HOCHSCHULE
HANNOVER
UNIVERSITY OF
APPLIED SCIENCES
AND ARTS
–
Fakultät IV
Wirtschaft und
Informatik

Requirements and Impact Analysis of Applying a Service Mesh to a Distributed off-the Shelf Software System

Marc Herschel

Master Thesis in Applied Computer Science

December 1, 2023

Author Marc Herschel
marc@herschel.io
166337

First examiner Prof. Dr.-Ing. Arne Koschel
Department of Computer Science, Faculty IV
Hochschule Hannover
arne.koschel@hs-hannover.de

Second examiner Prof. Dr.-Ing. Holger Peine
Department of Computer Science, Faculty IV
Hochschule Hannover
holger.peine@hs-hannover.de

Statement of Authorship

I hereby declare that I have written this submitted Master's thesis independently and without external help and that I have not used any sources other than those indicated by me and that I have marked any passages taken verbatim or in terms of content from the works cited.

Hannover, December 1, 2023 Signature

A service mesh is a dedicated infrastructure layer that facilitates secure, reliable communication between services in a distributed system. HAFAS is a distributed off-the-shelf timetable information system maintained by Hacon, a Siemens Mobility subsidiary. Introducing a service mesh to HAFAS is a base scenario for a call of research to understand better the requirements and impacts of a service mesh on distributed systems. This thesis examines introducing a service mesh to a distributed software system and investigates the impact on the example of HAFAS. Scenarios for HAFAS that profit from the features of a service mesh were identified and mapped to functional requirements. The service mesh technological landscape was investigated by defining additional non-functional requirements and identifying a project satisfying the requirements. A subset of prioritized scenarios emerged after investigating the previous scenarios more closely in the context of the identified service mesh project. By introducing the identified service mesh project and the scenario subset to a HAFAS environment as a case study, valuable data on the feasibility of the operation has arisen. The resulting data allowed the investigation of value added by the service mesh and impacts on latency and resource consumption. Data analysis showed that introducing a service mesh has an acceptable impact on latency, memory, and CPU resources. The benefits gained through mTLS encryption improve the system's overall security by introducing the security properties of confidentiality, integrity, and authenticity to HAFAS's data in transit. The results of this thesis help organizations that plan on adopting a service mesh to their technology stack.

Keywords: *Service Mesh*, *Cloud-native Computing*, *Distributed Systems*, *IT Security*

Since the beginning of March 2021, the author of this thesis has been employed at Hacon, a Siemens subsidiary, as a working student in the department of hosting and operations. During this time, he worked as a DevOps engineer and was exposed to one of Hacon's main products, HAFAS. HAFAS is a timetable information system that public transportation agencies use. During this time, Fabian Korte, a cloud architect mentoring the author at Hacon, mentioned service mesh technology. Hacon has yet to investigate the introduction of service mesh technology to its technology stack, but the topic has sparked interest in software architectural meetings at the company. Naturally interested in this theme, the idea of investigating the requirements of a service mesh and its impacts on HAFAS as a master thesis topic was born.

I want to acknowledge and thank everyone at Hacon, specifically Fabian Korte and Jan Fricke, for assisting me in carrying out this research and gaining insight into the company and HAFAS while providing me with the necessary resources to conduct my practical case study.

Table of Contents

List of Abbreviations

API ..... Application Programming Interface
AWS ..... Amazon Web Services
CA ..... Certificate Authority
CaaS ..... Container as a Service
CI/CD ..... Continuous Integration/Continuous Delivery
CIDR ..... Classless Inter-Domain Routing
CLI ..... Command-line Interface
CNCF ..... Cloud Native Computing Foundation
CRD ..... Custom Resource Definition
CSR ..... Certificate Signing Request
CSV ..... Comma-separated values
CVE ..... Common Vulnerabilities and Exposures
DNS ..... Domain Name System
EC2 ..... Amazon Elastic Compute Cloud
EKS ..... Elastic Kubernetes Service
GUI ..... Graphical User Interface
HAFAS ..... Hacon Timetable Information System
HCI ..... HAFAS Client Interface
HDC ..... Hafas Data Cockpit
HIM ..... Hafas Information Manager
HTTP ..... Hypertext Transfer Protocol
IaaS ..... Infrastructure as a Service
IaC ..... Infrastructure as Code
JSON ..... JavaScript Object Notation
MaaS ..... Mobility as a Service
mTLS ..... mutual Transport Layer Security
OSI ..... Open Systems Interconnection
OSS ..... Open Source Software
PaaS ..... Platform as a Service
PDF ..... Portable Document Format
RBAC ..... Role-based Access Control
RPS ..... Request Per Second
SaaS ..... Software as a Service
SLA ..... Service Level Agreement
SMI ..... Service Mesh Interface
TLS ..... Transport Layer Security
XML ..... Extensible Markup Language

List of Figures

List of Tables

List of Listings

1 Introduction

The hosting world has significantly evolved over the last two decades[Kra18]. While local data centers primarily operated in the past, a rethink of this approach and the outsourcing of services to the cloud has occurred. Initially, this was mainly done with the help of managed virtual machines such as *Amazon EC2*[1] and now increasingly through containerization with the help of elastic platforms with open-source projects such as *Kubernetes*[2]. As the complexity of distributed software architectures grows, the need for fine-grained control over the internal communication of distributed systems arises. Service Mesh[LLG+19] technology is one approach to filling this gap. A service mesh is a dedicated infrastructure layer that facilitates communication between services in a distributed system and operates in Kubernetes clusters. The *Hacon Ingenieurgesellschaft mbH*[3], a transportation, traffic and logistic oriented software company and member of *Siemens Mobility*[4] develops and distributes *HAFAS*[Wik][Hace] (*HaCon Fahrplan-Auskunfts-System*), a timetable information system with international customers such as *Deutsche Bahn*[5], *Zürcher Verkehrsverbund*[6], *Rejseplanen*[7] and *Bay Area Rapid Transport*[8]. Hacon sells HAFAS to customers as an off-the-shelf distributed software system following a modular design principle. Since HAFAS is a software system that emerged over thirty years ago, with the first version released at the end of the 1980s, a continuing modernization and migration process occurs at Hacon. An increasing number of customers' HAFAS systems are hosted in the cloud on virtual machines, and a migration towards Kubernetes is already taking place. Since service mesh projects operate with Kubernetes clusters, this migration step paves the way for introducing service mesh technology to HAFAS environments. The motivation of this thesis is thus to use this practical scenario as a call to research the requirements of a service mesh and the impact of introducing it to an already existing distributed software system in a real-world scenario. Part of this thesis involves conducting a practical case study demonstrating that introducing a service mesh to HAFAS environments is possible. With promising results, adding service mesh technology to Hacon's technology stack is a possible outcome for the future.

Motivation

As service mesh technology has been mainly industry-driven, scientific discourse is arriving. The paper *Service Mesh: Challenges, State of the Art, and Future Research Opportunities*[LLG+19, p. 1-2] from 2019 provides an initial scientific entry point to the topic and states:

> *"To the best of our knowledge, this is the first paper to systematically review the literature of service mesh and the key technologies behind the theme. Given the current state of the research on service mesh and the cloud-native evolution, we believe our effort is just-in-time. We expect that it can attract*

[1] `https://aws.amazon.com/ec2/`
[2] `https://kubernetes.io/`
[3] `https://www.hacon.de/`
[4] `https://www.mobility.siemens.com/global/de.html`
[5] `https://www.bahn.de/service/mobile/db-navigator/`
[6] `https://www.zvv.ch/zvv/de/home.html`
[7] `https://rejseplanen.dk/webapp/`
[8] `https://www.bart.gov/planner`

> *valuable research interest and inspire more practical research work in this exploited area."*

As practical research is deemed to be sparse, this thesis connects to the paper mentioned above. For this purpose, various existing service mesh projects are compared and evaluated according to their intended use cases and capabilities. It will then be outlined which opportunities and risks exist for an organization such as Hacon when adapting a service mesh and how it can be introduced to an already existing distributed system most sensibly. In order to study this in a practical scenario, the applicability of a service mesh to HAFAS will be examined and conducted as a case study. The knowledge and experience gained from this case study will then help evaluate aspects of introducing a service mesh to an already existing distributed system in practice. The findings uncovered while conducting this case study might partly be transferred to other distributed systems with comparable complexity and architecture. Additionally, an attempt is made to determine whether a service mesh's benefits outweigh the overhead. The results of this thesis aid in better understanding what applying a service mesh to an already existing distributed system entails and how a proof of concept can be realized and verified.

Structure

Chapter 2 presents the necessary concepts concerning cloud computing models and container orchestration through Kubernetes. Additionally, on service meshes, the reader is introduced to service mesh technology, its key concepts, features, challenges, and the current technological landscape. Afterward, more details on HAFAS are supplied before moving on to Chapter 3, which introduces the methodology applied in this thesis. In-depth information concerning the current state of HAFAS is provided to the reader, and a desired state involving service mesh technology is defined. Lastly, a research assignment is defined based on the desired state, research questions, and non-goals. With Chapter 4, the requirements for service mesh projects are defined, resulting in a selection process to find a service mesh suitable for the practical case study of this thesis that satisfies the previously defined requirements sufficiently. Once completed, a variety of scenarios based on practical features provided by the selected service mesh project are investigated before defining the foundation of the case study. Subsequently, a subset of scenarios investigated in the case study context is chosen. With Chapter 5, the HAFAS environment for the case study is configured in a Kubernetes cluster, the selected service mesh project, Linkerd, is then introduced to HAFAS, and the selected subset of scenarios is validated. Lastly, load tests targeting the possible latency and resource overhead introduced by the service mesh of the case study are conducted, which will be analyzed in Chapter 6. Finally, this work's research assignment and findings are reviewed before heading to the outlook in Chapter 7.

Typographic Conventions

Abbreviations are defined at the first occurrence in the format of *Long Abbreviation* (LA). In the following text, only the abbreviation is used. All abbreviations used in this thesis can also be found in the list of abbreviations at the beginning of the thesis. Emphasis on, e.g., technical terms, questions, or foreign language words is highlighted using this *italic* font. This `True Type` font is used for highlighting file paths, IP addresses, CRDs, and shell commands. Referencing all instances of defined

identifiers throughout the document, such as for example, questions in the format *Q1: abc, Q2: abc, Q3: abc, Q4: abc* is shortened by using the notion of *Q1..Q4*.

Appendix

Since this work deals with computer science, and thus digital artifacts exist, there is a physical and an *Electronic Appendix*. The physical Appendix represents the final section of this thesis and contains further information about the *Electronic Appendix*, which is included in this thesis as a DVD copy.

2 Fundamentals

This section will deal with the fundamental knowledge required for the following Chapters. Some context for the current cloud computing landscape development is given to understand better the ongoing migration towards Kubernetes at Hacon and the motivation for service mesh technology, which is elaborated further on in Chapter 3.2. Some basics of Kubernetes and an overview of service mesh terminology and features are provided. Finally, an introduction to the capabilities of HAFAS is given to understand the domain the software system aims to serve.

2.1 Paradigm Shift in Cloud Computing

The constant evolution of technology can be observed in all aspects of computing. As cloud computing is changing and has evolved from a past where companies hosted their infrastructure on-premise, managing all their local hardware and operations by themselves, this approach is getting more and more unusual as new cloud computing models emerge. Figure 1 depicts an overview of different cloud computing models. Cloud computing models such as *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS), and *Software as a Service* (SaaS) exist next to on-premise which offer less organizational overhead due to external providers managing the colored parts in Figure 1.

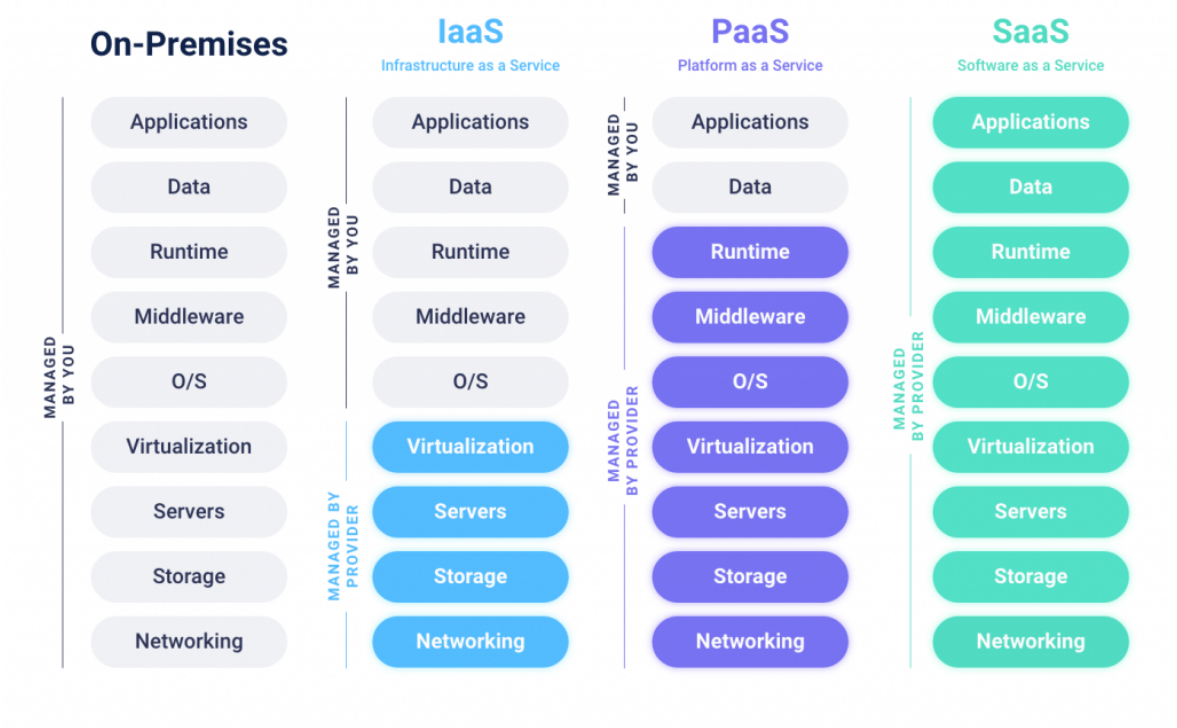

Figure 1: Overview of cloud computing models and their level of autonomy[1]

The difference between IaaS, PaaS, and SaaS lies in their level of autonomy and operational overhead outsourced to an external company. IaaS[2] provides a virtualization platform; the user still has to set up an operating system, a runtime environment,

[1] [Elb21]
[2] Examples for IaaS: AWS, Microsoft Azure, Digital Ocean, and Google Compute Engine

and the application they want to host themselves. PaaS[3] outsources the runtime environment and operating system to an external provider. Users must bundle up their software for deployment and handle their data themselves. SaaS[4] provides no autonomy anymore and delivers software over the internet; an external provider manages the software and all operational aspects of it. In this case, the user becomes an end user who uses a product managed for them. Operational overhead minimizes to near zero. A suitable cloud computing model can be chosen depending on the requirements and complexity of a distributed system.

2.2 Containers and Orchestration

Containers are lightweight, portable, and isolated units of software that package applications and their dependencies, allowing them to run consistently across various computing environments. Containers encapsulate an application's code, runtime, libraries, and configuration, ensuring that operation, regardless of the underlying infrastructure, is guaranteed. They enable rapid deployment, scalability, and efficient resource utilization, making them a popular solution for building, shipping, and running applications in the cloud. *Docker*[5] is a widely used containerization platform that has significantly popularized the use of containers in cloud computing. The Kubernetes documentation[Kubj] mentions three deployment eras as seen in Figure 2.

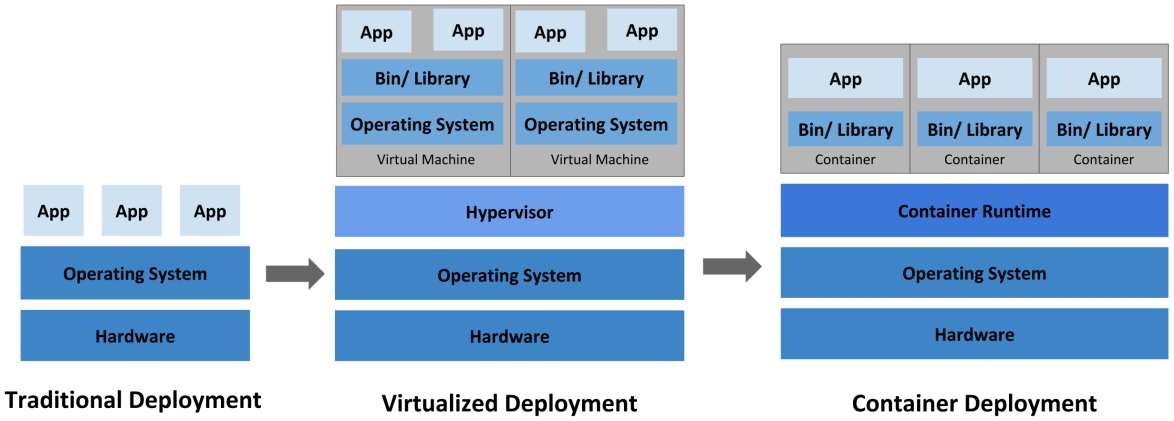

Figure 2: Evolution of software deployment eras[6]

Traditional deployment involves deploying applications on physical servers. This can cause performance issues in defining efficient resource allocation boundaries as multiple applications on one machine compete next to each other for resources. This approach makes efficient scaling and resource management difficult and expensive for organizations. Introducing **Virtualization** allows the definition of multiple virtual machines on a single host. Virtualization improves isolation between applications and simplifies resource management and utilization. Because each virtual machine is an independent instance running its operating system, a resource overhead exists with this approach. **Containers** are similar to virtual machines, but their isolation properties are not as sharply defined. They share parts of the underlying operating system and utilize the concepts of *cgroups*, *namespaces* and *unison file systems*[BW18, p. 37

[3]Examples for PaaS: Google App Engine, OpenShift, and Heroku
[4]Examples for SaaS: Dropbox, Google Drive, and HAFAS
[5]`https://www.docker.com/`
[6][Kubj]

- 42]. Containers have their isolated file system and share of CPU and memory. Decoupling from the underlying execution platform improves portability across various cloud vendors and operating systems. A container image bundles an application and its required execution environment, while a container is defined as the running instance of an image. Container image creation, versioning, deployment, and rollback simplify administrative efforts compared to handling virtual machines. Containers also adhere better to *Continuous Integration/Continuous Delivery* (CI/CD)[DMG07][HF10] and separate the concerns of image creation and image deployment cleanly. Loose coupling of applications via containers, environmental consistency, and observability count as the gained benefits. Container orchestration[Red22] describes the automation of deployment, management, scaling, and networking configuration of containers. Container orchestration also allows deploying the same application across different staging environments without any redesign to the application required. Open-source elastic platforms such as Kubernetes manage the orchestration of containers and allow the deployment of distributed applications that span multiple containers while automating their scheduling, scaling, routing, and monitoring aspects.

2.3 Kubernetes: Automating Container Orchestration

Kubernetes[Kubj] is a portable, open-source platform for automating, managing, and orchestrating containers. Before taking a closer look at service meshes in Chapter 2.4, a brief knowledge of fundamental Kubernetes key concepts is required since most service meshes work in symbiosis with Kubernetes.

2.3.1 What is Kubernetes?

Chapter 2.2 mentions the advantages of containers and that container orchestration helps automate the process of handling and monitoring containers. Instead of having humans on standby in case a container shuts down, Kubernetes can automate the initialization of new containers.

In the Kubernetes documentation[Kubq] the following core features are listed:

- Kubernetes can **load balance** traffic between service instances so the deployed application keeps running stable in high load situations.
- **Service discovery** exposes the containers behind a service via a DNS name, avoiding the need for hard-wiring container's IP addresses.
- **Rollouts** and **rollbacks** can be utilized to manage the version of deployed containers in a controlled manner. Deploying a new service release is as easy as creating a new container image with that version and replacing the currently deployed containers with new containers from that image.
- **Resource management** is implemented by assigning resource quotas to containers. Kubernetes then fits container instances according to those quotas into a node so that optimal resource utilization is archived.
- Kubernetes is **self-healing** by performing user-defined health checks on containers and restarting or replacing them as necessary.

- **Secret and configuration management** allows handling sensitive information such as auth tokens, passwords, or secret keys. Carrying out changes to configuration and secrets works without needing to rebuild the underlying container images each time.

Kubernetes does not see itself as a traditional PaaS platform[Kubp]. It is a container orchestration engine that can be described more fittingly as *Container as a Service* (CaaS). An IaaS layer below is required to provide the virtual machines that Kubernetes manages. Users are free to integrate their own logging, monitoring, and alerting solutions as they are designed to be optional and pluggable. Kubernetes provides the majority of the simplicity of PaaS with the flexibility of IaaS, thus enabling portability between infrastructure providers.

2.3.2 Core Components

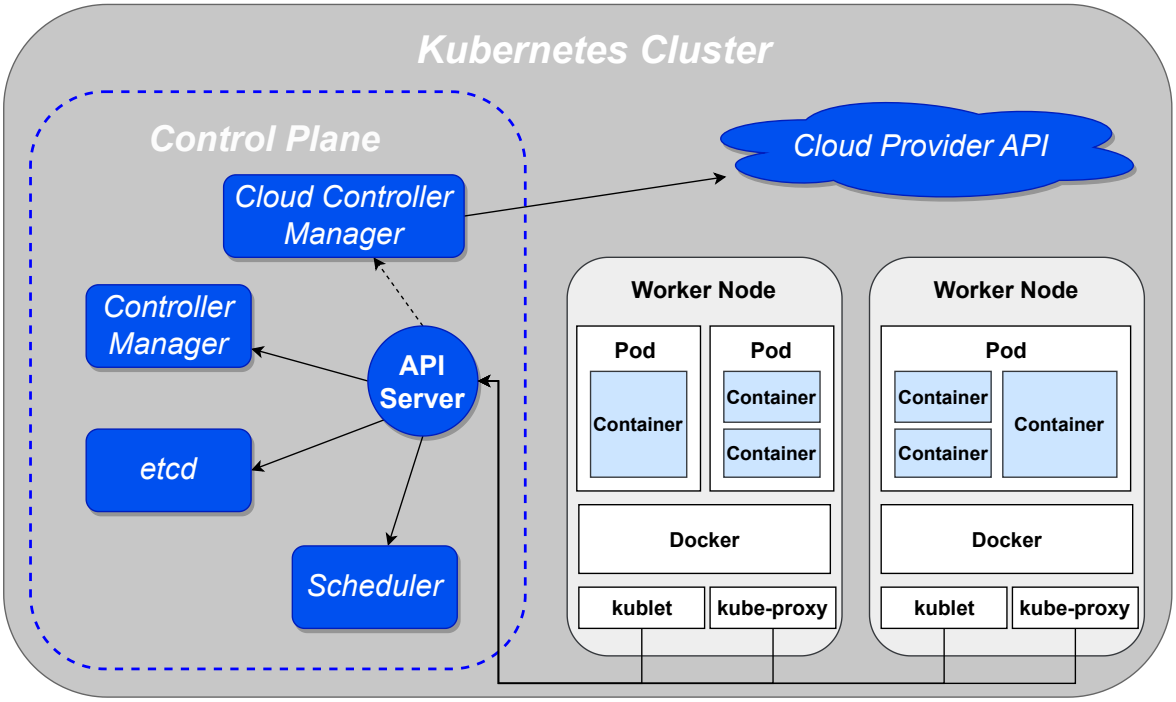

Figure 3: The architecture of a Kubernetes cluster[7]

In Figure 3, the components of Kubernetes are illustrated in detail: the key components are *Cluster*, *Node* and *Pod*[Kubf]. Once successfully initiated, a cluster consists of worker machines called nodes that run the containers in a pod. Worker nodes host pods[Kubk], which represent the smallest deployable unit of computation that Kubernetes manages. The resulting hierarchy is $Node \subset Cluster$ and $Pod \subset Node$. A pod is a group of one or more containers sharing storage and networking resources. A pod can contain one or more containers representing the service running on the pod. Pods can run single or multiple containers grouped and encapsulate a service composed of multiple applications. An additional relevant component is the control plane, which is used to configure and manage the cluster. An API server exposes the Kubernetes API to other machines for remote control plane access. A *kublet* represents an agent that runs on each node and ensures that the containers in a pod are healthy and running.

[7]Own illustration, based on [Kubf]

A *kube-proxy* on each node handles network traffic and configuration of network rules to ensure network connectivity.

2.3.3 Workloads, Services and Custom Resource Definitions

The API of Kubernetes is extensible over *Custom Resource Definitions* (CRDs)[Kubc] and already contains a sizeable list of default building blocks. Additionally, workloads[Kubd] provide declarative management of how Kubernetes should run a defined resource on a pod. The most important types of Kubernetes concepts for this work include, but are not limited to:

- **Namespaces**[Kubh] aid in organizing and bundling resources in the cluster. A namespace serves as a global configuration endpoint for all its contained resources.
- **Labels**[Kubg] in Kubernetes is metadata in the format of key-value pairs attached to resources such as pods that are used in dynamic configuration scenarios.
- **Services**[Kubm] abstract the access to pods. As pods are dynamically scheduled to Nodes, their IP addresses are not permanent and are an ever-changing landscape. Services define domain identifiers that route traffic permanently to the attached pods.
- **Deployments**[Kube] is a declarative way to manage and maintain sets of identical pods, ensuring application scalability, self-healing, and ease of updates.
- **StatefulSets**[Kubn] are a resource type used to manage stateful applications by providing stable network identities and ordered deployment, ensuring pods maintain their unique identities and storage.

2.3.4 Advantages and Disadvantages

The advantages of Kubernetes lie in its scalability, portability, and high availability. Kubernetes allows scaling applications based on demand, and clusters can be hosted on-premise or managed by an external vendor. As a standardized container orchestration platform, Kubernetes helps to avoid vendor lock-in due to the platform's portability. Health checks and failover features improve resiliency, which helps recover services in case of node or pod failures. Disadvantageous is the added complexity of Kubernetes and the resource overhead its internal components introduce. In practice, the benefits outweigh the disadvantages for organizations, which makes Kubernetes, according to the 2021 CNCF report, a widely[Mei22] adopted choice of technology in the cloud native space.

2.4 Service Mesh: A Dedicated Infrastructure Layer

After briefly mentioning the term service mesh in Chapter 1, it is time to define the concepts applied to a service mesh. What features it offers, how it operates, and overlooking the current technological landscape of service mesh projects.

2.4.1 What is a Service Mesh?

A service mesh[KBB+21] is a dedicated infrastructure layer for handling and controlling service-to-service communication[LLG+19] in a distributed system. It creates an independent and standardized way of measuring and controlling traffic[DMFC23]. Features offered by a service mesh enhance the capabilities of a distributed system so that no additions to the codebase of the services are required. This approach cleanly separates the business logic in services from the network logic used for service-to-service communication by outsourcing networking responsibilities to the service mesh. When applied correctly, this *separation of concerns*[Mit90][p. 5] allows stakeholders to have more precise control over the distributed system, improving reliability, security, and observability in the process. A service mesh thus standardizes and abstracts the configuration and handling of service-to-service communication in a distributed system to an additional infrastructure layer. Features offered by a service mesh include but are not limited to service discovery, load balancing, fault tolerance, traffic monitoring, circuit breaking, authentication, and access control. A service mesh consists of a data plane and a control plane. A set of proxies, which are called sidecar proxies[Tiw17], make up the data plane. Their naming is based on them applying the sidecar pattern. Those sidecar proxies handle and control all the network traffic between services within the service mesh. Figure 4 illustrates the sidecar pattern applied to a service mesh. Services within the mesh combine the service instance and a sidecar proxy. The network traffic of an instance is handled over the sidecar proxy, and the service instances do not directly communicate with each other. The book *Microservices for the Enterprise* supplies the following definition[IS18][p. 266] for the sidecar pattern:

> *"A sidecar is a software component that is co-located with the primary application but runs on its own process or container, providing a network interface to connect to it, which makes it language-agnostic. All the core application functionalities are implemented as part of the main application logic, while other commodity crosscutting features, which are not related to the business logic, are facilitated from the sidecar. Usually all the inbound and outbound communication of the application take place via the sidecar proxy."*

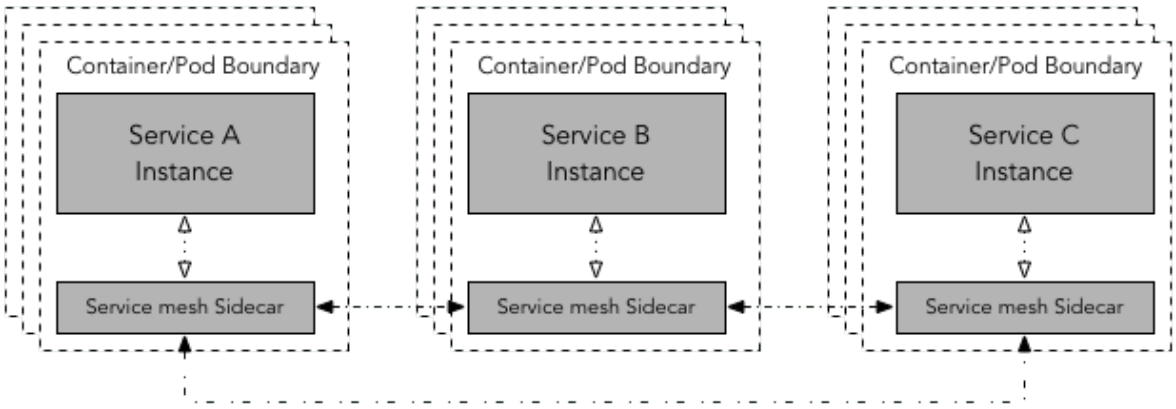

Figure 4: The sidecar pattern as a building block for service meshes[8]

[8] [Tiw17]

The control plane can be seen as the central unit of the service mesh, as it controls the sidecar proxies of the data plane. Figure 5[SG18] provides a graphical illustration of the divide into the data plane and control plane and serves as a high-level architectural overview of the design of service meshes. The following two bullet points give a detailed description of the two distinct planes:

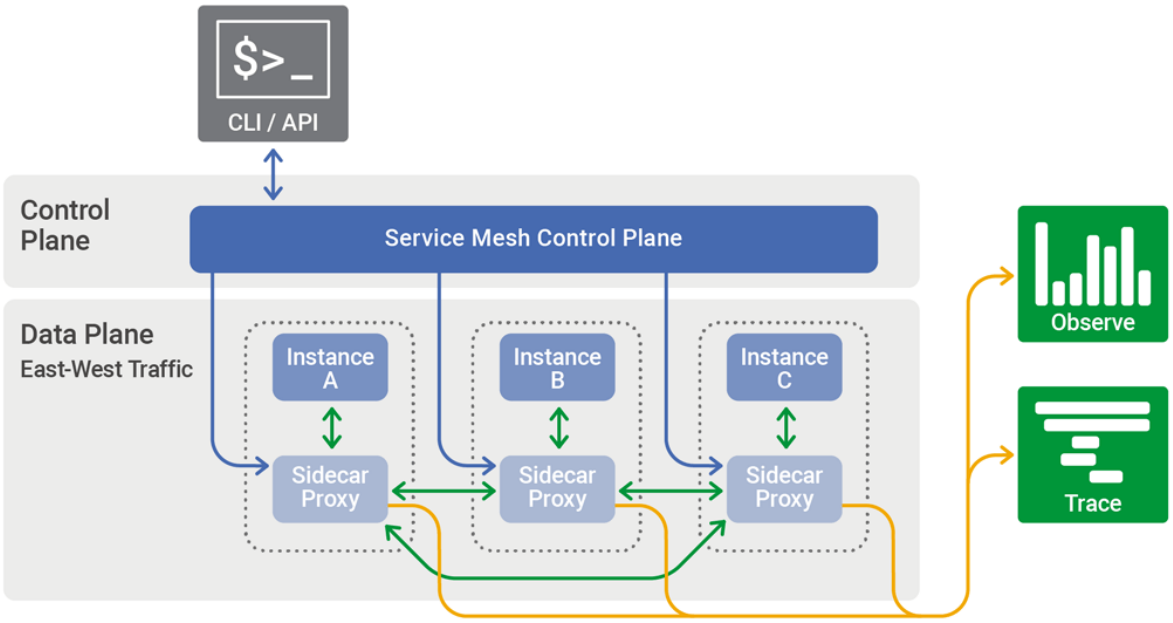

Figure 5: Overview of the two distinct planes in a service mesh[9]

- The **data plane**[IS18][p. 272] is composed of a set of sidecar proxies that route and control network traffic between the different services. Features such as health checking, circuit breaking, authentication, and authorization may also be provided by the data plane[CB20b][p. 7] to the distributed system. Service instances communicate with each other through the attached sidecar proxy. Each sidecar proxy can be configured individually depending on the required features and preferred policies. Service instances themselves do not count as part of the service mesh; besides noting their presence, the service mesh is unaware of them. Instead, the sidecar proxies attached to the service instance and report to the control plane make up the service endpoints within the mesh. A service instance itself is thus unaware of the service mesh and its configuration; the only change affecting a service instance when *meshing* it is the attached sidecar proxy. A service in a service mesh is thus always seen as a sidecar proxy and service instance combination.

