-source Indicates whether a service mesh system is available in the form of open-source software. To determine this, we identified source code repositories for the individual service mesh systems if they existed.

NFR3 Thoroughly Documented Indicates whether the system is extensively documented. This is determined by examining the vendor documentation on the system. The requirement is not satisfied if there is no vendor documentation available. The requirement is partially satisfied if there is some form of vendor documentation, but not extensive or up to date. It is fully satisfied if there is extensive and well-written documentation on the system.

NFR4 Cloud Native Computing Foundation Project

Indicates whether a service mesh system is recognized as an official CNCF project and if so, which level of maturity it currently has. The CNCF binds a level of maturity to each of its projects, and imposes certain guidelines to improve the maturity rating (52). Projects are categorized in three levels of maturity[1], ranging from (in ascending order) *Sandbox*, to *Incubated* to *Graduated*. The requirement is not satisfied if it is not a formal project, partially satisfied if it is in the *sandbox* or *incubating* stage, and fully satisfied if it is a *graduated* project.

NFR5 Community Recognition

Indicates the recognition as per the community of users. We determine this by looking at various metrics and surveys regarding the usage of individual systems. For example, with open-source projects, we use community-related metrics such as GitHub stars. The requirement is partially satisfied if it has more than 10.000 GitHub stars or more than 10 percent usage in production, according to the CNCF survey conducted in 2021 (11). To fully satisfy this requirement, the project needs more than 20.000 GitHub stars or has more than 25 percent usage in production.

3.4.3 Comparing Service Mesh Systems

In this section, we present the results of the data synthesis based on the identified Functional and Non-Functional Requirements (Section 3.4.2.1, Section 3.4.2.2). First, in Section 3.4.3.1, we compare the service mesh systems on their service-to-service proxy. Secondly, in Section 3.4.3.2, we compare the systems based on their observability capabilities. Thereafter, in Section 3.4.3.3, we compare the systems on the most common security features. Following that, in Section 3.4.3.4, we evaluate the resiliency features of the systems. Subsequently, we analyse the application level support from these systems in Section 3.4.3.5. Finally, in Section 3.4.3.6, we show the obtained non-functional requirements regarding the systems. These results provide a stepping stone to the analysis in Section 3.5.

Service Mesh		Service Proxy Capbilities			
ID	Name	Service Proxy	Proxy Arch.	Proxy TCP	Proxy UDP
SM1	AWS App Mesh	Envoy	Per-Service	✓	✓
SM2	Cilium	Cilium (eBPF)	In-Kernel	✓	✓
SM3	Consul	Envoy*	Per-Service	✓	✓
SM4	Ease Mesh	Easegress	Per-Service	✓	✗
SM5	Istio	Envoy	Per-Service	✓	✓
SM6	Kuma	Envoy	Per-Service	✓	✓
SM7	Linkerd2	Linkerd2 Proxy	Per-Service	✓	✗
SM8	Nginx Service Mesh	NGINX Plus	Per-Service	✓	✓
SM9	Open Service Mesh	Envoy	Per-Service	✓	✓
SM10	Traefik Mesh	Traefik Proxy	Per-Node	✓	✓

Table 3.4: Comparing the proxy capabilities of identified service mesh systems.

✓: Indicates that the feature is supported.
✗: Indicates that the feature is no supported.
*: Can be interchanged for other service proxies.

3.4.3.1 Proxy Systems

First off, we compared the service-to-service proxy mechanisms of the identified service mesh systems. At the core of a service mesh system lies the service-proxy, it serves as the primary component of the data-plane and can have a considerable impact on the flow of traffic. The result of this comparison can be seen in Table 3.4. While some service mesh systems implement a custom service proxy, such as *Linkerd2*, others utilize popular open-source proxy implementations such as *Envoy*. Not all service proxies work and behave the same, the architecture of a proxy has a major impact on the traffic flow of a service mesh system.

During this survey, we have identified three different types of proxy architectures. First off, we identified a *per-service proxy*, such as used by most of the identified service proxies. This means that every service gets its own service proxy, and thus the number of proxies linearly scales with the number of logical services. The second architectural style identified uses a *per-node proxy*, this is used by the *Traefik Proxy*. This means that there is a single proxy per node, and that all services running on a node use that same proxy. The third and final architectural style identified also makes use of a single proxy per node but manages the traffic *in-kernel* and utilizes eBPF technology to do so. This technology allows sandboxed programs to execute in the operating system kernel, which enables efficient network packet

[1] `https://www.cncf.io/projects/`

Service Mesh		Observability Capbilities			
ID	Name	Metrics	Monitoring	Tracing	Dashboard
SM1	AWS App Mesh	AWS CloudWatch	AWS CloudWatch	AWS CloudWatch	✓
SM2	Cilium	Prometheus	Grafana	Jaeger, Open Telemetry	✓
SM3	Consul	Prometheus	Grafana	Jaeger, Zipkin, OpenTracing	✓
SM4	Ease Mesh	Ease Monitor	Ease Monitor	OpenTracing	✗
SM5	Istio	Prometheus	Grafana	Jaeger, Zipkin	✓*
SM6	Kuma	Prometheus	Grafana	Jaeger, Zipkin	✓
SM7	Linkerd2	Prometheus	Grafana	Open Telemetry	✓
SM8	Nginx Service Mesh	Prometheus	Grafana	Jaeger	✗
SM9	Open Service Mesh	Prometheus	Grafana	Jaeger	✗
SM10	Traefik Mesh	Prometheus	Grafana	Jaeger	✗

Table 3.5: Comparing the observability capabilities of identified service mesh systems.
✓: Indicates that the feature is supported.
✗: Indicates that the feature is no supported.
*: Available through third party, open-source offering *Kiali*.

processing solutions.

Furthermore, we identified for each of the proxy systems if it was able to proxy both TCP and UDP traffic. All the systems supported the ability to proxy TCP traffic, however, not all of them offered support for UDP traffic. Service mesh systems without UDP traffic proxy support can still have UDP traffic in their system, however, with the traffic not passing through the proxy it will provide none of the features that a service mesh traditionally provides.

3.4.3.2 Observability

Observability is a key feature and reason to use a service mesh. The proxying of service-to-service communications allows systems to implement per-service metrics such as latencies, request volumes and error rates. We compared the identified service mesh systems on key observability requirements, and the results of that can be seen in Table 3.5.

First off, we identified the way in which the systems report metrics. Most of the identified systems made use of *Prometheus*, a popular open-source time-series metric aggregator and database, whereas two other service mesh systems in the form of *AWS App Mesh* and *Ease Mesh* used their own, proprietary solutions. Furthermore, we compared the ways the systems enabled monitoring. Once again, service mesh systems resort to similar solutions, this time in the form of *Grafana*, a monitoring and observability platform. *AWS App Mesh* and *Ease Mesh* once again used their own, proprietary solutions. Tracing support, on the other hand, showed more variability. Most of the identified systems support *Jaeger*, an open-source distributed tracing system. *Zipkin* was another popular system with support, and some systems even supported open standards such as *Open Telemetry*, allowing for

Service Mesh		Security Capbilities		
ID	Name	Mutual TLS	Auth. Policies	External CA
SM1	AWS App Mesh	✓	✓	✓
SM2	Cilium	✓	✓	✓
SM3	Consul	✓	✓	✓
SM4	Ease Mesh	✓	✓	✓
SM5	Istio	✓	✓	✓
SM6	Kuma	✓	✓	✓
SM7	Linkerd2	✓	✓	✓
SM8	Nginx Service Mesh	✓	✓	✓
SM9	Open Service Mesh	✓	✓	✓
SM10	Traefik Mesh	✗	✗	✗

Table 3.6: Comparing the security capabilities of identified service mesh systems.
✓: Indicates that the feature is supported.
✗: Indicates that the feature is no supported.

any tracing system that supports that standard. Some service mesh systems come with a bundled dashboard or have popular open-source extensions which enable this. These dashboards provide additional domain-specific insights, such as provide1 graph tools to visualize the service mesh traffic topology.

3.4.3.3 Security

A service mesh can help to harden the security of the infrastructure. We compared the identified systems on the most common security characteristics and properties that a service mesh architecture can provide. The result of this can be seen in Table 3.6.

First, we identified if a service mesh system can provide mutual TLS with little to no additional effort. This means that any service-to-service traffic is then encrypted and potential intruders in a network cannot inspect or intercept that traffic. All the systems, except *Traefik Mesh*, support automatic mutual TLS encryption. Additionally, we compared the service-to-service authorization capabilities of these systems. This enables fine-grained control of access between services and users. Once again, all the systems, except *Traefik Mesh*, have some form of authorization policies that they support. Finally, we checked the service mesh systems if they supported external Certificate Authorities (CA) for the TLS encrypted connections. All the identified systems that supported mutual TLS encryption, supported the usage of an external certificate authority.

Service Mesh		Resiliency Capbilities					
ID	Name	Retry Functionality	Timeout Settings	Rate Limiting	Circuit Breaking	Fault Injection	Delay Injection
SM1	AWS App Mesh	✓	✓	✗	✓	✗	✗
SM2	Cilium	✓	✓	✓	✓	✓	✓
SM3	Consul	✓	✓	✓	✓	✗	✓
SM4	Ease Mesh	✓	✓	✓	✓	✗	✗
SM5	Istio	✓	✓	✓	✓	✓	✓
SM6	Kuma	✓	✓	✓	✓	✓	✓
SM7	Linkerd2	✓	✓	✗	✗	✓	✗
SM8	Nginx Service Mesh	✓	✓	✓	✓	✗	✗
SM9	Open Service Mesh	✗	✗	✗	✗	✗	✗
SM10	Traefik Mesh	✓	✓	✓	✓	✗	✗

Table 3.7: Comparing the resiliency capabilities of identified service mesh systems.
✓: Indicates that the feature is supported.
✗: Indicates that the feature is no supported.

3.4.3.4 Resilience

Managing applications in distributed systems is a complex task and can often lead to undesired failures. Networks, compute nodes or applications can fail and cascade these failures to other components in a SOA. To mitigate or resolve these problems, we can introduce distributed systems best practices to improve the resiliency of a system. In Table 3.7, we compared the resiliency functionalities of identified service mesh systems.

First off, we compared the systems on the ability to retry failed service requests. Most of the systems, except *Open Service Mesh*, supported this functionality. Furthermore, we identified and compared the systems on their ability to allow and tweak the timeout settings of service-to-service communications. Subsequently, we identified and compared the systems on their ability to support rate limiting. Rate-limiting in this context refers to the ability to tweak maximum request per second settings on a per-user or per-service level. Next, we evaluated the systems on their ability to support circuit breaking, this allows the system to prevent cascading failures by for example preventing traffic to reach an individual service instance if this instance fails requests too frequently. Finally, we compared the systems on their support for fault injections and delayed fault injections. These features allow the user to evaluate the resiliency of their applications by injecting faulty requests and timeouts, and can catch bugs in service integrations.

Service Mesh		Protocol Support						
ID	Name	HTTP1.1	HTTP2	HTTP3	Websockets	gRPC	TLS	Additional Protocols
SM1	AWS App Mesh	✓	✓	✓	✓	✓	✓	-
SM2	Cilium	✓	✓	✓	✓	✓	✓	Kafka
SM3	Consul	✓	✓	✓	✓	✓	✓	-
SM4	Ease Mesh	✓	✓	✗	✓	✗	✓	MQTT
SM5	Istio	✓	✓	✓	✓	✓	✓	Mongo, Redis, MySQL
SM6	Kuma	✓	✓	✓	✓	✓	✓	Kafka
SM7	Linkerd2	✓	✓	✗	✓	✓	✓	-
SM8	Nginx Service Mesh	✓	✓	✗	✓	✓	✓	-
SM9	Open Service Mesh	✓	✓	✓	✓	✓	✓	-
SM10	Traefik Mesh	✓	✓	✓	✓	✓	✓	-

Table 3.8: Comparing the application level protocol support of identified service mesh systems.
✓: Indicates that the feature is supported.
✗: Indicates that the feature is no supported.

3.4.3.5 Application-Level Protocol Support

Not all service proxies are the same, some are built to be very minimal and fast, whereas others try to provide as much functionality as possible. We evaluated the identified service mesh systems on their application level protocol support, as shown in Table 3.8. To have advanced proxy capabilities such as routing based on application protocol details and rich metrics support, the protocol must be determined and understood by the proxy.

All the systems support the most common service-to-service protocols in the form of *HTTP/1.1 HTTP/2.0* and gRPC. This means that the user of such system can see the HTTP status codes in the metrics or route the traffic based on the HTTP headers present. Additionally, all the identified systems support web sockets and TLS encryption. Some systems have support for the newer *HTTP/3.0* protocol, however, for all the identified systems that support this, the feature is in beta and bound to change. Finally, we identified some systems that support additional application level protocols.

3.4.3.6 Non-Functional Requirements

Finally, we identified several non-functional requirements of each of the identified service mesh systems. The result of this can be seen in Table 3.9. We identified the launch date of a service mesh system by either the first official vendor blog post or commit in an open-source repository, if available. Furthermore, we identified which company or initiated the project, this can give an indication on the amount of backing a project can get. Next up, we determined if the service mesh system is part of any formalCNCF project, and if so at which stage it currently is, this can give a sense of the level of maturity. We also determined if the system is available as open-source software. If the project was open-source, we were

Service Mesh		Non-functional Attributes							
ID	Name	Launched	Initiator	CNCF Project	OSS	GitHub Stars	Version	Licence	Lang.
SM1	AWS App Mesh	27/03/2019	Amazon	✗	✗	-	-	-	-
SM2	Cilium	02/12/2021	Isovalent	◐	✓	11149	Beta	Apache	Go
SM3	Consul	26/06/2018	Hashicorp	✗	✓	24423	1.11.4	Mozilla	Go
SM4	Ease Mesh	01/02/2021	MegaEase	✗	✓	396	1.2.0	Apache	Java
SM5	Istio	19/11/2016	Google	✗	✓	29770	1.13.2	Apache	Go
SM6	Kuma	13/03/2019	Kong	○	✓	2637	1.5.0	Apache	Go
SM7	Linkerd2	05/12/2017	Buoyant	●	✓	8207	1.7.5	Apache	Go
SM8	Nginx Service Mesh	12/10/2020	Nginx	✗	✗	-	1.4.0	-	-
SM9	Open Service Mesh	13/12/2019	Microsoft	○	✓	2343	1.0	Apache	Go
SM10	Traefik Mesh	03/03/2019	Traefik	✗	✓	1622	1.4.5	Apache	Go

Table 3.9: Comparing the non-functional attributes of identified service mesh systems.
Circles in the table indicate the level of maturity for CNCF projects:
○: Sandbox
◐: Incubating
●: Graduated
✓: Indicates that the attribute conforms.
✗: Indicates that the attribute does not conform.

able to determine other requirements, such as the current version number, licence, and main programming language used as well as the amount of stars the project has which can give an insight in community recognition. Every identified open-source system had a remote repository on GitHub, for which we extracted the previously mentioned attributed through their public API[1].

3.5 Analysis

In the previous section (Section 3.4), we conducted a data synthesis process on the data we obtained during the systems survey. This process resulted in domain-specific characteristics of identified service mesh systems. In this section, we continue with the data and analyse and discuss notable differences we discovered during the data synthesis process.

3.5.1 Qualitative Comparison of Service Mesh Systems

In Table 3.10, we present a qualitative evaluation of state-of-the-art service mesh systems. In this comparison, we present the architectural style the service proxy uses and map the identified requirements and their level of satisfaction (Section 3.4.2 to each individual system.

[1] `https://docs.github.com/en/rest`

	Data Plane		Functional Requirements				Non-Functional Requirements				
Service Mesh	**Service Proxy**	**Proxy Arch.**	**FR1**	**FR2**	**FR3**	**FR4**	**NFR1**	**NFR2**	**NFR3**	**NFR4**	**NFR5**
AWS App Mesh	Envoy	Per-Service	●	●	◐	●	◐	○	●	○	○
Cilium	Cilium eBPF	In Kernel	●	●	●	●	●	●	◐	◐	◐
Consul	Envoy	Per-Service	●	●	◐	●	◐	●	●	○	◐
Ease Mesh	EaseGress	Per-Service	●	○	◐	○	◐	●	○	○	○
Istio	Envoy	Per-Service	●	●	●	●	●	●	●	○	●
Kuma	Envoy	Per-Service	●	●	●	●	●	●	●	◐	○
Linkerd2	Linkerd2 Proxy	Per-Service	●	●	◐	●	◐	●	●	●	●
Nginx Service Mesh	NGINX Plus	Per-Service	●	●	◐	○	◐	○	◐	○	○
Open Service Mesh	Envoy	Per-Service	●	●	○	○	◐	●	◐	◐	○
Traefik Mesh	Traefik Proxy	Per-Node	●	○	◐	○	◐	●	◐	○	○

Table 3.10: Qualitative comparison between state-of-the-art service mesh systems. There are three different symbols in the table, each of them represents how well a requirement is satisfied. ●: Fully Satisfied, ○: Not Satisfied, ◐: Partially Satisfied.

We now present a list of the most interesting findings we obtained:

F1 Most of the identified service mesh systems share a similar data plane architecture.

The first finding from the obtained data is that most of the systems use a similar architecture for the data plane of the service mesh. Out of the ten identified service mesh implementations, eight used a per-service proxy architecture. Furthermore, out of those eight using a per-service proxy architecture, half of them used the same service proxy implementation in the form of *Envoy*, an open-source general purpose proxy.

F2 Per-node data plane architecture prevents support for common security features.

The second finding is that only a single identified system used a service proxy architecture, where the service proxy was implemented on a per-node granularity. *Traefik Mesh* is the identified service mesh that used such an architecture, and most notably it was the only implementation that did not support any of the security capabilities listed (**FR2**). In particular, it does not support automatic mutual TLS encryption between services, a killer feature to enhance the security in any environment. To further elaborate on this, we dive deeper into the characteristics of data plane architectures in the next section (Section 3.5.2).

F3 Most identified systems do not support all functional requirements.

The third finding is that out of all ten identified service mesh systems, just three systems fully satisfy all the functional requirements (**FR1-FR4**), namely, *Cilium*, *Istio* and *Kuma*. This finding can be explained by taking a closer look at the maturity

levels and the goals that the individual service mesh implementations have. The three identified services that achieve all functional requirements are mature projects or heavily emphasize their feature set and extensibility. *Istio* is the most mature platform, with large backing and is the most used service mesh implementation out of all identified systems according to the CNCF survey conducted in 2021 (11). It also supports the most features and therefore hits all the functional requirements. *Kuma* on the other hand, is a much smaller project. However, its focus lies on having a broad feature set with lots of extensibility options as well, thus achieving all functional requirements as well. *Cilium* on the other hand, is a new entry in the service mesh landscape (as can be seen by the release date in the list of non-functional attributes (Table 3.9). However, this story also does not paint the entirety of the picture as it builds upon the *Cilium* networking solution, a mature networking solution that already facilitates most of the functional requirements listed.

F4 Maturity does not translate to support of requirements.

A fourth finding is that maturity does not necessarily translate to the support of functional and non-functional requirements. This refers to *Linkerd2* in particular, which does not not fully satisfy the resiliency requirements **FR3** and also lacking in the additional application level protocol support **NFR1**. Despite its maturity, large-scale usage in production environments (11), and graduated CNCF project status, the system seems to be lacking in features. This, however, can be attributed to the goals of the project. Whereas *Istio* has the most features of any system, it is also dubbed as a complex system. *Linkerd2*, however, aims to optimize for simplicity and speed. Instead of using a generic proxy with numerous features and capabilities, such as *Envoy* or *NGINX*, it uses a custom proxy specifically tailored to the service mesh, optimizing for simplicity, security, and performance (53).

F5 Lack of additional layer 7 protocol support.

A fifth finding is that just a few of the identified systems support additional application level protocols (**NFR1**). Since service mesh systems can greatly benefit from additional application level support from its service proxies, it can be a promising area. It can enable application level aware traffic management, rich metric collection or can lead to advanced authorization policies. *Cilium* and *Kuma* for example, provide support for the Apache Kafka protocol and therefore can target specific messages when dealing with events of the platform. Although we did identify only a few implementations that provided support for this, we did identify an external project in the CNCF landscape that tries to enable this. *Aeraki*[1], attempts to close this gap by creating an extensible platform that enables layer 7 support for many

[1] `https://www.aeraki.net/`

protocols such as *Redis*, *Kafka* and *ZooKeeper*. This project, however, only provides support for *Istio* and therefore currently does not support any other service mesh implementation.

F6 New technologies enable additional data plane architectures.

A final finding is that only a single project uses a kernel-based proxy. *Cilium* uses a kernel-based proxy approach by utilizing eBPF applications to implement the service proxy. Together with *Traefik proxy*, it is the only service mesh that does not use a per-service based data plane architecture. We can attribute this finding to the fact that eBPF is a relatively new technology, and that the *Cilium* networking project has been banking on that technology before extending its networking solution to include the properties of a service mesh in late 2021 (13). Although this is a fairly new approach, it has been adopted by one of the leading Kubernetes service providers in the form of *Google Kubernetes Engine* (54). Furthermore, other service mesh systems also experiment with eBPF technology to improve upon existing functionalities and support new features (12, 55).