- The **control plane**[IS18][p. 269] is a centralized component that handles the orchestration of the sidecar proxies in the data plane as it injects the sidecar proxies into the service instances marked for injection. Configuration changes are sent to the control plane and are distributed to the individual sidecar proxies. In order to change configurations externaly[SG18], the control plane is accessible via an API or *command-line interface* (CLI). Most service mesh projects also offer a *graphical user interface* (GUI) that connects to that API to visualize the state of the service mesh and its involved services. The control plane of a service mesh

[9] [SG18]

is also, to some extent, integrated[CB20b][p. 7] in the underlying orchestration platform (i.e., Kubernetes). Access to critical data such as the service registry and the deployed service's metadata is required, as sidecar proxies are injected into service instances based on the orchestration platform's service configuration. In Kubernetes, service instances are mostly marked for injection via the Labels from Chapter 2.3.3.

Since a service mesh also offers features of **observability** and **tracing**, the sidecar proxies also plug into those respective interfaces and collect metrics from the ongoing network traffic. The next chapter gives a more in-depth view of a service mesh's tracing and observability properties.

2.4.2 Fundamental Features

The main advantage of a service mesh is that it separates the business logic of a single service from the security, communication, and monitoring aspects. Introducing a service mesh to a distributed system allows the teams behind maintaining each service to focus solely on business logic decisions. In contrast, the operative team configures the service mesh to their requirements. A service mesh can be applied to any distributed system with service instances that allow attaching sidecar proxies to them; in practice, however, a service mesh is mainly applied to microservice[JPM+18] architectures. Fundamental features of a service mesh fall into the broad categories of **traffic management**, **resiliency**, **security** and **observability**[Pra23][Cha23]. In detail, the core features of a service mesh can be enumerated as[LLG+19][p. 123-124]:

- **Service discovery** allows each service to add itself and its current resource locator to an for all other services accessible service registry. The registry keeps track of active and inactive services in the service mesh. In most cases, distributed systems are not statically limited to the number of services. The location of a service may change over time as dynamic scaling adds or shuts down services depending on the experienced system-wide load.

- **Load balancing** provides means for controlling traffic flow between services. Various parameters such as health status, load factor, and customizable per-service definable weights allow complex routing scenarios. Canary deployments or blue-green rollouts can, in such a way, be implemented through a service mesh[Tig].

- **Fault tolerance** is improved by providing fault injection capabilities for resiliency testing purposes[Tig]. Retry logic encapsulates retrying requests to a service n-times in case the previous *1..n-1* requests have failed. An error is only reported to the caller once the threshold of n nonsuccessive request attempts is reached.

- **Circuit breaking** prevents endpoints deemed overloaded or otherwise unhealthy from being accessed. When the instance is considered unhealthy, requests will not be routed to that instance until it is deemed healthy again[CB20a][Ch. 4].

- **Monitoring** capabilities are enhanced by collecting metrics on the internal traffic flow between services through the attached sidecar proxies and reporting on request volume, success rates, and latency. Features such as distributed tracing

can also help visualize how a request flows through the whole distributed system and showcase which services are involved in its fullfilment[Tig], which improves debugging capabilities.

- **Authentication and access control** offer a security instrument that maps access policies onto a service mesh to define which services are allowed to be accessed by whom and how.
- **Encryption for data in transit** ensures that the internal data flow of a distributed system is protected from eavesdropping and interception if operational security is breached by hostiles taking control over a service instance. This is implemented by *mutual Transport Layer Security* (mTLS) traffic encryption[CB20b] and certificate management. Certificates are distributed automatically to the sidecar proxies, and the service mesh also handles the renewal of certificates.

2.4.3 Challenges and Opportunities

The main challenges of a service mesh[LLG+19][p. 124-125] can be summarized as being related to:

- **High performance**: Since the sidecar proxies are attached to the service instances and all of their passing traffic is intercepted and mediated, a lightweight and performant solution is crucial not to impede the overall performance of the distributed system. Components in the control plane that offer monitoring features and are required to aggregate large amounts of data must also adhere to specific performance standards to be considered beneficial. Another point is the overhead introduced by a service mesh that has to stay within certain boundaries, so its additional consumption of resources is not considered uneconomical, as, for every service instance, there is now also a sidecar proxy required.
- **High availability**: High availability is vital since a service mesh is deeply ingrained in a distributed system. Frequent outages of the sidecar proxies cause services within the service mesh to be unreachable and should be minimized so as not to impede operation. Outages in the control plane can cause issues with the authentication and authorization features, which can lead to an inconsistent application state and security risks, as requests not allowed to be accepted by a service could now potentially be accepted and processed.
- **Adaptability**: A service mesh must offer certain levels of configurability, extensibility, and pluggability. Supporting a wide range of cloud-native orchestration platforms in order to avoid vendor lock-in is a desirable property.

Additionally, a service mesh introduces complexity and increases configuration overhead. Service meshes are also relatively new, and their applied maturity in practice has not yet been proven by the test of time[Tig]. Service meshes have not entered mainstream use yet but are on the way, and any organization wishing to adapt a service mesh could be required to train or hire specialized staff. Integrating a service mesh into an existing infrastructure or supporting legacy applications may prove problematic. Opportunities lie in the added security benefits, such as data in transit encryption via mTLS and automated certificate management. By outsourcing operational concerns such as traffic management and resiliency away from the business logic and onto the

service mesh, the efficiency of development cycles could be improved[Tig]. The overall complexity of the service's codebase could also decrease, and operational features could be standardized between services by defining them in the service mesh configuration. DevOps concepts such as *Infrastructure as Code* (IaC) could be strengthened by this approach. Resiliency testing and debugging could improve for the involved teams due to features such as fault injection and distributed tracing. Managing configurations and policies for large-scale distributed systems could be simplified as a service mesh provides a unified and consistent way of doing so. Since service mesh projects aim to be platform-agnostic, the adverse effects of vendor lock-in for organizations could also be weakened.

2.4.4 Technological Landscape

Service mesh adoption in CNCF surveys has increased from 27% in 2020 to 47% in 2022[SF23]. The *CNCF Cloud Native Interactive Landscape*[CNCc] lists twenty-one service mesh projects as of October 2023. Two projects: *Linkerd*[10] and *Istio*[11] have been classified as *CNCF Graduated Projects*, meaning they are from a technical standpoint guaranteed to be stable and ready for productive use. Of those twenty-one projects, four count as *CNCF Sandbox Projects*, meaning they are in an early stage, and two have ended up in the archive, meaning they no longer receive support from the CNCF. The other thirteen projects count as CNCF member and non-member projects and have received no comments on their maturity level from the CNCF. There are also likely more than twenty-one service mesh projects as of right now, as projects such as the *NGINX Service Mesh*[Doc] are not cataloged by the CNCF. The exact number of service mesh projects cannot be defined, but given the CNCF numbers, one could estimate that between twenty and thirty serious service mesh projects exist at the moment. The Technological landscape of service meshes is seen as diverse and free of monopolies, as two mature projects exist with Istio and Linkerd. Not all these projects are open-source; some have licenses with varying restrictions. The vast technological landscape does match the adoption numbers of the CNCF survey. As more projects reach production-grade maturity levels, the landscape will likely be saturated with ongoing time. There appears to be no data regarding service mesh adoption outside CNCF surveys targeting companies and technical stakeholders with no solid exposure to cloud technology. This can indicate that service mesh technology is still in the adoption phase and not widely spread yet.

2.5 The HAFAS Timetable Information System

Now that an introduction to the key concepts and features of service meshes and an overview of their technological landscape have been given, it is time to take a closer look at HAFAS. As illustrated in Figure 6, Hacon[12] which belongs to Siemens Mobility[13] envisions and distributes components of a large-scale *Mobility as a Service* (MaaS) platform to customers that integrates various means of public transportation. Providing travelers with a comfortable and simple solution that only requires a mobile app on the end user's behalf to partake in public transportation[Hacc] is the idea behind the MaaS platform. Various mobility service providers using different data exchange

[10] `https://linkerd.io/`
[11] `https://istio.io/`
[12] `https://www.hacon.de/`
[13] `https://www.mobility.siemens.com/de/de.html`

formats will be integrated into this platform to deliver a seamless end user experience. A pamphlet about the MaaS platform[Hacc] describes the vision as following:

> *Travelers wish to get from door to door as quickly and conveniently as possible. Out of a customer-centric view, MaaS solutions combine all modes of transport in just one app. This way passengers can plan, book and ride with one account.*

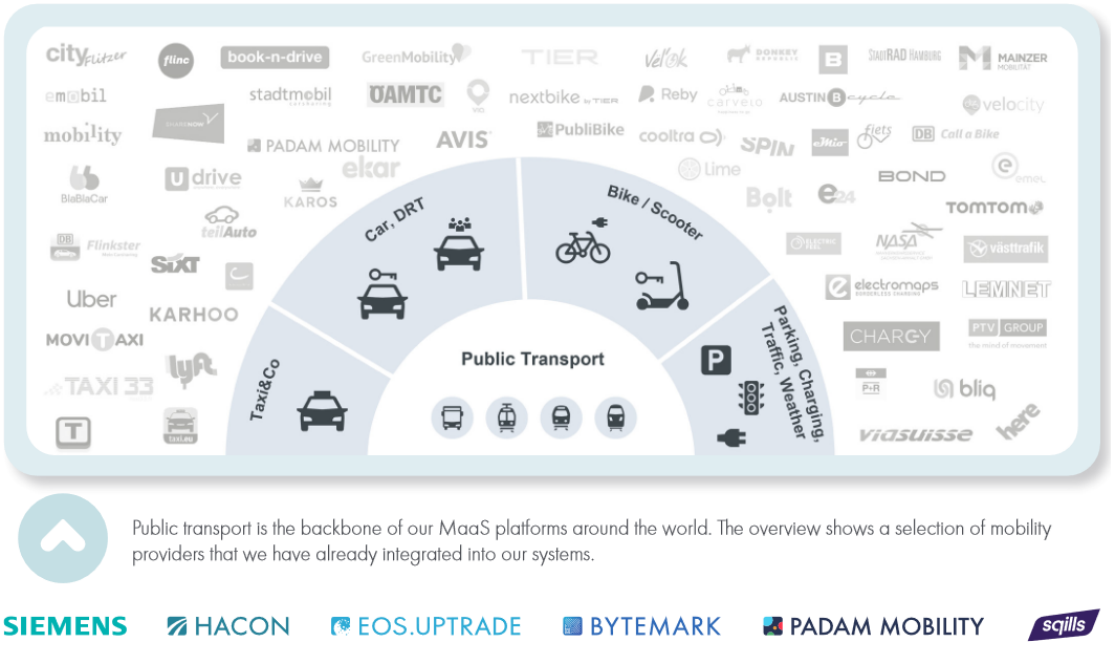

Figure 6: Overview of the MaaS ecosystem and involved partners[14]

Trip planning plays a vital role in implementing a MaaS platform. The HAFAS timetable information system (*deviated from the German term **Ha**con **F**ahrplan-**A**uskunfts-**S**ystem*[Wik]) is a distributed software system with trip planning capabilities. It is delivered by Hacon to various international customers such as the *Deutsche Bahn*[15], *Rejseplanen*[16], *Bay Area Rapid Transit*[17] or *Schweizerische Bundesbahnen*[18] and can be seen as an off-the shelf distributed software system that follows a modular principle. Singular components can be customized for customers by enabling and disabling features; the backend can range from a basic deployment including only a few services to a complex distributed system spanning multiple interconnected services.

The basic core components of a HAFAS system include but are not limited to:

- **Engine**[Hace]: A core system that is provided with timetable data from mobility providers and calculates ideal connections spanning multiple means of public transportation, including buses, trains, trams, scooters, ferries, bikes, car sharing, taxis, and even air travel.

[14] [Hacc]
[15] `https://www.bahn.de/`
[16] `https://www.rejseplanen.dk/webapp/`
[17] `https://www.bart.gov/`
[18] `https://www.sbb.ch/de`

- **API Server**[Hacb]: A gateway that allows HAFAS functionalities to be integrated into external applications and made accessible to third parties. Authorization and authentication are enforced, as are access quotas. It also allows customers to access internal and real-time data.
- **Web Application**[Hacd]: A web-based solution for accessing and using the HAFAS timetable information system for end users. It allows for trip planning and displays departure and arrival boards for stations. Real-time data and pricing information for user-queried journeys can be included depending on the scale of the configured HAFAS system.
- **Mobile App**[Haca]: Like the web-based solution for end user access, this mobile app supporting iOS and Android devices provides the same features natively for mobile devices.

Introducing a service mesh to HAFAS is part of the research questions handled in the next chapter of this thesis. Under the hood, the system is more complex than this brief introduction of components; there are also hidden components that play a vital role that have not been listed explicitly here, as this list is mainly of interest from the point of view of a transportation agency that wishes to add HAFAS to their portfolio.

2.6 Conclusion

The fundamentals are summarized as follows. Chapters 2.1 and 2.2 introduce different cloud computing models and discuss the evolution of software deployment eras, emphasizing the significance of containers and orchestration platforms like Kubernetes for achieving scalability in distributed systems. Chapter 2.3 delves into Kubernetes, outlining its core features and architecture and introducing the concepts of clusters, nodes, and pods. Chapter 2.4 defines and explores the concept of a service mesh as a dedicated infrastructure layer for handling and controlling service-to-service communication in a distributed system, detailing its fundamental features, challenges, and opportunities and presenting the technological landscape of service mesh projects. Lastly, Chapter 2.5 introduces the HAFAS Timetable Information System, a crucial component of a large-scale Mobility as a Service platform focusing on providing a seamless end user experience for public transportation through mobile and web-facing interfaces. The context and background for the following chapters should suffice with these fundamentals provided.

3 Analysis

Chapter 2 establishes the necessary vocabulary and concepts for understanding service mesh technology. Now, it is time to move on to the analysis part of this thesis. Firstly, the applied methodology is introduced and described in Chapter 3.1. Chapter 3.2 explains the current state of HAFAS, which includes an architectural overview of HAFAS and its involved components. Chapter 3.3 briefly states the desired state and expectations of introducing a service mesh to HAFAS. Chapter 3.4 verbalizes a research assignment supported by research questions and non-goals acting as delimiters for restricting the scope of the research. Finally, Chapter 3.5 offers a short conclusion about the progress made in this Chapter before transitioning to the conceptional stage of Chapter 4.

3.1 Introduction to Applied Methodology

The five main pillars of the methodology applied in this thesis are literature research, an expert interview, a service mesh scenario analysis, an examination of the service mesh technology landscape, and a practical case study. Additionally, the current state of HAFAS is covered; this includes studying the software architecture of HAFAS as a sixth pillar. The six research methods have been enumerated with the identifiers *M1..M6* and are listed in Table 1 and explained in detail below.

Identifier	**Methodology**
M1	Literature Research
M2	Studying the Software Architecture of HAFAS
M3	Expert Interview at Hacon with Cloud Architect
M4	Examination of Service Mesh Technology Landscape
M5	Conducting a Service Mesh Scenario Analysis
M6	Case Study on Introducing a Service Mesh to HAFAS

Table 1: List of methodology applied in this thesis

M1: Literature Research

To better understand service meshes in scientific literature and identify current research opportunities, the paper *Service Mesh: Challenges, State of the Art, and Future Research Opportunities*[LLG+19] from 2019 serves as the main starting point. As stated in a quote from the paper in Chapter 1, practical research is still rare, so no practical service mesh-related paper has been identified covering a topic comparable to this thesis. The papers *A Look at Service Meshes*[KBB+21] and *Service Mesh Patterns*[DMFC23] additionally serve as a general introduction to the topic. Some literature, such as *Microservices for Enterprise*[IS18] or *Building secure microservices-based applications using service-mesh architecture*[CB20b] have been also identified as being helpful. The topic of service meshes appears to be primarily an industry-driven domain, with most knowledge available in the documentation of service mesh projects, talks of tech conferences, or blog posts published by software architects, developers, and DevOps engineers who work with service meshes daily. Specialized literature such as *Service Mesh Patterns*[CJB23] also helped gasping service mesh concepts and their appliance in practice.

M2: Studying the Software Architecture of HAFAS

There are no official, publicly available sources on the architecture of HAFAS. 2.5 dealt with all publicly available information already. To gain an overview of the HAFAS architecture, consulting the documentation in Hacon's knowledge base takes place. Chapter 3.2 focuses on creating an architecture diagram of HAFAS, its components, and their respective system context. It also aims to describe the inner workings and interconnection of the systems components. The internal documents supporting the system architectural overview are not included in the Appendix of this thesis in order to publish this work without a blocking notice or leakage of corporate secrets. The only exception is the MaaS platform's architecture diagram in the Appendix in Figure 41. This diagram provides a rough overview of the overall HAFAS system. Access to company secrets and HAFAS software artifacts is given per this author's work contract with Hacon. The author is not authorized to release these company secrets and artifacts as per the work contract, and doing so would yield legal consequences for the author. Hacon has defined the degree to which HAFAS-related information in this thesis is permitted for publication. Releasing all these artifacts with the *Electronic Appendix* would be preferable so individuals interested in the case could reconstruct the case study themselves. However, due to HAFAS closed-source licensing, this is impossible, and that way of handling it is the most sensible in this case. The resulting system architecture diagram and description should be sufficient to understand HAFAS in the context of this thesis without having access to further internal documents and artifacts.

M3: Expert Interview at Hacon with Cloud Architect

To aid research method *M5*, an expert interview will be conducted with Fabian Korte, the cloud architect at Hacon. Due to his expertise and yearlong experience with cloud-hosted HAFAS systems, he has been identified as the expert with a complete overview of all components. As the topic of a service mesh is operation-heavy, it has not been deemed target-oriented to interview multiple stakeholders, such as project managers or software development teams, with isolated viewpoints of HAFAS. The interview guidelines and transcripts are found in the Appendix under *Guidelines for an Expert Interview* and *Transcript of the Expert Interview*. The primary purpose of this expert interview is to identify potential scenarios for *M5*. The expectations and resulting scenarios of introducing a service mesh to HAFAS are explored in Chapter 3.3. Criteria are defined for deciding which service mesh projects are deemed of interest in the case of HAFAS. Those criteria will later help in *M4* to identify which service mesh projects are deemed reasonable candidates for *M6*. The recorded interview is transcribed, and parts of it will be referenced in this thesis to gain insight into the current and desired situation of HAFAS.

M4: Examination of Service Mesh Technology Landscape

Chapter 4.1 will define requirements for choosing a service mesh project for *M6* aided by the criteria and scenarios uncovered in *M3*. With those requirements in mind, the service mesh technology landscape described in Chapter 2.4.4 will be examined in Chapter 4.2. Part of this examination is a selection process that singles out service mesh projects not deemed a good fit with the requirements defined in 4.1. At the end of the process, a single service mesh project has been identified that is deemed a good match for *M6*. That service mesh project is also the reference point in *M5*.

M5: Conducting a Service Mesh Scenario Analysis

With the understanding of service meshes and their capabilities as of *M1* and the insights provided by *M2*, after conducting *M3*, it is possible to explore the scenarios of introducing the service mesh project selected in *M4* to HAFAS in detail. Those scenarios will be investigated further in Chapter 4.3. For each scenario, a problem and a solution will be identified and described; the results of *M3* guide in understanding the decisive background of each scenario. A strategy is described in how a scenario-related service mesh feature could be applied in practice, and it is attempted to assess the feasibility of the scenario with HAFAS. Out of the defined scenarios, a subset of scenarios is chosen as the main scenarios of interest in Chapter 4.4.1 that is then investigated in *M6*.

M6: Case Study on Introducing a Service Mesh to HAFAS

In order to gain data on introducing a service mesh to HAFAS, a practical case study is carried out. In this case study, the main scenarios of interest defined in Chapter 4.4.1 are implemented by introducing the selected service mesh project from Chapter 4.2 to a HAFAS environment run on a Kubernetes cluster. The results of the case study serve as a proof of concept of introducing a service mesh to HAFAS. The gained data will then aid in evaluating the feasibility of a service mesh introduced to HAFAS in Chapter 6.

3.2 Describing the Current State of HAFAS

In order to understand how HAFAS works under the hood and identify which components interact with each other, a look at the system architecture[ML] is necessary. Note that this chapter only aims to provide fundamental knowledge of specific thesis-relevant HAFAS key services; a complex HAFAS environment can be made up of a long list of specialized services that will not be included here. Since the whole MaaS platform from Chapter 2.5 is beyond the scope of this thesis, only the relevant HAFAS-related section will be investigated further. Figure 7 (on the next page) showcases the general system architecture of an advanced HAFAS environment. The system is divided into sections in the architectural diagram described in Table 2.

Table 2: Overview of the different sections in the architectural diagram of Figure 7

Section	Description
Trip Planning Front-end	The trip planning front-end includes the web application and the platform dependant mobile apps for Android and iOS.
Trip Planning API	The trip planning API serves as an API gateway to the backend components of HAFAS and manages access by front-end clients and third parties
Backend Components	The backend components include the components that are required to calculate the most basic trip planning operations.
Integration Components	Integration components enhance the trip planning features of HAFAS and are additional to the most basic functional environment.

External Data	External data is provided by either public transportation agencies or transportation companies over external interfaces and is required by some integration components.

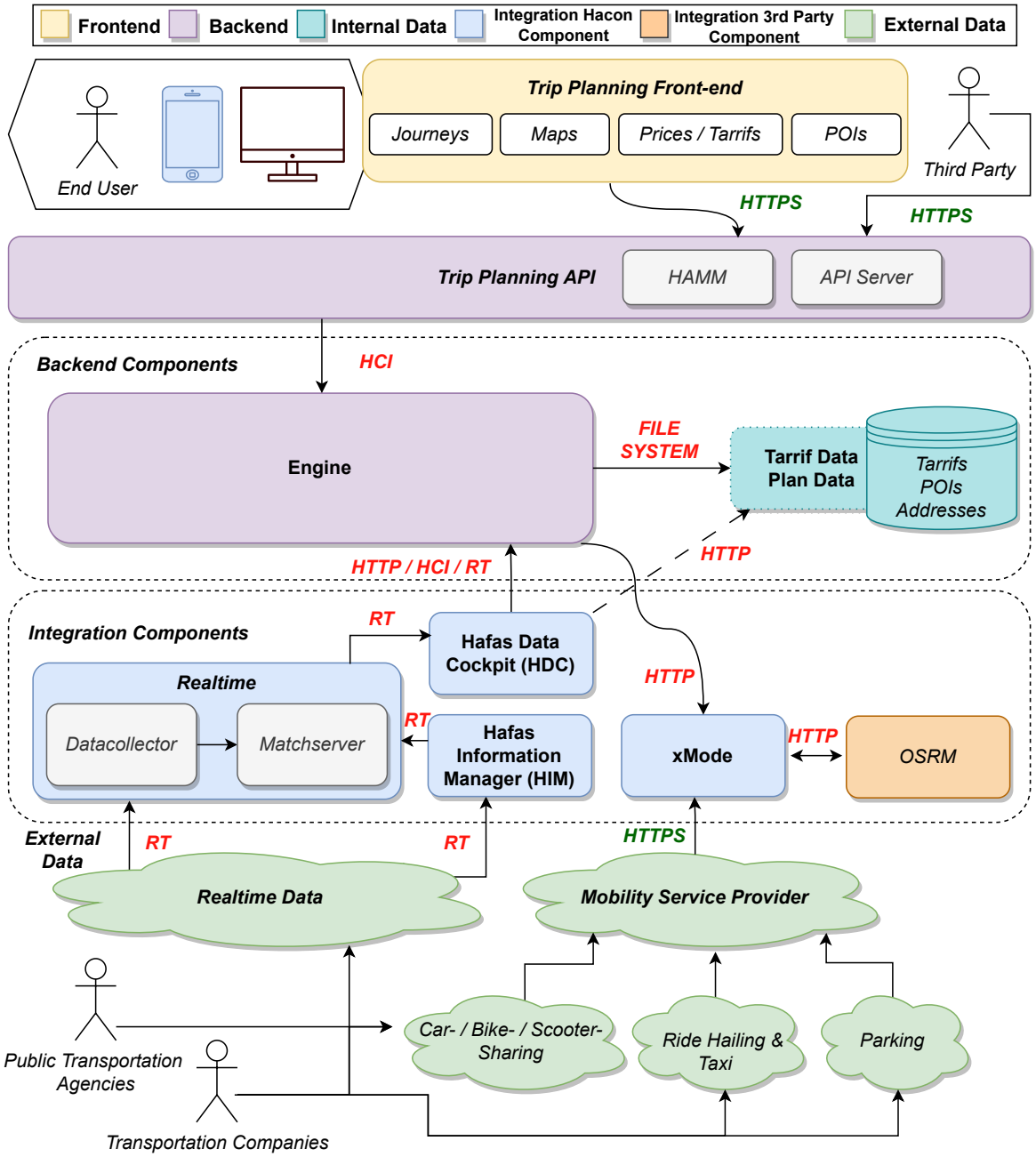

Figure 7: A high-level architectural overview of HAFAS (own illustration)

As illustrated in Figure 7, the front end is divided between the clients of a web application and platform-dependent mobile apps for Android and iOS. All front-end requests made by the end user hit the trip planning API as their first stop through the backend. They are then routed further to the engine. The web application is designed with a standardized assembly kit approach, allowing different customers to turn on or off certain features and change the design while using one continuously developed version.

This eliminates the requirement of keeping multiple teams maintaining similar yet varying web application versions for different customers. Once the engine has calculated a request, the request is sent back to the front-end clients, where the results of the trip calculation are rendered comprehensibly to the end user. The trip planning API is seen as an API gateway with some load distribution and authorization responsibilities. In Figure 8, the inner workings of the trip planning API are illustrated. It consists of two independent services: *HAFAS Application Massaging Middleware* (HAMM) and *API Server*.

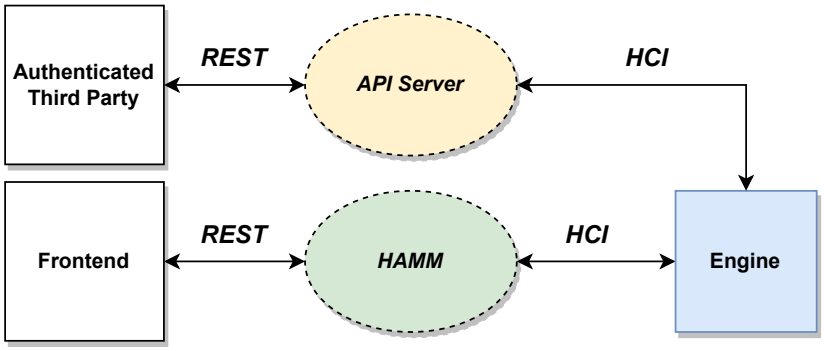

Figure 8: A detailed look inside the trip planning API (own illustration)

For handling load distribution aspects, the HAMM acts as a middleware and lightweight API gateway that distributes the HCI requests to the engine instances according to internally tracked capacity metrics. An optional part of the Trip Planning API is the API server. It allows authenticated third parties to access HAFAS features and internal data to build a customized front-end. The HAMM validates front-end requests and checks if the sending client is registered with the engine configuration. Different clients can have different features assigned that change the engine's behavior when processing a request. Each engine has a set number of threads that can be used to calculate a trip concurrently; if all threads are in use, incoming requests are added to a queue. The HAMM tries to route requests so that each engine's queues remain empty. The protocol that is used to communicate with the engine is called *HAFAS Client Interface* (HCI) and is a proprietary protocol that works by sending commands via JSON or XML over a *Transmission Control Protocol* (TCP) connection. HAMM and the API Server are thus required to translate the inbound RESTful calls into outbound HCI requests.

The heart of every HAFAS environment is the engine, which calculates trips and fulfills the requests of end users and third parties made via the trip planning API. Such a trip calculation request contains a specified time interval, a departure and destination location, and optional trip parameters that define calculation behavior properties (i.e., include individual transportation such as e-scooters or ride-sharing). In a realistic scenario, the load produced by end users is too much to be handled by one engine, so the engine is scaled horizontally in productive environments. Each engine has an upper boundary of requests that can be handled simultaneously; that boundary can be configured and ultimately depends on how many resources the engine has and how many threads run stable in parallel. Each thread can do one trip calculation sequentially; if all threads are busy, a new request must wait in a queue. In a realistic scenario, distributing requests to multiple engines is essential to spread the load evenly among the engine instances; for the engine to calculate routes for trip planning requests,

tariff and plan data are supplied to each instance via the local file system. This data contains the different tariff zones, points of interest, addresses, stations, and timetables of the operating transportation agency's various train and bus lines. Distributing the tariff and plan data by hand would be tedious; the *Hafas Data Cockpit* (HDC) of the integration components automatically distributes plan, tariff, and real-time data to the engines. Engines register with the HDC and are assigned a data pool. The data pool is updated if new data is available, and the registered engines are notified about the update. They will then proceed with downloading and using the newly available data.

Up until now, the essential components required to have a functional HAFAS system in place have been described. The additional components seen in Figure 7, which have not been mentioned yet, enhance the feature set of that basic HAFAS environment in the following ways:

- The **Realtime** component consists of two services named *data collector* and *match server*. The data collector collects and bundles real-time data from various sources, such as mobility providers or internal tools. The match server is supplied with the bundled data by the data collector and is then tasked with validating the data. The real-time data is checked against the plan data to identify which routes are affected by cancellations and diversions. Lastly, that enriched real-time data is converted to a format compatible with the engine. The HDC transmits the real-time data to the various engine instances, which includes it in their calculations.

- The ***Hafas Information Manager*** **(HIM)** allows transportation agencies and companies to attach customized information to a trip that is then displayed to the end users. Such information includes gate and platform changes, cancellations, and customized texts supporting multiple languages. Once information is associated with a trip, it is transmitted to the data collector as real-time data. If the engine calculates as a trip and finds such attached information, it includes it in its response, relaying the message to the end user.

- To also integrate third-party mobility providers such as e-bikes, e-scooters, taxis, and ride-sharing services, the **xMode** can be added to the system. The xMode collects data from third parties and converts it to a format the engine can use when calculating trips. An *Open Source Routing Machine* (OSRM)[1] server supports the xMode by calculating parts of a trip and their corresponding estimated times that take place by foot, bicycle, scooter, and car. For calculating the shortest path in a road network, the OSRM server relies on OpenStreetMap[2] data.

HAFAS can be accurately described as a service-based architecture. It cannot be considered a microservice architecture, as the services involved have too many obligations and are too large. HAFAS also cannot be considered a *Service Oriented Architecture* (SOA)[Jos07] as there is no *Enterprise Service Bus* (ESB)[Jos07][Ch. 5] or equivalent, acting as a centralized middleware. HAFAS also cannot be described as a monolith[Ing18][p. 260-262], as it is broken up into different services developed independently. So, describing HAFAS as a service-based architecture is the closest to putting a label on it. HAFAS can also be described as a distributed off-the-shelf

[1] `https://project-osrm.org/`
[2] `https://www.openstreetmap.org/`

software system, as each component has one team assigned to it and is continuously developed. Flavors of components may exist, but components are not developed independently per customer. Setting up a HAFAS environment can be done by turning on and off the features that the customer needs.

3.3 Describing the Desired State of HAFAS

After having conducted *M3*[3], the desired state of HAFAS can be described as follows: Currently, the migration towards Kubernetes has not been completed as a whole yet. Some systems, however, run as productive environments on a Kubernetes cluster already. Application scenarios have emerged where a service mesh could be helpful. Due to the planned future migration of all systems to Kubernetes and the resulting technological foundation, introducing service mesh technology is considered a noteworthy option. The HAFAS application-relevant scenarios primarily focus on a service mesh's security-enhancing features. Features for controlling network traffic appear attractive but are not the primary reason a service mesh should be introduced to HAFAS. The current Prometheus-based monitoring stack is deemed satisfactory. Therefore, the monitoring properties of a service mesh would not represent a selection criterion for this particular technology. It would be considered a bonus if the additional monitoring properties are helpful. Scenarios that would mainly support adding service mesh technology to HAFAS's hosting stack are listed in Table 3.

Identifier	**Scenario**
S1	Data in Transit Encryption with mTLS
S2	Certificate Management for Data and Control Plane
S3	Service-to-Service Authorization Policies
S4	Improving Resilience with Circuit Breaking
S5	Testing Robustness with Fault Injection

Table 3: Prioritized scenarios that warrant introducing a service mesh to HAFAS

Scenario *S1* is motivated by the fact that there is currently no data in transit encryption enforced by all services of HAFAS. Only selected services dealing with sensitive user data enforce encryption with their implementation maintained on a service level by development teams. Introducing mTLS helps to ensure that traffic flow is secure and trusted in both directions between services. mTLS provides an additional layer of security and prevents attackers from eavesdropping, tampering, and launching impersonation attacks. Scenario *S2* is closely related to *S1* and relates to rotating, issuing, and rotating the certificates used in mTLS encryption. Scenario *S3* relates to network policies that govern how services can communicate with whom and how. Service meshes introduce zero trust network policies enforced at OSI-Layer 7. These policies offer more flexibility and security than the Layer 3 and 4 Kubernetes network policies regarding improving the isolation of systems in multi-tenant environments where HTTP endpoints are shared between systems. It also restricts the possible endpoints a potential attacker can access on a compromised workload. Scenario *S4* addresses a possible improvement to system-wide resilience by introducing a circuit breaker that limits the impact of failing services. This feature already exists with some standalone components of HAFAS, resulting in whether a service mesh-operated circuit breaker

[3]Accessible in the Appendix under *Transcript of the Expert Interview*

provides additional advantages. The last scenario *S5* targets the testing of robustness by injecting faulty requests directly into the service mesh instead of having to set up application-specific debug code. This feature is not particularly important from a hosting and operations standpoint but might prove helpful for developer and testing teams. These five scenarios have been identified as prioritized and thus support introducing a service mesh to HAFAS.

Table 4: Scenarios not investigated further in this thesis

Scenario	**Comment**
Traffic Redirection Traffic Splitting	Use cases such as blue-green or canary deployments cannot be solely handled on the network level, as data dependencies complicate the matter.
Traffic Mirroring	Interesting for testing environments, but it is not a priority to introduce a service mesh to HAFAS.
Retry Logic	Services within HAFAS already have their retry logic implemented, thus not being seen as a priority for introducing a service mesh.
Distributed Tracing	Requires the passing through of tracing headers, leading to large-scale code changes across all services involved in HAFAS. The observability features of a service mesh do not warrant introducing large-scale changes to HAFAS.

Table 4 lists scenarios not investigated further in this thesis as they have been deemed irrelevant for HAFAS. These scenarios are attractive to some extent, but they do not justify introducing a service mesh to HAFAS alone or provide limited use as they require extensive changes to be made to HAFAS. The feasibility of these scenarios is, to a greater extent, unclear, as the comments indicate, making them a non-priority for now. In this thesis, they will be considered out of scope and not investigated further. At this point, the main scenarios *S1-S5* have been identified that warrant introducing a service mesh to HAFAS, and only those will be looked into further.

3.4 Defining a Research Assignment

After identifying adequate scenarios of use that support introducing a service mesh to HAFAS, it is time to define a research assignment for this thesis.

RA: What is the practical benefit in introducing a service mesh to an already existing distributed software system?

In this research assignment, the already-existing distributed software system is HAFAS, and the following five questions define the scope of this thesis research assignment in more detail:

Defining Research Questions

Q1: What are the chances and risks of using a service mesh in practice?

Challenges and opportunities of a service mesh have been briefly mentioned in Chapter 2.4.3. A question that arises is how the findings of that Chapter turn out after

introducing a service mesh to an already existing distributed software system such as HAFAS.

Q2: What does the current technological landscape of existing service mesh projects look like?

Chapter 2.4.4 has given a short overview of the current technological landscape of existing service mesh projects. Now, with the five scenarios *S1..S5* identified in Chapter 3.3, the question arises: Which of those projects are compatible with those scenarios and match the requirements that will be defined in Chapter 4.1 as well?

Q3: What are the implications and restrictions of introducing a service mesh to an existing system?

Since every distributed system has its architectural challenges, the question is whether, in the example of HAFAS, certain boundaries in the services making up the system would prevent a service mesh from being introduced successfully. If such restrictions exist, finding and overcoming them could aid in establishing a framework for introducing service meshes to already existing distributed systems whose architecture has not been planned to be service mesh-compatible from the ground up.

Q4: How do the complexity and overhead of a service mesh impact performance and allocation of resources?

One of the three main challenges of service meshes discussed in Chapter 2.4.3 has been related to performance. Introducing a service mesh to a system creates more complexity and could lead to impacted performance and increased resource consumption. This question assesses if the added benefits of a service mesh outweigh the disadvantages; in this case, the results of a practical scenario are attractive for gaining data and insights.

Q5: What are the complications of introducing a service mesh in a practical scenario, such as HAFAS?

This research question targets the unknown and assumes that there will be some complications when introducing a service mesh to a distributed software system that has not been designed from the ground up to be operated in combination with a service mesh.

Defining Non-Goals

Non-goals help refine the targeted scope of the thesis for the previously defined research questions. The following four non-goals have been defined for the previous five research questions:

NG1: Evaluating every single feature offered by a service mesh

Since this thesis has a HAFAS-related focus, introducing and evaluating the general scenario of every single feature that a service mesh provides is not the goal of this thesis. Instead, features are evaluated based on whether they are deemed attractive for a HAFAS environment; the most attractive feature is then investigated and evaluated in the practical section of this thesis.

NG2: Introducing a service mesh to HAFAS production environments

Since there is no guarantee that, in its current state, HAFAS would seamlessly support a service mesh, it is ruled out that any artifact ready for productive use will result from this thesis. Instead, the practical research will serve as a starting point for the future discussion of whether a service mesh is desirable for productive use and what changes are required to head in that direction.

NG3: Providing a detailed framework for introducing a service mesh to an already existing distributed system

Since HAFAS is a unique software system, it is unlikely that other systems will behave the same when confronted with a service mesh. The findings of this thesis may help serve as a reference in some cases; however, they will not be nearly detailed enough in the edge cases covered to create a framework for introducing a service mesh to an existing distributed system.

NG4: Providing an in-depth cost and performance analysis for introducing a service mesh to an already existing distributed system

An attempt at providing a performance and cost estimation will be made; a detailed report, however, is reserved for a thesis that follows up on the topic. This thesis emphasizes introducing a service mesh to a distributed system such as HAFAS and creating an early proof of concept.

3.5 Conclusion

The methods *M1..M6* used in this thesis's scope have been introduced, and the current state of HAFAS has been described. Building on that, the expert interview mentioned in *M3* has aided in generating a list of prioritized scenarios when introducing a service mesh to HAFAS. Those scenarios *S1..S5* will be handled in more detail in Chapter 4.3. Lastly, a research assignment has been defined that is based on research questions and non-goals. The following chapters aim to investigate the identified scenarios in more detail and select a service mesh project for the case study that matches the scenarios from Chapter 3.3 and requirements of Chapter 4.1. Lastly, the practical case study will be conducted with the selected service mesh project and a chosen subset of scenarios of interest. This case study serves as a proof of concept for introducing a service mesh to HAFAS and will aid in generating the data needed to answer the research questions *Q1..Q5* in Chapter 6.

4 Conception

The chapter on conception follows up on the ideas presented in Chapter 3. The defining requirements for a service mesh project are discussed in Chapter 4.1. Functional and non-functional requirements are established for this purpose. These requirements are subsequently used to select a service mesh project by research method *M4* in Chapter 4.2. This service mesh project selection is followed in Chapter 4.3 by the service mesh scenario analysis, as addressed in research method *M5*. This scenario analysis is based on the scenarios *S1..S5* identified for HAFAS in Chapter 3.3. Finally, Chapter 4.4 defines the foundations for the case study. A subset of the scenarios will be selected and trialed in practice with the service mesh project selected in Chapter 4.2 being introduced to a dummy HAFAS environment used for this case study. It is also determined what data is collected in the context of this case study and how that data is used in answering the research questions from Chapter 3.4.

4.1 Defining Requirements for Service Mesh Projects

In order to analyze the technological landscape of existing service mesh projects, it is necessary to define the requirements of a service mesh project. These requirements are declared to remain within the reasonable scope of HAFAS. The expert interview from method *M3* is used in this case to include a HAFAS-oriented perspective to the matter. Insight into the expert interview and the corresponding guidelines are found in the appendix under sections *Guidelines for an Expert Interview* and *Transcript of the Expert Interview*. In this chapter, defined requirements are divided into *functional* and *non-functional* requirements[AA05][6.3]. With that in mind, functional and non-functional requirements in the context of a service mesh are defined as:

- **Functional requirements** are detailed specifications describing the specific functions, features, and capabilities a service mesh project must have to meet its intended purpose.
- **Non-functional requirements** specify how a service mesh should perform in terms of performance, reliability, security, usability, maintainability, stability, et cetera.

In this case, the prioritized scenarios *S1..S5* previously identified in Chapter 3.3 correlate to a specific service mesh feature that translates to a functional requirement. When considering service mesh projects for closer investigation in Chapter 4.2, only projects implementing those features can be selected for the latter-conducted case study. The resulting functional requirements for the service mesh project have been listed in Table 5 below.

Scenario	Functional Requirement Id	Targeted Service Mesh Feature
S1	FR1	mTLS Encryption
S2	FR2	Certificate Management
S3	FR3	Service-to-Service Authorization Policies
S4	FR4	Circuit Breaking
S5	FR5	Fault Injection

Table 5: Functional requirements, their corresponding scenarios, and targeted service mesh features

At this point, scenarios have been linked to functional requirements and a targeted service mesh feature. A description formulating the details of each functional requirement is found in Table 6 and completes the definition of functional requirements,

Id	Details of Functional Requirement
FR1	The service mesh project should provide features that ensure setting up encrypted mTLS connections targeting meshed service-to-service communication successfully. Turning this feature on and off on a per-service basis should be possible.
FR2	The service mesh project should provide features that aid in certificate management. This includes changing the root certificate of the *Certificate Authority* (CA) that is used to derive the certificates used by the sidecar proxies for mTLS encryption. The service mesh project should allow automating the rotation of the root certificate.
FR3	The service mesh project should contain features that allow configuring policies that determine which services or components can communicate with each other, helping enforce security and access control within the service mesh. It should be possible to define authorization policies on a per-service basis.
FR4	The service mesh project should provide a circuit-breaking feature that helps prevent service failures and overloads by automatically temporarily blocking traffic to a service once its health status deteriorates, ensuring system stability in the process.
FR5	The service mesh project should contain tooling that allows for intentionally introducing errors into a service's communication in order to test its resilience and ability to handle unexpected issues under controlled circumstances.

Table 6: List of functional requirements and their details

The following non-functional requirements for service mesh projects have been identified in Table 7.

Non-functional Requirement Id	Non-functional Requirement
NFR1	Technical Maturity
NFR2	Compatibility
NFR3	Maintainability
NFR4	Configurability
NFR5	Usability
NFR6	Portability
NFR7	Usage Cost
NFR8	Licensability

Table 7: Non-functional requirements for service mesh projects

As per the expert interview, technical maturity has come up, as there is no interest in introducing a cutting-edge service mesh project to HAFAS that has not yet seen any production use elsewhere. Additionally, compatibility is essential, as the service mesh project has to support Kubernetes environments and should not require software changes to HAFAS. A desired solution should not add additional costs on its own. There are no strict restrictions on licensability, so open-source and proprietary

projects are considered, although open-source is preferable. There are weak restrictions on portability: if desired, a project should be migrated to other cloud platforms if necessary. However, due to Hacon's reliance on Amazon Web Services, a solution that deeply integrates into it could be considered if it outweighs all other solutions by a large margin. Outside the interview, configurability, usability, and maintainability will also be included in the non-functional requirements. These aspects target the configuration aspects of a service mesh project and whether it provides ways to ease user interaction, such as a CLI that aids in configuration or a dashboard to visualize the current state of the service mesh. With the defined non-functional requirements for service mesh projects, a description drafting the details of each non-functional requirement is found in Table 8.

Id	**Details of Non-functional Requirement**
NFR1	The non-functional requirement of technical maturity requires that other companies actively use the service mesh project in production scenarios. In the case of commercial products, it is assumed here that a released product has sufficient maturity. For open-source projects, the maturity level of the CNCF is consulted. The project should have had security audits completed.
NFR2	There are three decisive criteria concerning compatibility features. Firstly, the service mesh project must be compatible with a Kubernetes cluster. Secondly, no drastic changes must be made to HAFAS to implement the service mesh. The service mesh must support tunneling HTTP and raw TCP connections through its sidecar proxies.
NFR3	For the maintainability of the service mesh project, it is necessary to have an understandable and well-documented mechanism for upgrading the project that prevents the service mesh from experiencing outages during the upgrade process.
NFR4	Concerning the configurability requirement, it is necessary for the service mesh project to have an understandable, detailed, and well-documented configuration mechanism.
NFR5	For the usability requirement, the service mesh project must offer tooling for user-friendly interaction. Tooling includes, for example, a command-line interface to manage the configuration or a graphical user interface that reports on the current state of the service mesh.
NFR6	For the requirements of portability, it is desirable, but not necessary, that the service mesh project can also be used in cloud environments other than AWS.
NFR7	With regard to the cost requirements, it is considered unavoidable that the control plane of the service mesh and the necessary sidecar proxies will increase the use of resources. This increase in computing power will result in basic costs. It is, therefore, necessary that using the service mesh project does not result in any licensing costs.
NFR8	Regarding licensability, the service mesh project is preferred to be considered a healthy *Open Source Software* (OSS) project. . A way of measuring is to see if the project has obtained an *OpenSSF Best Practice* badge[Ope]. Non-OSS projects would only be preferable if they integrated into AWS at ease.

Table 8: List of non-functional requirements and their details

With those five functional requirements *FR1..FR5* and eight non-functional requirements *NFR1...NFR8* a selection process is conducted to identify projects that correspond to these requirements and would be a suitable fit for HAFAS. As per *NG4:*

Providing an in-depth cost and performance analysis for introducing a service mesh to an already existing distributed system and *Q4: How do the complexity and overhead of a service mesh impact performance and allocation of resources?* There have been no defined non-functional requirements for performance and availability. These can hardly be provided by every service mesh project and depend on the distributed system to which the service mesh is introduced and the targeted usage scenarios. Suppose multiple service mesh projects appear to be on par and sources that allow a rough performance comparison are available. This metric will be used to decide which project to prioritize for the case study. As per *NG2: Introducing a service mesh to HAFAS production environments*, no requirements for professional support via paid options are to be considered at this stage. For some open-source service mesh projects such as Linkerd enterprise versions such as Buoyant[1] exist that provide advanced support and features that are absent with the open-source version.

4.2 Selecting a Service Mesh Project for Closer Investigation

With the functional and non-functional requirements for a service mesh project defined in the previous chapter, it is time to apply them to a set of projects to filter out a suitable candidate for the case study. In Chapter 2.4.4, it has been established that the CNCF Cloud Native Landscape[CNCc] currently lists twenty-one service mesh projects. These twenty-one projects are seen as a base list of service mesh projects for the selection process. Chapter 2.4.4 also mentioned that some service mesh projects, such as the NGINX Service Mesh, appear not to be listed by the CNCF. Projects not listed by the CNCF are not deemed noteworthy, so this selection process will only focus on the twenty-one projects in the CNCF Cloud Native Landscape. As seen in Figure 9, the CNCF also provides access to the Cloud Native Landscape in an interactive way[CNCb].

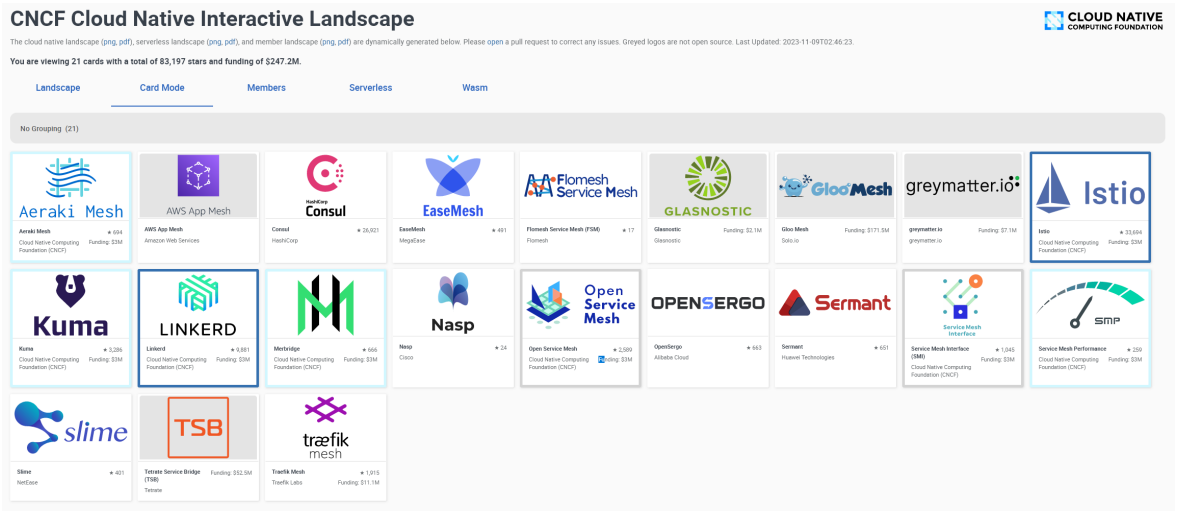

Figure 9: Interactive overview of the CNCF Cloud Native service mesh landscape[2]

This allows quick filtering and grouping of the projects by pre-defined criteria. As comparing twenty-one projects and their responses to the previously defined requirements is not feasible, pre-culling is required to shorten the list of possible candidates. The following steps define this pre-culling process:

[1] `https://buoyant.io/`
[2] Screenshot of [CNCb]

- Aggregate date of the twenty-one projects from the CNCF Interactive Cloud Native Landscape[CNCb] as seen in Table 9. This data[3] includes the project name, organization behind, license information, and number of GitHub[4] stars.
- For OSS projects, it is assumed that the amount of GitHub stars correlate with the popularity, user commitment, and distribution of a project, as the platform is considered a central hub of the OSS community[git23]. This approach utilizes the amount of GitHub stars as a rough indicator of project maturity, assuming that mature projects have a higher impact and, thus, a higher reach.
- For projects, not OSS, there is no data on the GitHub stars, as development occurs behind closed doors. As OSS projects are preferable because of *NFR8*, Amazon's AWS App Mesh would be the only exception.

Project	**Organization**	**License**	**GitHub Stars**
Aeraki Mesh	CNCF	OSS	694
AWS App Mesh	AWS	Not OSS	-
Consul	HashiCorp	OSS	26921
EaseMesh	MegaEase	OSS	491
Flomesh Service Mesh	Flomesh	OSS	17
Glasnostic	Glasnostic	Not OSS	-
Gloo Mesh	Solo.io	Not OSS	-
greymatter.io	greymatter.io	Not OSS	-
Istio	CNCF	OSS	33694
Kuma	CNCF	OSS	3286
Linkerd	CNCF	OSS	9881
Merbridge	CNCF	OSS	666
Nasp	Cisco	OSS	24
Open Service Mesh	CNCF	OSS	2589
OpenSergo	Alibaba Cloud	OSS	663
Sermant	Huawei	OSS	651
Service Mesh Interface	CNCF	OSS	1045
Service Mesh Performance	CNCF	OSS	259
Slime	NetEase	OSS	401
Tetrate Service Bridge	Tetrate	Not OSS	-
Traefik Mesh	Traefik Labs	OSS	1915

Table 9: The aggregated data of twenty-one projects listed in the category service mesh in the CNCF Cloud Native Interactive Landscape

In order to analyze this data, it is first loaded into R[5], then a Q-Q Plot[Dod08][p. 437-439] plot is created in order to verify the distribution of the GitHub stars for the service mesh projects. In this Q-Q plot, the distribution of GitHub star data points is compared against the normal distribution of the data. We are interested in data points that heavily diverge from the normal distribution. The source code for Figure 10 and Table 10 on the next page is available in Listing 1.

[3]Data in the table from 2023-11-09T02:46:23
[4]`https://github.com/`
[5]`https://www.r-project.org/`

```
# Create Q-Q Plot
    x = c(17, 24, 259, 401, 491, 651, 663, 666, 694, 1045,
       1915, 2589, 3286, 9881, 26921, 33694)
    qqnorm(x)
    qqline(x, col=2, lwd=2, lty=2)
# Calculate Quantiles
    quantile(x, seq(0.1, 1, by=.15))
```

Listing 1: R-Code used for analyzing OSS projects GitHub star data

The generated Q-Q Plot is seen in Figure 10. Three data points strongly diverge from the normal distribution of data and are thus considered projects of interest for comparing their responses to the requirements of Chapter 4.1. Those three data points respond to Istio, Consul, and Linkerd and are labeled for identification in the Figure.

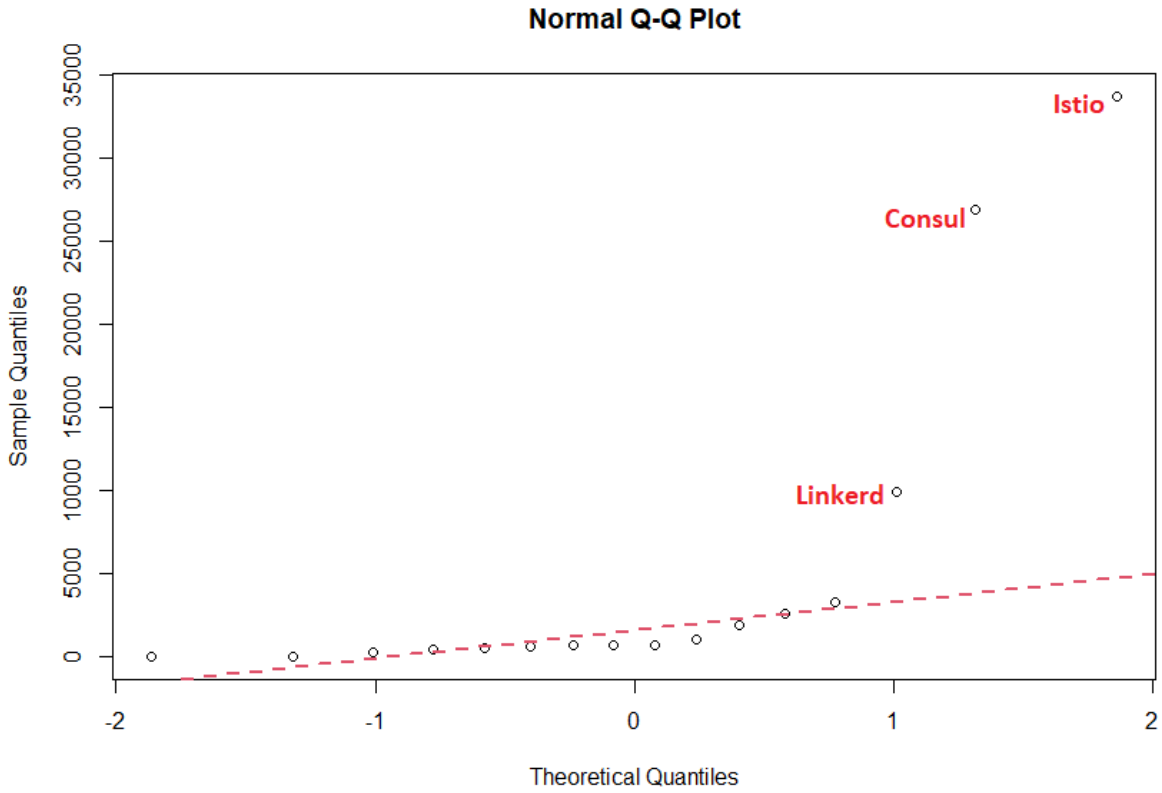

Figure 10: Q-Q plot of GitHub star data of OSS service mesh projects[6]

Percentile	10%	25%	40%	55%	70%	85%	100%
Data Point	141.50	468.50	663.00	781.75	2252.00	8232.25	33694.00

Table 10: Percentiles of GitHub stars of OSS projects

Table 10 shows the results of calculating percentiles by increasing the value of the percentile for each data point by 15 percent. It is observed that the cut-off point for the upper 15th percentile is 8232.25 GitHub stars for all service mesh projects in the CNCF Cloud Native Landscape. This finding indicates that Linkerd, Consul, and Istio represent the upper 15th percentile of service mesh projects when measured by GitHub

[6]Generated with source code from Listing 1

star popularity. As per the previously defined steps, this means that of the twenty-one service mesh projects, the following four candidates will be subjected to the in Chapter 4.1 defined requirements:

- Linkerd[7] (Upper 15th percentile of service mesh projects by GitHub stars)
- Istio[8] (Upper 15th percentile of service mesh projects by GitHub stars)
- Consul[9] (Upper 15th percentile of service mesh projects by GitHub stars)
- AWS App Mesh[10] (Non-OSS project with strong integration into AWS due to Amazon)

The final process of selecting a service mesh project that is to be used for the case study goes as follows:

- Compare the four candidates' ability to satisfy the functional requirements *FR1..FR5* from Chapter 4.1. Eliminate candidates that do not have sufficient features to support the functional requirements.
- Compare the remaining candidates' ability to satisfy the non-functional requirements *NFR1..NFR8* from Chapter 4.1. Eliminate candidates who do not have sufficient prerequisites for at least two non-functional requirements.
- Out of the remaining candidates, select one service mesh project for the case study. If multiple possible candidates exist, considerations of which candidate appears to be most sensible for the case study compared to the others must be made.

The results of the first step of the selection process are seen in Table 11. For each functional requirement and corresponding service mesh feature, it has been checked if the service mesh projects include the required feature in their offered functionality.

FR	**Feature**	**Linkerd**	**Istio**	**Consul**	**App Mesh**
1	mTLS Encryption	Yes[Linb]	Yes[Istf]	Yes[Cona]	Yes[AWSc]
2	Certificate Management	Yes[Linc]	Yes[Istd]	Yes[Cond]	Yes[AWSc]
3	Service-to-Service Authorization Policies	Yes[Linn]	Yes[Istf]	Yes[Conc]	***No***[AWSb]
4	Circuit Breaking	Yes[Line]	Yes[Istb]	Yes[Conb]	Yes[AWSa]
5	Fault Injection	Yes[Link]	Yes[Istc]	***No***	***No***

Table 11: Comparison of service mesh candidates regarding their ability to satisfy the functional requirements

Table 11 shows that while Linkerd and Istio satisfy all five functional requirements derived from the scenarios *S1..S5*, Consul does not satisfy fault injection functionalities, while AWS App Mesh on top of that does not support service-to-service authorization features. This eliminates Consul and AWS App Mesh from the considerable service

[7] `https://linkerd.io/`
[8] `https://istio.io/`
[9] `https://www.consul.io/`
[10] `https://aws.amazon.com/app-mesh/`

mesh candidates pool. As the second step, the assessment of the non-functional requirements *NFR1..NFR8* of Linkerd and Istio is done in Table 12 and 13.

Table 12: Assessing the non-functional requirements of Linkerd

Non-functional Requirements	**Linkerd**
NFR1 Technical Maturity	Linkerd has achieved CNCF graduated status in 2021[CNCa]. It is the first service mesh project listed by the CNCF that has archived graduated status. Linkerd lists adopters such as Xbox, Adidas, and DB Schenker on their website that are using it productively[11]. There have been three security audits in 2019, 2020 and 2022[CNCb].
NFR2 Compatibility	Linkerd is compatible with Kubernetes environments[CNCb]. All TCP protocols are supported, including HTTP and raw TCP protocols. Some server-first protocols require additional configuration for mTLS support[Lini].
NFR3 Maintainability	Linkerd upgrades are done separately and are broken down into CLI, control plane, control plane extensions, and data plane upgrades. Utilizing Kubernetes rollouts, zero downtime upgrades are made possible. The CLI offers a check mechanism that can verify the success of the upgrade[Lino].
NFR4 Configurability	Linkerd provides configuration management via Kubernetes Labels. Resources are annotated to persist configuration changes. Linkerd introduces its own CRDs on installment that act as configuration building blocks. The Linkerd CLI[Linf] offers commands such as `inject` that offer automatic injection of configuration values into Kubernetes YAML files. Linkerd offers extensive configuration documentation with examples on its website.
NFR5 Usability	Linkerd offers a CLI[Linf] for administrative purposes. The viz[Ling] extension provides observability and visualization components and offers a graphical overview of the meshed services.
NFR6 Portability	Linkerd can be deployed to any cloud-hosted or on-premise Kubernetes environments[Linl].
NFR7 Cost	Linkerd is free to use. An enterprise version exists with Buoyant[12].
NFR8 Licensability	Linkerd is licensed under the Apache License 2.0[CNCb] and satisfies OpenSSF best practices[Mora].