3.5.2 Analysis of Common Service Mesh Architectures

In this section, we analyse the identified service mesh architectures in more detail. For every identified architecture, we present a reference topology and discuss the advantages and disadvantages one such implementation has. First, in Section 3.5.2.1, we discuss the most common identified approach, a service mesh using a per-service proxy. Next, in Section 3.5.2.2, we discuss architectures leveraging a per-node proxy. Finally, in Section 3.5.2.3, we discuss architectures using a kernel-based eBPF approach.

3.5.2.1 Per-Service Proxy

During the systems survey, we identified ten different service mesh systems. Out of those ten systems, eight of those shared similar architectural characteristics. For all of these systems, we identified that the mesh network was established by introducing a proxy for every individual software service present. This architectural style is so common within the Kubernetes ecosystem, that the pattern is often referred to as the *sidecar pattern*.

In Figure 3.3, we present a reference architecture of this design pattern for service meshes in a Kubernetes cluster. This shows the data path of a packet throughout its lifespan in a system that implements this architecture. To explain the data path of a packet, we first introduce the individual components depicted, as most of them are present in the other architectures as well.

First off, we have the node (❶), this represents a worker machine in Kubernetes, which can run workloads. Next up, we have a Pod (❶), this is the smallest unit of deployment

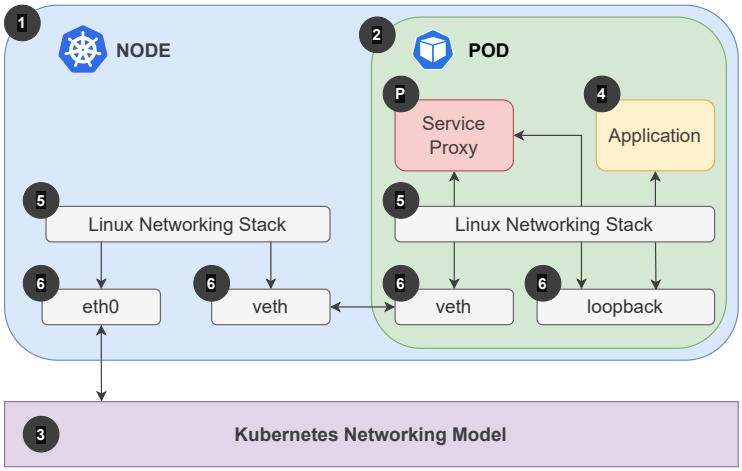


Figure 3.3: The data path of a network packet in a per-service data plane.

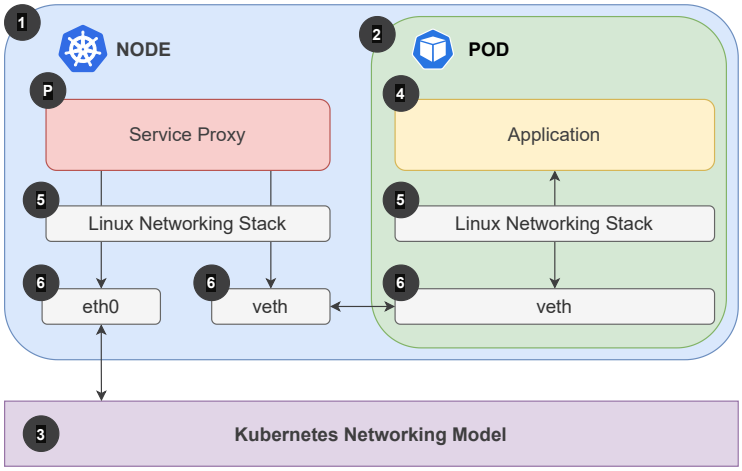


Figure 3.4: The data path of a network packet in a per-node data plane.

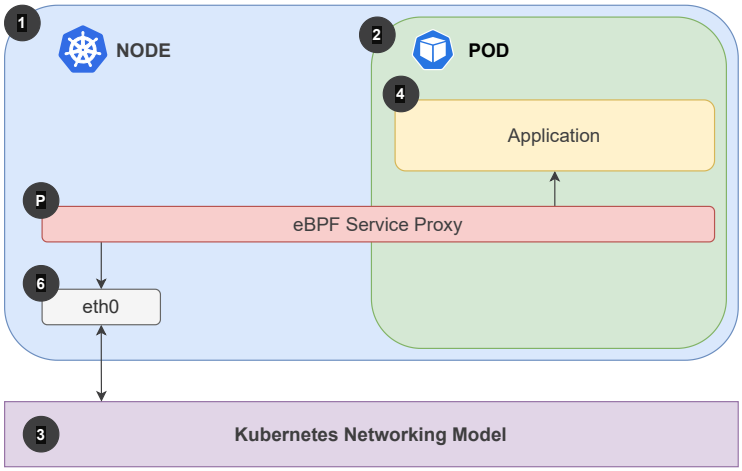


Figure 3.5: The data path of a network packet in an *eBPF-based* data plane.

within Kubernetes and consists of one or more application containers. The last Kubernetes related concept is the Kubernetes Networking Model (❸)[1], which can be implemented in various ways, but has as goal to connect the nodes and pods within the cluster. Within the Pod, lives the actual application (❹), which can for example be a web server. The *Linux Network Stack* (❺) is abstracted in this design as one single component, but consists of routing and filtering tables used to manage packets within the system. An important note, is that both the node and the Pod implement a separate networking stack, which isolates the pod network from the host network through network namespaces. The traffic flows to physical and virtual network interfaces (❻) connected through one another, to reach its destination. Note that in actual deployments, more complex models are often present, such as bridge networks to support multiple pod networks or tunnel interfaces to create overlay networks, for instance. However, for this example, the added complexity does not help to illustrate the architectural differences at hand.

With the basic components of the reference architecture explained, we can take a closer look at the data path of a service request. Whenever a request to a service has been made that lives within the application pod (❹) a packet first enters the node. The inbound packet enters through the network interface (❻), has to pass through the kernel's networking stack (❺) to reach the Pod's network namespace. After this, the traffic will reach the service proxy (Ⓟ) because it is configured to intercept all network traffic inside the pod. From there onwards, it has to travel through a loopback interface to finally reach the application. Replies from the application will traverse a similar, but inverse, data path.

One of the advantages of such an architecture is that you can have fine-grained control over the traffic, as it is intercepted the moment it enters the namespace of the Pod. This characteristic allows the security features discussed during this systems review (**FR2**), and can for example enable encrypted connections between all traffic leaving a Pod. Furthermore, the *sidecar pattern* allows for a very generic implementation, and does not require much if any modifications from the user's perspective. This enables a user-friendly user experience, while also allowing support for general purpose proxies such as *Envoy* or *HAProxy*. These advantages, make it so that this is the most common architectural approach identified. A disadvantage, however, is that this design introduces additional overhead. First off, it introduces an overhead in system resources, as there now is a network proxy for every software service. Furthermore, it introduces additional latency for the traffic, as for every service request, the packets travel through the proxy twice (ingress/egress).

3.5.2.2 Per-Node Proxy

Another identified service mesh architecture makes use of the per-node proxy. This architectural style is used by a single identified service mesh system, *Traefik Mesh*. We present

[1] `https://kubernetes.io/docs/concepts/cluster-administration/networking/`

this form of service mesh architecture in Figure 3.4. Although many of the components in this system are similar and previously explained in Section 3.5.2.1, some differences change several characteristics drastically.

The most significant difference compared to the per-service architectural style is that service proxy (Ⓟ) is now not embedded within the Pod but lives within the node, thus makes use of the host network namespace. This is because this architectural style tries to limit the additional overheads caused by the introduces proxies of a service mesh. This style of service proxies results in a single proxy per node, and thus does not scale linearly with the number of services on a node. This leads to the advantage that it consumes less system resources in the form of CPU cycles and memory. An additional benefit of this approach is that the traffic in the data path now has to traverse the proxy once in a service-to-service request. This halves the amount of service proxy hops compared to the previous solution.

However, this architectural pattern comes at a cost. Since the traffic is not passing through the service proxies from within the pod, traffic enters the host network in a potentially compromised state. To elaborate this, the per-service approach allowed the traffic between proxies to be automatically encrypted using a mutual TLS connection, a characteristic evaluated during the system's review (**FR2**). This is, however, not possible as the data path outside the Pods is not fully controlled and therefore has to be implemented manually. A downside is that when this is ignored, service-to-service data flows unencrypted through the node and therefore breaches well established security best practices such as the notion of zero trust networks (56).

3.5.2.3 eBPF Proxy

The last of the identified service mesh architectures uses a vastly different approach. The approach used by *Cilium* uses eBPF to establish a mesh network between the services. What started out as a Kubernetes networking solution and evolved to become a fully fledged service mesh implementation (13).

In Figure 3.5, we present a simplified data path of a service mesh implemented using *Cilium*. Many of the components are shared with the other architectures and are explained in Section 3.5.2.1. However, compared to the other identified architectures, the data path as depicted here is much shorter. This is because the routing and proxying is done in the kernel through eBPF (Ⓟ).

To explain the shortened data path, we first briefly have to introduce the technology powering it. eBPF[1] is a Linux Kernel technology that allows users to run sandboxed programs in the operating system kernel. Furthermore, it is event-driven and allows these programs to execute on certain hooks. This is used throughout *Cilium* to implement the observability, security and networking functionalities of the service mesh. eBPF allows

[1] `https://ebpf.io/what-is-ebpf/`

for hooks throughout the entire lifecycle of a network packet. This combined with the programmable sandbox environments allows it to replace the role of service proxies in the other architectural styles. Since the eBPF programs are aware of the Kubernetes endpoints and services, packets can be routed straight from the kernel, enabling a much more direct route.

This architectural style has several advantages compared to the previously introduced per-service and per-node architectures. First, it has potential to decrease latency in its data path. It does this by not requiring additional hops in service-to-service communications. Additionally, it allows for greater programmability of the mesh network, as certain programs can be compiled and then run in a sandboxed environment. However, it also can have some disadvantages. First off, the technology and this implementation of it in particular is fairly new, even in a beta phase at the time of writing. This means that the service mesh implementation could be unstable and not suitable for production usage. Furthermore, using user programs in kernel space could introduce potential security risks.

3.6 Summary

To conclude the systems survey, we return to the initial research question (**RQ1**), *How do existing service mesh implementations compare?* The answer to this question is provided in various levels of detail throughout the systems survey. The most detailed and objective answer is provided within the data synthesis and results section (Section 3.4) in which we identified domain-specific characteristics and features and compared the systems based on that. A summarized result is presented in the form of established functional and non-functional requirements (Section 3.5), in which we combine results from the data synthesis to provide a more generic, but more useful result. Finally, in Section 3.5.2, we take a deep-dive into the architectural differences of identified systems in which we answer the research question by exploring key characteristics in more detail.

Throughout this systems survey, we observed that service mesh systems can have different approaches and goals and that systems within this the area are constantly changing. With many of the systems adopting different strategies and bleeding-edge technologies, the area is exciting in many ways.

4

Design and Implementation of a Mesh Bench, a Service Mesh Benchmark

In the previous chapter (Chapter 3), we conducted a systems survey and identified several state-of-the-art service mesh systems. We examined the architectural differences between these systems and compared them on domain-specific functional and non-functional requirements. We then discussed the theoretical performance implications of the architectural styles that we had identified. These performance implications have an effect on the performance of all software services in an environment and can cause a major bottleneck. With these concerns in mind, we dive deeper into the performance of service mesh systems.

In this chapter, we provide an answer to research question **RQ2**. *How to design and implement a service mesh benchmark which evaluates the performance of service mesh systems?* During the systems survey, we observed that there was no relevant work found in the Primary Literature. Furthermore, preliminary research has shown that previous and current efforts made to evaluate these performance characteristics are either in early stages or have limited functionality. To solve these shortcomings, we present a service mesh benchmark instrument and implement a prototype that supports various workloads, application level protocols and service mesh systems.

The remainder of this chapter is structured as follows. First, in Section 4.1 we introduce the goals of the benchmarking instrument. Second, in Section 4.2 we describe the components that are part of the System Under Test. Afterwards, in Section 4.3 we present a requirement analysis in which we formalize the stakeholders, use cases and system requirements of the instrument. Next, in Section 4.4 we present the design of the *Mesh Bench*, a benchmarking instrument for service mesh systems. Afterwards, in Section 4.5 we discuss the implementation details of a prototype that implements the benchmark design. Finally,

in Section 4.6 we summarize and conclude this chapter.

4.1 Benchmarking Objectives

The goal of a benchmarking instrument is to objectively compare, and decide on the best system for a particular domain. However, the concrete definition of the best system depends on the underlying benchmarking objective (14). Therefore, it is important to first establish a set of goals for the benchmarking before designing or implementing it. In this section, we will introduce the benchmarking objectives, related to the main research question **RQ2**.

To design a benchmark that evaluates key characteristics of a service mesh system, we guide the process by following industry best practices for measuring and monitoring distributed systems. Following the best practices and principles of the Google Site Reliability Engineering handbook (57), we can establish a base set of metrics and signals that are important to measure in distributed systems. More specifically, we can follow the guidelines for *white-box monitoring* and implement the *Four Golden Signals* in our benchmarking instrument.

The following list presents the objectives that the benchmarking instrument has.

O1 Support System Resource Measurements

The first objective of the benchmarking instrument is to measure the amount of system resources a service mesh system uses.

O2 Support Throughput Measurements

The second objective is to measure the amount of throughput a system can handle. The throughput in this context refer to the number of requests per pre-defined time frame a service mesh system can handle. This is used to see if the service mesh system introduces any bottleneck in terms of throughput.

O3 Support Latency Measurements

The third objective is to measure the amount of latency requests have. This is used to measure how much latency a service mesh system introduces.

O4 Support Resiliency Measurements The fourth objective of the instrument is to measure the resiliency of the systems by measuring the error rates within the system.

O5 Enable Explorative Research The fifth objective is to enable explorative research, this objective aligns with the overarching goals of this thesis. A design that supports explorative research should be easily configurable and provide a fast experiment feedback loop.

O6 Highly Configurable The final objective is to design and a highly configurable benchmarking system. This means that the system will be able to perform many types of experiments by enabling a variety of situations through configuration.

4.2 System Under Test

To design a benchmarking instrument that evaluates service mesh systems, we first have to define the system and the components that are included in the benchmark. The components relevant to the benchmark are described in the System Under Test. However, even though the benchmark has to deal with numerous components related to the service mesh system ant its surrounding environment, not all are relevant to measure to satisfy the benchmarking objectives.

Therefore, we split the components of the System Under Test in two categories according to the practices outlined by Folkerts et al. (14). The first group of components are labelled *Components of Interest*. These components are of principal interest to the benchmark and have to be measured. The second group of components is labelled as *Purely Functional Components*. The components in the latter group are present in the System Under Test in order for the benchmarking instrument to work correctly. However, measuring these components is not of importance relating to the objectives of the benchmark.

In Figure 4.1 we present the System Under Test for the benchmarking instrument. In the following sections, we elaborate on the *Components of Interest* (Section 4.2.1) and *Purely Functional Components* (Section 4.2.2) present in this system.

4.2.1 Components of Interest

The *Components of Interest* are aligned with the objectives as presented in Section 4.1. In the context of a service mesh system, we aim to measure and evaluate the components present in the data path of a software service. Therefore, in the context of a Kubernetes cluster running a service mesh, we evaluate the performance of software services that live in the Kubernetes specific abstraction, the Pod ❶ .

To measure the service and the service traffic, we measure the application container ❷ and the data plane components of a service mesh ❸ .

4.2.2 Purely Functional Components

To be able to run the benchmark, we require the several components. First, we run the benchmark on a Kubernetes cluster F1 . Furthermore, a service mesh system introduces several control plane components, which are required for the service mesh to correctly function F2 . However, since our objectives are to measure the data path of a service mesh

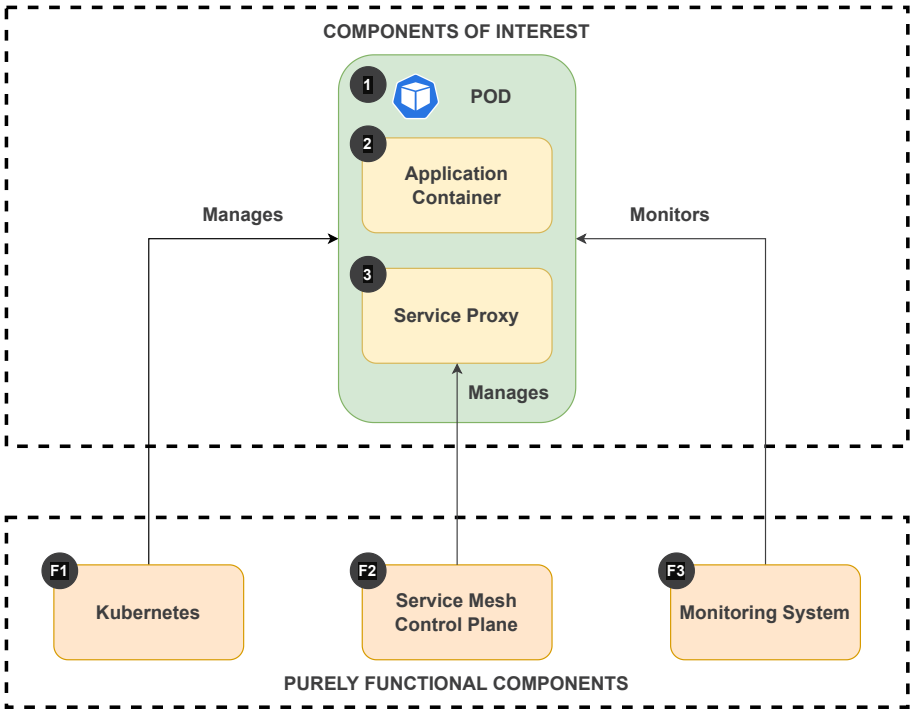


Figure 4.1: The System Under Test for benchmarking service mesh systems.

they are not of importance to the benchmarking instrument. Additionally, we introduce a metric collector and monitoring system in the cluster (F3), which will be used to collect and time series data of the pods and containers residing in the Kubernetes cluster.

4.3 Requirements Analysis

In this section, we expand on the details and requirements of the benchmarking instrument. To properly define the requirements of the benchmark, we first have to define the stakeholders and their use cases for such an instrument. After that is defined, we can define the actual requirements of the instrument based on those.

In Section 4.3.1 we define the stakeholders. Afterwards, in Section 4.3.2 we define the use cases for the benchmarking instrument. Finally, in Section 4.3.3 we define the actual requirements.

4.3.1 Stakeholders

To uncover the use cases of this benchmarking instrument, we first have to identify the primary stakeholders. This is because stakeholders can have different applications for such a system, and therefore have different requirements. The following list of stakeholders (**S1-S3**) introduces the identified stakeholders and their concerns.

S1 DevOps Engineers

This group of stakeholders is the primary user of a service mesh system. They manage and control the infrastructure applications and services run on, and therefore can consider implementing a service mesh system. Their primary concern with the benchmarking instrument is to evaluate the performance and resiliency characteristics of a service mesh implementation on existing infrastructure or compare different implementations to make an informed decision to choose between the various implementations. For this group of stakeholders, it is important that the service mesh matches their goals and requirement. This is exemplified by a DevOps engineer bound to predetermined Service-Level Agreements that define contracts on service latencies.

S2 Service Mesh Engineers

The second group of stakeholders consists of the engineers that develop the service mesh systems itself. For these engineers, it is important to track the performance of their system to make sure that it meets the predetermined performance requirements. Furthermore, it can enable the engineers to compare their system to other offerings in the service mesh landscape, which can help this group to set expectations and requirements.

S3 Scientific Researchers

The final group of stakeholders represents researchers from academic institutions. This group uses the benchmarking instrument to conduct scientific research on the service mesh systems, and their workloads. For this group of stakeholders, it is important that the benchmark instrument reports detailed information on both the system and the workloads running on it so that they can reason on the results. Furthermore, it is important that the results the instrument produces are reproducible, meaning that experiments conducted with similar inputs produce similar outputs (bearing in mind the inherent variability in distributed environments). Finally, it is important that the instrument can run different workloads so that they can evaluate different situations and environments.

4.3.2 Use cases

Previously, we have identified the stakeholders of a benchmarking instrument, (**S1**-**S3**) as it is important to know the users and their concerns. This helps us to understand *who* the users of the benchmark are, and *why* they would use such an instrument.

To understand *how* such a benchmark is used, we need to identify the use cases. Drawn from the concerns of the stakeholders, we present a list of use cases that (**UC1**-**UC3**) this

benchmarking instrument can fulfil.