Table 13: Assessing the non-functional requirements of Istio

Non-functional Requirements	**Istio**
NFR1 Technical Maturity	Istio has achieved CNCF graduated status in 2023[CNCa] and lists adopters such as Airbnb, eBay, and Atlassian on their website[13] that are using it productively. There has been one security audit in 2023[CNCb].

NFR2 Compatibility	Istio is compatible with Kubernetes environments[CNCb]. All TCP protocols are supported. This includes HTTP and raw TCP protocols. Some server-first protocols require additional configuration[Iste] for mTLS support.
NFR3 Maintainability	Istio is upgradable through either an in-place or canary upgrade[Istg]. Canary upgrades are the recommended upgrade methods and are deemed safer than in-place upgrades. Utilizing Kubernetes rollouts, zero-downtime upgrades are made possible.
NFR4 Configurability	Istio supplies its own CRDs for configuration. Some configuration values can also be applied through Labels. The Istio CLI[Isth] also aids users by injecting configuration values into Kubernetes YAML files. Istio offers extensive configuration documentation with examples on its website.
NFR5 Usability	Istio offers a CLI[Isth] for administrative purposes. A graphical representation of the service mesh can be created with the addition of Kiali[Isti].
NFR6 Portability	Istio can be deployed to any cloud-hosted or on-premise Kubernetes environments[Ista].
NFR7 Cost	Istio is free to use.
NFR8 Licensability	Istio is licensed under the Apache License 2.0[CNCb] and satisfies OpenSSF best practices[Raj].

With Linkerd and Istio having their non-functional requirements assessed, their assessment results are compared in Table 14. L stands for Linkerd and I for Istio; in the case that a non-functional requirement is satisfied to a comparable extent, this is expressed with $L = I$. The $>$ and $<$ operators symbolize the case where one project satisfies a non-functional requirement to a more matching extent.

NFR	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
vs	$\boldsymbol{L > I}$	$L = I$	$\boldsymbol{L < I}$	$L = I$	$\boldsymbol{L > I}$	$L = I$	$L = I$	$L = I$

Table 14: Comparison of Linkerd vs. Istio responses to non-functional requirements

In five instances, the two projects can fully satisfy the non-functional requirements and are considered comparable. In the instance of *NFR3: Maintainability*, Istio has deemed a better match; this stems from the fact that Istio's upgrade mechanism offers a safer way of upgrading through the canary deployment than Linkerd's in-place method. Regarding *NFR1: Technical Maturity*, Linkerd is considered a better match; it is the first service mesh project that the CNCF has granted graduated status and has held that status for a longer time. While Istio's security audit is more current than Linkerd's, Linkerd has generally had more audits. Linkerd is also considered a better fit concerning *NFR:5 Usability* as it offers an out-of-the-box dashboard with its viz add-on. The dashboard makes Linkerd a more suitable candidate than Istio for fulfilling the non-functional requirements. Linkerd also provides faster performance and a smaller resource fingerprint than Istio. Due to Linkerd's sidecar proxy being written in

[11] `https://linkerd.io/community/adopters/`
[12] `https://buoyant.io/`
[13] `https://istio.io/latest/about/case-studies/`

Rust compared to Istio's C++-written Envoy proxy, an entire class of memory-related *Common Vulnerabilities and Exposures* (CVEs) is being avoided due to Rust being a memory-safe language[Buo]. Istio provides extensive features for complex architectures and advanced use cases. At the same time, Linkerd stands out for its simplicity, security, and performance, making it an attractive option for lightweight and straightforward service mesh needs. For this reason, Linkerd is selected as the service mesh project of choice for the case study.

4.3 Investigating Service Mesh Application Scenarios for HAFAS

Selecting Linkerd for the service mesh of choice for the case study in Chapter 4.4 now requires a more in-depth look into the scenarios *S1..S5* identified in Chapter 3.3. Each scenario receives its sub-chapter investigating its functionalities. With the service mesh of choice for the case study selected, the technical details of each scenario will relate to Linkerd. The functionalities and background are covered for each scenario, and a short assessment of how this feature could be used with HAFAS is carried out. As per *NG2: Introducing a Service Mesh to HAFAS that is Ready for Productive Use*, this is not an in-depth analysis of the feasibility and usefulness of each scenario for HAFAS. The primary motivation is to identify high-priority scenarios that will be investigated further in the case study to create an early proof of concept for introducing a service mesh to HAFAS.

4.3.1 S1: Data in Transit Encryption with mTLS

Data in transit encryption with mTLS is a security feature of a service mesh that ensures high confidentiality and integrity in service-to-services communication[Fly23]. Unlike traditional TLS, mTLS requires both the client and the server to present valid certificates, establishing a mutual trust relationship. This two-way authentication helps prevent unauthorized access and man-in-the-middle attacks within the service mesh. By encrypting data in transit, mTLS adds an extra layer of protection, making it significantly harder for malicious actors to intercept or tamper with sensitive information exchanged between services, thereby enhancing the overall security of the distributed system. Figure 11 highlights the difference between a distributed system devoid of mTLS encryption and one deploying mTLS in its service-to-service communication.

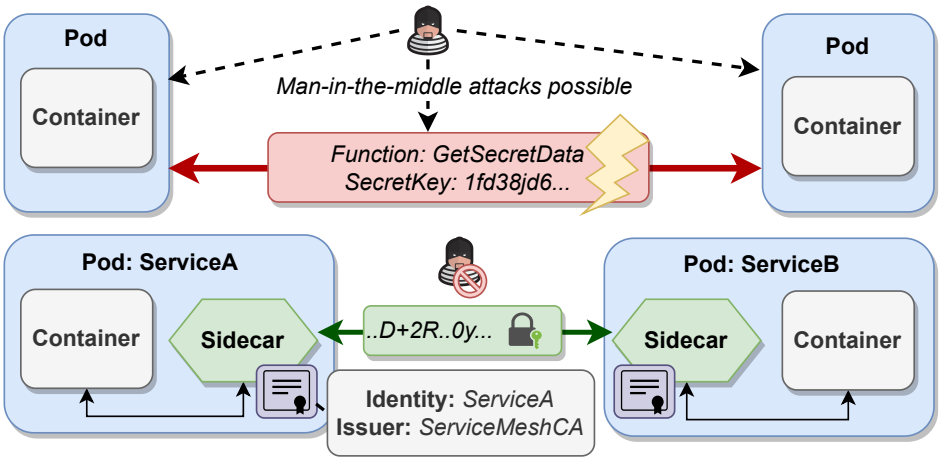

Figure 11: Data in transit encryption with mTLS in action (own illustration)

The above example in Figure 11 describes a system lacking the security properties of confidentiality, integrity, and authenticity. The lack of confidentiality allows a man-in-the-middle attacker to eavesdrop on the data in transit. The absence of integrity permits the attacker to modify data in transit. Moreover, without authenticity, the attacker can pretend to be an otherwise trusted service and conduct restricted operations that way. TLS certificates provide a critical component of mTLS encryption. Each certificate identifies a subject and is certified to be that subject by an issuing entity. TLS in a client-server scenario on the web only involves the client verifying the server's certificate; the server does not verify a client's certificate as they provide none. mTLS stands for mutual TLS and requires the additional verification of a client compared to TLS. In the lower example of Figure 11, *ServiceA* opens up a connection to *ServiceB*, and they conduct a key exchange to set up an encrypted tunnel. Then, they both exchange and verify each other's end-entity certificates. Upon completing this validation step, the services can begin exchanging encrypted data. mTLS thus adds the security properties of confidentiality, integrity, and authenticity to the service-to-service communication of the distributed system. Authenticating both communication participants allows authorization, a building block for zero trust functionalities. Kubernetes does not offer an automated solution for implementing mTLS encryption; manual ways involving adding mTLS capabilities per service codebase are cumbersome. A service mesh solves this problem with the sidecar proxy approach. Figure 12 showcases the architecture of Linkerd. The identity service of the control plane handles certificate signing. It acts as a TLS Certificate Authority, issuing end-entity certificates to the proxies at proxy initialization time to implement mTLS encryption.

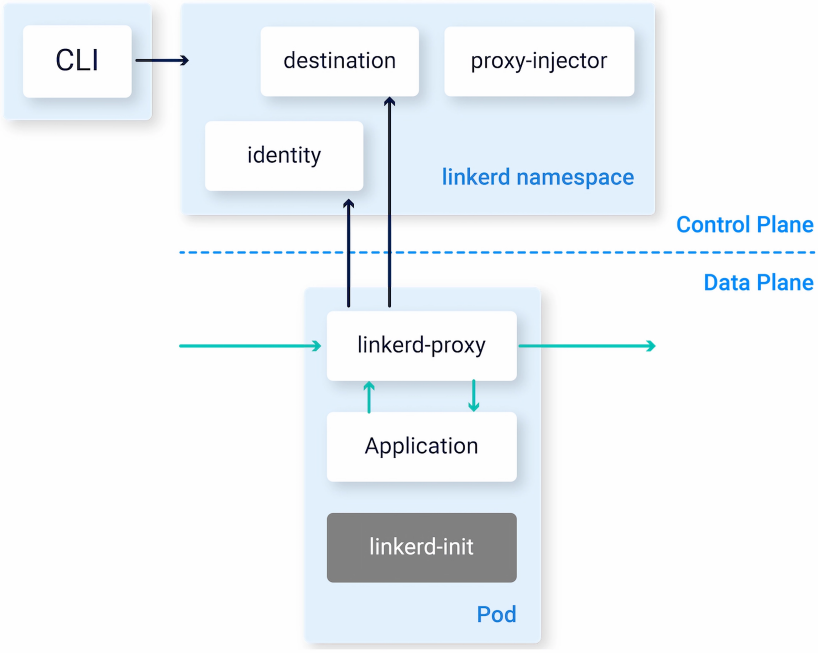

Figure 12: Architecture of Linkerd[14]

Deriving end-entity certificates in Linkerd works by including the Kubernetes `Service Account` token as a unique identification point for a workload's identity. Linkerd deploys a two-layer hierarchy of trust: The identity service from Figure 12 handles issuing and rotating certificates to workloads. Linkerd's trust anchor handles the issuing of

[14] [Lina]

the certificate for the identity service. Rotating end-entity certificates is automated, while rotation of the trust anchor is a manual task for the service mesh operator. More on that will be handled in Chapter 4.3.2 as this is related to Scenario *S2: Certificate Management for Data and Control Plane*. Linkerd enables mTLS encryption between services as a default setting for all meshed services. There is, however, no enforcement of mTLS encryption between meshed and non-meshed services, meaning that workloads in the service mesh are accessible by non-mTLS connections by default unless explicitly defined by the authorization policies of Scenario *S3* discussed in Chapter 4.3.3. Linkerd's mTLS feature supports routing any TCP-based traffic. Linkerd, however, also relies on a feature called protocol detection[Pre21]. Protocol detection allows Linkerd to detect the protocol used in a TCP connection automatically. Determining the protocol is done by inspecting traffic between the connected services. This feature is in place to avoid having the user define the protocol manually for each port. The information gathered by the used protocol aids Linkerd in load balancing and gathering observability metrics. While Linkerd can only balance raw TCP connections on a connection level, it can advance this to the request level once HTTP is involved. Protocol detection fails as soon as *server-speak-first* protocols are involved. These protocols work by having the client establish a connection and then waiting for the server to respond. Linkerd is, however, only able to detect the protocol with the server's response. It means it cannot establish an mTLS tunnel with only the client's request as a source of information. The concept of opaque ports has been introduced; it allows the service mesh operator to mark ports used by such protocols as opaque with a specific annotation at the workload level in Kubernetes. Linkerd will then skip protocol detection for the protocols on these ports and continue routing traffic through an mTLS-encrypted connection. Before introducing opaque ports, the only way of bypassing this was by excluding the involved ports from the sidecar proxy and establishing the connection directly from container to container. This approach made it impossible to use mTLS encryption with protocols that failed the protocol detection.

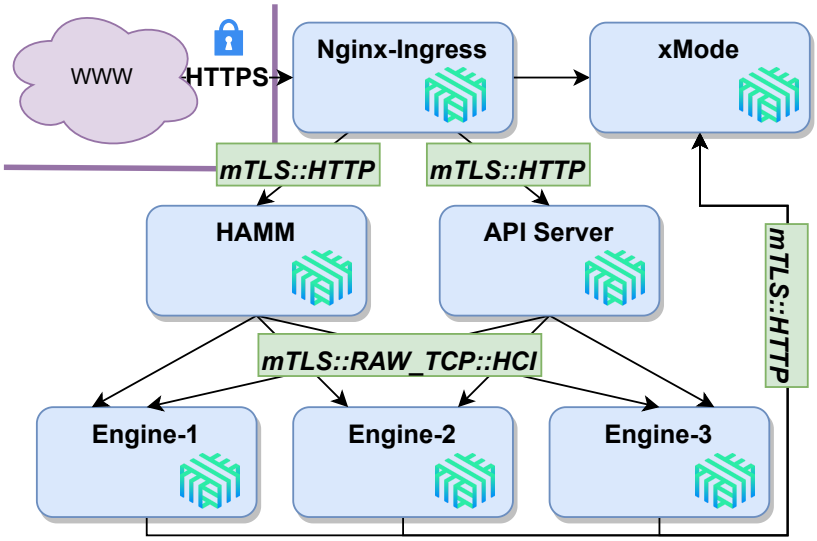

Figure 13: Introducing mTLS encryption to HAFAS (own illustration)

Introducing mTLS encryption to a HAFAS environment could look like in the graphical illustration of Figure 13. Linkerd's mTLS feature encrypts all communication between the meshed services in the Kubernetes cluster. As the sales platform portion of the MaaS platform evolves, personalized user data will likely enter the system. Introducing

the security features of mTLS to HAFAS will thus become an essential factor in securing the platform. The two protocols that play a role in this are HTTP and HCI[15], both supported by Linkerd for mTLS encryption. Additionally, mTLS sets the foundation for introducing zero trust network security via authorization policies to HAFAS, which further helps isolate the systems in multi-tenant Kubernetes environments. Lastly, it must be stated that mTLS is not a catch-all solution; the added security features are beneficial in preventing eavesdropping, spoofing attacks, and data manipulation in transit. It does, however, not protect from attacks on data stored by the services or man-in-the-middle attacks on the loopback device that Linkerd uses to communicate between the proxy and workload. The mTLS encryption functionality offered by a service mesh is highly relevant for HAFAS and warrants introducing it to Kubernetes environments, given that the case study supports the technical requirements on performance and latency.

4.3.2 S2: Certificate Management for Data and Control Plane

The previous chapter discussed the scenario of data in transit encryption with mTLS and explained Linkerd's implementation. Linkerd provides its own Certificate Authority that fulfills *Certificate Signing Requests* (CSRs)[NK00] submitted by the sidecar proxies. Figure 14 visualizes the process of generating end-entity certificates for sidecar proxies in Linkerd. Kubernetes `ServiceAccount` tokens help determine the identity of the workload attached to the sidecar. Additionally, each sidecar proxy generates a private key used for signing certificates. A CSR containing the sidecar's public key and a digital signature is sent to Linkerd's Certificate Authority when the sidecar proxy generates a certificate. The Certificate Authority verifies the digital signature with the public key, ensuring the applying sidecar proxy possesses the private key, validating its identity. In the case of a verified identity, the Certificate Authority then issues and signs an end entity certificate to the sidecar proxy used for establishing mTLS connections in the service mesh.

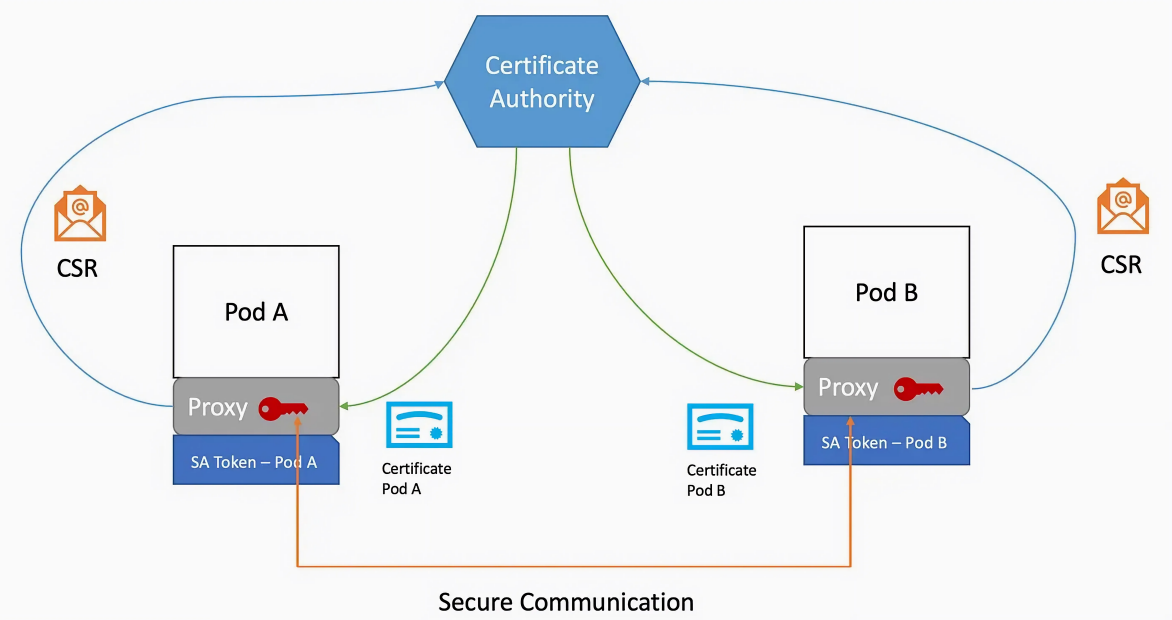

Figure 14: End-entity certificate request procedure for sidecars in Linkerd[16]

[15] As a raw TCP protocol unknown to Linkerd, HCI is a likely candidate for opaque port capabilities

Linkerd automatically rotates end-entity certificates for workloads. Rotating the identity issuer certificate of the Certificate Authority and the trust anchor used to sign the identity issuer certificate are manual tasks that the service mesh operator must manage. By default, Linkerd uses a pre-generated trust anchor upon installation that is valid for 365 days and will cause certification expiration issues once that timeframe has passed[Linm]. Independent of the trust anchor, the Certificate Authority's issuer certificate can also expire in that timeframe. It is possible to set up a certificate manager such as *cert-manager*[17] that automates the process of rotating the identity issuer certificate. Rotating an expired trust anchor involves generating a new one, bundling it with the old one, rotating the issuer certificate for the Certificate Authority, and removing the old trust anchor. Linkerd permits this trust anchor rotation procedure without downtimes. Utilizing *cert-manager*, generating a long-lasting trust anchor and short-lasting automatically rotated identity issuer certificates is possible. Linkerd thus provides optimal certificate management capabilities, as the trust anchor can be stored away safely while intermediary certificates with short validity periods are renewed frequently. The certificate management features also allow for a quick zero downtime recovery if a secret is lost or compromised. Figure 15 showcases a Linkerd certificate management architecture based on Kubernetes secret management capabilities and *cert-bot*. The trust anchor has a ten-year validity, while issuer certificates are renewed every five hours and are only valid for a maximum of 6 hours. Sidecar proxies renew their certificates every 24 hours as a default strategy. End-entity certificates are rotated every three hours to demonstrate the possible short validity of sidecar proxy certificates.

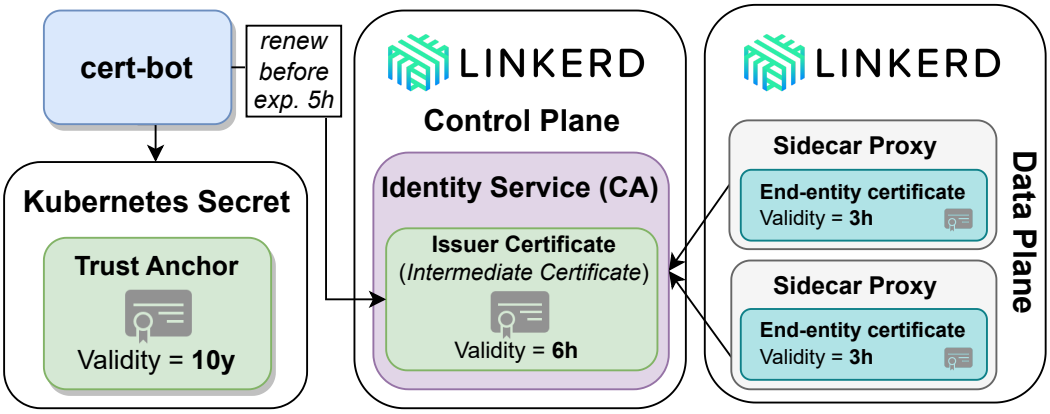

Figure 15: Example of a Linkerd certificate management architecture (own illustration)

Linkerd provides sufficient certificate management capabilities for its mTLS functionality. Automation and issuing short-lasting intermediate certificates minimize the risk of having a compromised certificate at bay and shorten the timeframe in which attacks are made possible. The revocation process of tainted certificates improves by having short living certificates[TSHJ12] that expire after a few hours. Even if the trust anchor is compromised, Linkerd provides an uncomplicated method of quickly changing it to a new one with zero downtime and a quick cascade down to the end-entity certificate level. Scenario *S1* depends on Scenario *S2* to fulfill its full potential.

[16] [Bal23]
[17] `https://cert-manager.io/`

4.3.3 S3: Service-to-Service Authorization Policies

Authorization policies in a service mesh are pivotal in governing and securing communication between services within a distributed architecture. These policies define rules and permissions that dictate which services can interact with each other and under what conditions. Leveraging *Role-based Access Control* (RBAC) and other fine-grained access management techniques, authorization policies ensure that only authenticated and authorized services can exchange data, mitigating the risk of unauthorized access or data breaches. The aspect of mTLS encryption mentioned in Chapter 4.3.1 introduced the security properties of confidentiality, integrity, and authenticity. Authorization policies extend this set of security properties by providing authorization. By implementing a robust authorization framework, service meshes enhance the overall security posture of distributed systems, safeguarding sensitive information and enforcing strict access controls between service-to-service communication. Linkerd offers authorization policies that can operate at Layer 7 of the OSI model[Ped21]. Kubernetes network policies[Kubi] operate on OSI Layer 3 and 4. Unlike Kubernetes network policies, Linkerd's authorization policies build upon the workload identities (i.e., certificates issued to workloads) provided by the mTLS encryption feature. Linkerd's network authorization policies work by restricting access to individual services and namespaces in the service mesh to all actors not defined in the authorization policy. The sidecar proxy enforces the authorization policies, which means only meshed resources are secured this way[Linn]. Authorization policies are defined as a *default policy*, enforced at the cluster, namespace, or pod level through Kubernetes annotations or a set of CRDs specifying fine-grained policies targeting specific ports and routes. The *default policy* can allow or deny unauthenticated and authenticated (meaning mTLS secured) requests on a global or cluster-wide scope. The fine-grained authorization policies offer a tighter amount of control through the definition of CRDs, which is visualized in Figure 16.

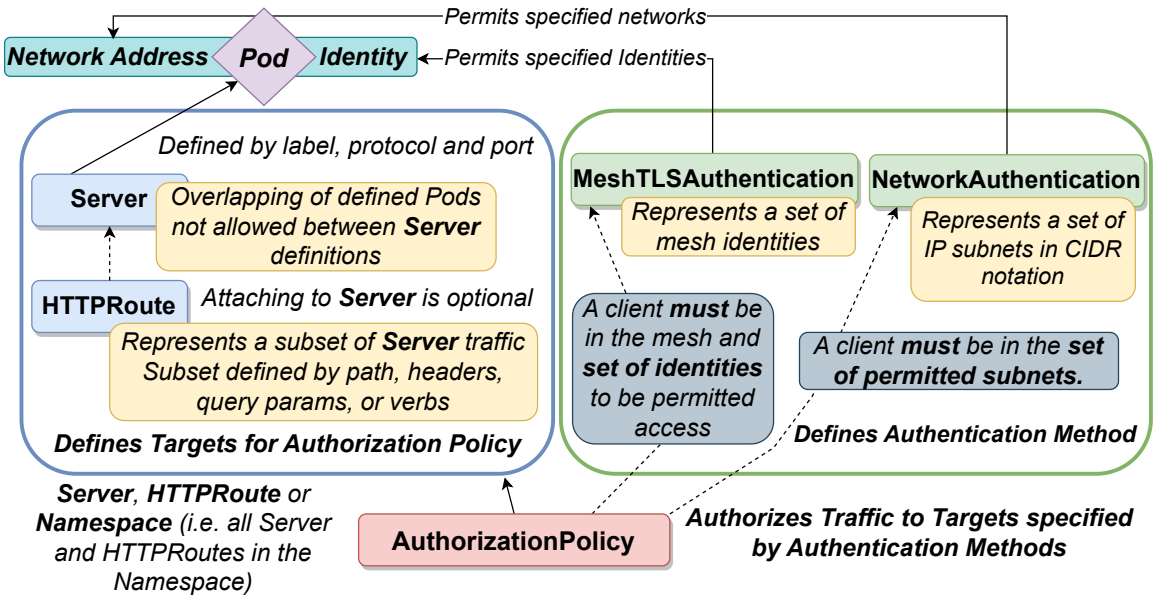

Figure 16: Overview of Linkerd's fine-grained authorization policies (own illustration)

The `AuthorizationPolicy` CRD represents the top-layer configuration block of Linkerd's fine-grained authorization policies. It references resource definition CRDs such as `Server` or `HTTPRoute`. Defining a `Server` is done by selecting one or more pods and

their respective ports and protocols to which the authorization policy should apply. `HTTPRoute` CRDs can be attached to a `Server` in case the policy should only target a subset of the `Server` CRDs HTTP traffic. This way, an authorization policy applies to a specific URL path or targets specific headers, query parameters, or HTTP request methods. This approach allows flexibility in defining the scope of an authorization policy that cannot be archived with Kubernetes Layer 3 and 4 network policies. Linkerd offers two methods of authentication. The `MeshTLSAuthentication` automatically excludes all traffic not belonging to the service mesh. A set of *Identities* that are permitted access is specified. Wildcards can be applied to allow all identities of the service mesh or limit access to specific namespaces. The `NetworkAuthentication` CRD brings the Layer 3 and 4 features of Kubernetes network policies to Linkerd. Defining a set of permitted subnets is done in the *Classless Inter-Domain Routing* (CIDR) notion. The authorization policy ensures that only requests matching the allowed subnets will make it to the pods. In the case of HAFAS, Linkerd's authorization policies could aid in improving system isolation in multi-tenant clusters. Figure 17 provides an example of such a scenario.

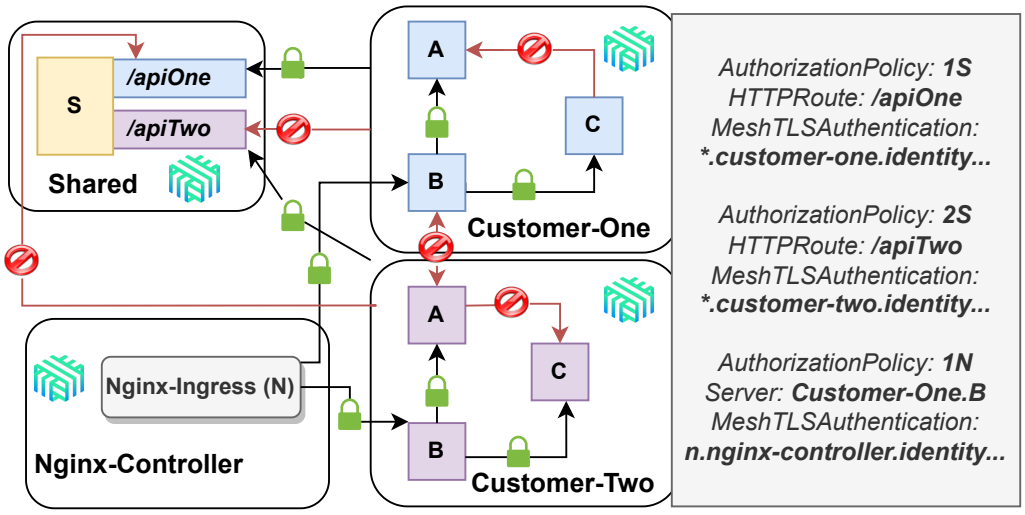

Figure 17: Example of Linkerd's authorization policies in practice (own illustration)

All namespaces have all their pods added to the Linkerd service mesh. *Customer-One* and *Customer-Two* are the namespaces of the two customer environments. In both customer namespaces, the enforcement of mTLS encryption is configured by denying all communication via a *default policy* and enabling the allowed connections via fine-grained policies. Not all services are allowed to communicate with each other within the cluster. Because service *A* of *Customer-One* contains sensitive data, only service *B* is allowed to access it. The Nginx controller that handles ingress can only connect to service *B* in both clusters. In this scenario, a shared service *S* with two API endpoints on the same pod and port exists. Both customer clusters can only access the endpoint with a matching `HTTPRoute` and `MeshTLSAuthentication` defined. The two customer namespaces are fully isolated, and the enforced mTLS encryption ensures authenticity, confidentiality, and integrity, while the authorization policies ensure authorization. Should an attacker, for example, take over Service C in cluster *Customer-One*, they would be unable to connect to service *A* and *B* as no permitting `MeshTLSAuthentication` policy exists. They would also be unable to contact any service in the cluster of *Customer-Two*. For HAFAS, the authorization policy's main benefit is improving system isolation in multi-tenant environments. Secondly, sharing

resources between customers or allowing two related transportation agencies access to a sub-functionality of their respective systems are valid scenarios. Lastly, the authorization policies provided by Linkerd [Morb] can aid organizations in implementing zero-trust policies[Fre22] for Kubernetes clusters.

4.3.4 S4: Improving Resilience with Circuit Breaking

Circuit breaking functionalities in a service mesh are critical components contributing to a distributed system's reliability and resilience[CJB23][Chapter 4]. These features act as intelligent safeguards by monitoring the health and responsiveness of individual services within a distributed system. When a service experiences performance issues or failure, the circuit breaker can dynamically interrupt the flow of requests to that service, preventing potential cascading failures across the entire application. This proactive mechanism helps to isolate and contain issues, allowing affected services to recover and maintain overall system stability in the face of varying conditions and disruptions. At this point, some software components at Hacon already have their own circuit breaking features baked in. The circuit breaking functionalities of a service mesh would thus not introduce the concept to services at Hacon. Circuit breaking functionalities of a service mesh offer more configurability. The engine has a simple integrated circuit breaker that ignores a faulty xMode and responds to requests that involve the xMode with an error instead of waiting for the timeout of the xMode for every request. The abstract concept of a circuit breaker is described in Figure 18.