UC1 Competitive Benchmarking, *concerns:* **S1**, **S2**, **S3**

The first and most common use case of the benchmarking instrument is to perform competitive benchmarking. The benchmarking instrument produces quantitative data on the performance characteristics of a service mesh system. The results of the benchmark can then be used to compare service mesh implementations and decide on the best performing system in a domain.

UC2 Strategic Benchmarking, *concerns:* **S2**, **S3**

A second use case of the benchmarking instrument is to enable critical insights into the best performing systems. A benchmarking instrument gives insights into the performance of various characteristics of a system. This allows users to analyse the best performing systems in certain individual areas. Analysing those systems in further detail allows users to develop an understanding on how these systems achieved their results and allows them to learn from it. These critical insights can then be translated into other systems applying these best practices.

UC3 Evaluate Periodic Performance, *concerns:* **S1**, **S2**, **S3**

A third use case for the benchmarking instrument is to capture the performance of a system over time. When conducting a benchmark, you evaluate a system and capture the performance of that system at a specific time. However, it can also be used to monitor and evaluate the performance of a system over a larger time frame to determine if, and by how much, the performance characteristics alter over time.

UC4 Validation of Performance Optimizations, *concerns:* **S2**, **S3**

Another use case for this benchmarking instrument is to analyse the impact of performance optimizations applied to service mesh systems. When such an optimization is made, users can evaluate the system pre- and post-optimization to measure the impact of the optimization.

UC5 Validate System Requirements, *concerns:* **S1**

A final use case for this benchmarking instrument, is to evaluate if a service mesh meets the requirements of the end user. End users can have different requirements, such as performance and resiliency requirements. This benchmark will enable those users to evaluate a service mesh on their infrastructure to see if the system meets their requirements.

4.3.3 Requirements

To design a benchmarking instrument that resembles any value, we satisfy the concerns and use cases of the identified stakeholders. To ensure that we satisfy this, we formulate a set of requirements, which we present in the following sections.

4.3.3.1 Functional Requirements

Functional requirements are requirements that the system *must* implement, to function correctly. The following list (**FR1 - FR7**) introduces the functional requirements of the benchmarking instrument and to which use case they relate.

FR1 Support the most popular service mesh implementations., *related to:* **UC1**, **UC3**, **UC4**

The benchmarking instrument must support the most popular service mesh implementations. During the systems survey (Chapter 3), we identified that two of the identified service mesh implementations currently dominate the landscape in terms of usage (11). Therefore, the benchmarking instrument has to support both *Istio* and *Linkerd* to be relevant for most users.

FR2 Support systems using different data plane architectures, *related to:* **UC1**, **UC3**

The benchmarking instrument must support all the identified data plane architectures (Section 3.5.2). During the systems survey, we identified four different data plane architectures with various performance implications. To analyse this, the benchmark must support at least one of the identified systems for each identified data plane architecture. The system already has to support systems adhering to the *per-service* through the first established functional requirement (**FR1**). Since the other architectures only relate to a single service mesh implementation, it means that the system must additionally support *Cilium* and *Traefik*.

FR3 Implement and Report the Four Golden Signals, *related to:* **UC1**, **UC2**, **UC3**, **UC4**, **UC5**

The benchmarking instrument must implement the *Four Golden Signals*. To report meaningful data, we have to define meaningful metrics that the benchmarking instrument will report. To achieve this, we define the metrics by following industry best practices as established in the Google SRE handbook (57) and implement the *Four Golden Signals*. This means that the benchmarking instrument must implement relevant metrics to report the *Latency*, *Traffic*, *Errors*, and, *Saturation* of the System Under Test.

FR4 Support Varying Workloads, *related to:* **UC2**, **UC3**, **UC5**

The benchmarking instrument must support varying workloads. This allows the benchmark to be relevant for the different use cases it can encounter and simulate the different workloads a service mesh can see in the real world. To support this, the benchmark must provide a mechanism to specify the workload that the benchmark will run. Additionally, it has to provide a way in which the user can specify how the benchmark can extract relevant metrics. This is done by specifying how the workload, or service, can be consumed (e.g. HTTP endpoints).

FR5 Support Varying Cluster Environments, *related to:* **UC2**, **UC3**, **UC5**

The benchmarking instrument must support varying cluster environments. This means that the benchmark must support any valid Kubernetes cluster environment. This allows users to evaluate service mesh implementations on within their own environment and enables environment-specific use cases such as **UC5**.

FR6 Report results in a well-known data format, *related to:* **UC1**, **UC2**, **UC3**, **UC4**

The benchmarking instrument must report its results in a well-known data format. In order for the results to be useful to the various stakeholders, data must be presented in well-known formats such as *JSON*, *CSV* or *YAML*.

FR7 Produce Reproducible Results, *related to:* **UC1**, **UC2**, **UC3**, **UC4**, **UC5**

The benchmarking instrument must produce reproducible results. To provide meaningful results, the instrument has to be able to produce similar results when running a single experiment in the same environment. If the instrument is given the same inputs, similar outputs have to be expected.

4.4 Design of a Service Mesh Benchmarking Instrument

In this section, we present the design for *Mesh Bench*, a benchmarking instrument for service mesh systems. In Figure 4.2 we depict an overview of the benchmark and all of its components. This design shows how a user interacts with the benchmark, and how it can evaluate the components within the System Under Test (see Section 4.2). In the remainder of this section we will introduce the annotated components (as indicated by the black circles ❶ - ❽) and discuss the functionalities of them.

4.4.1 Benchmark Interface

The *Benchmark Interface* ❶ is the entry point for the end user and enables them to conduct performance experiments on service mesh systems. The main goal of the Benchmark

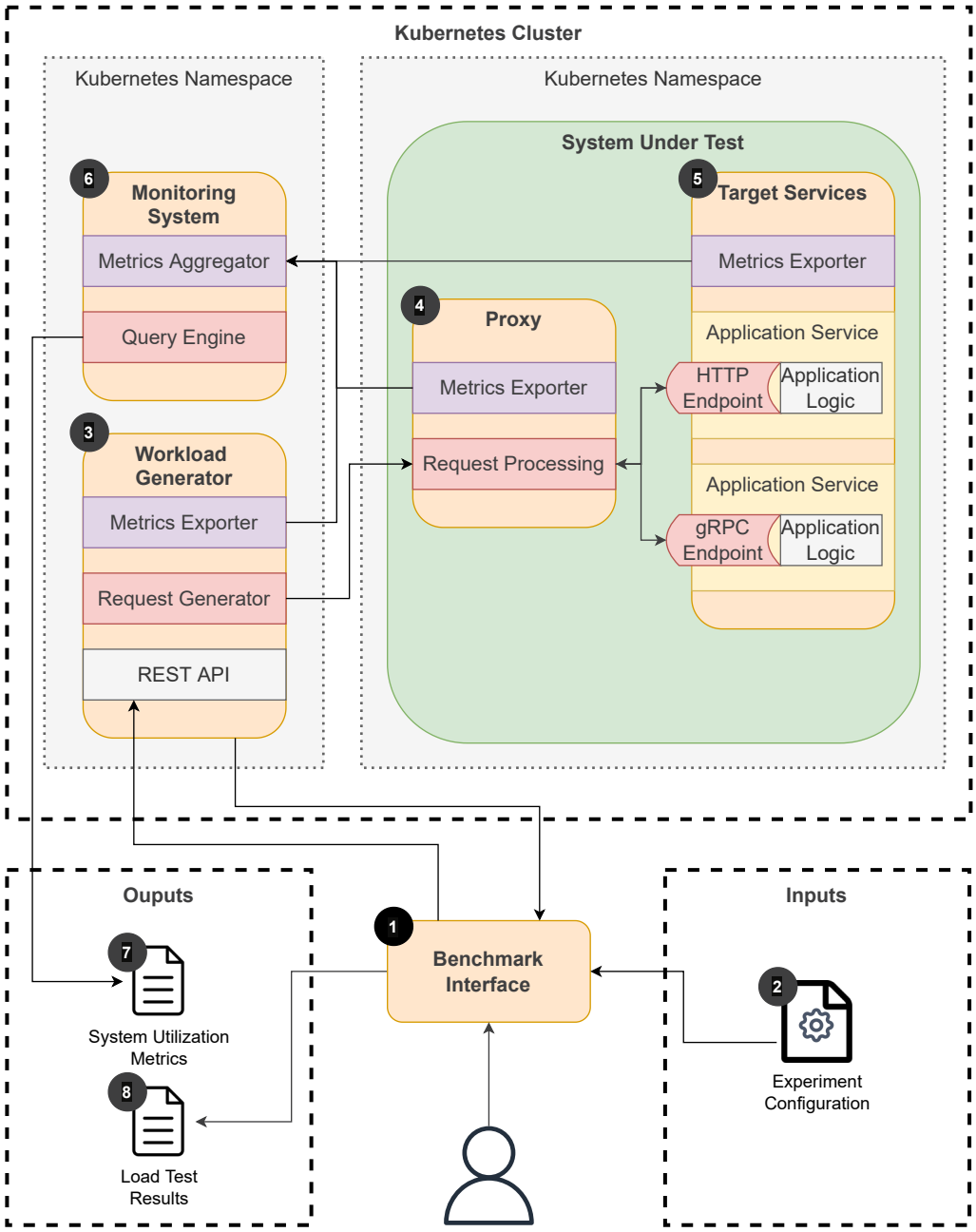


Figure 4.2: A design overview of *Mesh Bench*, a benchmark for service mesh systems.

Interface is to manage experiment executions. The lifecycle of a single experiment consists of three phases. The first phase consists of initialization and configuration. This is done by parsing and processing an *Experiment Configuration* ❷ . After this, it instructs the *Workload Generator* ❸ , to generate experiment-specific workloads based on the supplied configuration. Once an experiment is finished and the workload generator has reported back to the benchmark interface, it finalizes the experiment and stores the obtained results

❽ .

4.4.2 Experiment Configuration

The benchmark consists of several performance experiments, as introduced in Chapter 5. Each of these experiments consists of several configuration parameters, which allows the user to change several aspects of the experiment. Notable configurations include the ability to modify the type of workload, the frequency of the workload, the duration of the experiment and what type of workload to use, and therefore which target service ❺ . These configuration parameters of a single experiment, forms the *Experiment Configuration* ❷ and is the primary input of the benchmark interface ❶ .

4.4.3 Workload Generator

The *Workload Generator* ❸ is used to generate synthetic workloads and measure the performance of the components within the System Under Test. This is arguably the most important component in the benchmark, and it has to support various modes of operation to match the functional requirements as defined in Section 4.3.3.1. The workload generator has to be able to perform load test experiments, in which the *Request Generator* generates a certain application workload for a given *Target Service* ❺ . It has to be able to do so in a constant throughput fashion, i.e., a fixed number of requests per second while also supporting a mode that enables us to test the maximum throughput of a given System Under Test. Another important aspect is that it should be configurable at run-time, so that the benchmark interface ❷ can configure it, allowing us to design various performance related experiments. This aspect is done through the *REST API* as depicted in the benchmark design. The workload generator also exposes system utilization metrics through the **Metrics Exporter**.

4.4.4 Proxy

The *Proxy* ❹ is the core component of a service mesh architecture. The entire benchmark is designed to measure the performance implications of the proxy component. This component varies based on the service mesh system that is evaluated (see Section 3.5.1 for the proxies used per service mesh system). Furthermore, it is unmodified, which means that it will intercept all the requests from and to the target service ❺ , and do the *Request Processing* as originally intended. However, we do export the resource utilization metrics through the *Metrics Exporter*.

4.4.5 Target Services

At the receiving end of the requests as generated by the workload generator are the *Target Services* ❺ . The goal of this component is to mimic a generic service as encountered in a production environment. In this benchmark, we have two *Application Services* to mimic different types of workloads. The first type of service accepts workloads through the *HTTP endpoint*. The second type of service accepts workloads through its *GRPC endpoint*. Both of the backing services accept the synthetic workloads, and perform minimal to no computations on it (*Application Logic)*. This minimizes the impact on end-to-end latencies observed and keeps the resource overheads to a minimum. This allows us to focus on the performance implications of the proxy, instead of the backing services. Just like the other components in the benchmarking system, we export resource utilization metrics through the *Metrics Exporter*.

4.4.6 Monitoring System

The *Monitoring System* ❻ is responsible for monitoring the systems within the *Kubernetes Cluster*. This is used primarily to collect resource utilization metrics for all the components within the system through the *Metrics Aggregator*. The monitoring system has to support the collection of these metrics at a fine granularity. More specifically, it has to be able to distinguish resource utilization metrics at a pod and container granularity. This allows us to identify resource utilization patterns for the components of interest. A user should be able to query the aggregated metrics through a *Query Engine* and output these results in a common format to stable storage in a common format ❼ .

4.4.7 Load Test Results

The *Load Test Results* ❽ are the results created by the *Workload Generator* ❸ . These results contain the performance results of an experiment. More specifically, it contains the request latencies as generated by the workload generator. Additionally, it comes with metadata such as the experimental configuration and environment data.

4.5 Implementation

In this section, we discuss the implementation details of a benchmark prototype that adheres to the design of *Mesh Bench* as presented in Section 4.4. The remainder of this section will discuss the design decisions for the core components in more detail. We present our implementation details, and the alternatives we had considered and how they relate to our requirements as discussed in Section 4.3.

Workload Generator		Workload Support		Configurability		Other
ID	Name	HTTP/2	GRPC	Payload	Output	Server Process
WG1	wrk2	○	○	◐	◐	○
WG2	k6	◐	◐	◐	◐	◐
WG3	hey	●	○	●	●	○
WG4	ghz	○	●	●	●	○
WG5	fortio	●	●	●	●	●

Table 4.1: Comparing workload generator systems on relevant properties and features. There are three different symbols in the table, each of them represents how well a requirement is satisfied. ●: Fully Satisfied, ○: Not Satisfied, ◐: Partially Satisfied.

4.5.1 Workload Generator

The workload generator is arguably the most important component of the benchmark and has to support a large amount of functional requirements as discussed in Section 4.3.3.1.

Foremost, the workload generator has to support *varying workloads* (**FR4**). This means that it has to support the two most common service-to-service communication protocols in the form of HTTP and gRPC (as derived from our system survey results in Section 3.4.3.5). Furthermore, to support a wide variety of experiments the workload generator would have to support varying workload sizes and payloads so that the user can specify and evaluate the impact it would have. In addition to this, it would be beneficial to us to be able to configure this on the fly. This would mean that the workload generator would be deployed as a server-side process, that can receive instructions from a client-side process.

The final two functional requirements for the workload generator are related to the data it produces. First off, it would have to implement three of the four golden metrics as discussed in **FR3**. It would have to measure the end-to-end *latency* of any request that it generates towards a target service. Additionally, it would have to measure the amount of *traffic* at any given time, measured in the form of *queries per second*. Finally, it would have to check for any *errors*, this would mean it has to evaluate the HTTP and gRPC status codes to check if the request was successful. The last requirement has to deal with the format the data is presented in **FR6**. The resulting data would ultimately be reported back to the user to analyse the traffic and its performance.

To decide on a fitting workload generator we evaluated numerous systems as seen in Table 4.1. We performed simple load tests with each of these systems and assessed their support for the relevant functional requirements.

WG1 wrk2

wrk2 was the first workload generator that we had considered as it was a battle-tested

solution. It is an evolution of *wrk* that can achieve highly accurate latency details (i.e., 99.9999%'ile). It also consumed the least amount of system resources which would translate in the ability to evaluate systems under more load. Configurability was done through command line interface options, as most of the evaluated systems did. However, it did not support many modern features, most notably it did not support http/2 and only supported the older protocol versions. Additionally, it did not support reporting in common formats nor was it able to modify request payloads out of the box, but required Lua scripts to modify its behaviour. Furthermore, it would have to be paired with a load generator that would support the gRPC protocol as per the functional requirements, as it did not support that either.

WG2 k6

k6 was another promising solution, as the programmable take on load testing would allow us to design and create any form of synthetic workload that our benchmark would require. Although this approach is great for complex load testing environments (i.e. those that requires specific flows such as authentication procedures for instance), it was less useful to perform relatively simple load testing experiments. The solution that required pre-programmed scripts that represent a load test scenario is a bit too convoluted for experiments that changed a singular variable. We also noted that k6 used a lot of system resources whilst generating workloads, that could potentially influence the outcome of load-test experiments if the load generator and target service operate on the same machine.

WG3 hey

hey is another load testing tool for HTTP-based workloads. It is a modern tool that supported all the required functionalities and was a prime candidate to act as a component within the workload generator.

WG4 ghz

ghz is a benchmarking and load testing tool for gRPC based services. It matched all of our functional requirements and was considered for the implementation of the workload generator.

WG5 fortio

fortio, however, was the implementation we ultimately decided upon using. As this solution was designed to evaluate the performance of services under various environments and conditions. It was able to generate both HTTP and gRPC based workloads, which allowed us to use a single tool for both types of workloads. Additionally, it supported a wide variety of configuration options, that allows us to modify

payloads. A decisive feature, however, was that it was able to operate as a server-side process. This meant that we could run this process in the cluster as depicted in the benchmark design Figure 4.2 and instruct it from a separate component, controlled by the user.

4.5.2 Target Service

The workload generator generates requests to a target service, which has to process this request and reply afterwards. As previously mentioned, we want to minimize the impact the target service has on the end-to-end latencies as observed in the data path. To accomplish this we first developed simple services, that replied the content of the requests back to the requesting entity.

However, fortio comes bundled with a set of testing services that we could use for our use case. This meant that we could instrument our benchmarking prototype with fortio as the load generator, and as target services. It comes bundled with an HTTP echo service that acts similarly as described above. In addition to the HTTP echo service, it comes with a simple gRPC based service that implements a common health checking protocol[1].

4.5.3 Monitoring System

The workload driver allows us to capture and report metrics related to service requests. However, to measure the amount of system resources the components in our System Under Test use we have to implement a system that exports the related metrics in a fine granularity which can differentiate various container processes. In addition to that we need a system that can extract these metrics and aggregate them. Furthermore, the reported metrics should be stored as time-series data, so we can analyse the metrics at a given time in conjuncture with our experiments. Finally, the monitoring system should support extensive querying capabilities and the ability to export data in a common format **FR6**.

For the implementation of our monitoring system we decided to use *Prometheus*[2]. This tool satisfies all the requirements and is a common cloud-native tool that matches our use case. As a matter of fact, it is so common within the Kubernetes ecosystem that most of the service mesh systems as identified, provide support for this tool (see Section 3.4.3.2).

To extract resource utilization metrics, we can make use of the *cAdvisor*[3] daemon that is already embedded into a standard Kubernetes component (the *Kubelet* process). cAdvisor collects, aggregates and processes resource utilization metrics and is aware of the isolation parameters of a container. By introducing Prometheus into our benchmark prototype,

[1] `https://github.com/grpc/grpc/blob/master/doc/health-checking.md`
[2] `https://prometheus.io/`
[3] `https://github.com/google/cadvisor`

and configure it to scrape and collect the metrics exposed by cAdvisor, we can analyse experiment results and relate it to resource utilization data per pod and container.

Additionally, we use *Grafana*[1] in our prototype. This is another common, open-source tool that acts as an observability platform. It natively supports Prometheus and can be used to analyse and explore the collected data. Although we do not use it for our data visualizations as presented in the experimental evaluation (Chapter 5), we did use it to quickly validate and explore results of experiments.

4.6 Summary

To summarize and conclude this chapter, we return to the initial research question (**RQ1**), *How to design and implement a benchmark that evaluates the performance of service mesh systems?* To design and build a benchmark that properly evaluates the performance characteristics of service mesh systems we first used the learnings from our extensive system analysis as conducted in Chapter 3. Based on these learnings, we focussed on a sub set of components that were of interest to us and the performance implications they could have. Before starting the design phase, we established a set of benchmarking objectives based on industry best practices (Section 4.1). After this, we clearly defined our System Under Test and separated our components of interest and purely functional components as to help with our design (Section 4.2). Subsequently, we then performed an extensive requirements analysis, in which we clearly defined the stakeholders and use cases (Section 4.3). With the components, stakeholders and use cases clearly defined, we started our design process (Section 4.4). After many reiterations to make the design more concise and modular, we finished the design of *Mesh Bench*. Finally, after the design was completed, we implemented a prototype and evaluated the working prototype.

[1] `https://grafana.com/`

5

Experimental Evaluation of Service Mesh Systems

In the previous chapter (Chapter 4), we defined the System Under Test and the metrics that are important to evaluate the performance aspects of service mesh systems. This ultimately led to the design, and implementation of *Mesh Bench*, a benchmarking system for service mesh systems. In this chapter, we present the experimental evaluation of service mesh systems using the aforementioned benchmarking tool. With this, we provide an answer to research question **RQ3**. *What are the differences between the different service mesh systems in terms of overhead, throughput, and latency?*

The remainder of this chapter is structured as follows. First, in Section 5.1 we introduce the design of the experimental setup. After this, in Section 5.2 we evaluate and measure the impact of our experimental environment and functional components of our benchmark through several micro benchmarks. Subsequently, in Section 5.3 we present the results of our experimental evaluation and perform an extensive analysis. After that, in Section 5.4, we discuss any threats to validity for our experimental evaluation. Finally, in Section 5.5 we provide a summary of the contents discussed in this chapter.

5.1 Experiment Design

The goal of the experimental evaluation is to gain insights into the performance characteristics of service mesh systems. We do this by designing and conducting experiments to extract and gauge relevant metrics for these systems; and by analysing these results in detail at the end of each experiment. This section builds upon the knowledge gained in previous chapters and details all the relevant aspects of the experiments.

Service Mesh	Version	Reason for inclusion
Cilium	v1.12.0-rc1	The singular identified service mesh system that uses an in-kernel, *eBPF*-based data plane architecture.
Istio	1.14	Prevalence and market share.
Linkerd2	2.11.2	Prevalence and market share.
Traefik Mesh	1.4.5	The singular identified service mesh system that uses a per-node service proxy in its data plane architecture.

Table 5.1: The service mesh systems used in the experiments and the reason of inclusion.

5.1.1 Service Mesh Selection

The selection of service mesh systems to include in the experiments is based on two criteria. First, we aimed to include a representative selection of systems that are used in real-world, production grade environments. Secondly, we aim to include systems based on different architectural approaches. In Chapter 3, we survey the existing state-of-the-art service mesh systems in the industry and have identified the most frequently used, production grade systems. Furthermore, we identified three different architectural styles that could have major implications on the performance of these systems.

With this in mind we present the list of mesh configurations that is used throughout the experiments, additionally in Table 5.1 we summarize this selection and present the exact versions of the systems used during our experimental evaluation.

SM1 None (baseline)

To establish a baseline of performance, we start by performing experiments without any service mesh at all. This enables us to not only compare different mesh systems to one another, but also allows us to paint a picture of the performance overhead compared to an unmeshed environment.

SM2 Istio

The first mesh of the selection is *istio*, we chose to include this service mesh because of its prevalence in the industry. During our system survey we have identified that *istio* is the most frequently used, production grade system at the time of writing.

SM3 Linkerd

The second mesh of the selection is *linkerd*, we chose to include this service mesh because of its popularity. With a service proxy built for service mesh workloads and a focus on performance, this system quickly gained popularity and established itself as a dominant entry in the industry.

SM4 Traefik

The third mesh included in the experiments is *traefik mesh*. We chose to include this mesh because it uses a vastly different architecture for its data plane. During our system survey we established that most of the identified service mesh systems used a per-service proxy. *Traefik mesh* was the only system that we identified that used a single proxy *per-node*. By including this mesh in our experiments we aim to uncover the performance implications of this type of architectural approach.

SM5 Cilium

The final mesh included in the experiments is *cilium*. A rather new entry in the world of service mesh systems that we chose to include because of its architecture. During our system survey we identified that *cilium* was the only service mesh system which used a eBPF based data plane proxy. By including this mesh in our experiments we aim to expose the performance implications such architecture can introduce.

5.1.2 Metrics

To evaluate the performance characteristics of service mesh systems we have to extract relevant metrics from the System Under Test. In Chapter 4, we designed and implemented a benchmarking instrument to evaluate service mesh systems. During the requirements analysis (Section 4.3) we identified the stakeholders, discussed their use cases and formed a list of requirements. During this, we established a set of metrics that are used to evaluate the performance characteristics based on industry best practices.

Below we present the list of metrics used throughout the experiments

M1 CPU utilization

The first metric that we use during the experiments is related to resource utilization. This helps us to establish how much computational overhead the data plane proxies of a service mesh introduce. All the CPU utilization results are expressed in fractions of a CPU core per second, e.g. the test system has 2 CPU cores and maximum utilization at a given time would be 2.0. More specifically, we use the `container_cpu_usage_seconds_total` metric as exposed by the *Kubelet* process. We obtain this result through the *Prometheus* API by using the query as seen in Listing 5.1.

```
1       sum(rate(
2           container_cpu_usage_seconds_total{
3               container!="",
4               pod=~"(target-fortio|traefik-mesh-proxy|cilium).*"
5           }[1m]
6       )) by (pod, container)
7
```

Listing 5.1: PromQL query for CPU metric.

M2 Memory utilization

The second metric that we use in the experiments is also related to the resource utilization. We capture the memory utilization of service proxies by measuring the amount of memory they consume during operation. More specifically, we use the `container_memory_working_set_bytes` metric as exposed through the *Kubelet* process. We obtain this result through the *Prometheus* API by using the query as seen in Listing 5.2.

```
1       sum(rate(
2           container_memory_working_set_bytes{
3               container!="",
4               pod=~"(target-fortio|traefik-mesh-proxy|cilium).*"
5           }[1m]
6       )) by (pod, container)
7
```

Listing 5.2: PromQL query for CPU metric.

M3 Requests per Second

The third metric that we use is the amount of requests the system has handled expressed in requests per second. This metrics allows us to identify to measure the amount of throughput or traffic that a system can handle.

M4 Request Latency

The fourth metric that we use in our experiments captures the amount of time it takes for a request to be served. This metric, is expressed as latency in milliseconds allows us to establish the overhead caused in the data path of every request in the system.

In the experiments we do not evaluate metrics related to the resiliency of the system. In our requirements analysis we established a requirement to capture this characteristic

based on established best practices. However, is not included in the current experimental design. The reason for this is twofold. First of, we uncovered that the service mesh systems handle timeouts and errors in requests differently[1] and that it involves balancing resiliency capabilities and system load. This means that we cannot compare the systems based on their default settings. Second, to create a fair experiment we have to configure these systems to have identical fault tolerance behaviours, which not all systems support. However, this is an interesting opportunity for future research.

5.1.3 Workloads

The selection of workloads aims to represent varying real-world scenarios that a service mesh might encounter. Since a service mesh handles most, if not all the traffic between software services, it can encounter a wide variety of workloads depending on the use case. With this in mind, we design our experiments around various generic workload patterns.

During the systems survey conducted in Chapter 3 we analysed the state-of-the-art service mesh systems and identified their key Functional Requirements and Non-Functional Requirements. We identified that service mesh systems handle service traffic in an application aware manner. Furthermore, we identified the most common application level transport protocols in terms of support. With this in mind, we designed the experiments around the two most common application level protocols.

5.1.3.1 HTTP Workload

For the experiments that make use of HTTP workloads we use a simple HTTP service. This service represents a simple echo service and replies with the data a client sends to it. This service requires relatively little system resources to operate and scales well to handle large amounts of connections. Additionally, it is also very configurable which enables us to test HTTP workloads with varying conditions and response payloads.

Although this is a synthetic workload which cannot be compared to any real-world workload, it allows us to capture and evaluate the general performance characteristics of service mesh systems under varying circumstances.

5.1.3.2 gRPC Workload

To evaluate how a service mesh performs under varying application level workloads we included a gRPC-based workload pattern. During the system survey we observed that all the identified service mesh systems supported the gRPC protocol and noted that it was a common communication standard in distributed computing. With this in mind we introduced a simple gRPC-based workload.

[1] `https://linkerd.io/2.11/features/retries-and-timeouts/`

GKE Configuration	
Compute Region	europe-west4
Kubernetes Version	1.22.8-gke.201
Kubernetes Release Channel	regular
VPC-native-routing	enabled
Dataplane V2	disabled
Node Configuration	
Compute Zone	europe-west4-a
Node Count	1
vCPU	2
Memory	8 GB

Table 5.2: The cluster configuration used throughout the experiments.

To use the gRPC protocol, both the client and server need to know how the service operates, which is defined in the service definition[1]. Best practices dictate that gRPC-based services should to include a health checking endpoint[2]. We use this to our advantage by using a simple gRPC service that includes this endpoint. For this type of workload we configure our workload generator to generate requests towards the gRPC service endpoint, to evaluate the performance implications of other (non-HTTP) application level workloads on service mesh systems.

5.1.4 Experimental Environment

It is important for experiments to be repeatable and that the variance is a little as possible. This allows others to reproduce and verify the results obtained and therefore be of scientific relevance. To minimize this variance and create reproducible results we present and discuss our experimental environment in detail.

5.1.4.1 Cluster Configuration

All the experiments were conducted on a Kubernetes cluster running on Google Cloud Platform. The cluster was created through their managed Kubernetes service Google Kubernetes Engine. The decision to use their managed service to construct the cluster was based on their flexible, but vast array of network configuration options. In Table 5.2 we present the most important details of the cluster and node configurations used. Which we will now discuss in further detail.

[1] `https://grpc.io/docs/what-is-grpc/core-concepts/#service-definition`
[2] `https://github.com/grpc/grpc/blob/master/doc/health-checking.md`

First, we conducted all the experiments in the *europe-west4* region. More specifically, all the compute nodes are located in the *europe-west4-a* data centre location which is located at Eemshaven in the Netherlands. We chose this location as it was closest to our own location. Secondly, we used the most recent version of Kubernetes available at the time of writing and chose this version from the stable release channel. Thirdly, we made use of *VPC-native-routing*[1] which implements the Kubernetes networking model[2]. Additionally, we disabled *Dataplane V2*, an upcoming networking solution based on *cilium* that would conflict with the service mesh systems that we evaluate.

For the node configurations in the cluster we used a single *e2-standard-2* node. This type of machine has 2 *vCPUs* and 8 GB of memory[3] and is the default option for Google Cloud Platform and is suitable for many types of workloads. We designed the experiments to be run on a single node cluster to abolish any variance in network latencies during the experiments. If for example, we chose to run the load generator on a single node and the target service on another node we introduce additional variance of network latencies. Furthermore, we validate this design decision in a micro benchmark, in which we test both single and multi-node cluster configurations (Section 5.2.1).

5.1.4.2 Metric Collection

We capture two types of metrics from the experiments. The group of metric is related to the load generator (**M3**-**M4**). These metrics are collected by capturing the JSON output that the load generator produces. The second group of metrics is related to resource utilization (**M1**-**M2**). To analyse the resource overheads of the data plane proxies, we capture the resource utilization metrics at a fine level of granularity, more specifically, at the level of a Pod.

To collect resource utilization at this level of granularity we bootstrap the cluster with various monitoring instruments. More specifically, we rely on *Prometheus*[4], a metric collection and monitoring system for time-series data (see Section 4.5.3).

5.1.5 Experiment Variables

When performing load testing experiments we can observe and control many of the variables present in the environment. We can do so on three areas of the benchmark. First of, we can control which service mesh is used, this will be done across all experiments. Section 5.1.1 details which service mesh configurations are used and why they were chosen. Additionally, we can control the settings of the workload generator. This allows us to tweak what kind of

[1] `https://cloud.google.com/kubernetes-engine/docs/concepts/alias-ips`
[2] `https://kubernetes.io/docs/concepts/services-networking/#the-kubernetes-network-model`
[3] `https://cloud.google.com/compute/docs/cpu-platforms`
[4] `https://prometheus.io/`

Variable	Description	Default
Mesh Configuration	What type of service mesh configuration is used in the experiment. (Section 5.1.1)	-
Workload Type	What type of workload the load generator should send (Section 5.1.3).	HTTP
RPS	How many requests per second the load generator should generate.	100
Connections	Number of simultaneous connections to use when making requests.	32
Duration	How long the load generator should generate load to the target service before aggregating and outputting results.	15 minutes
Payload Size	The size of the response payload in bytes.	0

Table 5.3: Overview of the experiment variables.

workload we produce (Section 5.1.3), but also enables us to control the frequency in terms of constant throughput that it is producing. The final area which we can tweak is on the side of the target services. The target services are defined as the receiving endpoints of the workload generator. This allows us to adjust the way a service responds, allowing us to introduce artificial delays, introduce faults or change application payload sizes.

In Table 5.3 we present the variables that we control across the experiments in this thesis. Additionally, it includes the default values that were used for these variables if they are not introduced as factor variables within an experiment.

5.1.6 Experiments

We now introduce the four experiments that were designed and used to evaluate the performance characteristics of service mesh systems. Tables 5.4, 5.5, 5.6 and 5.7 introduce the four individual experiments. For each of the experiments we introduce their primary goal, discuss how they are executed and reason why we chose this particular design. Additionally, Table 5.8 presents an overview of all the experiments, what type of workload they use, which metrics they capture and which factors are used to control the experiment.

5.2 Microbenchmarks

Before conducting our designed experiments, we evaluate our experimental setup through several micro benchmarks. The goal of these benchmarks is to measure the impact of our functional benchmark components and experimental environment.

EXP 1 - HTTP Maximum Throughput	
Goal	To find the maximum throughput each of the meshed configurations can achieve.
Methodology	The load generator will run in an unrestricted mode and generate as much load as possible without holding back and without any request timeouts. The maximum throughput is determined by looking at the actual number of requests per second in the configuration was able to achieve.
Workload	HTTP (Section 5.1.3.1)
Factors	Service mesh configuration (Section 5.1.1)
Reasoning	This experiment will serve as a baseline to determine the maximum throughput of all meshed configurations. The results will be used to determine sensible defaults for the experiments with a pre-defined constant throughput.

Table 5.4: Experiment Design: Experiment 1.

EXP 2 - HTTP Constant Throughput	
Goal	To evaluate how service mesh configurations behave under varying levels of load.
Methodology	The load generator will generate load in a constant and uniform throughput setting. This means that the load generator will produce a set number of requests per second and will not deviate or try to catch up when requests are not processed in time.
Workload	HTTP (Section 5.1.3.1)
Factors	Service mesh configuration (Section 5.1.1) Requests per second
Reasoning	This experiment evaluates the service mesh configurations under similar levels of load. This allows us to compare these configurations to one another when they undergo the same workloads.

Table 5.5: Experiment Design: Experiment 2.

EXP 3 - HTTP Payload Response	
Goal	To evaluate how service mesh configurations behave with varying payload sizes.
Methodology	The load generator will produce HTTP requests to the target service, which in turn will respond with pre-determined payload sizes.
Workload	HTTP (Section 5.1.3.1)
Factors	Service mesh configuration (Section 5.1.1) Payload size
Reasoning	This experiment introduces several pre-determined payload sizes to evaluate the effects of additional application data being transferred in meshed environments. This allows us to study the effects and potential additional overheads extra data transferes may cause.

Table 5.6: Experiment Design: Experiment 3.

EXP 4 - gRPC Maximum Throughput	
Goal	To evaluate how meshed configurations behave with alternative communication protocols.
Methodology	The load generator will generate gRPC traffic towards the gRPC service and do so in an unrestricted manner, meaning it will try to generate as much load as possible without holding back and without enforcing any timeouts. The maximum throughput is determined by looking at the actual number of requests per second in the configuration was able to achieve.
Workload	gRPC (Section 5.1.3.2)
Factors	Service mesh configuration (Section 5.1.1)
Reasoning	This experiment allows us to evaluate the effects of alternative application level protocols. During the system survey we uncovered that service mesh systems inspect and use this data to for both observability and routing functionalities. This experiment allows us to uncover the effects of alternative, but common, application level protocols widely used within the industry.

Table 5.7: Experiment Design: Experiment 4.

Experiment Overview			
Experiment	**Workload**	**Metrics**	**Factors**
EXP 1	HTTP	**M1**, **M2**, **M3**, **M4**	**Mesh Configuration**
EXP 2	HTTP	**M1**, **M2**, **M4**	**Mesh Configuration** **RPS** *(1, 100, 500, 1000)*
EXP 3	HTTP	**M1**, **M2**, **M4**	**Mesh Configuration** **Payload Size** *(0, 1kb, 10kb)*
EXP 4	gRPC	**M1**, **M2**, **M3**, **M4**	**Mesh Configuration**

Table 5.8: Overview of the experiments, their metrics and factors.

5.2.1 Measuring the impact of a multi-node setup

The first micro benchmark that we have conducted compares two experimental environments. In Section 5.1.4, we introduced our experimental environment and discussed the cluster configuration that we have use for our experimental evaluation. We discussed that we designed the experiments to be run on a single node cluster to abolish any variance in network latencies. However, this also implies that the benchmark system is running on the same node and this can have an effect on the results of the experiments.

To evaluate the impact of such a cluster configuration we constructed a micro benchmark. In this micro benchmark, we perform an experiment similar to the design of our first experiment, in which we try to find the maximum throughput of our experimental environment. In this experiment, however, we only perform it on an unmeshed (baseline) configuration to simplify and limit any variance a service mesh system can impose and run it for a total duration of 15 minutes. We compare two experimental environments, in which we change the cluster configuration. The first environment has a single node cluster which runs the benchmark system and the target services. The second environment has a two node cluster, one of these nodes runs the benchmark system whereas the other node runs the target service.

The results of the micro benchmark can be seen in Figure 5.1. The environment using a single node reached an average throughput of 22851 requests per second over the duration of the 15-minute experiment. The environment using two nodes, however, only managed to reach 20520 requests per second. Although the benchmarking system in the two node environment has more system resources to work with, it was unable to reach the throughput levels of the single node environment. Therefore, we can conclude that the experimental

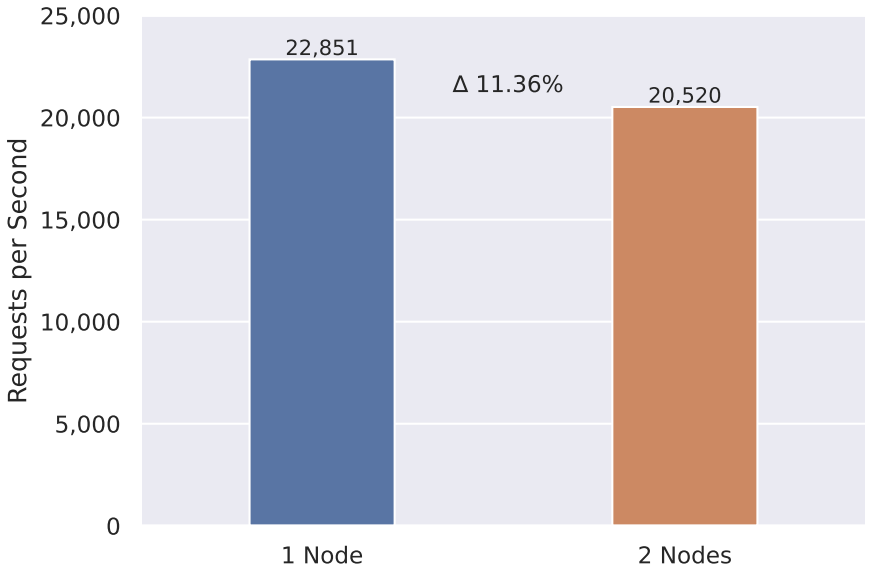


Figure 5.1: Comparing the average maximum throughput of two cluster configurations by using an HTTP-based load test in an environment without a service mesh.

environment using one node does not cause any system resource related bottlenecks and that our proposed design using one node is a valid approach for our experiments.

5.2.2 Measuring the impact of the impact of our target service

The second micro benchmark that we conduct tries to establish if the target services can be the cause of a bottleneck during our experimental evaluation. As discussed in our analysis of the System Under Test (see Section 4.2), we measure the data path towards our service to evaluate the overheads of a service mesh system and its proxy architecture. However, to evaluate this, we need to emulate a software service that acts as a purely functional component in our benchmark. In our prototype implementation of *Mesh Bench*, we use a simple HTTP and gRPC service (see Section 4.5).

To determine if the target service implementation can be the cause of a bottleneck, we used a different workload generator than the one we use for our benchmarking system. Although *fortio* has a flexible feature set and is highly configurable, *wrk2* (as discussed as alternative in Section 4.5) was the most performant workload generator we have evaluated and can generate more load than the other systems mentioned. During our micro benchmark, we generate several levels of load beyond that of our designed experiments, to make sure that the target service is not a bottleneck in any of the results.

In Figure 5.2 we present the results of the micro benchmark. This chart shows the requested throughput of the workload generator on the y-axis and shows the actual throughput

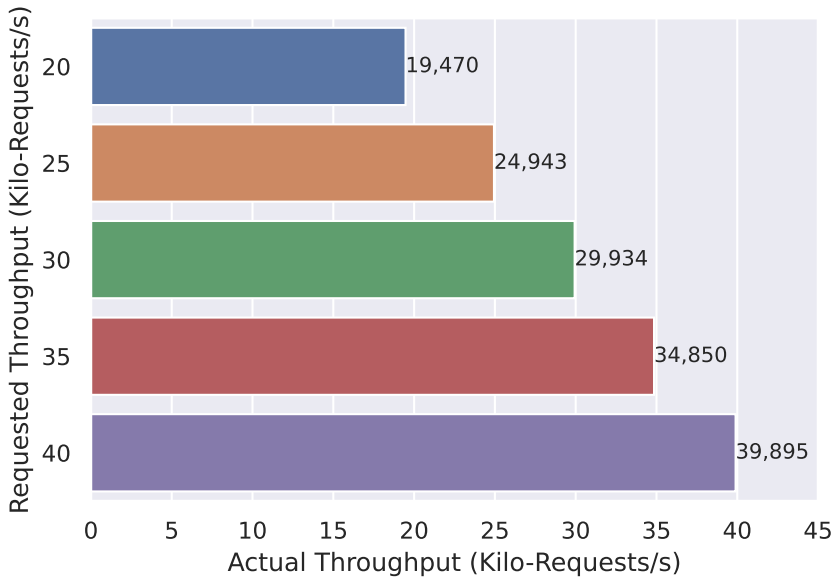


Figure 5.2: Measuring the maximum sustained level of throughput for the target service by performing HTTP-based load tests in an environment without a service mesh.

of successful requests on the x-axis. From these results, we can derive that the target services had no problems handling the levels of throughput generated as the actual throughput levels are very close to the requested levels. Note that these levels of throughput exceed the maximum throughput of the experiment results as discussed in the next section, which shows that the target service as functional component in our benchmark system is not the cause of any bottlenecks.