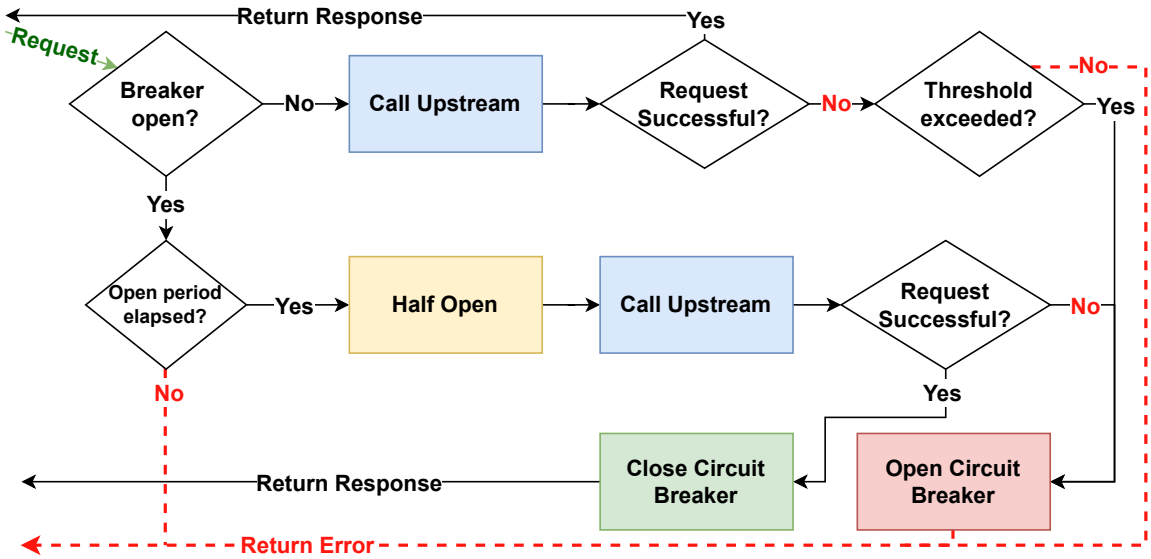

Figure 18: Abstract functionality diagram of a circuit breaker[18]

A circuit breaker is made up of an error threshold that contains the maximum of errors that are acceptable in a given timeframe from a service. The circuit breaker opens up if the error threshold exceeds its internal counter. At this point, every request that hits the circuit breaker will be replied to with an error without contacting the service behind it. After a given timeframe, the circuit breaker switches to a state of half openness in which an erroneous response from the service behind it causes the circuit breaker to reopen. In contrast, a successful response closes the circuit breaker and resets the error threshold. Linkerd implements circuit breaking functionality over

[18](Own illustration, based on [CJB23])

the sidecar proxy[Line]. Currently, only HTTP requests are supported, which excludes its functionality from all HCI-based services as their protocol is raw TCP. The HTTP status codes identify failures; at the moment, Linkerd classifies every 5XX code as an error. It is impossible to define custom error codes, which is problematic when a service responds with 401 Unauthorized if a misconfiguration with API access is encountered. This edge case would not trigger the circuit breaker in a Linkerd service mesh. If the error threshold of a service is exceeded, the circuit breaker is activated, and the service is marked as unavailable. The load balancer ignores services that are marked as unavailable. In the case that all services of a kind are unavailable, a 503 Service Unavailable error is forwarded to callers. The only failure accrual policy available for Linkerd works by counting consecutive failures. If an endpoint fails five times consecutively and the error threshold is set to five, the circuit breaker will be activated. This policy is inconvenient if a service fails in a zig-zagging pattern where enough calls succeed and never reach the consecutive threshold, but still, enough failures occur to slow down the system. The only way to encounter this would be to set a low consecutive threshold, like one or two failures in succession. A policy that allows tracking failures over a timeframe would be more suitable in that case; however, Linkerd does not currently support it. Once the circuit breaker opens and an endpoint is deemed unavailable, it enters a state of probation after a fixed backoff period. This probation period works analog to the half-open circuit breaker described in Figure 18. After the backoff period, a probe request is allowed to reach the service. If the probe request succeeds, the service is marked available again. Failing the probe requests results in a new backoff period, which raises exponentially on consecutive probe failures until it hits a maximum ceiling.

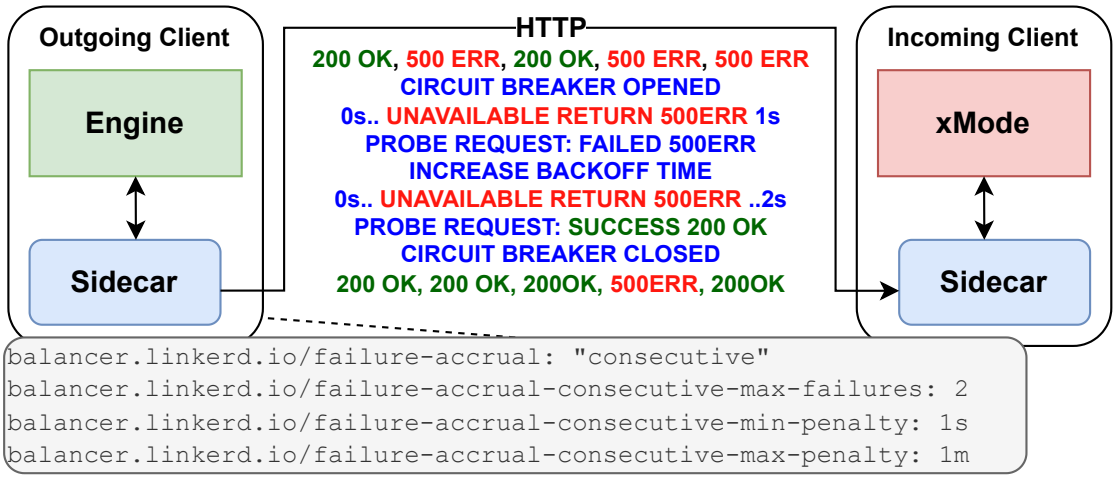

Figure 19: Example of a circuit breaker involving engine and xMode (own illustration)

Figure 19 showcases the scenario of a Linkerd circuit breaker configured for the engine with `max-failures` of 2 and a penalty between `1s` and `1m`. It communicates with the xMode, and the circuit breaker opens up after two consecutive HTTP 500 errors. It takes two probation periods until the circuit breaker closes again. The consecutive requests do not trigger the circuit breaker again. In principle, managing circuit breakers by a service mesh offers a valuable addition to HAFAS. Due to some components already having implemented basic circuit breaking functionalities, it is not the main reason for considering a service mesh for HAFAS, however. Suppose Hacon chooses to roll out a service mesh; the developers of components with partial circuit breaking functionalities should consult with hosting and operations if they are more comfortable

maintaining their service-scoped implementation or are ready to outsource the responsibility to the service mesh. Due to technicalities, only one xMode is deployed per HAFAS environment; removing those technicalities paired with circuit breaking would drastically improve HAFAS's resilience. The missing support of raw TCP is considered a negative due to the resulting feature incompatibility with HCI services.

4.3.5 S5: Testing Robustness with Fault Injection

Fault injection is a chaos engineering technique[Ros20]. Fault injection features[Link] refers to the deliberate introduction of faults or failures into a distributed system to assess its robustness and identify potential weaknesses. Service meshes provide tools for controlled fault injection, allowing developers to simulate network errors, delays, or service unavailability scenarios. By systematically testing how a system responds to such disruptions, teams can better understand and improve their application's reliability and fault tolerance. Looking at HAFAS, there are currently no scenarios where fault injection is helpful in an operative context. The expert interview has identified its use in development and testing. Because these fault injection features are an upcoming feature, they might see some benefit if Hacon decides to introduce service meshes to their regular technology stack. Figure 20 describes fault injection in practice. In the pictured scenario, the engine communicates with an xMode instance, possibly to request routing options related to shared mobility services. Both engine and xMode have been integrated into a service mesh and can only communicate over their respective sidecar proxies. The protocol is HTTP; this is an essential detail as Linkerd, for example, does not appear to support raw TCP protocols. The lack of support for raw TCP protocols means this feature is incompatible with HCI-based services. In Figure 20, the two faults introduced are an HTTP status code fault that results in 50 % of the requests to the xMode being answered with the error code 500 and a 6s delay that occurs in 10 % of the requests simulating an outage of the xMode that could cause configured circuit breakers to fire.

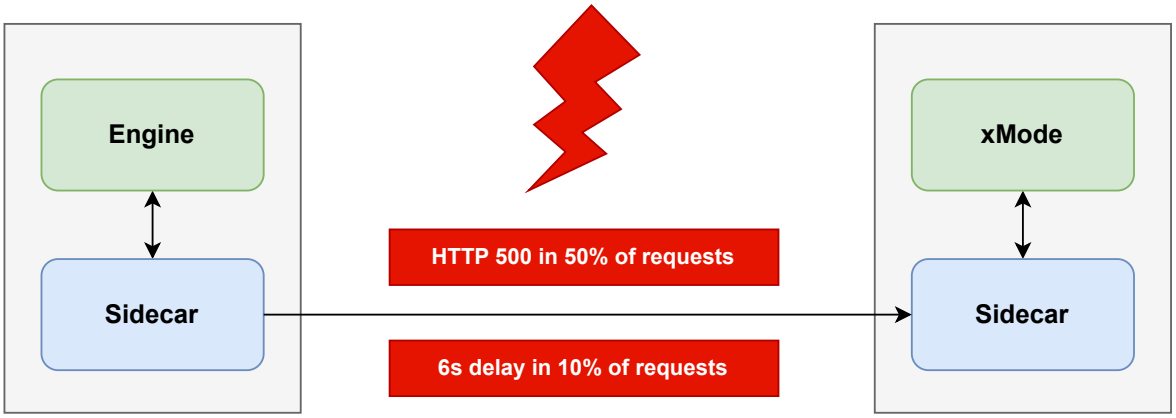

Figure 20: Example of using fault injection features with HAFAS (own illustration)

Utilizing this feature of a service mesh allows for quick and easy testing and debugging of HTTP service-to-service communication for developers and testers. If chaos engineering practices are applied to HAFAS, the service mesh will allow it to handle all configuration aspects efficiently. Linkerd's fault injection capabilities support timeouts, custom payloads, and error codes. The configuration is done by setting up a customized nginx container that specifies the faulty HTTP endpoints. Using one

of Linkerd's CRDs, the configuration of weights and redirection to the faulty service is done with a customized `HTTPRoute`-CRD[Linj]. Timeouts can be specified with a specialized configuration parameter. This scenario is not explored further, as fault injection is considered a bonus for testing and development if a service mesh is to be permanently introduced to the HAFAS technology stack.

4.4 Defining the Foundation of the Case Study

With the five scenarios (*S1..S5*) from Chapter 3.3 investigated in Chapter 4.3, this chapter will now lay the groundwork for the practical case study as mentioned in research method *M6*. Firstly, out of the five scenarios *S1..S5*, a subset of prioritized scenarios of interest for the case study is defined in Chapter *scenofinterest*. Implementing all scenarios in the practical case study is deemed out of scope and not detrimental to answering the research assignment. The scenarios considered to have the highest priority by the expert interview from research method *M3* will be picked. This approach allows collecting initial data to analyze the impacts of introducing a service mesh to HAFAS. Introducing all cases to the practical case study would also require collecting this data between introducing each scenario to eliminate possible regressions due to features not working well together as per the non-goals, *NG1: Evaluating every single feature offered by a service mesh*, *NG2: Introducing a Service Mesh to HAFAS that is Ready for Productive Use* and *NG4: Providing an in-depth cost and performance analysis for introducing a service mesh to an already existing distributed system* implementing all scenarios in the case study has been ruled out. Introducing a subset of scenarios is deemed sufficient for a first impression of how HAFAS fares by adding a service mesh. Once the scenarios of interest are established, the HAFAS environment initiated for conducting the case study is described in Chapter 4.4.2. To minimize technical complications and as per the non-goals *NG2: Introducing a service mesh to HAFAS production environments* and *NG3: Providing a detailed framework for introducing a service mesh to an already existing distributed system*, a smaller and scaled-down HAFAS environment is used that does not include all the features discussed in Chapter 3.2. The HAFAS environment of the case study can not be compared to a productive environment. This is a conscious decision backed by the idea that the case study only serves as an early proof of concept for studying the initial feasibility. Lastly, Chapter 4.4.3 discusses how the resulting data of the case study is collected and analyzed.

4.4.1 Selecting Scenarios of Interest

Of the five scenarios *S1..S5* identified and investigated in Chapter 4.3, *S1: Data in Transit Encryption with mTLS* strikes out as the scenario to prioritize in the case study. As of the expert interview, it has been established that introducing a service mesh to HAFAS is primarily motivated by the requirement for mTLS encryption in service-to-service communication. For this reason, *S1* has been selected as the scenario that is focused on in the case study. Scenario *S2: Certificate Management for Data and Control Plane* is closely related to *S1*, yet primarily focuses on the administrative aspects of the service mesh that deals with pre-conditions that are required to provide mTLS encryption for service-to-service communication. Because of this, the certificate management functionalities of Linkerd will also be included in the case study. Scenario *S3* is more of a theoretical scenario of use at this point and is thus not investigated in the case study. Some HAFAS components already provide circuit breaking functionalities;

for this reason, the case study will also not include *S4*. If service meshes are to be seriously considered by Hacon, it might be interesting if the circuit breaking features of a service mesh offer advantages over the existing solution. Lastly, Scenario *S5* has been identified as primarily related to software testing and development. Because of the missing operational focus, it is also not seen as a priority for the case study. It will likely be investigated further if Hacon advances its rollout of service meshes for HAFAS in the future.

4.4.2 Defining the HAFAS Environment

As mentioned in Chapter 4.4 the HAFAS environment in the case study will not feature all software components described in Chapter 3.2. It represents a HAFAS environment that provides basic trip searching and API capabilities. Any real-time, external mobility provider importing and pedestrian routing capabilities are absent from this basic HAFAS environment. This is motivated by the fact that trip-searching capabilities are the most important aspect of any HAFAS environment, and introducing a service mesh to a complex HAFAS environment is out of the scope of this thesis. Figure 21 shows the proposed basic HAFAS environment that will be set up for the case study.

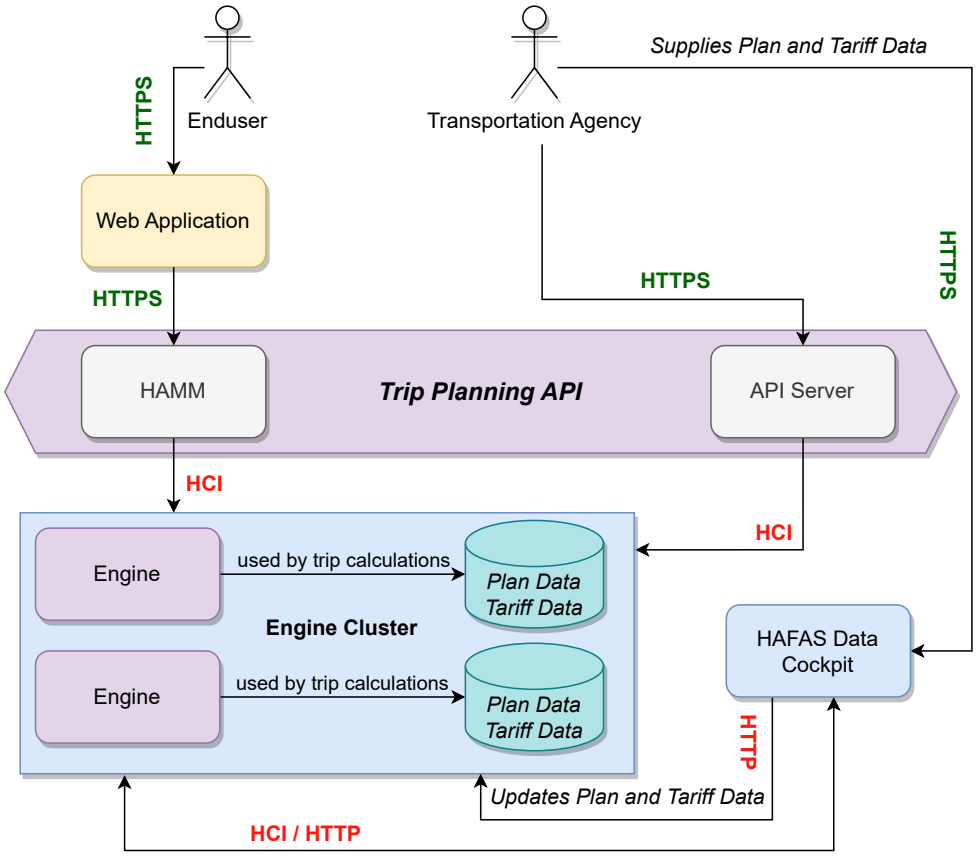

Figure 21: Architectural overview of the case studies HAFAS environment (own Illustration)

The basic HAFAS environment for the case study consists of the following software components:

- The **web application** provides a front-end that connects to a **HAMM** that routes the requests to the backend. End-users can use the web application to perform trip calculations.

- An **API Server** provides external API access to the trip-searching capabilities of the engine, which simulates a common scenario where transportation agencies rely on the API Server to integrate HAFAS functionalities into their services. The API Server also provides an excellent way to perform load tests on the whole system, which will be specified further in Chapter 4.4.3.
- A group of two **engines** provides the backend capabilities required to perform trip calculations.
- To supply the engines with plan and tariff data, a **HAFAS Data Cockpit** handles data distribution to the engines.

This basic HAFAS environment allows trip searches targeting the railway network. It does not include buses, airplanes, ferries, trams, metros, electric scooters, taxis, or ride-hail services. The routing is station-to-station, meaning that trips must start and end at a railway station included in the plan data set. In this HAFAS environment, combining different means of transportation is impossible, and access to searchable routes is limited to the railway stations from the data set. Configuring an HAFAS environment from the ground up is a complex task outside this thesis's scope. For this reason, the engine configurations and plan data are taken from a small transportation agency in central Germany. Basic API capabilities are provided through the API Manager, which is configured in a standardized manner. The HDC is configured so that plan data has to be uploaded and registered through the user interface instead of being supplied via network storage options and then distributed to engine instances that report missing plan data. The plan data that is used covers the range from *2022-12-22* till *2023-10-14* and is considered expired, which is no problem for testing purposes, as trip planning requests can also target past dates.

4.4.3 Collection and Analysis of Data

The collection and analysis of data are necessary to fulfill the research assignment and answer research question *Q4: How do the complexity and overhead of a service mesh impact performance and allocation of resources?* In the expert interview of research method *M3*, it has been established that an additional resource overhead of ten percent would be deemed unacceptable for Hacon. In order to generate data, a scenario must be created in which load is generated on the HAFAS environment of the case study for which the API Manager will be used.

API Server Features		
Arrival board	GeoFeature by Coordinate	Departure board
Data info	GeoFeature in a Bounding Box	Interval search
GIS Route by Context	Location search by coordinate	Reconstruction
Journey Detail	Location search by name	Trip search
Location Details	Location search in a bounding box	

Table 15: List of features provided by an API Server with a standard configuration

The offered features of the API Server will be used to generate this load. Table 15 lists the standard features of the API server. Of these fourteen API methods, two are particularly interesting for load testing. These are *Trip search* and *Location search by name*. *Trip search* uses the route search options of the engine; for complex routes, a

lengthy search process is started, which can sometimes take up to one second. The engine uses many CPU resources to calculate the route. If clients request several route calculations in parallel, the load increases even more. Therefore, this resource-intensive API method is only used when searching for connections. *Location search by name* is an API method that allows you to search for locations in the plan data. Such locations are, for example, train stations to which the user should travel. The web application uses this method to update user information. Therefore, it is a non-resource-intensive method processed within a few milliseconds and used frequently by a user when utilizing HAFAS. The procedure for data collection is, therefore, as follows. The necessary data relates to the latency of requests within HAFAS. Once the service mesh is introduced to the case studies HAFAS environment, the latency data for the two API methods is collected once with and without the service mesh. The two API methods will be examined separately. A simple connection and a longer and more complex connection are also selected for *Trip search*. The load of the requests is increased within the test to observe how an increase in load affects resource consumption and latencies. *Trip search* simulates resource-intensive use of services. In contrast, *Location search by name* ensures a lot of network traffic, and paying attention to the latency time is particularly important. As the HAFAS environment for the case study will be deployed to one of Hacon's EKS clusters for development purposes, a degree of isolation is required to prevent falsification of results. For this reason, all involved pods will be allocated to a particular set of nodes dedicated to the case studies HAFAS environment. Finally, the resources allocated by the service mesh to provide the data plane and the footprint of the sidecar proxies are also evaluated. The data is to be collected in Chapter 5.3 and then analyzed using the statistics software R[19] in Chapter 6.

4.5 Conclusion

To summarize this chapter: In chapter 4.1, five functional requirements have been defined that map the scenarios *S1..S5* from chapter 3.3 to a feature of a service mesh. In addition, eight non-functional requirements were established, which place additional demands on a service mesh that must be fulfilled in the context of HAFAS and the case study. In chapter 4.2, a selection process was conducted to find a suitable project for the case study from the 21 projects listed in the CNCF Cloud Native Landscape. Once the list of 21 candidates had been narrowed down to four candidates: Linkerd, Istio, AWS App Mesh, and Consul, Linkerd and Istio were compared regarding non-functional requirements. AWS App Mesh and Consul were removed from the pool of candidates due to not meeting all functional requirements. Linkerd was able to stand out from Istio as a suitable candidate primarily because it displays a higher level of technical maturity. In chapter 4.3, the five scenarios were examined from a Linkerd-centered perspective. For the case study, it emerged in chapter 4.4 that scenario S1 is considered a scenario of interest. As scenario *S2* is very close to *S1*, Scenario *S2* is also included in the scenarios of interest for the case study. The scenarios *S3..S5* would go beyond the scope of the case study and are therefore not considered further. In chapter 4.4.2, a HAFAS configuration focusing on the basic features of route calculation emerged as the optimal HAFAS environment for the case study. With the plan for collecting and evaluating data described in chapter 4.4.3, nothing in terms of planning stands in the way of the practical part of this thesis in chapter 5.

[19] `https://www.r-project.org/`

5 Solution

With the conceptional work completed in Chapter 4, this Chapter focuses on conducting the case study and load tests to generate data aiding in carrying out the research assignment and answering the research questions *Q1..Q5*. Chapter 5 describes setting up the HAFAS environment discussed in Chapter 4.4.2. Chapter 5.2 then focuses on introducing Linkerd to said HAFAS environment. Chapter 5.2.1 describes setting up the control plane and data plane. The Chapter also looks into the observability features that Linkerd introduces, such as the dashboard or Prometheus[1]. Chapter 5.2.2 contains validating the mTLS data in transit encryption for service-to-service communication of Scenario *S1* described in Chapter 4.3.1. Chapter 5.2.3 deals with setting up the automated certificate management features of Scenario *S2* described in Chapter 4.3.2. Finally, Chapter 5.3 focuses on load testing utilizing the API server's location search and trip search capabilities.

5.1 Setting up the HAFAS Environment for the Case Study

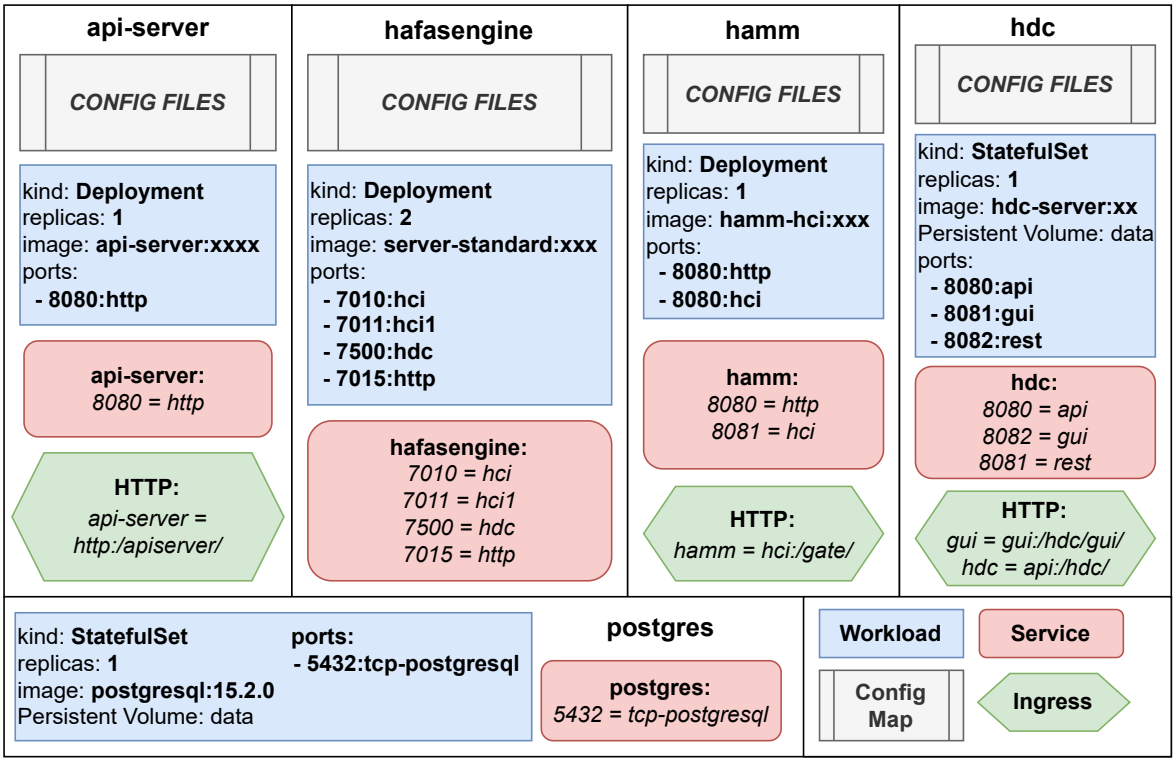

Figure 22: Kubernetes configuration for the HAFAS environment (own illustration)

Configuring a HAFAS environment from scratch exposes one to complex component configurations resulting from HAFAS being an off-the-shelf distributed software system. The system configuration of an existing on-premise hosted HAFAS environment of a small transportation agency operating in central Germany is consulted to provide a reference point. Figure 23 illustrates the Kubernetes configuration effort required for setting up the HAFAS environment for the case study. Configuring five different Kubernetes workloads is necessary to match the functionality of Chapter 4.4.2. One of

[1] `https://prometheus.io/`

Hacon's Kubernetes test clusters hosted with Amazon EKS provides the runtime environment for the system. The API server, engine, and HAMM are stateless components without additional persistent volumes required and configured as `Deployment`. The HDC depends on a Postgres[2] database for organizing its internal state. Both HDC and Postgres are deployed as `StatefulSet`, with a persistent volume attached. Only the engine replicates two instances; auto-scaling features are not configured for this HAFAS environment as they are out of scope. All components require the configuration of Kubernetes Services to expose their respective ports. Only the API server, engine, and HAMM will be web-facing. The Kubernetes cluster that the system is deployed to already has an *Nginx Ingress Controller*[3] configured for handling ingress. In-depth configuration of the HAFAS environment is considered out of scope for this thesis; the Kubernetes configuration files matching Figure 23 are found in the Appendix under *Kubernetes YAML Configuration Files*[4]. `ConfigMaps`[Kubb] are configured with the individual components' configuration files in place. All components are deployed by using Hacon-internal Helm charts[5]. As those are considered company secrets, neither the Helm charts nor the associated docker images are accessible to the public. A web application is not deployed to the Kubernetes cluster. Hacon provides internal tooling to configure off-the-shelf web applications that are hosted on-premise. Because none of the load-bearing tests involve the front end, skipping the configuration overhead of dealing with the web application is preferable. Configuring the web application is done by applying default settings and specifying the web-facing endpoint of the HAMM.

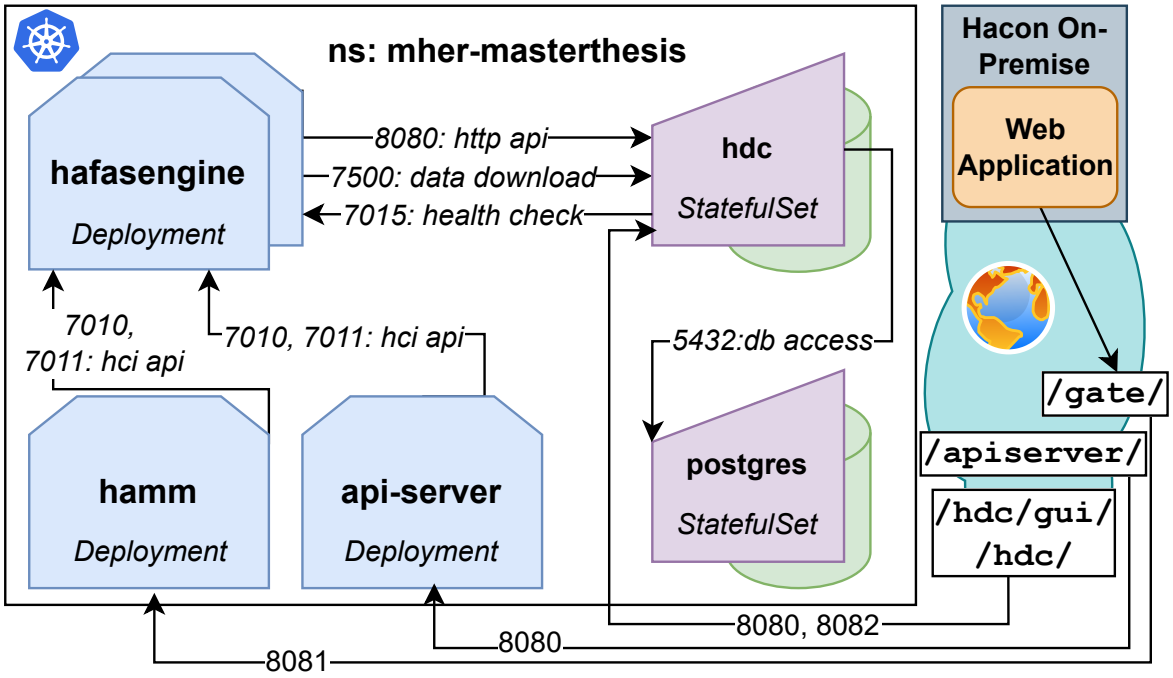

Figure 23: Overview of HAFAS environment deployed to Kubernetes (own illustration)

Figure 23 provides an overview of workloads, service-to-service communication routes, and web-facing endpoints of the HAFAS environment once deployed to the EKS clus-

[2] `https://www.postgresql.org/`
[3] `https://www.nginx.com/products/nginx-ingress-controller/`
[4] Section in the Appendix contains details regarding the configuration files, the actual YAML files are part of the *Electronic Appendix*
[5] `https://helm.sh/`

ter. To isolate the system from other Kubernetes resources, the namespace *mher-masterthesis* is defined. The engine is configured to utilize two threads for processing HCI requests. Each thread claims one port accepting HCI connections. As the system is hosted in the cluster for the time of the thesis, cost considerations have to be made. Assigning more threads to the engine increases resource consumption, which is not required in the scope of this case study. HAMM and API Server are configured to connect to the engine's exposed HCI ports to forward their requests. Once deployed, the HDC needs further manual configuration effort to automatically distribute the plan data to HAFAS instances. The plan data is taken from the system that serves as the reference point, as engine configuration must match the plan data format. After deploying the Kubernetes configuration files from the Appendix and configuring the HDC to distribute the plan data to engine instances automatically, Figure 24 shows the resulting pods in the namespace. Linkerd still needs to be installed; no sidecar proxies are running, resulting in the count of one container per pod.

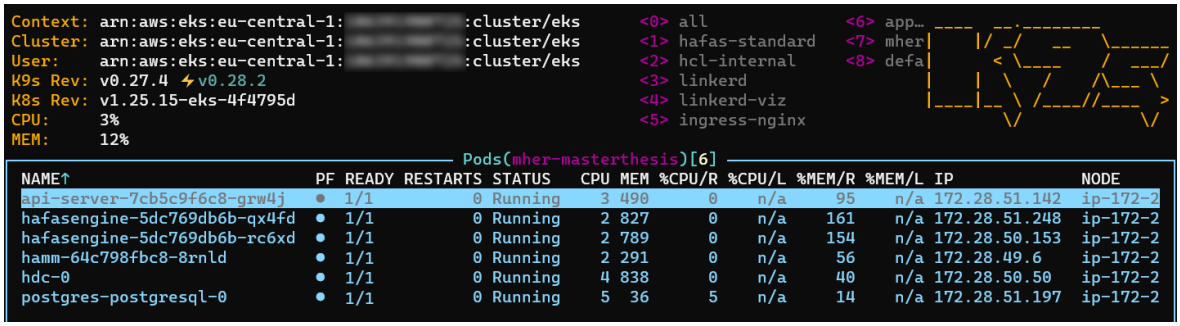

Figure 24: HAFAS environment up and running in the cluster

The web application is tasked with searching for a connection from *Hildesheim Hbf* to *Kassel Bahnhof Willhelmshöhe* in Figure 25 to verify the correct behavior of HAFAS. The engine responds with a set of possible trip options, and one possibility is rendered on the map, confirming that the HDC distributes the plan data correctly and the HAMM relays the incoming trip search requests from the front end to the engine instances in the backend successfully.

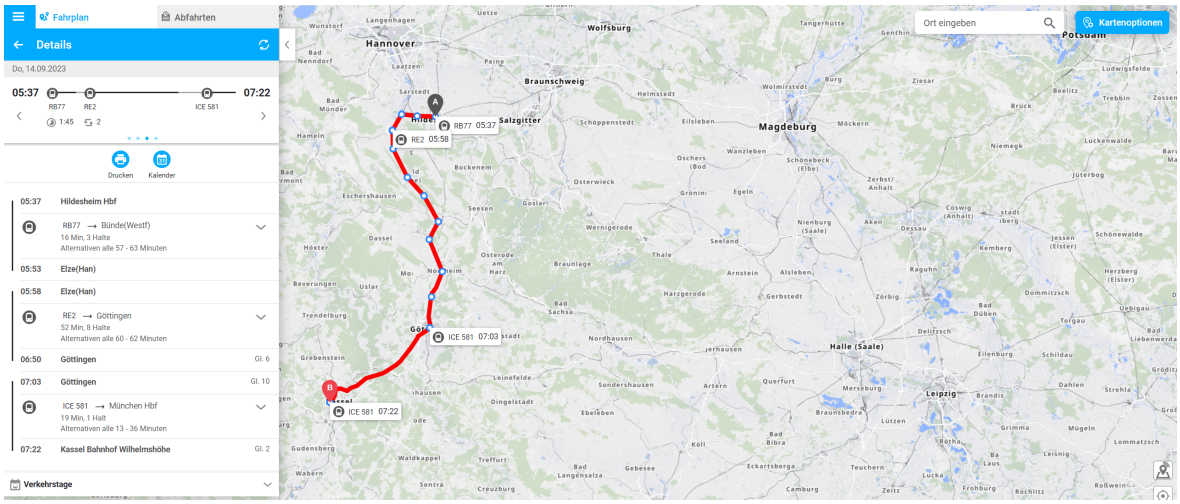

Figure 25: A trip search with the web application confirms the correct configuration of HAFAS

Lastly, to verify the API server's correct configuration, its health check endpoint accessible under `/apiserver/systeminfo` is queried. The resulting JSON reply depicted in Figure 26 confirms all to be in working order. The API server can communicate with the engines and is ready to serve user requests.

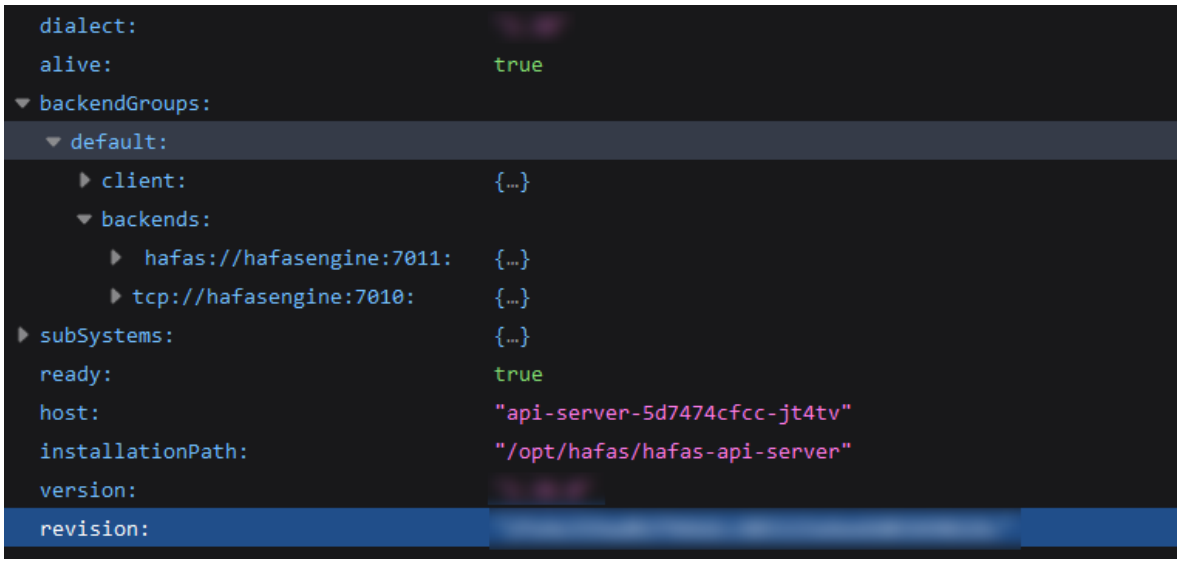

Figure 26: Results of querying the API server's health check endpoint

The following *Location Search by name* request is dispatched to verify correct behavior and configuration of API key:

```
GET <HOST>/apiserver/location.name?input=Hildesheim%20Hbf&accessId=<key>
```

Listing 2 depicts excepts of the resulting XML from the request and confirms that the API server is in working order. Searching for *Hildesheim Hbf* returns the correct train station and associated information. The HAFAS environment for the case study is in working order and ready for the addition of Linkerd as a service mesh.

```
1  <LocationList serverVersion="XXX" dialectVersion="XXX"
       requestId="5p6a7j6m4icy82wx">
2    <TechnicalMessages> ... </TechnicalMessages>
3    <StopLocation id="A=1@O=Hildesheim Hbf@X=9953495@Y
       =52160627@U=80@L=8000169@B=1@p=1678992734@" extId="
       8000169" name="Hildesheim Hbf" lon="9.953495" lat="
       52.160618" weight="7931" products="79">
4      <LocationNotes>
5        <LocationNote key="AR" type="A" txtN=""/>
6      </LocationNotes>
7      <productAtStop name="RB77" lineId="RSNM___RB77" cls
         ="4">
8        <icon res="prod_gen"/>
9      </productAtStop>
10     ...
11 </StopLocation>
```

Listing 2: API server correctly responds to *Location Search by name* request

5.2 Introducing Linkerd to HAFAS

With a working HAFAS environment running in the Kubernetes cluster, it is time to introduce Linkerd. The first step involves installing Linkerd to the cluster and then adding all workloads of the namespace `mher-masterthesis` to Linkerd and validating that the service mesh is set up correctly and its dashboard and monitoring features are operating. Afterward, a look into mTLS encryption between the services is done to validate that the service-to-service communication is indeed encrypted. Lastly, the automated certificate management described in Scenario *S2* is configured and validated. This results in the HAFAS environment for the case study having successfully implemented the Scenarios *S1* and *S2* and paves the way for conducting the load tests in Chapter 5.3.

5.2.1 Setting up the Control Plane and Data Plane

Installing Linkerd is conducted via the Linkerd CLI[Linf]. At present, the current Linkerd version is *2.14*. Before performing the installment, a pre-check is required. The pre-check is performed with the command `linkerd check -pre` and ensures that the target Kubernetes cluster is configured correctly and satisfies specific version requirements. Lastly, the pre-check ensures that no previous version of Linkerd exists and no CRDs with the same identifier exist on the cluster. With the pre-check completed, the following commands install Linkerd on the target cluster:

`linkerd install -crds | kubectl apply -f -` generates Kubernetes configuration files of the CRDs necessary to operate Linkerd. The command output is directly piped into `kubectl` and then deployed to the cluster.

`linkerd install | kubectl apply -f -` generates Kubernetes configuration files required for setting up the Linkerd control plane. It includes defining the `linkerd` namespace and the workloads of Linkerd's control services like the identity service or proxy injector service. The command output is directly piped into `kubectl` and then deployed to the cluster.

At this stage, Linkerd is installed in the cluster, and the pods of the control plane have initialized successfully in the namespace of `linkerd` as Figure 27 demonstrates. Linkerd has also injected its sidecar proxies to the workloads of the control plane, indicated by the `2/2 READY` metric of the screenshot.

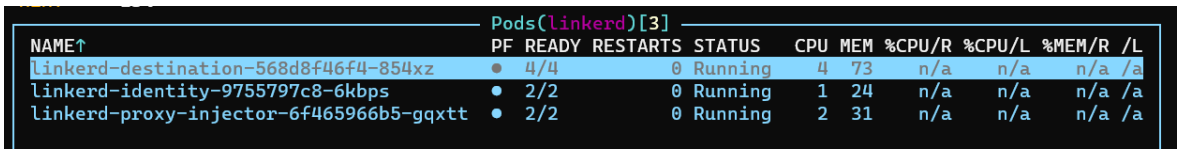

Figure 27: Linkerd control plane up and running

To gain access to the dashboard and service mesh observability metrics, the Linkerd *viz*[Ling] extension is installed with the command `linkerd viz install | kubectl apply -f` in the namespace `linkerd-viz`. In order to access the Prometheus metrics collected by the *viz*-extension, the visualization tool Grafana[6] is deployed to the

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

`linkerd-viz` namespace and configured to import metrics from Prometheus. Additionally, the dashboard is exposed to the web and password secured via an Nginx ingress rule to prevent tunneling into the cluster for access[7]. Figure 28 shows the resulting pods of the `linkerd-viz` namespace and indicates that all workloads have been added to the data plane of the service mesh in the installation process.

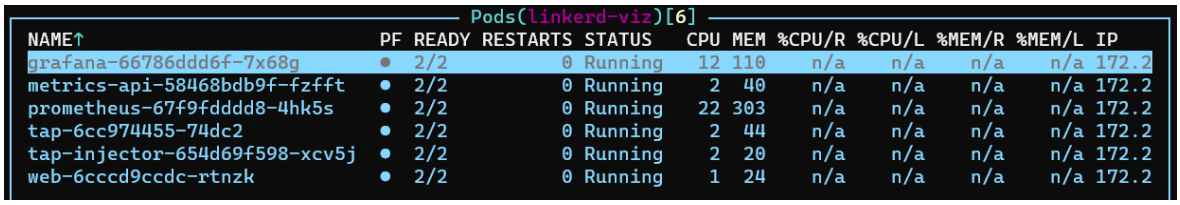

Figure 28: Linkerd's *viz*-extension and Grafana are up and running

With Linkerd and its visualization extension installed, it is time to introduce Linkerd to HAFAS by adding the individual services of the namespace *mher-masterthesis* to the data plane. The automated sidecar proxy injection is enabled by either manually annotating the individual workload definitions or target namespace with `linkerd.io/inject: enabled` or using Linkerd's CLI `inject` command on Kubernetes resource definitions. In this case, the `Deployment` and `StatefulSet` workloads in the namespace are automatically marked for automatic sidecar proxy injection by using the following commands on the `mher-masterthesis` namespace:

```
kubectl get deploy -o yaml | linkerd inject - | kubectl apply -f -
kubectl get statefulset -o yaml | linkerd inject - | kubectl apply -f -
```

The commands query the Kubernetes workload definitions as a YAML and then perform the text manipulation functionality of the Linkerd CLI `inject` command that adds the required annotation. The result is consecutively deployed to Kubernetes, and the proxy injector service of Linkerd's control plane immediately attaches sidecar proxies to the pods as indicated by the `2/2 READY` status in Figure 29.

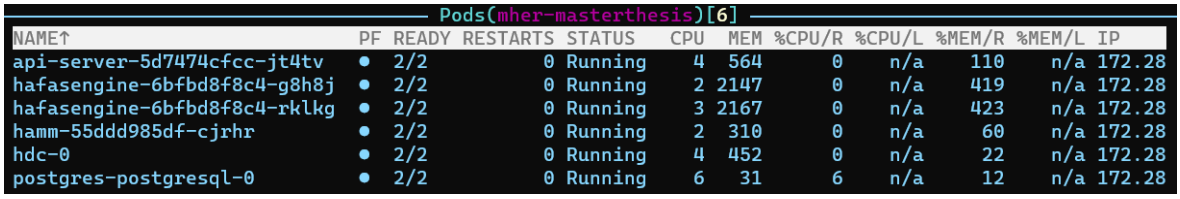

Figure 29: HAFAS environment `mher-masterthesis` added to Linkerd's data plane

Figure 30 shows the attachment process in detail. The container `linkerd-init` initializes the *iptables* configuration before the sidecar proxy starts up[Dav21].

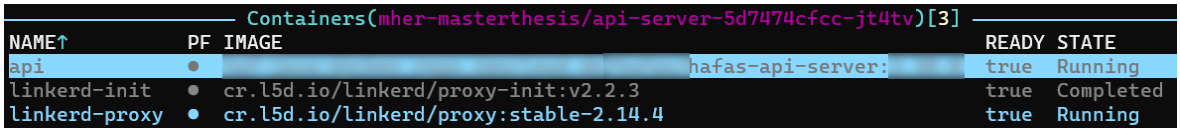

Figure 30: Linkerd's sidecar proxy is attached to a pod

[7]The details on this are found in the Appendix under the section *Setting up Grafana and Exposing the Linkerd Dashboard*

The sidecar proxy then intercepts and filters all incoming requests to the pod and outgoing application container requests and establishes a connection via the loopback interface. Linkerd automatically enables mTLS encryption for all meshed resources. Because HCI and Postgres use raw TCP protocols that prevent protocol detection by Linkerd's sidecar proxy, the configuration of opaque ports is necessary. Running `linkerd check -proxy` outputs the warning seen in Listing 3 and confirms the verdict.

```
1 !! opaque ports are properly annotated
2     * service hafasengine targets the opaque port hci through
          7010; add 7010 to its config.linkerd.io/opaque-ports
          annotation
3     * service postgres-postgresql targets the opaque port tcp-
          postgresql through 5432; add 5432 to its config.linkerd.
          io/opaque-ports annotation
4 see https://linkerd.io/2.14/checks/#linkerd-opaque-ports-
    definition for hints
```

Listing 3: Linkerd's CLI has detected a missing configuration of opaque ports

The HAMM cannot communicate with the engine without configuring the correct opaque ports with mTLS enabled. The proxy fails at protocol detection and never establishes a connection. Annotating the offending workloads and services with:

Engine: `config.linkerd.io/opaque-ports: 7010,7011`
Postgres: `config.linkerd.io/opaque-ports: 5432`

solves this issue and leads to a successful response when running `linkerd check -proxy` [8]. At this stage, Linkerd is successfully introduced to the HAFAS environment of the case study, and all of its components have been meshed and are part of the data plane. With the opaque ports configured, the system works exactly as in Chapter 5.1. The workloads of the clusters *Nginx Ingress Controller* have also been meshed to ensure data in transit encryption for terminated TLS connections[Linh]. Figure 31 showcases an excerpt of Linkerd's dashboard providing HTTP request metrics for meshed pods.

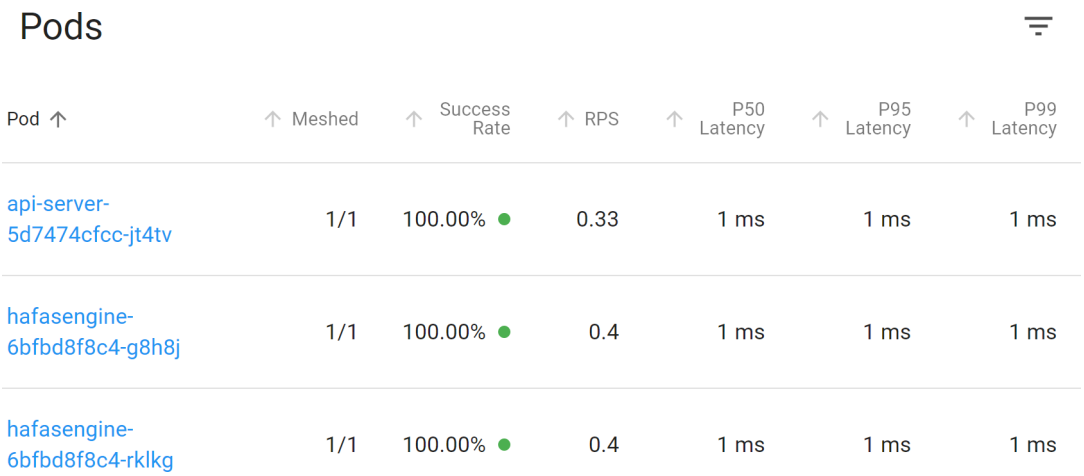

Figure 31: Excerpt from Linkerd's dashboard providing HTTP request metrics

[8] The total output of the command is found in Listing 7 under *Linkerd Proxy Verification Output* in the Appendix

5.2.2 Validating Service-to-Service mTLS Encryption

With Linkerd introduced to the HAFAS environment of the case study and the interplay of all components working correctly, a closer look at the mTLS encryption between the meshed services is taken. For this instance, the *Location Search by Name* functionality of the API server acts as a practical example for mTLS encryption of otherwise unencrypted traffic. To validate that mTLS encryption takes place, the debug sidecar[Linp] feature of Linkerd is helpful. There is no way of starting a shell inside the sidecar proxy container to keep it lightweight. The debug sidecar attaches between the sidecar and application container, thus allowing network package interception. In this case, network packages transmitted during the *Location Search by Name* request[9] to the API server are recorded with *tshark*[10] on the debug sidecar of one of the engines and then analyzed with *Wireshark*[11]. Figure 32 provides an excerpt of the recorded network packages[12] and captures the API server and engine exchange related to a *Location Search by Name* request. The exchange is between the API server (`172.28.51.167`) and the engine (`172.28.49.84`) and captures the whole TCP connection[Dor20] starting with the API server initiating with `SYN` and ending with `FIN, ACK` from the engine. The exchanged data is marked pink and categorized as *TLSv1.3 Application Data*; inspecting the data frame of such a package confirms that the exchanged data is encrypted, prooving that the service mesh fulfills the requirements of Scenario *S1*.

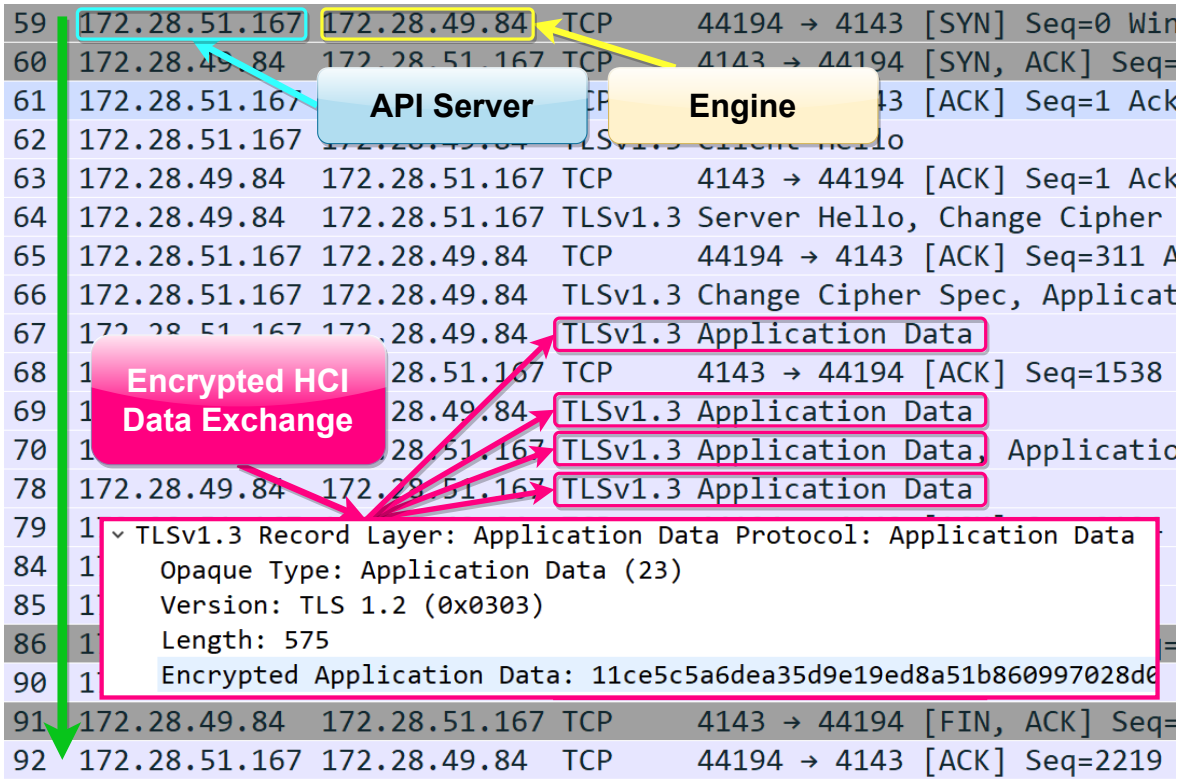

Figure 32: An in-depth look at mTLS encrypted traffic between services of the HAFAS environment

[9] The same request as in Listing 2, Chapter 5.1 is used for this occasion
[10] `https://www.wireshark.org/docs/man-pages/tshark.html`
[11] `https://www.wireshark.org/`
[12] The raw data of the capture that Figure 32 is based on is found in the *Electronic Appendix* under the file `mTLS-api-server-engine.pcap`

Another way of validating Linkerd's mTLS encryption between services is looking at the metrics collected by its monitoring stack. Linkerd provides detailed metrics for HTTP connections but only records rudimentary metrics for raw TCP protocols such as HCI. Looking at Linkerd's recorded metrics via Grafana and Prometheus, the following monitoring metrics exist for raw TCP protocols: *tcp_close_total, tcp_open_connections, tcp_open_total, tcp_read_bytes_total* and *tcp_write_bytes_total*. Once enough traffic has hit the system; the following metrics have been recorded[13] for `tcp_close_total`. For the API server, all communication concerning the two engine workloads and the two available ports assigned to each thread has always been TLS encrypted. As per Linkerd's sidecar proxy's *iptables* rules[Dav21], traffic hitting the loopback interface has not been TLS encrypted.

tcp_close_total *ns = mher-masterthesis, pod = api-server-7fbff778cf-jd5gm, direction = outbound*					
Target	**IP**	**Port**	**TLS**	**No TLS Reason**	**TCP Close Total**
Engine	172.28.49.84	7010	Yes	-	116
Engine	172.28.49.84	7011	Yes	-	90
Engine	172.28.51.111	7010	Yes	-	253
Engine	172.28.51.111	7011	Yes	-	266
Sidecar	10.100.60.160	7010	No	loobback	383
Sidecar	10.100.60.160	7011	No	loopback	383

Table 16: Sample `tcp_close_total` Prometheus metrics for the API server logging Linkerd TLS connections heading outbound

It has been validated that Linkerd's mTLS capabilities correctly secure the service-to-service communication for meshed services in the `mher-masterthesis` namespace and thus fulfill the functional requirements of *FR1*.

5.2.3 Configuring Automated Certificate Management

With the requirements of Scenario *S1* implemented, it is time to look at Scenario *S2*. Certificate management and issuance of end-entity certificates for the data plane are the essential building blocks of Linkerd's mTLS encryption. By default, the trust anchor used to derive the intermediate certificate for the CA is only valid for one year. The intermediate certificate is also only valid for one year and is not rotated automatically. End-entity certificates used by the sidecar proxies are renewed every 24 hours. Once the intermediate certificate expires, it is impossible to generate fresh end-entity certificates, which causes outages for all services in the mesh, as secure mTLS communication is not guaranteed anymore. Using Linkerd in productive environments is recommended to set up an automated way of handling certification renewal to prevent the just mentioned scenario. Short-lived end-entity certificates in the order of hours are recommended to limit attacks on services, as the timeframe for an impersonation attack to occur is shortened significantly with the timely expiration of a stolen certificate. Having to re-steal that certificate increases the difficulty for an attacker[CB20b][p. 13]. Shorter certificate validation periods also improve security when applied to the intermediate certificate of Linkerd's CA[TSHJ12]. The feature is helpful as certification revocation for compromised intermediate certificates is handled with a short expiration date in

[13]The raw data used for populating this table is available in the *Electronic Appendix* as `tls_close_total_grafana_dataframe_export.json`

case of a breach. As the trust root certificate is manually rotated, an extended validity period is preferable to minimize outages caused by expiration; in the case of a trust root certificate leakage, the organization can quickly deploy a new trust root, and the short-lived intermediate certificates derived from the old trust root will expire within a few hours. Implementing the certificate management architecture from Figure 15 of Chapter 4.3.2 is possible with the following steps[Lind] in Linkerd[14]:

- Deploy *cert-manager*[15] to the Kubernetes cluster.
- Generate a long-lasting (10 years) trust root for Linkerd and store it as a Kubernetes secret.
- Configure *cert-manager* via its CRDs to reference the trust anchor when issuing intermediate certificates for the CA.
- Configure *cert-manager* to issue an intermediate certificate with the duration of 6 hours and automated renewal after 5 hours stored in the `linkerd`-namespace secret `linkerd-identity-issuer`.
- Install Linkerd over the Linkerd CLI with the flag `-identity-external-issuer` set.
- Configure a shorter than 24 hours validity (3 hours) period for end-entity certificates with the flag `-identity-issuance-lifetime 3h0m0s`

Long Living Trust Anchor (10y)

Property	Value
Issuer	CN = root.linkerd.cluster.local
Subject	CN = root.linkerd.cluster.local
Valid From	23 Nov 2023, 11:12 p.m.
Valid To	20 Nov 2033, 11:12 p.m.
Serial Number	FE:B4:71:37:ED:3F:7F:7B:B4
CA Cert	Yes
Key Size	256 bits
Fingerprint (SHA-1)	03:EB:D2:0D:83:C6:E5:A8:0
Fingerprint (MD5)	05:7A:AB:61:EC:37:42:40:B

Short Living Intermediate Certificate (6h)

Property	Value
Issuer	CN = root.linkerd.cluster.local
Subject	CN = identity.linkerd.cluster.local
Valid From	25 Nov 2023, 5:03 a.m.
Valid To	25 Nov 2023, 11:03 a.m.
Serial Number	FB:30:A4:AA:9C:21:8C:2A:9B:0
CA Cert	Yes
Key Size	256 bits
Fingerprint (SHA-1)	6A:4C:F3:53:D1:DA:3F:7F:3A:7
Fingerprint (MD5)	52:B9:37:11:9F:5B:EF:FB:65:AF

Figure 33: Automated certificate management resulting in long-lasting trust anchor certificate and short-lasting intermediate certificate

With the steps mentioned above applied, Figure 33 shows the resulting long-lasting (10-year) trust anchor and short-lasting (6 hours) automatically `cert-manager` renewed intermediate certificate used by the CA of Linkerd to issue end-entity certificates to

[14]The cert-bot configuration CRDs are in the *Electronic Appendix*. See *Kubernetes YAML Configuration Files* for more in the Appendix.
[15]`https://cert-manager.io/`

the sidecar proxies. Using Linkerd CLI with the command `linkerd identity -n mher-masterthesis hdc-0` returns the end-entity certificate for the specified workload. Listing 4 (next page) displays the resulting end-entity certificate. As marked in green and red, the certificate is valid for three hours before the sidecar automatically requests a new end-entity certificate, as mentioned in Chapter 4.3.2.

```
1  Certificate: POD hdc-0 (1 of 1)
2  Data:
3      Version: 3 (0x2)
4      Serial Number: 2 (0x2)
5  Signature Algorithm: ECDSA-SHA256
6      Issuer: CN=identity.linkerd.cluster.local
7      Validity
8          Not Before:  Nov 25 04:19:29 2023 UTC
9          Not After :  Nov 25 07:20:09 2023 UTC
10     Subject: CN=default.mher-masterthesis.serviceaccount.
          identity.linkerd.cluster.local
11     Subject Public Key Info:
12         Public Key Algorithm: ECDSA
13             Public-Key: (256 bit)
14             X:
15                 2e:a6:70:7f:7a:5d:c4:ba:6f:e2:ac:fc:08:94:3d:
16                 42:03:c8:a4:f0:cf:4e:a3:a1:31:58:e2:35:5b:c8:
17                 82:ed
18             Y:
19                 ce:9d:a1:29:6b:59:56:07:4f:60:07:8f:e1:fd:ad:
20                 b8:6f:d5:21:b7:93:49:0c:6c:ab:4e:24:b6:8b:29:
21                 db:69
22             Curve: P-256
23     X509v3 extensions:
24         X509v3 Key Usage: critical
25             Digital Signature, Key Encipherment
26         X509v3 Extended Key Usage:
27             TLS Web Server Authentication, TLS Web Client
                  Authentication
28         X509v3 Authority Key Identifier:
29             keyid:A5:F3:B8:C0:06:0C:7E:C4:C8:2F:47:FC:51:D2:DD
                  :24:5B:CD:B4:50
30         X509v3 Subject Alternative Name:
31             DNS:default.mher-masterthesis.serviceaccount.
                  identity.linkerd.cluster.local
32
33 Signature Algorithm: ECDSA-SHA256
34      30:44:02:20:6e:ec:48:c2:9d:73:60:04:a6:0b:1d:9c:35:7d:
35      31:57:fb:a7:13:7a:ff:92:1a:8d:44:ad:14:49:d8:d2:3e:5f:
36      02:20:1b:d1:7b:c0:dc:e0:49:f7:d8:ef:4f:64:1a:84:82:a8:
37      76:40:cd:c3:57:1f:ed:e7:6b:ee:45:f1:13:45:88:1f
```

Listing 4: End-entity certificate issued to sidecar of the HDC workload

It has been validated that Linkerd certificate management allows for an automated rotation of intermediate and end-entity certificates with very short-lasting certificate validity durations, thus satisfying the functional requirements of *FR2*.

5.3 Conducting Load Tests of Linkerd and HAFAS

Two load tests are conducted to collect metrics for assessment of the performance impact of introducing Linkerd to HAFAS. The load tests are split into latency and resource overhead load tests. The former assesses the latency introduced by Linkerd to the system, while the latter attempts to gauge how the sidecar proxy performs resource-wise. The latency load test defines two load test scenarios, fulfills the test with and without Linkerd introduced to the system, and then compares the resulting data. The resource overhead load tests utilize the Prometheus monitoring stack already present with the Kubernetes cluster to extract container CPU and RAM usage data. This section deals with planning and executing the tests; the data analysis takes place in Chapter 6. Section *Insights on the Execution of Load Tests* in the Appendix also provides additional information in the form of screenshots and explanations.