5.3 Experiment Results

In this section, we present the results obtained from the experiments as defined in Section 5.1.6. In Table 5.9 we present an overview of the main findings and their associated experiments. The remainder of this section presents the results of each experiment in detail and provide an extensive analysis on them.

5.3.1 EXP 1 - HTTP Maximum Throughput

The first experiment evaluates the service mesh systems when they are fully satiated, i.e. they are at full capacity and handling the maximum amount of load that they can process. This amount of load, or throughput, is measured by the number of requests per second the system can process.

Main Findings			
MF1	Using a service mesh can lead to a significant decrease in sustained throughput.	**EXP 1**	Section 5.3.1
MF2	There is significant variance in the amount of sustained throughput that service mesh systems can handle.	**EXP 1**	Section 5.3.1
MF3	The network latency overhead caused by the proxy of Cilium is lowest among all evaluated service mesh systems.	**EXP 1**	Section 5.3.1
MF4	The latencies observed from Istio under load have a large spread, and are worse at the tail end.	**EXP 1**	Section 5.3.1
MF5	Traefik mesh performs an order of magnitude worse than any other evaluated service mesh in terms of request latencies when the system is under full load.	**EXP 1**	Section 5.3.1
MF6	Traefik experiences bottlenecking behaviour under a load of approximately 500 requests per second resulting in a bimodal distribution of request latencies.	**EXP 2**	Section 5.3.2
MF7	There can be a significant difference in the amount of CPU utilization for different service mesh systems under similar levels of load.	**EXP 2**	Section 5.3.2
MF8	The memory footprint of data plane proxies is negligible and does not significantly increase under higher levels of throughput.	**EXP 2**	Section 5.3.2
MF9	The size of the application payload does not have a significant effect on the performance of data-plane proxies on resource utilization levels.	**EXP 3**	Section 5.3.3
MF10	The configuration using Traefik was unable to process any gRPC based requests.	**EXP 4**	Section 5.3.4
MF11	Both gRPC and HTTP-based workloads experience similar reductions in terms of sustained throughput.	**EXP 4**	Section 5.3.4

Table 5.9: Main findings of the experimental evaluation.

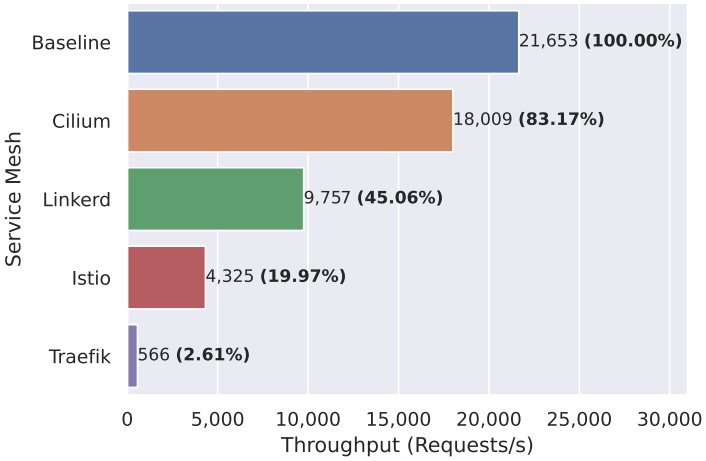


Figure 5.3: Average throughputs of service mesh systems under maximum load.

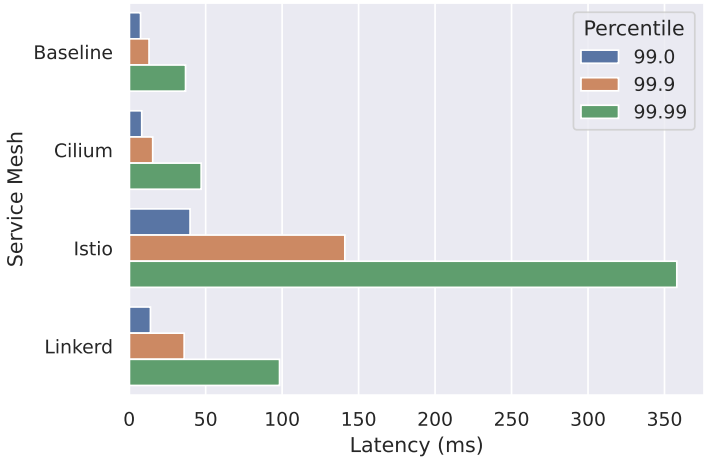


Figure 5.4: Tail end latencies of service mesh systems under maximum load.

5.3.1.1 Maximum Sustained Throughput Analysis

In Figure 5.3 we present a bar chart which depicts the average, sustained throughput that a service mesh system was able to process throughout the duration of the experiment. On the y-axis we present the different service mesh configurations, whereas the x-axis represents the throughput in requests per second. Next to each bar we present the actual observed value and the percentage of throughput a system was able to process compared to the best performing configuration, which in this case is the baseline.

From these results, we can derive that all the service mesh systems experience a loss of throughput compared to the baseline configuration. We can relate this observation to the fact that the service mesh systems introduce additional components in the form of proxies in the critical data path of requests. Every request has to be processed by these proxies, which in turn leads to the decrease in maximum sustained throughput. However, even through the decrease in throughput is expected, the amount of this decrease is rather significant. Cilium is the best performing service mesh configuration in this experiment in terms of throughput. However, it still experienced a massive 16.83% reduction compared to the baseline configuration. This observation leads to our first main finding:

MF1: Using a service mesh can lead to a significant decrease in sustained throughput.

Another observation we can make from these results is the amount of variance there is among the evaluated service mesh systems. We already established that Cilium is the best performing service mesh system in terms of throughput, even though it had a significant reduction compared to the baseline configuration. This reduction in throughput, however, is

little compared to the performance of other configurations. The configuration using Linkerd led to a 54.94% reduction and Istio saw a staggering 80.09% reduction in throughput. The worst performing configuration was using Traefik. This configuration only managed to serve a tiny fraction of the requests and experienced a massive 97.39% reduction in throughput compared to the baseline configuration. From these observations, we can conclude that there is a large variance in the observed throughputs among service mesh systems, where the reductions in throughput range from 16.83% to 97.39%. This leads to our second main finding:

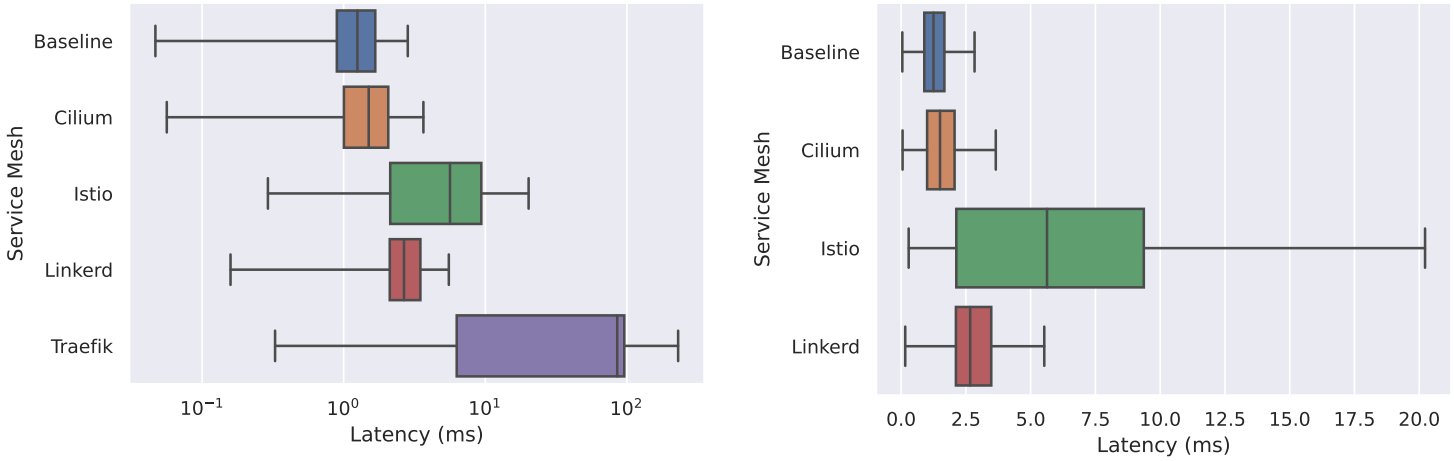
MF2: There is significant variance in the amount of sustained throughput that service mesh systems can handle.

5.3.1.2 Latency Analysis

In our previous analysis we looked at the amount of requests a configuration can handle. In this part of the analysis we take a look at the durations of the requests themselves. We measure the latency of each request. These latencies represent the time it takes for a request to complete. This is measured from the moment the workload generator sends a request until it successfully receives a response from the target service.

To fully understand the impact that the values and distributions in this analysis can have on real-world application performance, we have to refer to the manner in which applications are often constructed. When using a service-oriented architecture, and namely one using the granularity of microservices, you create an application of many interconnected services. Applications can consist of thousands of microservices (3, 27) and a user can indirectly use hundreds of backing services. To evaluate latencies in such a model, we follow the practices as described in the works of Gil Tene (58). Because of this model, the tail end latencies are more common than one might expect due to the nature of probabilities. As an example, the likelihood of experiencing the 99.995th percentile of latencies is greater than 99% when a procedure involves 200 microservices.

In Figure 5.5 we present a pair of box and whiskers plots that depict the locality, spread and skewness of latencies per service mesh configuration. The lines at the borders of the box represent the 25th and 75h percentiles of data. The line in the middle of the box depicts the median observed value and the whiskers of the box represent the minimum and maximum values. It is important to note that the outliers are omitted from these figures, as we cover these in greater detail later on. The plots depicted in the figure share the meaning of their axes, where the y-axis represents the service mesh configuration and the x-axis represent the observed latencies in milliseconds. The first plot (Figure 5.5a), shows all the service mesh configurations and displays the box plots on a logarithmic scale. The



(a) Logarithmic scale

(b) Linear scale, excluding Traefik

Figure 5.5: Latency distributions of service mesh systems under maximum load.

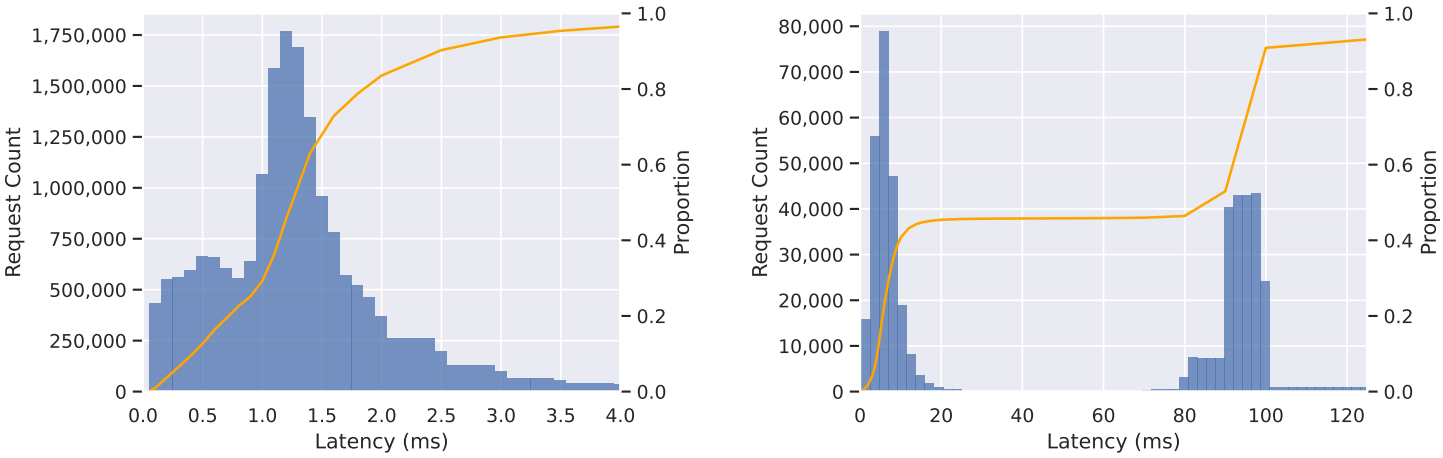
second plot (Figure 5.5b), displays the data on a linear scale but excludes Traefik.

In Figure 5.5a we present the observed latencies on a logarithmic scale. We do this because it shows that the observed latencies for Traefik surpass the other configurations by an order of magnitude. In Figure 5.5b we compare the service mesh configurations on a linear scale and exclude Traefik. From these results, we can observe that the distribution of observed latencies for Cilium is very similar to that of the baseline configuration. This indicates that the network overhead caused by the in-kernel proxy of Cilium is kept to a minimum which leads us to the third main finding:

MF3: The network latency overhead caused by the proxy of Cilium is lowest among all evaluated service mesh systems.

Another observation we can make is that Linkerd has a slightly higher median and spread, but is still performing relatively well as it does not deviate too much from the baseline configuration. Istio, on the other hand, appears to suffer significantly more. Not only is the median affected, the spread of observed latencies is significantly greater. We can relate this behaviour to the type of proxy used in their data planes and their design decisions (see Section 3.4 Table 3.4). Whereas Istio uses *Envoy*, a complex and feature rich general purpose proxy, Linkerd uses its own lightweight proxy (53).

In Figure 5.4 we present the tail end of latencies observed (above the 99th percentile) for the service mesh configurations excluding Traefik. On the y-axis we represent the latencies observed in millisecond whereas the x-axis represents the various service mesh configurations. The colours of the bars represent a certain percentile of latencies observed.



(a) Baseline

(b) Traefik

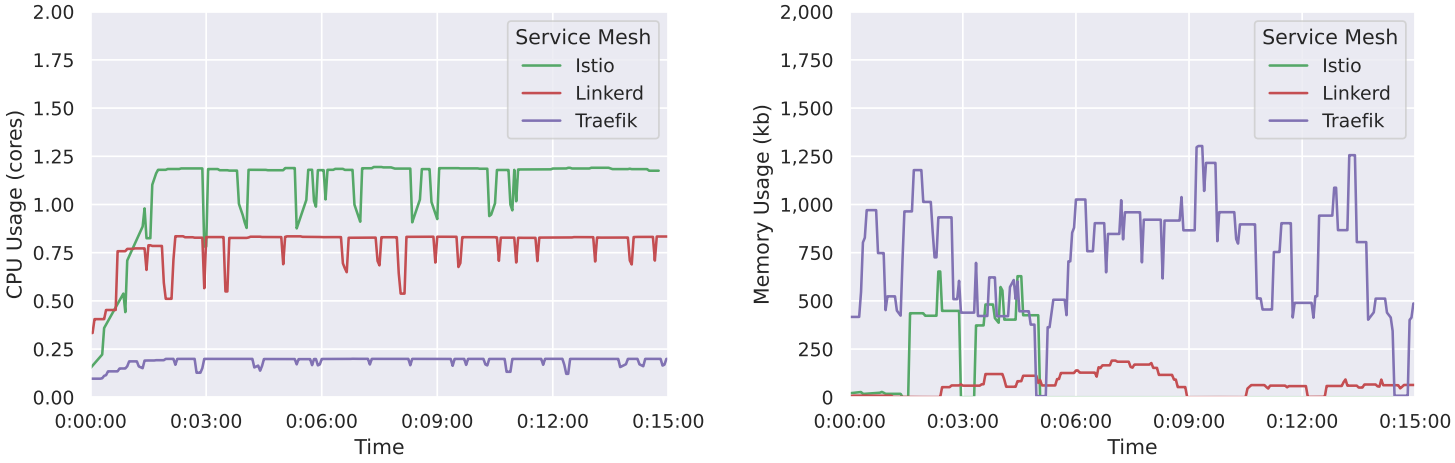
Figure 5.6: Histogram of observed latencies under maximum load per service mesh configuration.

From the chart in Figure 5.4 we can observe that Istio suffers the most in the observed tail end latencies. We observed a latency values of 40, 141 and 358 milliseconds for the 99th, 99.9th and 99.99th percentiles respectively. This observation is a continuation from the behaviour depicted in the box and whiskers plot (Figure 5.5b), where the results tied to the configuration using Istio also depicted a large spread in observed latency values. This leads to a fourth main finding:

MF4: The latencies observed from Istio under load have a large spread, and are worse at the tail end.

To evaluate the behaviour of Traefik in more detail, we depict the latency distributions in a histogram. In Figure 5.6 we depict two plots where each plot contains a histogram of latency distributions. The first plot contains the histogram of the baseline configuration (Figure 5.6a) and the second plot contains the histogram of Traefik (Figure 5.6b). Each histogram has a y-axis that represents the number of requests for a specific bin and an x-axis that represents the latency in milliseconds. In addition to the bins, we depict the cumulative density function that shows the proportion of requests smaller than the latency depicted on the x-axis.

From the results depicted in Figure 5.6, we can observe the distribution of latencies for the baseline configuration and that of Traefik. Note that the values next to the y-axis depict different values, there are more latencies observed for the baseline configuration, as the throughput was not fixed in this experiment. The shape of the distribution for



(a) CPU Utilization

(b) Memory Utilization

Figure 5.7: Resource utilization for service mesh systems under load.

the baseline configuration is similar to that of other configurations except for Traefik (see Appendix for a full comparison). From the shape of the histogram distribution of Traefik we can derive a peculiar finding, it has a bimodal distribution. The first mode is close to 8ms, whereas the second mode is around the 90 milliseconds. These observations lead to our fifth finding of the experiment:

MF5: Traefik mesh performs an order of magnitude worse than any other evaluated service mesh in terms of request latencies when the system is under full load.

5.3.1.3 Resource Utilization Analysis

The final type of analysis that we perform is related to the utilization of system resources. In Figure 5.7, we present two line graphs that depict resource usage over time. The first plot (Figure 5.7a) is related to the CPU utilization of service mesh proxies. The y-axis represents the fraction of CPU cores used for the service mesh proxy (on a two core system), and the x-axis represents the time delta of the experiment. The second plot depicts Figure 5.7b the memory utilization. For this, plot the y-axis represents the memory utilization in kilobytes, whereas the x-axis once again represents the time delta of the experiment. In these plots, we display three out of four service mesh systems. This is because we were only able to capture the user-level application containers related to Cilium, and not the kernel-level proxy. For this reason, we omit Cilium from the resource graphs, as the data gathered was inconclusive for that system.

The results as depicted in Figure 5.7 show the various configurations under varying levels

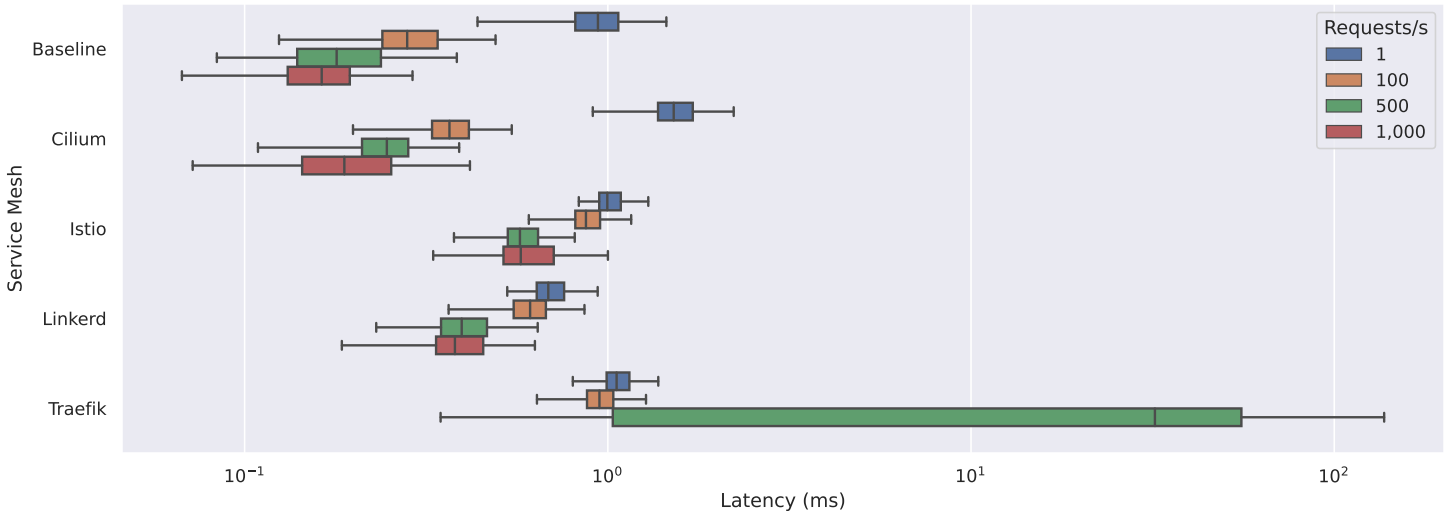


Figure 5.8: Distribution of observed latencies per service mesh system under various levels of constant throughput on a logarithmic scale.

of load. This means that the comparison cannot be considered fair. With that said, we can still observe some interesting behaviour. In Figure 5.7a we depict the CPU utilization for the three observed systems. We can see that the proxy of Istio is the largest consumer of the CPU, even though as previously observed, it had a significantly fewer requests to process compared to Linkerd. This can be related to the aforementioned design decisions of the proxy implementations (53). Another thing to note is that Traefik, the worst performing service mesh system consumes the least amount of CPU resources. This at the very least proves that the poor performance is not related to any CPU related bottlenecks.