5.3.1 Carrying Out Latency Overhead Load Tests

Apache JMeter[16] is selected as the tool of choice for latency load testing. The focus of these tests is on intra-cluster service-to-service HTTP and HCI communication with the mTLS feature of Linkerd enabled. Three additional nodes of the type *m6i.xlarge*[17] are added to the Kubernetes cluster for the testing period to ensure adequate isolation of workloads. Pods are assigned to the nodes by utilizing Kubernetes *affinity* features[Kuba]; to restrain unwanted pods from being scheduled to the three nodes adequate *taints and tolerations*[Kubo] are configured for the involved workloads. *Apache JMeter* is deployed to the `mher-masterthesis` namespace via a customized docker container[18] and the load tests are executed by triggering a customized script by accessing the shell of the container[19]. The retrieved data is then to be analyzed further in Chapter 6. The tests are conducted by using the *Location Search by Name* and *Trip Search* capabilities of the API server. To record enough data for a comparison, each test consists of five one-minute testing phases. First, data is collected for a baseline scenario depicting HAFAS hosted in the Kubernetes cluster without attaching Linkerd's sidecar proxies to its services. Secondly, HAFAS is added to Linkerd's data plane. From then on, HAFAS will be an active part of the service mesh, and all service-to-service traffic will be transmitted over mTLS-encrypted channels. The primary data of interest is the recorded latency time for each request to research the possible latency overhead introduced by routing traffic over Linkerd's sidecars. This load test is not designed to explore the *Request per Second* (RPS) limits of HAFAS; the focus is solely on the difference in latency times in the meshed and non-meshed scenarios. During the load tests, the replication value of the engine is changed to five to support a higher RPS output without risking the falsification of test results through performance bottlenecks. Linkerd allows defining resource policies for the sidecar proxies; for this test, the feature is not utilized as Linkerd's default configuration does not enable the feature, and the resulting resource usage data is required to analyze a possible resource overhead. The following two load tests have been identified as helpful in collecting data:

[16] `https://jmeter.apache.org/`

[17] Each node features four vCPUs and 16GB of RAM

[18] The docker container and load testing script is available in the *Electronic Appendix* under `jmeter-docker`. The Kubernetes deployment file is available in the folder `jmeter` of `kubernetes-config`.

[19] *Apache JMeter* executes test plans and exports the results as a *Comma-separated value* (CSV) file. The script takes a link pointing to the test plan, downloads it, executes it with *JMeter's* headless mode, and lastly, uploads the result to the web for easy retrieval.

- **Load Test #1: Location Search by Name** - 20 RPS over five minutes. Lightweight API server requests involving low computing power from the engine. A higher RPS rate is needed to observe Linkerd's behavior with short-lived requests.
- **Load Test #2: Trip Search** - 10 RPS over five minutes. Heavy API server requests involve a sizable chunk of computing power from the engine. A higher RPS rate would require more engine instances to avoid filling the queues, as each engine is configured with only two threads, conducted to observe Linkerd's behavior with expensive HAFAS operations.

Table 17 contains the exact *Apache JMeter* configurations used in both latency load tests. The test plans and raw CSV results are accessible in the folders `loadtest-plans` and `loadtest-results` of the *Electronic Appendix*.

Load Test #1 **Location Search by Name**	
JMeter Settings	1 Thread 1200 Repetitions Limit to 1200 Requests per Minute Results in 20 RPS Runtime: 60s / Repeat: 5x Results in 6000 Requests over 5 Minutes
HTTP Request	Protocol: *HTTP* Server Name: *api-server* Port: *8080* HTTP Request: *GET* Path: */apiserver/location.name* Parameters: - `input` = *Hildesheim* - `accessId` = *API_KEY*
Load Test #2 **Trip Search**	
JMeter Settings	1 Thread 600 Repetitions Limit to 600 Requests per Minute Results in 10 RPS Runtime: 60s / Repeat: 5x Results in 3000 Requests over 5 Minutes
HTTP Request	Protocol: *HTTP* Server Name: *api-server* Port: *8080* HTTP Request: *GET* Path: */apiserver/trip* Parameters: - `originExtId` = *8000169* - `destExtId` = *2200007* - `date` = *2023-09-01* - `accessId` = *API_KEY*

Table 17: *Apache JMeter* configuration used for the test plans

5.3.2 Carrying Out Resource Consumption Overhead Load Tests

Resource consumption load tests are conducted by executing modified versions of the latency load tests from the previous Chapter and exporting the recorded Prometheus resource metrics afterward. To incorporate scenarios with a higher load on the engine the *Trip Search* test plan from the previous Chapter has been modified[20]. It targets a route from *Kiel Hbf* to *Munich Airport Terminal* that spans through the entirety of Germany and requires more resources to compute. An idle period with no load is recorded to provide a baseline scenario for the following data analysis in Chapter 6.1.2. Three data recordings are made: an idle period, *Location Search by Name* from the previous latency load test, and a modified version of the *Trip Search* request that induces a higher load. After conducting the tests, Grafana exports the test data to the CSV file format with the Prometheus queries shown in Listing 5.

```
1  CPU Metrics FOR EACH:
2  (__POD__ = [apiserver, engine], __CONTAINER__ = [api / engine /
       linkerd-proxy])
3      node_namespace_pod_container:
          container_cpu_usage_seconds_total:sum_irate{namespace="
          mher-masterthesis", pod="__POD__", container="
          __CONTAINER__"}
4  Results in:
5      [Timestamp]   [CPU Utilization (1 = One Core)]
6      1701169460000,0.9195397489042423
7
8  Working Set Metrics FOR EACH:
9  (__POD__ = [apiserver, engine], __CONTAINER__ = [api / engine /
       linkerd-proxy])
10     container_memory_working_set_bytes{{namespace="mher-
          masterthesis", pod="__POD__", container="__CONTAINER__"}}
11 Results in:
12     [Timestamp]   [Working Set Size in Byte]
13     1701170240000,998244352
```

Listing 5: Prometheus queries pseudocode to extract resource metrics for further data analysis in Chapter 6.1.2

The resulting data points contain the memory footprint and CPU utilization for the containers of the API server, the five engine instances, and all sidecar proxies attached by Linkerd. A post-processing step that merges the resulting CSVs is implemented via a Python[21] to prepare the data for further analysis[22]. For the memory metrics, only the working set[Kubl] of the container is considered. The memory set is defined as the memory reported in use by the operating system of the container that cannot be cleared. The CPU metric targets the core utilization of the assigned nodes' resources.

[20] The test plan used in this Chapter is found in the *Electronic Appendix* under `loadtest-plans/Test-plan-2_higher_load.jmx`

[21] `https://www.python.org/`

[22] The script is located in the *Electronic Appendix* under `resource-test/process.py`. The resulting data is in the folder `resource-test/csv/` and consists of two CSV files for each memory and CPU utilization for the test scenarios. Through the script, the individual data points of all containers have been combined to a metric including all HAFAS components and the sidecar proxy through the arithmetic mean.

5.4 Conclusion

In conclusion, this Chapter demonstrates that HAFAS operates well in combination with a service mesh. Configuring Linkerd has not caused any issues with the functionality of the base system that was selected for the case study. The Scenarios *S1* and *S2* have been implemented and validated successfully. HCI has not caused any issues with Linkerd, and upon configuring opaque ports to bypass protocol detection, mTLS encryption involving HCI works, as shown in Chapter 5.2.2. The certificate management features of Linkerd work very well and allow for automated short certificate rotation for both end-entity certificates of the sidecar proxies and the intermediate certificate used by the CA. The security architecture of HAFAS improves by introducing the security properties of confidentiality, integrity, and authenticity to the service-to-service communication. While out of the scope of this case study, the security property of authorization is also implementable via Scenario *S3*. With a service mesh introduced to HAFAS, the foundation for implementing a zero-trust security architecture has been completed. The case study has proven that a basic HAFAS environment is compatible with service mesh technology. With the load tests from Chapter 5.3 conducted, Chapter 6 will now evaluate the collected data and successively answer the research questions of the research assignment from Chapter 3.4.

6 Evaluation

With Linkerd introduced to HAFAS successfully as part of the case study and the load tests of Chapter 5.3 conducted, the evaluation part of this thesis deals with analyzing the data collected in Chapter 6.1.1. Firstly, Chapter 6.1.1 deals with looking at network latency effects introduced by Linkerd to the distributed system. Following that, resource consumption is investigated in Chapter 6.1.2 to verify whether introducing a service mesh to HAFAS is punished with excessive resource overhead. Lastly, Chapter 6.1.3 deals with field observations from the case study. With the analysis part completed, Chapter 6.2 then reviews the research assignment from Chapter 3.4 with the insights gathered from the case study.

6.1 Analysis of Observations from the Case Study

The programming language R for statistical computing and graphics is selected as a suitable candidate for analyzing the raw CSV data generated by the latency and resource consumption load tests. Tables in this chapter depict processed data to the user; the plots resulting from this data are not placed in the Appendix to prevent unnecessary paging between the two chapters. Some plots take up a large amount of space and, for this reason, have been rotated to prevent skipping between the Appendix and this chapter. When referencing such a plot, the text will inform the reader of the corresponding page number. The raw data consists mainly of time series and is accessible via the *Electronic Appendix*, as storing it in the physical Appendix would unnecessarily bloat the page count of this thesis. For Chapter 6.1.1, the raw data of the latency load test generated by *Apache JMeter* is stored in the folder `loadtest-results/` as CSV files[1]. The corresponding R script file for processing and graphing this data is available under `load_test_analyse.R`. For Chapter 6.1.2, the raw data used in the R script `resource_test_analyse.R` is stored in the folder `resource-test/csv` as CSV files. This data has passed through a pre-processing step by the Python script `resource-test/process.py` as the original raw data exported from the Prometheus instance of the case studies Kubernetes clusters monitoring stack splits into twenty-four different CSV files for each test scenario, twelve of them each tied to the containers workload size of both Linkerd sidecar proxies and associated meshed HAFAS components and their CPU utilization during the test scenario. The merging done by the script is conducted with the arithmetic mean, averaging the data of the twenty-four CSV files by grouping them into *hafas* and *linkerd* subsystems following the pseudocode of the formulas below for each of the three resource consumption scenarios:

$$\frac{(\sum_{n=1}^{5} engine_n + apiserver)_{cpu} + (\sum_{n=1}^{5} engine_n + apiserver)_{mem}}{6} = hafas_{cpu,mem}$$

$$\frac{((\sum_{n=1}^{5} sidecar_n)_{engine} + sidecar_{api})_{cpu} + ((\sum_{n=1}^{5} sidecar_n)_{engine} + sidecar_{api})_{mem}}{6} = linkerd_{cpu,mem}$$

resulting in the following three data sets for each scenario of Chapter 5.3.2 that are

[1] The data of all four tests is stored in a separate CSV file. Each file contains the recorded time series of requests dispatched by *Apache JMeter* containing the timeframe, request, status code, and latency information. Only the latency information is relevant for the analysis as all requests have been successfully processed by HAFAS.

used as a foundation for the resource consumption analysis[2]:

$$load_{idle} = \begin{pmatrix} hafas \\ linkerd \end{pmatrix}_{cpu,mem} \quad load_{test1} = \begin{pmatrix} hafas \\ linkerd \end{pmatrix}_{cpu,mem} \quad load_{test2*} = \begin{pmatrix} hafas \\ linkerd \end{pmatrix}_{cpu,mem}$$

These averaged-out subsystems are then used as inputs when conducting the resource consumption analysis in Chapter 6.1.2 and allow comparing the proportions of allocated memory and CPU resources by HAFAS and the Linkerd data plane. This approach excludes services like the HDC or HAMM not involved in the load test based on only including components directly exposed to the load test and also takes no consideration for the Linkerd control plane. The Linkerd control plane is considered a static source of resource consumption that does not scale up with the number of active sidecar proxies in the data plane and is thus absent from this analysis.

6.1.1 Impacts on Network Latency

5m / Location Search by Name @ 20 RPS / Trip Search @ 10 RPS						
Baseline - Location Search by Name						
Mean / Median	**10.62ms / 10ms**		Min	**8ms**	Max	**56ms**
Percentile	25%	50%	75%	90%	99%	99.9%
Latency	**10ms**	**10ms**	**11ms**	**11ms**	**14ms**	**53ms**
Linkerd - Location Search by Name						
Mean / Median	**14.04ms / 14ms**		Min	**11ms**	Max	**658ms**
Percentile	25%	50%	75%	90%	99%	99.9%
Latency	**13ms**	**14ms**	**14ms**	**15ms**	**20ms**	**60ms**
Baseline - Trip Search						
Mean / Median	**34.21ms / 34ms**		Min	**32ms**	Max	**124ms**
Percentile	25%	50%	75%	90%	99%	99.9%
Latency	**33ms**	**34ms**	**35ms**	**35ms**	**38ms**	**42ms**
Linkerd - Trip Search						
Mean / Median	**38.58ms / 37ms**		Min	**35ms**	Max	**2198ms**
Percentile	25%	50%	75%	90%	99%	99.9%
Latency	**37ms**	**37ms**	**38ms**	**39ms**	**51ms**	**80ms**

Table 18: Latency data resulting from the load tests in Chapter 5.3.1

Table 18 depicts the data resulting from the four latency load tests of Chapter 5.3.1. When comparing baseline and Linkerd results for both load tests, a constant factor of *3-4 ms* difference in the median latency time within the cluster is observable by introducing Linkerd to the HAFAS environment of the case study. The requests have been routed from the *Apache JMeter* pod through the API server to one of the five engine instances and have thus passed two mTLS encrypted communication channels. From this data alone, it is not possible to determine if adding longer service-to-service communication chains would increase this latency overhead by a more significant margin or if it stays relatively constant. In both test scenarios, the *3-4 ms* latency overhead stays constant up until the 90th percentile. By the 99th percentile, the latency overhead diverges dramatically between *Latency Search by Name* and *Trip Search*. In both scenarios, severe outliers with at least ten times the latency of the baseline scenario in

[2] *test2** marks the modified test plan `Test-plan-2_higher_load.jmx`

the 99.9th percentile have been identified with the sidecar proxies attached. A source for the outliers' response time has not been identified, and they are thus considered negligible for the rest of the analysis with no impact attached to them. To remove them from distorting the results of further analysis, a subset of the data is declared by defining a lower and upper boundary by subtracting and adding *1.5* times the inter quantile range from the 25th and 99th percentile as follows in Listing 6:

```
1  quartiles <- quantile(csv1$Latency, probs = c(.25, 0.99))
2  IQR <- IQR(csv1$Latency)
3  Lower <- quartiles[1] - 1.5*IQR
4  Upper <- quartiles[2] + 1.5*IQR
5  data_no_outlier1 <- subset(csv1, csv1$Latency > Lower & csv1$
       Latency < Upper)
```

Listing 6: Removal of severe outliers deemed insignificant for further analysis from the data set

Figure 34 shows the distribution of the remaining outliers in the subset as a boxplot. In both cases, adding the sidecar proxies to the HAFAS services increases the probability of latency outliers. This observation is neglectable in the *Trip Search* scenario as the median latency time matches the first quartile. Interestingly, in the *Location Search by Name* scenario where a higher RPS with more computationally effortless requests is archived, the Linkerd median latency time matches the third quantile. In contrast, the baseline scenario median matches the first quantile. This finding might indicate that higher RPS rates and, thus, higher throughput for the sidecar proxy could cause a more significant latency overhead when considering scaling with Linkerd. Without further testing, this hypothesis is not verifiable with this data set.

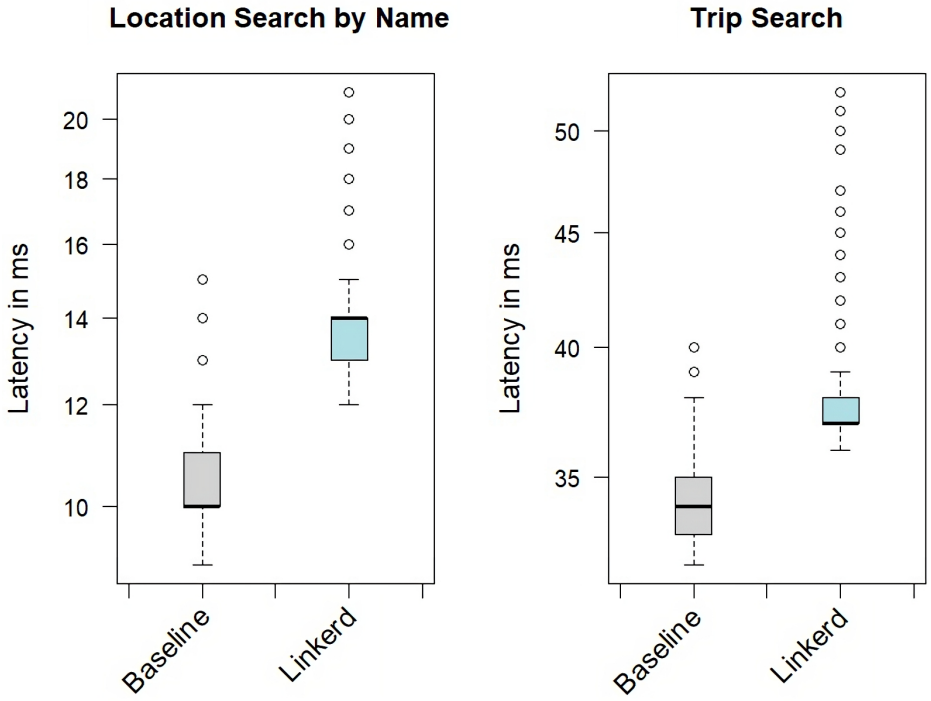

Figure 34: Latency comparison between baseline and Linkerd scenarios for the two latency load tests

The following two pages contain plots referenced in the third page from here.

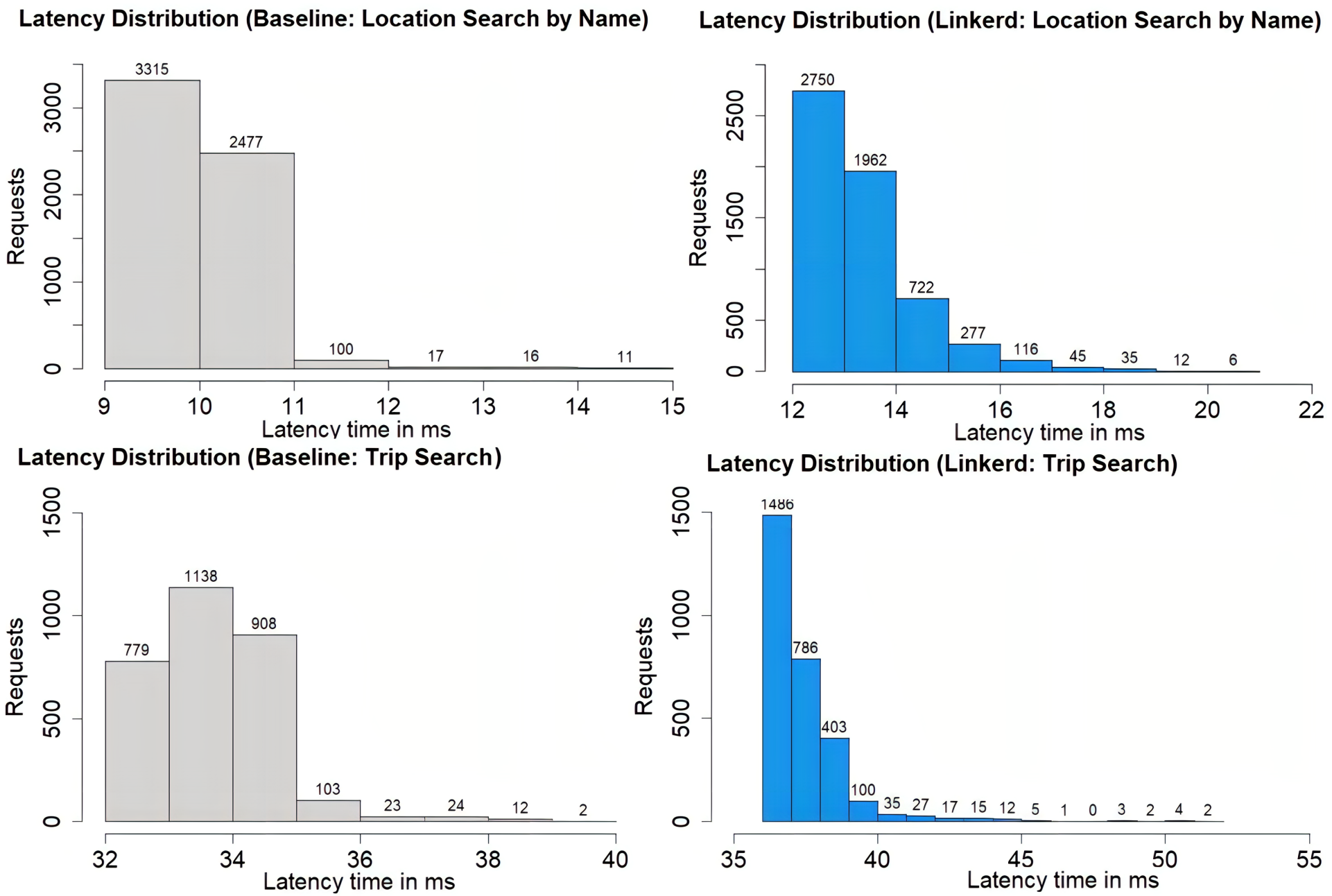

Figure 35: Latency distribution histograms resulting from the load tests

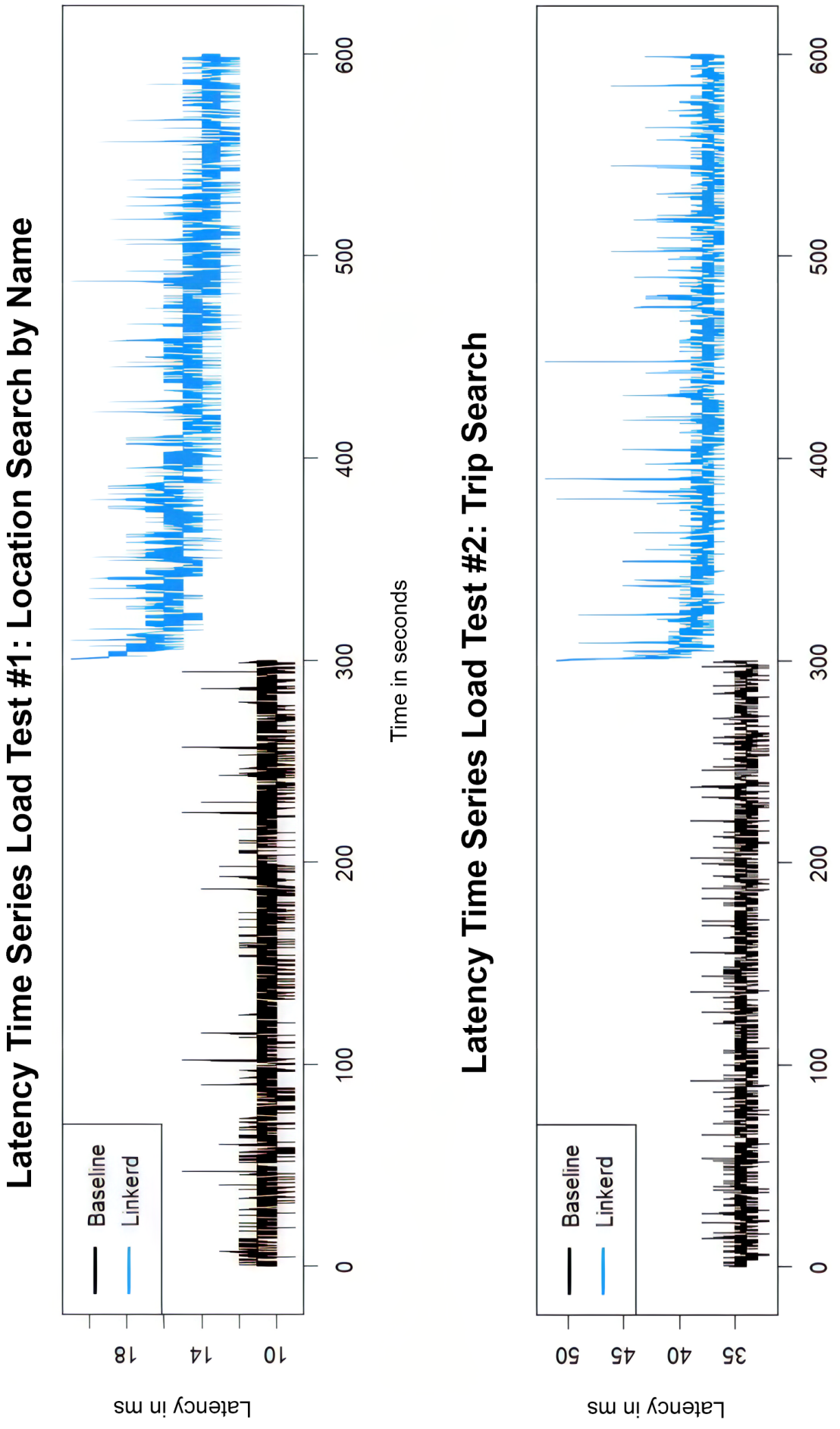

Figure 36: Latency time series resulting from the load tests

Figure 35 displays the latency distribution in both scenarios with a histogram bucket size set to one millisecond and also illustrates that the outliers introduced by Linkerd's sidecar proxies are a non-issue in a practical scenario. Figure 36 depicts the recorded data for both scenarios as a time series. The data has been recorded with gaps in between and is rendered without gaps to illustrate the differences between latency with and without Linkerd. The visual distribution of outliers appears to be nearly equal between baseline and Linkerd. Linkerd latency outliers result in a more significant delta between the outlier and median. Interestingly, both test cases of the baseline scenario result in a steady and stable distribution of latency times, with Linkerd's sidecars' latency time lowering over time. More comprehensive tests would be required to determine if *Apache JMeter* causes this anomaly or if Linkerd's sidecar proxy is prone to a warm-up time to reach stable performance. While these observations are interesting, the conclusion drawn from analyzing Linkerd's latency overhead is that they represent an acceptable trade-off in the context of HAFAS. The features of Scenario *S1..S5* provide value to HAFAS, and the measured constant median latency overhead of *3-4 ms* is not critical. HAFAS is not a high-performance real-time application where every millisecond of lost performance causes dramatic effects on stakeholders. An end user or transportation agency is unaffected by a constant overhead of *3-4 ms* due to HAFAS's user interface heavy focus. When considering rolling out Linkerd to a feature-complete productive HAFAS environment as described in Chapter 3.2, latency retesting should be conducted with RPS orientating at production values to validate similar behavior.

6.1.2 Impacts on Resource Consumption

Idle Scenario						
Linkerd Sidecars	**Min**	**Max**	**Median**	**p75**	**p90**	**p99**
CPU Load	0.00036	0.0004	0.00039	0.0004	0.0004	0.0004
Working Set (MB)	4.60	4.69	4.63	4.64	4.66	4.69
HAFAS	**Min**	**Max**	**Median**	**p75**	**p90**	**p99**
CPU Load	0.00072	0.0008	0.00076	0.00078	0.0008	0.00082
Working Set (MB)	475.32	475.34	475.32	475.32	475.33	475.34
Load Scenario (HAFAS = mean(Test1, Test2*))						
Linkerd Sidecars	**Min**	**Max**	**Median**	**p75**	**p90**	**p99**
CPU Load	0.0015	0.0078	0.0075	0.0077	0.0077	0.0078
Working Set Size (MB)	4.62	4.78	4.71	4.73	4.75	4.77
HAFAS	**Min**	**Max**	**Median**	**p75**	**p90**	**p99**
CPU Load	0.065	0.366	0.351	0.359	0.365	0.366
Working Set Size (MB)	950.61	950.8	950.77	950.78	950.79	950.8

Table 19: Resource consumption data resulting from the load tests in Chapter 5.3.2

Table 19 depicts the data from the resource consumption load tests from Chapter 5.3.2. As mentioned in Chapter 6.1.1, a post-processing step through Python has been involved, resulting in three averaged-out subsystems containing the metrics of one API server, five engines, and their respective Linkerd sidecar proxies that serve as the input for this analysis[3]. Additionally, the two subsystems *test1* and *test2** are treated as one

[3]The corresponding CSV files are located in the *Electronic Appendix* under `resource-test/csv`.

subsystem that depicts HAFAS under the load of both load tests by averaging them out using the arithmetic mean:

$$load_{full} = \frac{load_{test1} + load_{test2*}}{2}$$

The time series in Figure 37 illustrates why this averaging step occurs. Due to the different load scenarios mapped by the two tests, the CPU utilizations differ by quite a margin. *Test1* focuses on frequent shortlived and CPU consideriate *Location Search by Name* requests while *Test2** involves CPU heavy *Trip Search* calculations. The arithmetic mean is taken for comparing HAFAS in a load scenario to approximate real-world scenarios where both request kinds do not exist separately in a vacuum. As no data regarding the distribution of requests in a real-world scenario exists, both results are weighed equally.

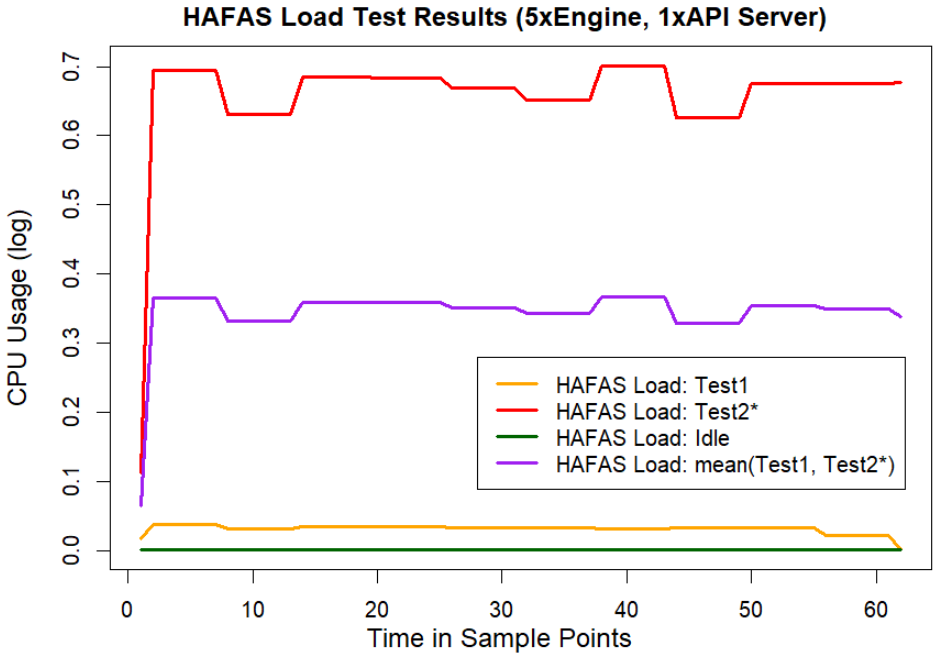

Figure 37: Comparison of HAFAS CPU utilization between load scenarios

The time series plots in Figure 38 (next page) based on the data from Table 19 paint a clear picture when it comes to the resource overhead introduced by Linkerd's sidecar proxies. It becomes apparent that the memory overhead of the sidecar proxy is far from concerning, with a slim median footprint of *4.63 - 4.71 MB* in idle and load scenarios, it is magnitudes lower than the footprint of HAFAS. Regarding CPU consumption at load, Linkerd's sidecar proxies add an overhead of roughly *2.1%*. In the idle scenario, the sidecar proxies and HAFAS nearly consume no CPU resources. Lastly, it has assessed that the resource overhead introduced by Linkerd's sidecar proxies to HAFAS is marginal and presents an acceptable trade-off, given the benefits gained. Additionally, a constant amount of resources allocated by the control plane must be considered when approximating Linkerd's resource overhead to a system. When introducing Linkerd to productive systems, resource consumption tests should be repeated with adequate load to reestimate the resource overhead, as the result from this test is likely only adaptive to other distributed systems to some extent.