From the results depicting memory utilization (Figure 5.7a) we can derive that the memory utilization for the data plane proxies is very minimal, even under maximum load. The highest values observed as depicted by the spikes in the graph, are less than 1500Kb and pose no significant bottleneck for most systems and environments.

5.3.2 EXP 2 - HTTP Constant Throughput

In the previous experiment (Section 5.3.1), we evaluated service mesh systems under maximum load. In this experiment, we evaluate these systems under varying, pre-defined levels of constant throughput. The results of these experiments aim to show how the service mesh systems scale, across varying levels of load.

5.3.2.1 Latency Analysis

In Figure 5.8 we present the latency distributions of the evaluated service mesh configurations under various levels of constant throughput. On the y-axis we present the service mesh configurations and on the x-axis the latency expressed in milliseconds on a logarithmic scale. The legend on the plots display the various levels of throughput depicted on this graph and the colours that represent them.

From Figure 5.8 we can derive that the latency distributions are generally close for each of the defined levels of throughput for most of the systems. This is an indication that the various levels of throughput are manageable by the evaluated configurations. This is exemplified by the average levels of throughput that these configurations were able to process as observed in our previous experiment in Section 5.3.1.

One exception to this, however, is that the configuration using Traefik is experiencing a significant increase in observed latencies when experiencing a constant load of 500 requests per second. Additionally, it is important to note that the results regarding 1000 requests per second have not been included in this figure, as it was unable to manage that level of throughput. As a matter of fact, it was unable to even fully sustain the constant throughput of 500 requests per second, as it only managed to process 419 requests per second in this particular experiment. This result falls in line with the observed behaviour from our previously conducted experiment, in which we evaluated these systems under maximum load. The reason that the observed throughput is even lower in this experiment compared to the previous one, is that the previous experiment allowed for dynamic levels of throughput. This experiment on the other hand, uses constant levels of throughput as generated by the workload generator. The workload generator, however, does not make up or increase the level of throughput to compensate if the system was unable to previously process the results in time.

Additionally, we analysed the tail end latencies of the evaluated configurations under these levels of constant throughput. The results that depict these high percentile latencies can be found in the Appendix (Figure A.2). However, at these levels of load, we did not observe any increase in high percentile latencies as we previously did in our first experiment. The values we observed for the constant levels of throughput align with the expected values of a normal distribution.

5.3.2.2 Traefik Bottleneck Analysis

To provide a more extensive analysis into the observed behaviour of Traefik, we visualized the latency distributions of the configuration using Traefik in violin plot as depicted in Figure 5.9. The y-axis of this plot represents the various levels of throughput, and the x-axis present the observed latency values in milliseconds. It is important to note that we

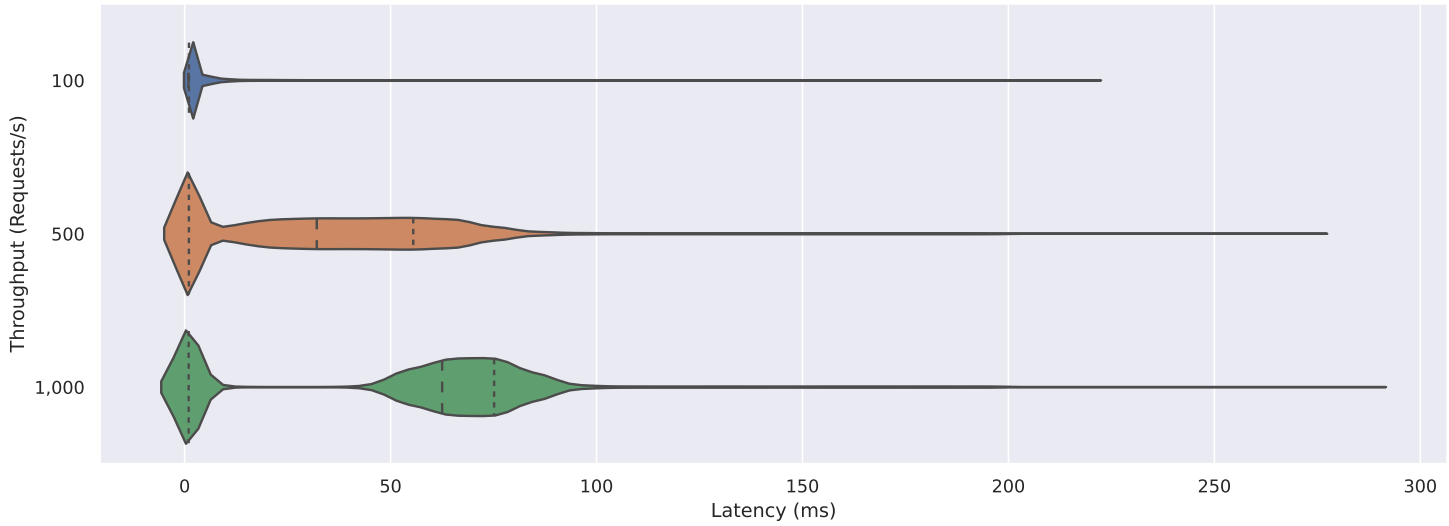
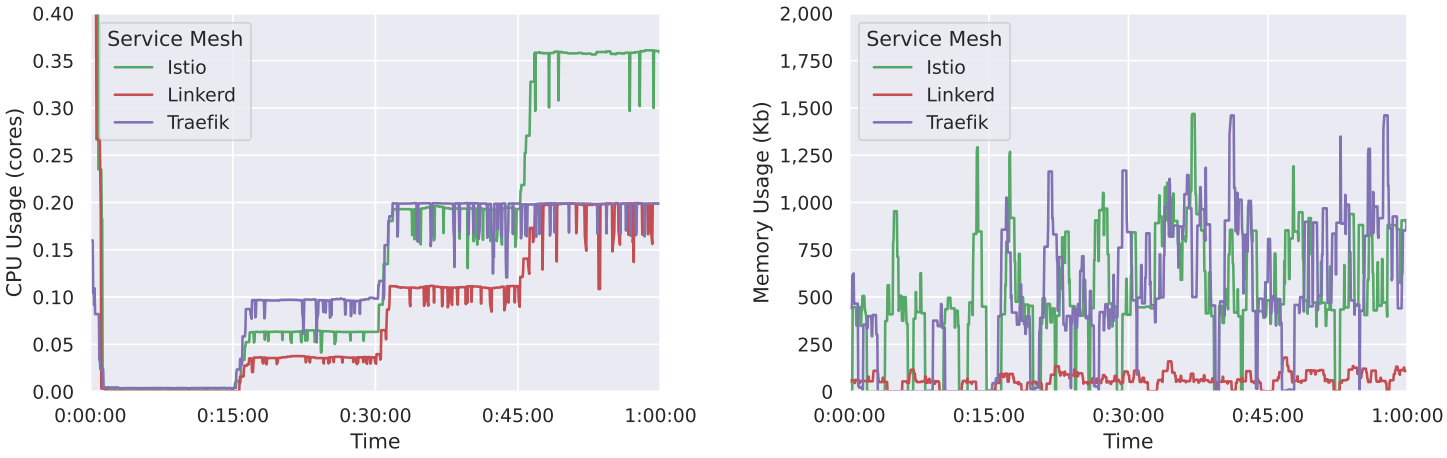


Figure 5.9: Latency distribution of Traefik under varying levels of constant throughput.

did include the results of when we evaluated Traefik under a constant throughput of 1000 requests per second, even if it did not manage to actually sustain this level of throughput as previously discussed.

Through the violin plots as depicted in Figure 5.9, we can observe the behaviour and bottlenecks of the system under load in more detail. The first violin plot shows the latency distribution when it is under a constant load of 100 requests per second. From this, we can derive that the system can process this amount and that the distribution of latencies is similar to that of other evaluated service mesh systems as previously seen. The second violin plot, however, shows the system under a constant load of 500 requests per second. This plot shows the first signals of a potential bottleneck in the system. We can observe that the observed latency values have a higher spread and that these values are often an order of magnitude higher, often reaching values well above 50 milliseconds. This provides a sharp contrast with the other systems that we evaluated, which performed significantly better under full or near full load. The third violin plot, however, shows the bottleneck in full effect. This plot depicts the system under a constant throughput of 1000 requests per second, which it was unable to process. This results in the previously observed bimodal distribution of requests latencies. This leads us to our sixth main finding:

MF6: Traefik experiences bottlenecking behaviour under a load of approximately 500 requests per second resulting in a bimodal distribution of request latencies.



(a) CPU Utilization

(b) Memory Utilization

Figure 5.10: Resource utilization for service mesh systems experiencing various levels of constant throughput. The level of throughput is increased at a 15-minute interval, and changes from 1, 100, 500, and 1000 requests per second respectively.

5.3.2.3 Resource Utilization Analysis

In our first experiment we already took a look at the resource utilization values of various service mesh systems under maximum load. However, those results could not be compared fairly, as they each experienced various levels of load. In this experiment, however, we can evaluate the resource utilization fairly as they are evaluated under similar circumstances.

In Figure 5.10 we present two plots related to the resource consumption of service mesh systems. Both plots can be interpreted similarly. On the y-axis we have the value of interest, which is the CPU utilization as expressed in fractions of a core for the first plot and is the memory utilization expressed in kilobytes (Kb) for the second plot. The x-axis presents the time delta of the experiment since the start. It is important to note the labels on the x-axis as these 15-minute intervals represent the time point in which the level of constant throughput was increased. The four intervals represent the constant levels of throughput from 1, 100, 500, and 1000 requests per second respectively (e.g. the interval from 0:30:00 to 0:45:00 represents the systems under a constant load of 500 requests per second). Additionally, the colours of the line represent the various service mesh systems that we evaluated. As previously noted, we are unable to evaluate the actual resource utilization of Cilium as the actual in-kernel proxy is not exposed as a user-space application in the form of a container.

The first plot (Figure 5.10a), shows the CPU utilization throughout the experiment for the various service mesh systems. The first thing we can observe the high values at the start of the experiment, these can be explained by the sampling rate used for the time series

data which overlaps with the results from the previous experiment. The second thing we can notice is that the line from Traefik does not increase at the 45:00 minute marker. This behaviour falls in line with the previously discussed bottlenecks in our extensive analysis of the bottlenecks of Traefik. Another observation that we can make is that the CPU utilization is relatively stable for all the evaluated systems throughout the duration of the experiment, we can derive this from the lack of extremes and spikes in the presented plot.

More importantly, however, we can observe how the systems scale when the load on these systems is increased. By looking at the first 15-minute interval, we can observe that the proxies require little to no CPU utilization when they experience next to zero load (1 request per second). Interestingly however, is the behaviour when the load is increased to 100 requests per second as it allows us to fairly compare these systems under a similar load. We can observe that the Traefik proxy utilizes the CPU the most, whereas the proxy in the Istio data plane requires just slightly more than half of Traefik's usage. The Linkerd proxy, however, puts the least amount of strain on the CPU requiring less than 5% of a CPU core at any given time. To analyse how these systems scale we take a closer look at the third and final time intervals. From this, we can observe that the CPU utilization scales linearly with the levels of throughput that a proxy is facing. We can also observe that the proxy empowering the Linkerd service mesh scales significantly better than the one found in Istio. This is exemplified by the fact that Istio under a load of 500 requests per second uses nearly identical amount of CPU resources when Linkerd serves double the number of requests per second. This leads us to a seventh main finding:

MF7: There can be a significant difference in the amount of CPU utilization for different service mesh systems under similar levels of load.

The second plot (Figure 5.10b), shows the CPU utilization throughout the experiment for the various service mesh systems. The first observation that we can make is that the amount of memory consumed for all systems does not seem to vary much across the various levels of throughput. Furthermore, we can observe that the linkerd proxy consumes the least amount of memory whereas Traefik and Istio's data plane proxy seem to consume similar amounts. More importantly, however, is that all the observed values are negligible for most environments. This brings us to our eighth main finding:

MF8: The memory footprint of data plane proxies is negligible and does not significantly increase under higher levels of throughput.

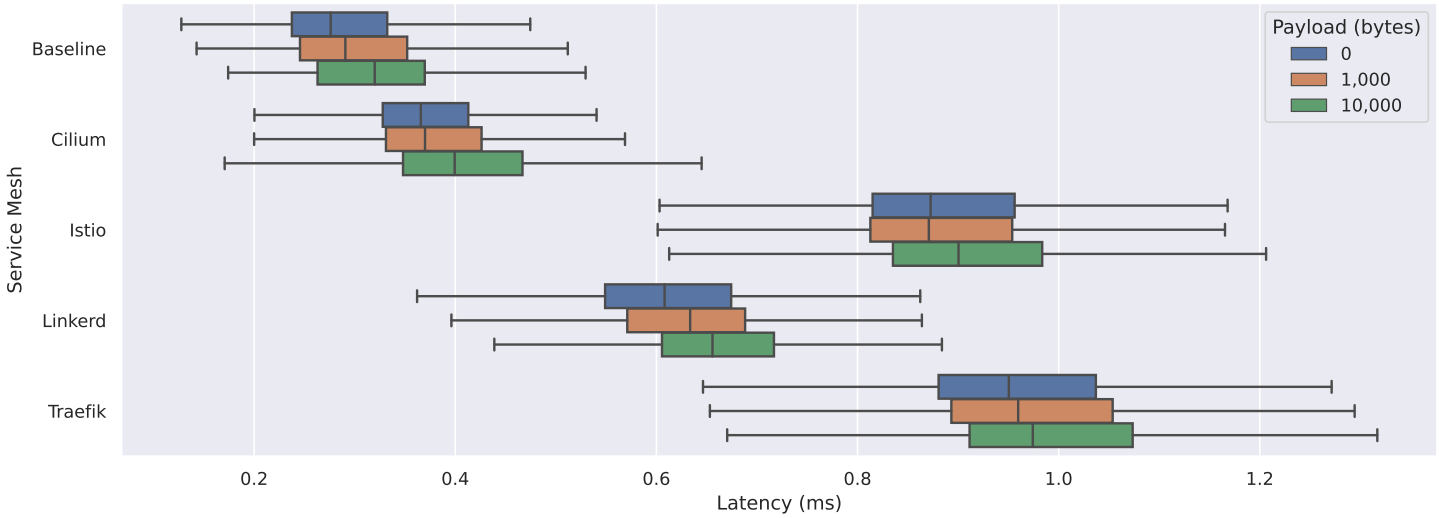


Figure 5.11: Distribution of observed latencies per service mesh system with varying application payload sizes.

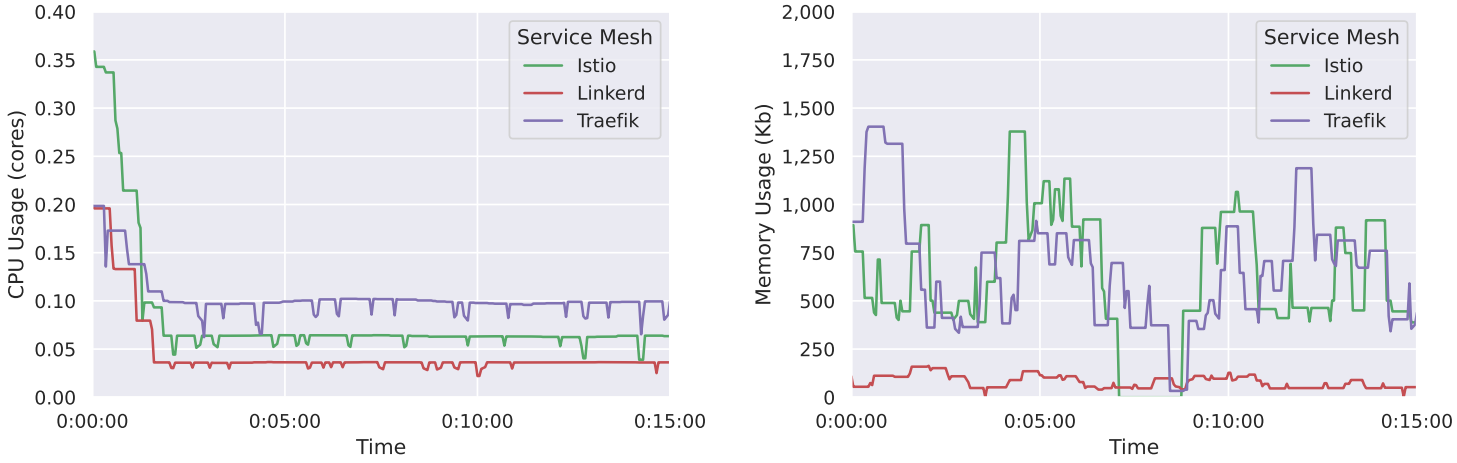
5.3.3 EXP 3 - HTTP Payload

In the third experiment we introduced variable application payloads. The goal of this experiment is to analyse the service mesh systems when they have to process various amounts of data as application payloads. During this experiment, we generate a constant throughput of 100 requests per second, however, the application payload of each of the HTTP responses vary between 0, 1000, and 10000 bytes in size.

In Figure 5.11 we present the observed latency distributions of the various service mesh configurations whilst processing varying levels of application payloads. On the x-axis we present the varying configurations, whilst the y-axis represents the observed latency values in milliseconds. The colours of the bars as indicated by the legend, represent the varying levels of application payload sizes encountered.

From the distributions as presented in Figure 5.11 we can observe that the observed latencies are slightly higher of the requests that returned a larger application payload. However, the difference is very minimal, and the selected payload sizes did not seem to impact the overall observed performance. Furthermore, the application payload size also did not seem to affect the tail end latencies, as can be seen in the comparison presented in the Appendix in which we compare the 99th, 99.9th and 99.00th percentile of latencies observed (Figure A.2).

In Figure 5.12 we present two plots related to the resource consumption of service mesh systems. These plots can be interpreted similarly, where on the y-axis we present the observed values that either represents the fraction of CPU cores used or the amount of



(a) CPU Utilization

(b) Memory Utilization

Figure 5.12: Resource utilization for service mesh systems experiencing varying application payload sizes.

memory consumed for a given data plane proxy. The x-axis once again represents the time delta of the experiment, which in total takes 15 minutes. The colours of the lines represent the various service mesh systems which we evaluate.

We observe no significant differences for both of the plots presented in Figure 5.12. The CPU utilization is stable, aside from the initial spiked caused by the manner in which the time series data is aggregated and sampled. Furthermore, both CPU utilization and memory consumption levels for the evaluated systems conform to their expected values. This leads us to our singular conclusion from this experiment:

MF9: The size of the application payload does not have a significant effect on the performance of data-plane proxies on resource utilization levels.

5.3.4 EXP 4 - gRPC Maximum Throughput

In the final experiment we utilize a different type of application workload. In the previous experiments we evaluated the different configurations and service mesh systems using various HTTP workloads. In this experiment, we use a different application level protocol to evaluate how the layer-7 aware data plane proxies perform with alternative layer-7 application protocols.

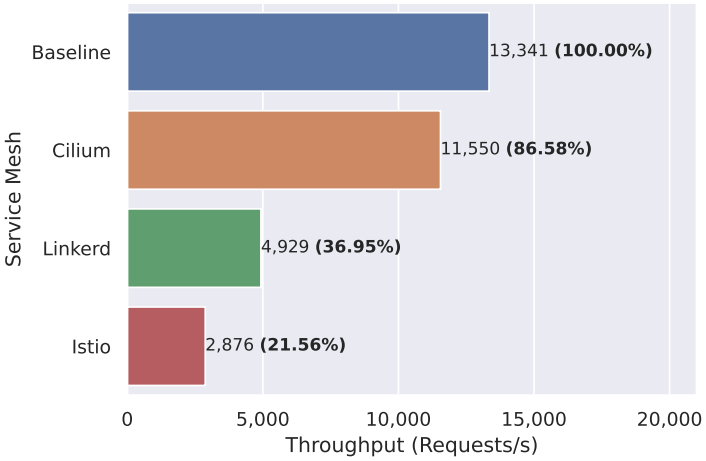


Figure 5.13: Average throughput of service mesh systems under maximum load using the gRPC protocol.

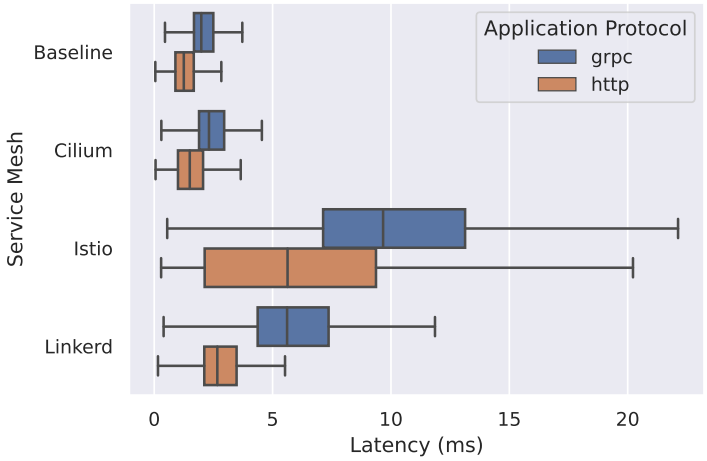


Figure 5.14: Comparing the latency distributions of service mesh systems under maximum load for both HTTP and gRPC workloads.

5.3.4.1 Maximum Sustained Throughput Analysis

In Figure 5.13 we present a bar chart which depicts the average throughput a service mesh system was able to process throughout the 15-minute duration of the experiment. During this experiment, the workload generator produced as many requests as the receiving system was able to process and did so in an unrestricted manner. The results are presented in the chart which can be interpreted as follows. On the y-axis we present the different service mesh configurations. The x-axis presents the aggregated average throughput value that a configuration was able to process, expressed in requests per second. The values next to the bars represent the actual observed value and the percentage next to it represents the percentage of throughput a system was able to process compared to the best performing configuration, which in this case is the baseline.

The very first thing to notice in Figure 5.13 is the absence of Traefik in the depicted chart. This is because the proxy in Traefik was unable to process any gRPC requests and any attempt in doing so resulted in errors even though historical articles produced by the vendor explicitly state the support for the gRPC protocol (59). This leads to our tenth main finding:

MF10: The configuration using Traefik was unable to process any gRPC based requests.

Another observation we can make is regarding the levels of throughput a service mesh system can process compared to the baseline configuration. Similarly to the first experiment, in which we evaluated the maximum sustained throughput whilst using HTTP-based

requests and responses (Section 5.3.1), we observe that there is a significant reduction of throughput for each of the evaluated service mesh systems. The best performing service mesh system is once again, Cilium, experiencing a 13.42% reduction in throughput compared to the baseline configuration. Linkerd is the second best performing system, which experiences a 63.05% reduction. The greatest reduction of throughput, however, is reserved for the Istio which experiences a massive 78.44% reduction in throughput compared to the baseline.

When we compare these results to the results from the first experiment we can observe a similar trend. We observe the same ranking whilst evaluating this metric. Additionally, the relative differences when comparing the reductions per protocol are minimal. Most of the evaluated systems experience similar reductions for both application level protocols. The largest difference observed is for Linkerd, which encountered a 54.94% reduction for the HTTP workloads whilst it encountered a 63.05% reduction for the gRPC based workloads. On average, there was a 4.39% difference between the levels of reduction experienced for the types of workloads used. This brings us to our eleventh main finding:

MF11: Both gRPC and HTTP-based workloads experience similar reductions in terms of sustained throughput.

5.3.4.2 Latency Analysis

In Figure 5.14 we present the latency distributions of the evaluated configurations under maximum load for both the gRPC and HTTP-based workload experiments. On the y-axis we present the evaluated configurations, excluding Traefik. On the x-axis we present the latency in milliseconds. The colours of the bar indicate the type of workload used. The blue bars (gRPC) present the data from the fourth experiment, whereas the orange bars (HTTP) present the observed latency values from the first experiment.

The first observation that we can make is that the observed latency values are slightly higher for the gRPC based workloads across all the evaluated configurations. Furthermore, we can observe that the spread of latency values for Linkerd is larger when using a gRPC-based workload, compared to the HTTP-based counterpart. Although there are some differences, we can conclude that there are no significant differences in the latency distributions based on the application protocol.

In Figure 5.15 take a closer look at the observed tail end latencies for both types of application level protocols used. On the y-axis we present the evaluated service mesh configurations and on the x-axis we present the latency in milliseconds. Within the plots we depict coloured bars, in which the colours represent the 99th, 99,9th, and 99.99th percentile respectively. The plot on the left displays the results for the gRPC-based workload whereas

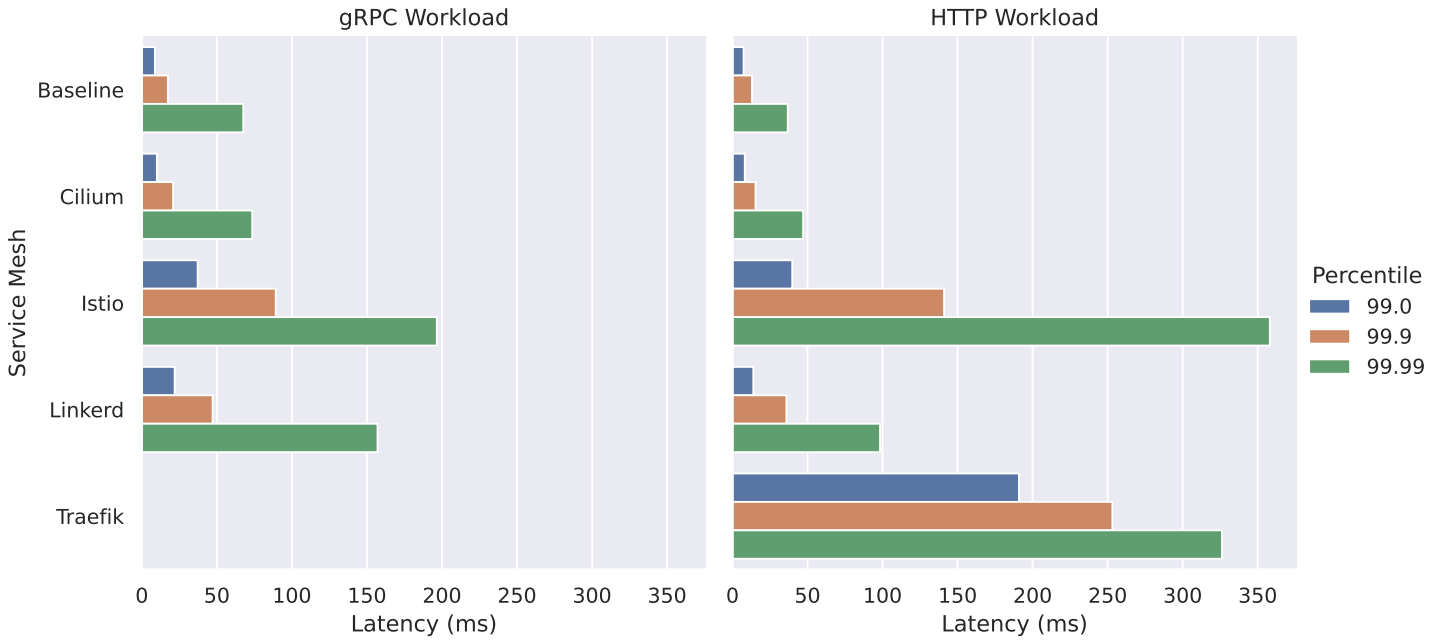


Figure 5.15: Tail end latencies of service mesh systems experiencing maximum load per application protocol.

the plot on the right presents the result for the HTTP-based workload.

From the plots as depicted in Figure 5.15 we can observe a couple of interesting findings. First, we observe that the tail end latencies, specifically the 99.99th percentile are negatively affected whilst using the gRPC based protocol for the baseline configuration and the configurations using Cilium and Linkerd. Interestingly however, is that the tail end latencies of Istio improved significantly when exposed to gRPC-based workloads compared to the HTTP counterpart. This could be related to the manner in which the data plane proxy, Envoy, processes gRPC based requests as they treat is as a first class citizen at both the transport and application layer[1].

5.4 Threats to Validity

In this section, we present identified potential threats to validity that we encountered during the experimental evaluation.

1. **Lack of representative, real-world infrastructure**

   The first threat to validity is the lack of a real-world environment to evaluate our experiments in. In Section 5.1.4 we presented our experimental environment and

[1] `https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/other_protocols/grpc`

discussed the reasons for choosing a single node cluster. We later performed a micro benchmark to validate that the single node cluster was not a cause for any bottlenecks in our experimental set-up (Section 5.2.1). However, this type of environment does not emulate a real-world environment as it does not reflect the environments of the target audience that uses Kubernetes and service mesh systems. Such environments would include multi-node high-availability clusters and even multi-cluster environments.

2. **Absence of real-world experiments**

   Another potential threat to validity is based on the lack of real-world experiments. In our implementation of *Mesh Bench* we use two simplistic, synthetic target services (Section 4.5.2). These target services accept workloads based on the two most common service-to-service communication protocols, HTTP and gRPC. However, the target services perform trivial, synthetic computations on incoming requests. Furthermore, they are isolated and do not constitute a full-sized service-oriented architecture in which they rely on, or utilize other services. Even though the synthetic experiments and workloads aim to minimize the variance in observed results, they do not emulate a real-world experiment such as a real-world application built using a microservices architecture.

3. **Absence of Resource Utilization Results for Cilium**

   During the implementation of *Mesh Bench*, we made several design decisions to capture the resource utilization at the granularity of Kubernetes related resources (Section 4.5.3). The implementation details seemed to fit our design and related objectives, and during early evaluations we validated the design. However, during the experimental analysis, we uncovered that the containers related to Cilium were reporting abnormally low resource utilization values. Upon further inspection, we found out that the reported values only captured the configuration of the related data plane proxies, and not the actual processing of data. Since it relies on in-kernel proxying programs powered by eBPF, we were unable to capture it with our metric collection, and monitoring system.

4. **Accuracy of reported tail latencies**

   Another potential threat to validity comes from the reported values of the workload generator. In the implementation of *Mesh Bench*, we utilized fortio as workload generator as discussed in Section 4.5.1. The workload generator is responsible for generating workloads and reporting on the observed latencies, status codes and errors that it observed for the underlying requests. The workload driver, however, aggregates the results during execution of an experiment. This prevents it from having to report detailed metrics of millions of requests per experiment. To properly

aggregate the data, it aggregates the request metrics in predefined bins. These bins are based on the expected latency per request, which in our experimental environment was relatively stable, especially in constant throughput experiments. However, this aggregated approach comes at the cost of accuracy, especially at the outliers in which the actual latencies are further from the expected latency. To prevent this problem from impacting our analysis we performed experiments with a duration long enough to gather significant data and validated the results of the tail-end latencies with other attempts of the same 15-minute duration experiment.

5.5 Summary

To summarize and conclude this chapter, we return to the initial research question (**RQ3**), *What are the differences between current service mesh systems in terms of overhead, throughput and latency?* Based on the design and prototype implementation of *Mesh Bench*, we designed several performance related experiments that provide an answer to our research question (Section 5.1). First, we establish an experimental environment, which aims to minimize variance from external, unrelated variables to improve the reproducibility of experiments. We then conducted several micro benchmarks to validate this (Section 5.2). After this, we conducted the performance related experiments that evaluate service mesh systems under various conditions and performed an extensive analysis Section 5.3. During our analysis we discuss the significant impact a service mesh system can have in terms of resource overheads, reductions in throughput and increase in (tail) latency to provide a detailed answer to our research question. We heavily emphasize on the costs associated with service mesh systems and clearly show how various systems compare to one another, and to an environment without a service mesh present. Furthermore, we show promising future directions based on the relatively new, in-kernel, eBPF-based approach of networking as seen with Cilium, the service mesh that outperformed all other systems in all of our experiments. Finally, in Section 5.4 we discuss potential threats to the validity of the experimental evaluation.

6

Conclusion and Future Work

The service mesh architecture is an emerging approach to address several challenges presented in distributed systems. Through a dedicated layer of networking infrastructure it manages to improve the infrastructure in four areas of interest; observability, reliability, security, and programmability. In the latter half of the past decade, many service mesh systems have surfaced, and have reached serious levels of adoption. Although performance is a crucial aspect for many, little is known about the performance implications of these systems. In this work, we address this shortcoming by taking an experimental approach to the emerging type of system, the service mesh.

6.1 Conclusion

In this work, we address three main research questions related to the performance evaluation of service mesh systems. The approach used to answer these research questions follows the vision of MCS, in which we heavily emphasize the design, implementation, deployment, analysis, and benchmarking of distributed systems. This approach divided our research on service mesh systems in three major chapters that each aim to answer a main research question. In Chapter 3 we performed an extensive system survey, to analyse the state-of-the-art service mesh systems. Then, in Chapter 4, we design *Mesh Bench*, a benchmark for service mesh systems based on our previous learnings. Last, in Chapter 5, we take an experimental approach to evaluating service mesh systems, by conducting performance related experiments and performing an extensive analysis on the results.

We now address the main research questions individually, to answer our research questions and conclude our work.

RQ1 How to compare, and evaluate service mesh systems?

We presented an extensive system survey in Chapter 3, in which we identified several state-of-the-art service mesh systems. During this survey, we identified several

domain-specific characteristics and features that these systems had and categorized them under functional and non-functional requirements of a service mesh system. Based on these findings we created a comparison framework, which allowed us to compare various service mesh systems based on these attributes. Furthermore, we identified the primary components that could impact the performance of these systems. We analysed the proxy components that make up the actual mesh, the layer of dedicated infrastructure often referred to as the data plane, and discovered various approaches and architectural designs that could heavily influence the performance of these systems. This analysis laid the groundwork for the work presented in this thesis, and established the direction we take to evaluate the performance of service mesh systems.

RQ2 How to design and implement a benchmark that evaluates the performance of service mesh systems?

Based on the results of our system survey, we designed and implemented a prototype of *Mesh Bench* in Chapter 4. To guide our design process, we established a set of benchmarking objectives that capture the performance related characteristics of distributed systems which are applicable to service mesh systems. We then used an iterative design process guided by a set of established best practices to design our benchmark. We bootstrapped this process by properly defining the System Under Test, which consists of the service mesh components that we aim to evaluate. Following that, we performed an extensive requirements analysis that identifies the stakeholders of such a benchmark system and their concerns and use cases. Based on this information we constructed a detailed design of Mesh Bench, our benchmark system for service mesh systems. We then concluded this chapter of our research with a prototype implementation, in which we discussed each implementation detail and their potential alternatives. The resulting prototype then served as the base for our experimental evaluation.

RQ3 What are the differences between current service mesh systems in terms of overhead, throughput, and latency?

The experimental evaluation as presented in Chapter 5 was set out to uncover the performance related intricacies of service mesh systems and provide an answer to our final main research question. It builds upon the obtained knowledge from the system survey, and used the prototype of Mesh Bench to perform various performance related experiments. We designed several performance related experiments that evaluate service mesh systems under various conditions by changing three primary dimensions. We found that service mesh systems can have a significant impact on maximum sustained throughput, e.g. an 80% reduction in one of the most commonly used

service mesh systems. Additionally, we provided evidence that tail latencies could be massively affected by the network proxies in certain cases, and heavily emphasized the impact tail latencies can have in a service-oriented architecture. Finally, we have shown that there is a huge discrepancy between common service mesh systems in terms of resource utilization, where competing systems can utilize twice the amount of CPU resources under similar levels of load. Finally, in our analysis we relate back to our architectural findings of the system survey, and identify promising future directions for eBPF-based networking solutions.

6.2 Future Work

In this work, we present an experimental approach to analyse the performance of service mesh systems. In a field that has previously received insufficient attention from academia, we present an extensive evaluation on a rapidly changing environment. During this work, however, we were only able to provide an answer to a limited amount of related questions and problems. Additionally, such an emerging field with many ongoing developments poses a wide range of many interesting research questions and points.

Based on the results from our work and the current challenges present in the field, we suggest the following opportunities for future research:

1. **Experiments on real-world infrastructures**

   In the design of our experiments we used a single node cluster as this limited the variance in results from our experiments. A single node set-up excels in such achieving this as it does not rely on networking between nodes in a cluster which could introduce variance. However, such a set-up does not reflect a real-world environment of the target audience of Kubernetes and service mesh systems. An opportunity for future research would include evaluating service mesh systems in multi-node cluster configurations and even multi-cluster environments.

2. **Real-world experiments**

   In our design and implementation of the experiments we used a synthetic workload and a single logical software service. This enabled us to accurately evaluate the performance of the data plane as it limited the number of components in our System Under Test and provided a focus on the critical data path of a network packet. However, a real-world environment would have more logical services and a longer chain of service-to-service communications. Benchmarking service mesh systems in a real-world environment would be an opportunity for future research.

3. **Enhance benchmark implementation to include resource utilization from eBPF programs**

During our experimental evaluation we measured the resource utilization levels of all containers in our Kubernetes cluster. This enabled our resource utilization analysis in which we compared the resource utilization of service mesh data plane proxies under various levels of load. However, in our implementation of the benchmark we were unable to measure the impact of the eBPF-based proxy. An opportunity for future research would be to modify the benchmark and implement a solution that can capture the resource utilization of eBPF programs.

4. **Evaluate the resiliency features of service mesh systems**

   One of the primary advantages of using a service mesh system is the suite of resiliency features such a system can bring. From retrying and delaying network requests to advanced circuit-breaking patterns to dissuade from cascading failures. The design and implementation of our benchmark has a focus on performance and did not include experiments that evaluate the resiliency of service mesh systems. An opportunity for future research would be to design and conduct experiments that evaluate the reliability features of service mesh systems by utilizing fault injection or chaos engineering techniques.

5. **Expand upon the experiment dimensions**

   In the design of our experiments we based the dimensions of the experiments on initial explorative findings. During this explorative phase, we used the service mesh systems that we set out to evaluate. However, as shown in our analysis, we observed large discrepancies in performance related aspects between these systems. We designed the experiments to accommodate all the selected service mesh systems and therefore set our experiment dimensions to values that were mostly reachable by all these systems. The result is that for some experiments, e.g. the constant throughput experiment, we used relatively small levels of constant throughput for the better performing systems. In addition to tweaking existing experiment dimensions, future research could include more dimensions to evaluate service mesh systems in additional environments.

Appendix A

Additional Experimental Results

A. ADDITIONAL EXPERIMENTAL RESULTS

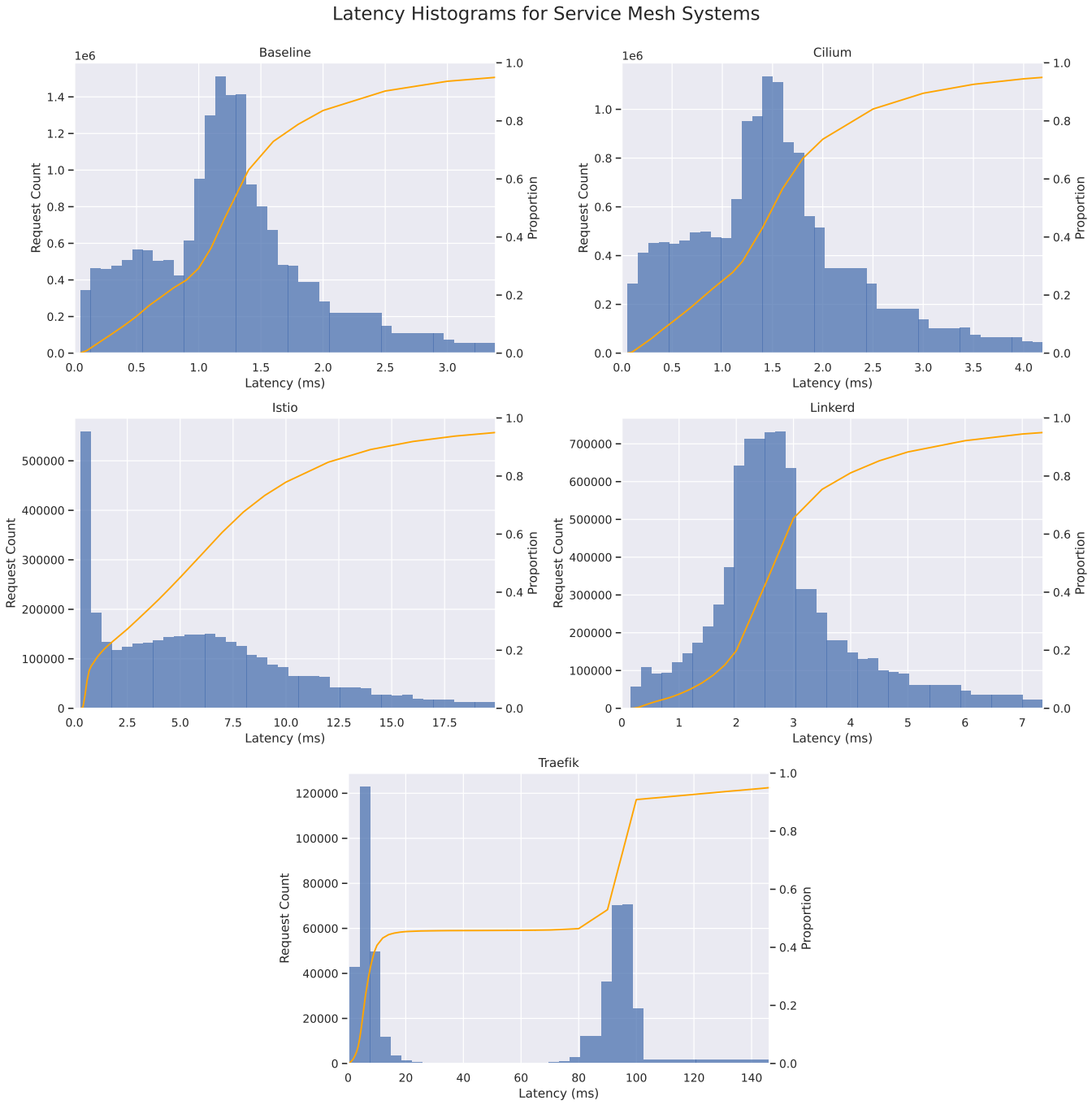


Figure A.1: Histogram of latencies under maximum load

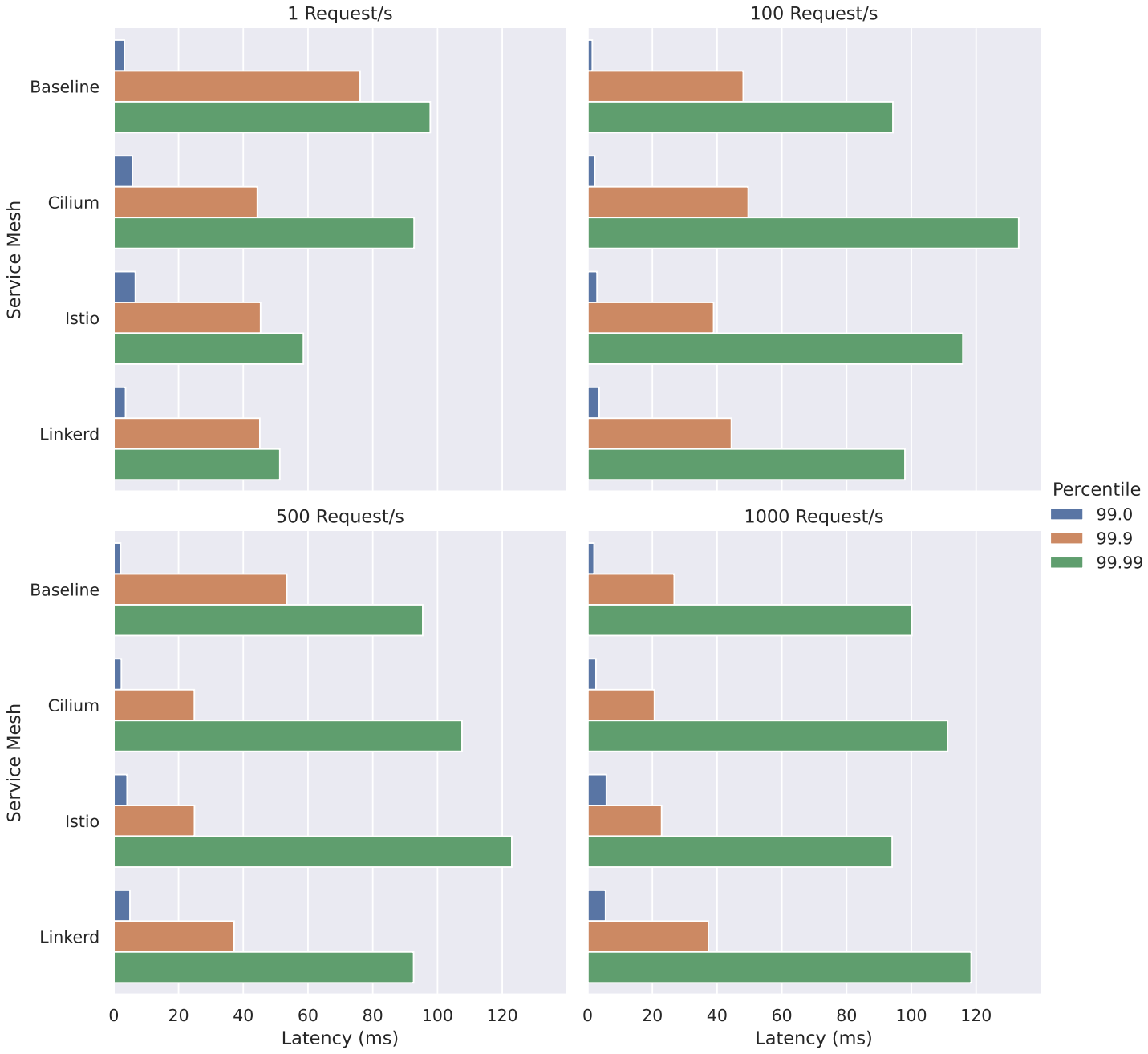


Figure A.2: Tail end latencies of service mesh systems under varying levels of constant throughput.

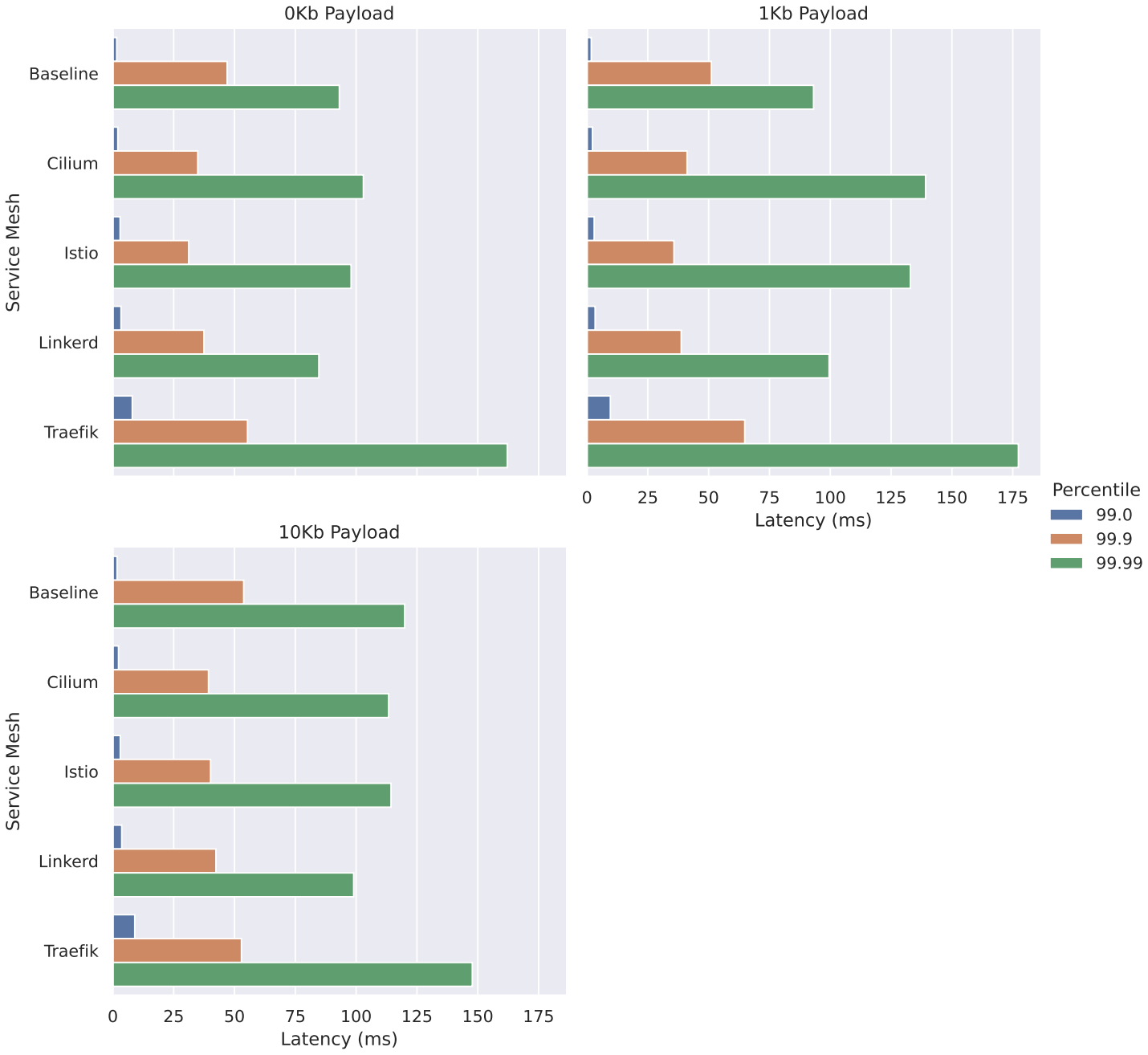


Figure A.3: Tail end latencies of service mesh systems when experiencing varying application payload sizes.

References

[1] Thomas Betts. **To Microservices and Back Again - Why Segment Went Back to a Monolith**. https://www.infoq.com/news/2020/04/microservices-back-again/, 2020. [Online; accessed 23. Jul. 2022]. 1

[2] Patrick Nommensen. **It's Time to Move to a Four-Tier Application Architecture**. https://www.nginx.com/blog/time-to-move-to-a-four-tier-application-architecture/, 2015. [Online; accessed 23. Jul. 2022]. 1

[3] Josh Evans. **Mastering Chaos - A Netflix Guide to Microservices**. https://www.infoq.com/presentations/netflix-chaos-microservices/, 2016. [Online; accessed 23. Jul. 2022]. 1, 14, 84

[4] Andrew Baird, Michael Connar and Patrick Brandt. **Coca-Cola: Running Serverless Applications with Enterprise Requirements**. https://www.youtube.com/watch?v=yErmil0ODYs, 2016. [Online; accessed 23. Jul. 2022]. 1

[5] Mary Pratt. **The 10 biggest issues IT faces today**. https://www.cio.com/article/228199/the-12-biggest-issues-it-faces-today.html, 2022. [Online; accessed 23. Jul. 2022]. 1

[6] Lionel Sujay Vailshery. **Organizations' adoption level of microservices worldwide in 2021**. https://www.statista.com/statistics/1233937/microservices-adoption-level-organization/, 2022. [Online; accessed 23. Jul. 2022]. 1

[7] Alexandru Iosup, Alexandru Uta, Laurens Versluis, Georgios Andreadis, Erwin van Eyk, Tim Hegeman, Sacheendra Talluri, Vincent van Beek, and Lucian Toader. **Massivizing Computer Systems: a Vision to Understand, Design, and Engineer Computer Ecosystems through and beyond Modern Distributed Systems**. *CoRR*, **abs/1802.05465**, 2018. 2, 3, 6

[8] Justina Alexandra Sava. **Full-time employment in the information and communication technology (ICT) industry worldwide in 2019, 2020 and 2023**. https://www.statista.com/statistics/1126677/it-employment-worldwide/, 2022. [Online; accessed 23. Jul. 2022]. 2

[9] Yoav Einav. **Amazon Found Every 100ms of Latency Cost them 1% in Sales**. https://www.gigaspaces.com/blog/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/, 2019. [Online; accessed 23. Jul. 2022]. 2

[10] Jake Brutlag. **Speed Matters**. https://ai.googleblog.com/2009/06/speed-matters.html, 2009. [Online; accessed 23. Jul. 2022]. 2

[11] Cloud Native Computing Foundation. **Cloud Native Survey 2020**. https://www.cncf.io/reports/cncf-annual-survey-2021/, 2021. [Online; accessed 23. Jul. 2022]. 3, 15, 39, 47, 59

[12] Kebe Liu, Xiaopeng Han and Hui Li. **Merbridge - Accelerate your mesh with eBPF**. https://istio.io/latest/blog/2022/merbridge/, 2022. [Online; accessed 23. Jul. 2022]. 3, 28, 48

[13] Cilium. **Cilium Service Mesh Beta**. https://cilium.io/blog/2021/12/01/cilium-service-mesh-beta/, 2021. [Online; accessed 23. Jul. 2022]. 3, 28, 48, 51

[14] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. **Benchmarking in the cloud: What it should, can, and cannot be**. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 173–188. Springer, 2012. 6, 54, 55

[15] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. **Microservices architecture enables devops: Migration to a cloud-native architecture**. *Ieee Software*, **33**(3):42–52, 2016. 9

[16] Lightstep. **Global Microservices Trends Report**. https://go.lightstep.com/global-microservices-trends-report-2018.html, 2018. [Online; accessed 23. Jul. 2022]. 9

[17] Marcelo Amaral, Jorda Polo, David Carrera, Iqbal Mohomed, Merve Unuvar, and Malgorzata Steinder. **Performance evaluation of microservices architectures using containers**. In *2015 IEEE 14th International Symposium on Network Computing and Applications*, pages 27–34. IEEE, 2015. 9, 13

[18] Solomon Hykes. **The future of Linux Containers**. https://pyvideo.org/pycon-us-2013/the-future-of-linux-containers.html, 2013. [Online; accessed 23. Jul. 2022]. 10

[19] Stack Overflow. **2021 Developer Survey**. https://insights.stackoverflow.com/survey/2021, 2021. [Online; accessed 23. Jul. 2022]. 10

[20] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. **Microservices: yesterday, today, and tomorrow**. *Present and ulterior software engineering*, pages 195–216, 2017. 11

[21] Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. **From monolith to microservices: a classification of refactoring approaches**. In *International Workshop on Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 128–141. Springer, 2018. 11

[22] Dirk Merkel et al. **Docker: lightweight linux containers for consistent development and deployment**. *Linux journal*, **2014**(239):2, 2014. 12

[23] Randall Perrey and Mark Lycett. **Service-oriented architecture**. In *2003 Symposium on Applications and the Internet Workshops, 2003. Proceedings.*, pages 116–119. IEEE, 2003. 12

[24] IBM. **SOA (Service-Oriented Architecture)**. https://www.ibm.com/nl-en/cloud/learn/soa, 2019. [Online; accessed 23. Jul. 2022]. 12

[25] Martin Fowler. **Microservices**. https://martinfowler.com/articles/microservices.html, 2014. [Online; accessed 23. Jul. 2022]. 13

[26] IBM. **SOA vs. Microservices: What's the Difference?** https://www.ibm.com/cloud/blog/soa-vs-microservices, 2021. [Online; accessed 23. Jul. 2022]. 13

[27] CloudZero. **Netflix Architecture: How Much Does Netflix's AWS Cost?** https://www.cloudzero.com/blog/netflix-aws, 2021. [Online; accessed 23. Jul. 2022]. 14, 84

[28] Zaigham Mahmood. **Service oriented architecture: potential benefits and challenges**. In *Proceedings of the 11th WSEAS International Conference on COMPUTERS*, pages 497–501. Citeseer, 2007. 14

[29] Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch*. Addison-Wesley, 2012. 14

[30] William Morgan. **https://thenewstack.io/history-service-mesh/**. https://thenewstack.io/history-service-mesh/, 2018. [Online; accessed 23. Jul. 2022]. 14

[31] Louis Ryan. **gRPC Motivation and Design Principles**. https://grpc.io/blog/principles/, 2015. [Online; accessed 23. Jul. 2022]. 14

[32] Netflix Technology Blog. **Introducing Hystrix for Resilience Engineering**. https://netflixtechblog.com/introducing-hystrix-for-resilience-engineering-13531c1ab362, 2012. [Online; accessed 23. Jul. 2022]. 14

[33] Twitter Engineering. **Finagle: A Protocol-Agnostic RPC System**. https://blog.twitter.com/engineering/en_us/a/2011/finagle-a-protocol-agnostic-rpc-system, 2011. [Online; accessed 23. Jul. 2022]. 14

[34] Kubernetes. **Kubernetes 1.0 Launch Event at OSCON**. https://kubernetes.io/blog/2015/07/kubernetes-10-launch-party-at-oscon/, 2015. [Online; accessed 23. Jul. 2022]. 16, 34

[35] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. **Borg, omega, and kubernetes**. *Communications of the ACM*, **59**(5):50–57, 2016. 16

[36] Karl Johan Åström and Richard M Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2021. 16

[37] Kirsten Morris and W Levine. **Control of systems governed by partial differential equations**. *The control theory handbook*, 2010. 16

[38] Kirsten Jacobs. **Container Networking From Scratch**. https://www.youtube.com/watch?v=6v_BDHIgOY8, 2018. [Online; accessed 23. Jul. 2022]. 16

[39] William Morgan. **The Service Mesh**. https://buoyant.io/service-mesh-manifesto, 2022. [Online; accessed 23. Jul. 2022]. 18, 28

[40] Falko Menge. **Enterprise service bus**. In *Free and open source software conference*, **2**, pages 1–6, 2007. 21

[41] Sam Newman. *Building Microservices, 2nd edition*. O'Reilly, 2022. 21

[42] Cloud Native Computing Foundation. **Cloud Native Computing Foundation ("CNCF") Charter**. https://github.com/cncf/foundation/blob/

c9caca27b50ee1d9315d6a134c2266b9c5f9bf72/charter.md, 2022. [Online; accessed 23. Jul. 2022]. 23

[43] Wubin Li, Yves Lemieux, Jing Gao, Zhuofeng Zhao, and Yanbo Han. **Service Mesh: Challenges, State of the Art, and Future Research Opportunities**. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, pages 122–1225, 2019. 24, 28, 31, 33

[44] Thilo Fromm. **Performance Benchmark Analysis of Istio and Linkerd**. https://kinvolk.io/blog/2019/05/performance-benchmark-analysis-of-istio-and-linkerd/, 2019. [Online; accessed 23. Jul. 2022]. 25

[45] William Morgan. **Benchmarking Linkerd and Istio: 2021 Redux**. https://linkerd.io/2021/11/29/linkerd-vs-istio-benchmarks-2021/, 2021. [Online; accessed 23. Jul. 2022]. 25

[46] Red Hat. **https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview**. https://www.redhat.com/en/resources/kubernetes-adoption-security-market-trends-overview, 2022. [Online; accessed 23. Jul. 2022]. 28

[47] Cloud Native Computing Foundation. **Cloud Native Survey 2020**. https://www.cncf.io/reports/cloud-native-survey-2020/, 2020. [Online; accessed 23. Jul. 2022]. 28

[48] B. Kitchenham and S. Charters. **Guidelines for performing systematic literature reviews in software engineering**. Technical report, Citeseer, 2007. 30

[49] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. **Systematic Mapping Studies in Software Engineering**. *12th International Conference on Evaluation and Assessment in Software Engineering, EASE 2008*, 6 2008. 31

[50] Kai Petersen, Sairam Vakkalanka, and Ludwik Kuzniarz. **Guidelines for conducting systematic mapping studies in software engineering: An update**. *Information and Software Technology*, **64**:1–18, 8 2015. 31

[51] Vahid Garousi, Michael Felderer, and Mika V. Mäntylä. **Guidelines for including grey literature and conducting multivocal literature reviews in software engineering**. *Information and Software Technology*, **106**:101–121, 2 2019. 31, 32

[52] Cloud Native Computing Foundation. **CNCF Graduation Criteria v1.3**. https://github.com/cncf/toc/blob/c936591a9c17eac011b890387fcd4b8a653184b0/process/graduation_criteria.md, 2022. [Online; accessed 23. Jul. 2022]. 39

[53] William Morgan. **Why Linkerd doesn't use Envoy**. https://linkerd.io/2020/12/03/why-linkerd-doesnt-use-envoy/, 2020. [Online; accessed 23. Jul. 2022]. 47, 85, 88

[54] Gobind Johar. **New GKE Dataplane V2 increases security and visibility for containers**. https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine, 2020. [Online; accessed 23. Jul. 2022]. 48

[55] NGINX. **NGINX Service Mesh Architecture**. https://docs.nginx.com/nginx-service-mesh/about/architecture/#nginx-service-mesh-sidecar, 2022. [Online; accessed 23. Jul. 2022]. 48

[56] Cloudflare. **Zero Trust security | What is a Zero Trust network?** https://www.cloudflare.com/learning/security/glossary/what-is-zero-trust/, 2022. [Online; accessed 23. Jul. 2022]. 51

[57] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: How Google runs production systems*. " O'Reilly Media, Inc.", 2016. 54, 59

[58] Gil Tene. **How NOT to Measure Latency**. https://www.infoq.com/presentations/latency-response-time/, 2015. [Online; accessed 23. Jul. 2022]. 84

[59] Traefik Labs. **Maesh, a Simpler Service Maesh.** https://traefik.io/resources/maesh-a-simpler-service-maesh-presented-by-the-traefik-team/, 2019. [Online; accessed 23. Jul. 2022]. 95