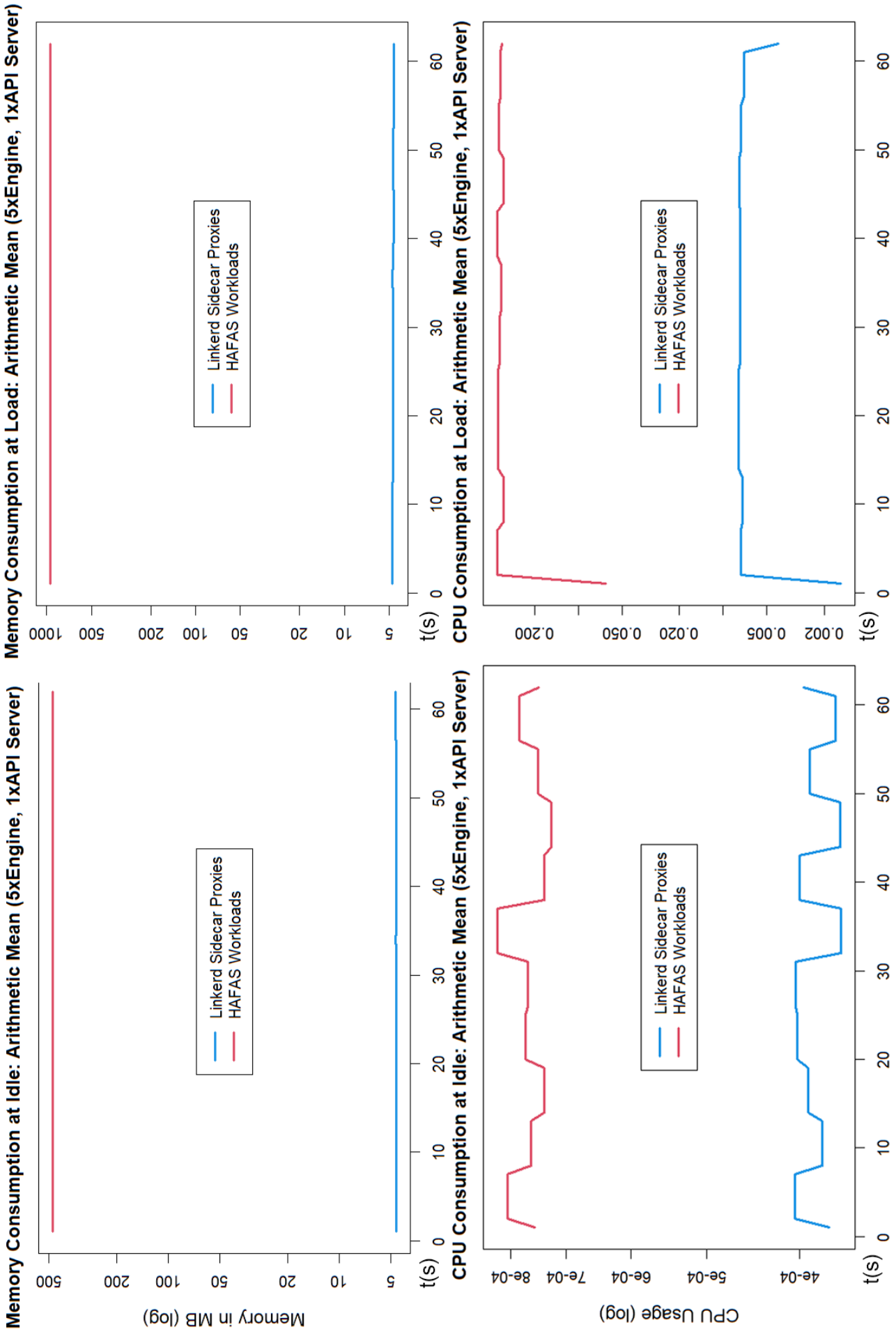

Figure 38: Memory and CPU consumption time series from the resource overhead load tests

6.1.3 Field Observations from the Case Study

The following additional observations have occurred while conducting the case study and interacting with Linkerd:

- Linkerd's dashboard and observability tools treat HTTP as a first-class citizen. Extensive monitoring features are provided for monitoring the HTTP connections between services, as Figure 39 illustrates. Linkerd's metrics allow monitoring of the RPS throughput of services and render the communication paths chosen between services in the service mesh. The same metrics are, however, not provided to other TCP protocols. At best, the current TCP throughput between services is rendered as a list of services. The screenshot of Figure 39 was taken while conducting load testing with Linkerd; at the time, the API server communicated with up to five engines not included in the request graph rendering. Organizations that rely heavily on raw TCP requests will find themselves without support for most of Linkerd's observability features, while architectures emphasizing HTTP as the primary communication protocol will find Linkerd's observability features useful.
- Linkerd's upgrade mechanism emphasizes the upgrade going right. There is no way not to upgrade the control plane in one go; all three upgrades conducted during the duration of the case study have been a smooth experience without causing issues. The Linkerd control plane also offers backward compatibility with sidecar proxies running an older version due to no restart injecting the new version yet. Chapter 4.2 has established this as an advantage of Istio; when introducing Linkerd to productive environments bound to *Service Level Agreements* (SLAs), this should be investigated further by organizations to prevent unwanted surprises.

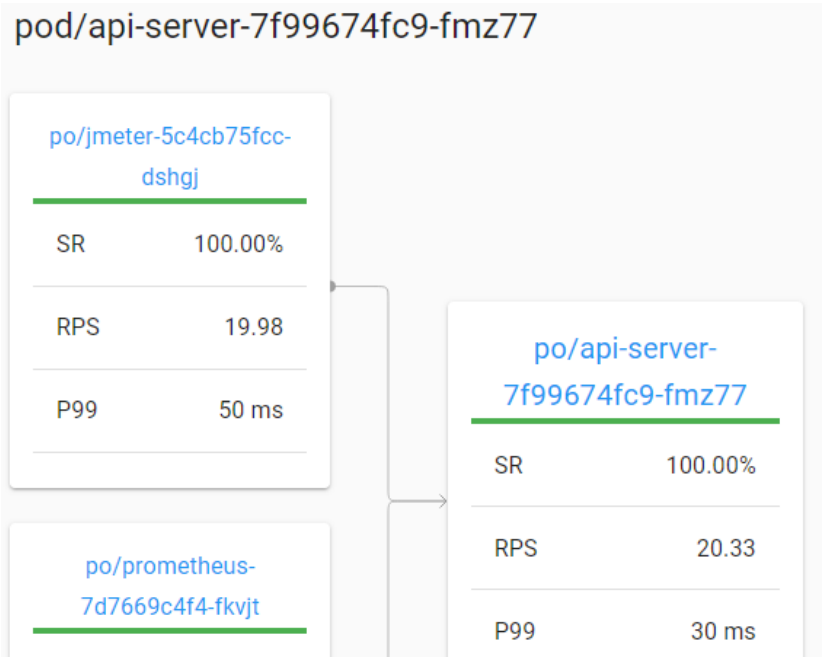

Figure 39: Linkerd's dashboard visualizes and reports HTTP connections between meshed pods

6.2 Review of the Research Assignment

With the case study conducted successfully, the set of selected scenarios for further investigation validated, and the case study observations analyzed, it is time to reflect on the research assignment from Chapter 3.4. The practical benefit of introducing a service mesh to an existing distributed software system stems from the introduced security, reliability, resilience, and observability features and the change in how a distributed system is configured and managed by operators. Service mesh technology strongly emphasizes infrastructure as code principles by outsourcing configuration to CRDs. An excellent example of this dynamic is Scenario *S2: Certificate Management for Data and Control Plane* from Chapter 4.3.2. In the case study, it was apparent that this feature improved the security aspects and resilience features of the distributed system by avoiding outages in the service-to-service communication through expired certificates that require manual rotation. *S2* also allows the configuration of absurdly short certificate rotation procedures that integrate seamlessly into the service mesh. Adding mTLS encryption is also considered a great addition to the service mesh. Improving the security properties of confidentiality, integrity, and authenticity to service-to-service communications strengthens the overall security features of a distributed system. It prevents attackers from eavesdropping or tampering from within should they gain access to a system. However, it also needs to be noted that a service mesh is not a catch-all security. Improvement: Data at rest must still be protected, and ransomware attacks are not prevented by simply having the client and server authenticate mutually and encrypt their communication. Attackers who gain access to a pod can also eavesdrop through the unencrypted communication between the sidecar proxy and service via the loopback device. While out of the scope of the case study, authorization policies and their ability to implement zero-trust architecture principles in Kubernetes clusters also offer remarkable opportunities to strengthen the security of distributed systems. The latency and resource overhead introduced by a service mesh like Linkerd appear manageable and are worth the trade-off for the added benefits. A service mesh, however, needs to be introduced with valid reasons in mind. Deploying a service mesh to a distributed system just because one feels it is not a good motivation, and before this step, determining the feasibility and requirements for the why of a service mesh is necessary. As the complexity of distributed systems increases, implementing and configuring capabilities via a service mesh does beat doing it on a service level. In the case of introducing mTLS to HAFAS, many components would require changes to them without the service mesh, an undergoing that appears to be almost not fertile.

For the research questions, the following answers are provided:

Q1: What are the chances and risks of using a service mesh in practice?

As mentioned, the chances of a service mesh lay mainly in introducing features to a large-scale distributed system without implementing them for each service individually. After introducing a service mesh to a distributed system, a migration plan could be worked out to outsource features already present in individual services to the service mesh, thus trading service architectural complexity against configuration complexity. Before considering this, the endeavor should be planned out meticulously by all stakeholders involved. Service meshes also introduce extra complexity and additional points of failure. Teams exposed to the service mesh face a learning curve before mastering the technology for safe and stable use in production. Wrong management of certificates could result in outages after a year if, for example, a default identity issuance

certificate expires. On the other side, outsourcing service features to the service mesh standardize the configuration of the distributed application, thus improving efficiency in an organization as soon as required new workflows have been established and propagated to stakeholders of the organization. Choosing the correct service mesh project is also crucial, as a wrong choice could jeopardize the use of certain features in the future. Organizations wishing to adopt a service mesh should not go from zero to one hundred but gradually, with lots of local testing and requirement engineering at hand. When realizing the full benefits of a service mesh, considering risks hand in hand with a well-thought-out implementation plan is necessary.

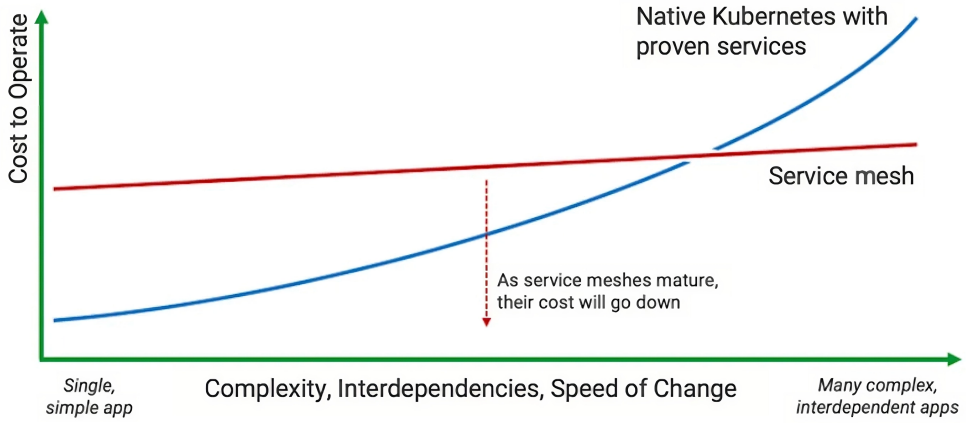

Figure 40: Factors on deciding whether introducing a service mesh to a distributed system is viable[4]

Figure 40 showcases finding the sweet spot of introducing a service mesh to a distributed system. A service mesh will always result in additional operation costs; the complexity of the distributed system is a deciding factor in whether or not implementing a service mesh is beneficial. Organizations should seek to explore the potential operative costs in detail before deciding on adopting service mesh technology.

Q2: What does the current technological landscape of existing service mesh projects look like?

As per Chapter 4.2, the current technological landscape of service meshes pulsates with life. Loads of small and upcoming projects exist in both the open- and closed-source space. Two Linkerd and Istio are two CNCF graduate projects. Both offer a mature and field-tested service mesh solution already used in production by large-scale companies worldwide. Service meshes appear to be on the way to an early majority adoption stage for cloud-native computing. The lack of monopolies in service mesh projects also ensures healthy competition. After conducting the case study with Linkerd as the selected service mesh project of choice, it would be interesting to see how the same would work with Istio. While both projects appear similar to the initial comparison conducted in Chapter 4.2, Istio offers more customizability at first glance. To confirm this, further investigation of Istio would be necessary.

[4]Sourced from [Bel21]

Q3: What are the implications and restrictions of introducing a service mesh to an existing system?

There are likely no restrictions on the service mesh itself, depending on the state of the system introduced. Certain features might only work with extensive changes to the system required. Looking at HAFAS, for example, the traffic redirection and traffic splitting features would likely work without issues on the network level itself; however, they are difficult to utilize to their full extent because data dependencies prevent the successful implementation of the primary usage scenarios such as canary or blue/green deployments. Changes on a component-level base would also be necessary to support distributed tracing, as the components need to ensure the pass-through of tracing headers for that feature to work. No restrictions were encountered during the case study when implementing Scenario *S1* and *S2*. Linkerd, for example, only supports the opaque ports feature since version *2.10*[Pre21]; without that feature present, restrictions with mTLS would certainly appear due to the HCI protocol. Without skipping protocol detection for *server-speaks-first* protocols, all services communicating via HCI would have to bypass the sidecar proxy and thus be excluded from the mTLS functionality of Linkerd. That would have been a severe restriction by the system that would have been avoided if communication had been undertaken using HTTP. In general, service meshes favor HTTP as the protocol of choice, so a distributed system like a lightweight microservice architecture that only communicates via Restful Interfaces would be least impacted by introducing a service mesh.

Q4: How do the complexity and overhead of a service mesh impact performance and allocation of resources?

As seen with the analysis conducted in Chapter 6.1.1 and 6.1.2, the effects on the HAFAS environment of the case study have been relatively mild, resulting in a four-megabyte footprint for each sidecar proxy, a constant latency overhead of three to four milliseconds up until the 90th percentile and a two percent CPU overhead targeting the sidecar proxies and enough resources to support the three to four pods of the Linkerd control plane. So far, these results look promising regarding considering service mesh technology for productive environments for HAFAS. The results from the case study are likely not transferable to every other distributed system. However, they indicate that the performance overhead of Linkerd and, thus, a service mesh is not a limiting factor.

Q5: What are the complications of introducing a service mesh in a practical scenario, such as HAFAS?

The complications have been mild or non-existent for the scenarios *S1* and *S2*. Misconfiguration of opaque ports for the engine has caused the HAMM and API Server to not communicate with the engine. However, in that case, Linkerd has also automatically issued a warning when running the check command. When comparing the HAFAS architecture diagram of Figure 7 in Chapter 3.2 with the case study HAFAS environment diagram of Figure 21 in Chapter 4.4.2, it becomes apparent that the case study is just the beginning and a mere proof of concept on feasibility. Introducing a service mesh to the additional components absent from the case study could introduce new challenges. Investigating the scenarios *S3..S5* that were out of the scope of the case study in practice could also introduce further challenges. The case study has partly answered this question; however, it is still relevant for the outlook.

7 Outlook

This work explored service meshes in a practical scenario. It demonstrated with a proof of concept case study that a service mesh, based on satisfying requirements defined by a system, can seamlessly integrate with a distributed system not designed with service mesh support in mind. The case study also demonstrates that the performance overhead introduced by the sidecar proxies of a service mesh is, in this case, minor and an acceptable trade-off for the gained features. Defining functional and non-functional requirements as criteria and exploring the service mesh technology landscape demonstrates that a vast array of service mesh projects exist and that the ecosystem is healthy. Linkerd and Istio represent two open-source CNCF-backed graduated projects displaying signs of maturity through utilization in productive staging areas by organizations worldwide. Through the identified scenarios and their practical application with the case study, the security properties of confidentiality, integrity, and authenticity targeting cluster internal service-to-service communication improve significantly through features such as mTLS encryption and automated certificate management provided by a service mesh. Generally, mTLS is one of the main aspects of a service mesh, as adding identities paves the way for zero-trust security architecture through features such as service authorization policies. Although not validated in the case study, features related to resiliency and robustness, such as circuit breaking and fault injection, provide the potential for organizations wishing to adopt service meshes to their technology stack to improve service stability and support their teams with testing and debugging of services. When diverging from the scope of the practical case study, service meshes demonstrate a remarkable ability to standardize the configurability of distributed applications operated in Kubernetes clusters. Organizations maintaining a large-scale set of services should investigate their requirements on a service mesh and, given the base of a stable Kubernetes infrastructure, investigate using a service mesh to outsource the functionality of their services to the service mesh. Doing so simplifies the development process for each team behind a service as the set of maintainable source code and architectural overhead minimizes. While the service mesh provides configuration management and unification for cloud-hosted distributed systems, a unified interface abstracting the configuration to multiple service mesh projects is absent from the ecosystem. An organization committing to service meshes thus needs to carefully select its service mesh project with long-term stability and feature compatibility in mind. Committing to service mesh projects implies, to some extent, a vendor lock to a configuration management provider: Say a large-scale organization outsources parts of its service level functionalities to a service mesh and, after a given timeframe, identifies its service mesh solution of choice not aligning with their goals anymore, migrating away to a different service mesh would involve a large scale migration as service mesh configurations are not compatible between service mesh projects. *Service Mesh Interface* (SMI) aims to solve this limitation by providing standardized interfaces for features implementable by projects wishing to improve mesh-to-mesh configuration compatibility. As of October 3rd, 2023, the CNCF has archived the project for inactivity[CNC23], and no equivalent is known. With service mesh technology appearing to be in the stage of early majority adoption, it is likely that with the continuous adoption of Kubernetes by organizations, the focus will shift to more mainstream adoption in the coming decade by organizations following cloud-native principles. With the further development of service mesh projects, their offered completeness and maturity will likely saturate, and the focus on intra-service-mesh configuration compatibility will likely improve the more

attention the topic receives, thus minimizing configuration vendor lock-in in the process. With full compatibility with Kubernetes, most service mesh projects have already eliminated any cloud provider-enforced attempts of vendor lock-in.

Coming back to the context of HAFAS, with the results from the case study, Hacon will investigate service mesh technology further to adapt it to their technology stack eventually. This process will require time and dedication, as deploying a service mesh to production is a different scenario than creating a proof of work case study. With the adoption of service meshes at Hacon, it seems likely that functionalities currently provided by services, such as the circuit breaker capabilities, will likely be outsourced to the service mesh to reduce service complexity for the developing teams. Lastly, the findings on performance impacts of the service mesh through the case study are likely not transferable to all other systems in that they match precisely. They provide a general hint that introducing a service mesh may not impede performance by negative means and that resource consumption is acceptable. However, organizations wishing to adopt a service mesh should run their tests to investigate cost concerns. A service mesh is likely to be viewed differently in real-time heavy application scenarios where one or two milliseconds of delay can already have severe consequences. Whether or not one needs a service mesh cannot be stated in one general sentence; each organization must investigate its needs and decide based on them.

Bibliography

[AA05] Aybüke Aurum, Claes W.: *Engineering and Managing Software Requirements*. 1. Springer, 2005. – ISBN 9783540250432

[AWSa] AWS: *AWS App Mesh introduces circuit breaker capabilities*. https://aws.amazon.com/about-aws/whats-new/2020/11/aws-app-mesh-introduces-circuit-breaker-capabilities/. – Version: 9.11.2020; Accessed: 10.11.2023

[AWSb] AWS: *How AWS App Mesh works with IAM*. https://docs.aws.amazon.com/app-mesh/latest/userguide/security-iam.html. – Accessed: 10.11.2023

[AWSc] AWS: *Transport Layer Security (TLS)*. https://docs.aws.amazon.com/app-mesh/latest/userguide/tls.html. – Accessed: 10.11.2023

[Bal23] Balasubramanian, Sundar: *mTLS in Linkerd*. https://faun.pub/mtls-in-linkerd-9ccb2754cabf. Version: March 2023. – Accessed: 1.11.2023

[Bel21] Belagatti, Pavan: *Service Mesh: Winning Ingredient for Cloud-Native Enterprises*. https://itnext.io/service-mesh-winning-ingredient-for-cloud-native-enterprises-10c714df7e8d. Version: April 2021. – Accessed at 29.11.2023

[Buo] Buoyant: *Linkerd vs Istio: A 2023 service mesh comparison*. https://buoyant.io/linkerd-vs-istio. – Accessed: 10.11.2023

[BW18] Baier, Jonathan ; White, Jesse: *Getting Started with Kubernetes*. 3. Birmingham, England : Packt Publishing, 2018. – ISBN 9781788994729

[CB20a] Calcote, Lee ; Butcher, Zack: *Istio - Service Mesh für Microservices*. Sebastopol : O'Reilly, 2020. – ISBN 978–3–960–10411–7

[CB20b] Chandramouli, Ramaswamy ; Butcher, Zack: *Building secure microservices-based applications using service-mesh architecture*. http://dx.doi.org/10.6028/nist.sp.800-204a. Version: May 2020

[Cha23] Chandrakant, Kumar: *Service Mesh Architecture with Istio*. https://www.baeldung.com/ops/istio-service-mesh. Version: July 2023. – Accessed at 7.8.2023

[CJB23] Calcote, Lee ; Jackson, Nic ; Bouwer, Paul: *Service Mesh Patterns*. O'Reilly Media, 2023. – ISBN 9781492086383

[CNCa] CNCF: *Cloud Native Computing Foundation Announces Linkerd Graduation*. https://www.cncf.io/announcements/2021/07/28/cloud-native-computing-foundation-announces-linkerd-graduation/. – Version: 28.7.2021; Accessed: 10.11.2023

[CNCb] CNCF: *CNCF Cloud Native Interactive Landscape: Service Mesh*. https://landscape.cncf.io/card-mode?category=service-mesh&grouping=no. – Accessed: 10.11.2023 / Last Update of Data: 2023-11-09T02:46:23

[CNCc] CNCF: *CNCF Cloud Native Landscape*. https://landscape.cncf.io/images/landscape.pdf. – Accessed: 1.10.2023 / Version: 1.0

[CNC23] CNCF: *CNCF Archives the Service Mesh Interface (SMI) Project*. https://www.cncf.io/blog/2023/10/03/cncf-archives-the-service-mesh-interface-smi-project/. Version: October 2023. – Accessed at 30.11.2023

[Cona] Consul: *How Service Mesh Works*. https://developer.hashicorp.com/consul/docs/connect/connect-internals. – Version: Consul v1.17.x; Accessed: 10.11.2023

[Conb] Consul: *Implement circuit breaking in Consul service mesh with Envoy*. https://developer.hashicorp.com/consul/tutorials/developer-mesh/service-mesh-circuit-breaking. – Accessed: 10.11.2023

[Conc] Consul: *Service intentions overview*. https://developer.hashicorp.com/consul/docs/connect/ca. – Version: Consul v1.17.x; Accessed: 10.11.2023

[Cond] Consul: *Service Mesh Certificate Authority Overview*. https://developer.hashicorp.com/consul/docs/connect/ca. – Version: Consul v1.17.x; Accessed: 10.11.2023

[Dav21] David, Matei: *How Linkerd uses iptables to transparently route Kubernetes traffic*. https://linkerd.io/2021/09/23/how-linkerd-uses-iptables-to-transparently-route-kubernetes-traffic/. Version: September 2021. – Accessed: 1.11.2023

[DMFC23] Duarte Maia, Joao T. ; Figueiredo Correia, Filipe: Service Mesh Patterns. In: *Proceedings of the 27th European Conference on Pattern Languages of Programs*, 2023

[DMG07] Duvall, Paul M. ; Matyas, Steve ; Glover, Andrew: *Continuous integration*. Boston, MA : Addison-Wesley Educational, 2007. – ISBN 9780321336385

[Doc] Docs, NGINX: *What is NGINX Service Mesh?* https://docs.nginx.com/nginx-service-mesh/about/what-is-nsm/. – Accessed: 1.10.2023

[Dod08] Dodge, Yadolah: *The Concise Encyclopedia of Statistics*. Springer New York, 2008. – ISBN 978–0–387–32833–1

[Dor20] Dordal, Peter L.: *An Introduction to Computer Networks*. 2. 2020

[Elb21] Elbe, Djalal: *PaaS vs IaaS vs SaaS — differences, pros, and cons.* https://www.artifakt.com/blog/paas/paas-vs-iaas-vs-saas-differences-pros-and-cons/. Version: July 2021. – Accessed at 31.07.2023

[Fly23] Flynn: *Workshop recap: A deep dive into Kubernetes mTLS with Linkerd.* https://linkerd.io/2023/01/30/mtls-and-linkerd/. Version: January 2023. – Accessed: 10.11.2023

[Fre22] Freter, Robert: *Department of Defense (DoD) Zero Trust Reference Architecture.* https://dodcio.defense.gov/Portals/0/Documents/Library/(U)ZT_RA_v2.0(U)_Sep22.pdf. Version: July 2022

[git23] *Open source's impact on the world's 100 million developers.* https://github.blog/2023-02-01-open-sources-impact-on-the-worlds-100-million-developers/. Version: Febuary 2023. – Accessed: 1.11.2023

[Haca] Hacon: *Apps For Seamless Mobility: Plan, Book, Pay and Ride.* https://www.hacon.de/fileadmin/user_upload/Portfolio/Factsheets/HAFAS/HAFAS.mobile_english.pdf. – Accessed at 7.8.2023

[Hacb] Hacon: *Integrating Applications: Programming Interfaces: Provisioning & Management.* https://www.hacon.de/fileadmin/user_upload/Portfolio/Factsheets/HAFAS/HAFAS.api_english.pdf. – Accessed at 7.8.2023

[Hacc] Hacon: *Mobility as a service: Plan, Book and Ride.* https://www.hacon.de/fileadmin/user_upload/Portfolio/Factsheets/HAFAS/HAFAS.maas_english.pdf. – Accessed at 7.8.2023

[Hacd] Hacon: *Route Planning On The Web: Responsive Timetable Information - Anytime and Anywhere.* https://www.hacon.de/fileadmin/user_upload/Portfolio/Factsheets/HAFAS/HAFAS.webapp_english.pdf. – Accessed at 7.8.2023

[Hace] Hacon: *Timetable Information System: The algorithm behind travel planning.* https://www.hacon.de/fileadmin/user_upload/Portfolio/Factsheets/HAFAS/HAFAS.engine_english.pdf. – Accessed at 6.6.2023

[HF10] Humble, Jez ; Farley, David: *Continuous delivery.* Boston, MA : Addison-Wesley Educational, 2010. – ISBN 9780321601919

[Ing18] Ingeno, Joseph: *Software Architect's Handbook.* Birmingham, England : Packt Publishing, 2018. – ISBN 9781788624060

[IS18] Indrasiri, Kasun ; Siriwardena, Prabath: *Microservices for the Enterprise: Service Mesh.* 2018 https://doi.org/10.1007/978-1-4842-3858-5_9

[Ista] Istio: *Application Requirements.* https://istio.io/latest/docs/ops/deployment/requirements/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Istb] Istio: *Circuit Breaking*. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Istc] Istio: *Fault Injection*. https://istio.io/latest/docs/tasks/traffic-management/fault-injection/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Istd] Istio: *Plug in CA Certificates*. https://istio.io/latest/docs/tasks/security/cert-management/plugin-ca-cert/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Iste] Istio: *Protocol Selection*. https://istio.io/latest/docs/ops/configuration/traffic-management/protocol-selection/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Istf] Istio: *Security*. https://istio.io/latest/docs/concepts/security/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Istg] Istio: *Upgrade Istio*. https://istio.io/latest/docs/setup/upgrade/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Isth] Istio: *Using the Istioctl Command-line Tool*. https://istio.io/latest/docs/ops/diagnostic-tools/istioctl/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Isti] Istio: *Visualizing Your Mesh*. https://istio.io/latest/docs/tasks/observability/kiali/. – Version: Istio 1.19.3; Accessed: 10.11.2023

[Jos07] Josuttis, Nicolai M.: *SOA in practice*. Sebastopol, CA : O'Reilly Media, 2007. – ISBN 9780596529550

[JPM+18] Jamshidi, Pooyan ; Pahl, Claus ; Mendonça, Nabor C. ; Lewis, James ; Tilkov, Stefan: Microservices: The Journey So Far and Challenges Ahead. In: *IEEE Software* 35 (2018), Nr. 3, S. 24–35. http://dx.doi.org/10.1109/MS.2018.2141039. – DOI 10.1109/MS.2018.2141039

[KBB+21] Koschel, Arne ; Bertram, Marvin ; Bischof, Richard ; Schulze, Kevin ; Schaaf, Marc ; Astrova, Irina: A Look at Service Meshes. In: *2021 12th International Conference on Information, Intelligence, Systems & Applications (IISA)*, IEEE, July 2021

[Kra18] Kratzke, Nane: *A Brief History of Cloud Application Architectures From Deployment Monoliths via Microservices to Serverless Architectures and Possible Roads Ahead - A Review from the Frontline (invited paper)*

[Kuba] Kubernetes: *Assigning Pods to Nodes*. https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#affinity-and-anti-affinity. – Last modified October 17, 2023 at 2:40 AM PST

[Kubb] Kubernetes: *ConfigMaps*. https://kubernetes.io/docs/concepts/configuration/configmap/. – Last modified August 24, 2023 at 6:38 PM PST

[Kubc] Kubernetes: *Custom Resources*. https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/. – Last modified August 7, 2023 at 10:10 PM PST

[Kubd] Kubernetes: *Deployments*. https://kubernetes.io/docs/concepts/workloads/controllers/. – Last modified September 06, 2023 at 10:16 PM PST

[Kube] Kubernetes: *Deployments*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/. – Last modified September 06, 2023 at 10:16 PM PST

[Kubf] Kubernetes: *Kubernetes Components*. https://kubernetes.io/docs/concepts/overview/components/. – Last modified July 12, 2023 at 1:25 AM PST

[Kubg] Kubernetes: *Labels and Selectors*. https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/. – Last modified September 1, 2023 at 12:50 PM PST

[Kubh] Kubernetes: *Namespaces*. https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/. – Last modified June 29, 2023 at 12:14 AM PST

[Kubi] Kubernetes: *Network Policies*. https://kubernetes.io/docs/concepts/services-networking/network-policies/. – Last modified August 24, 2023 at 1:36 PM PST

[Kubj] Kubernetes: *Overview*. https://kubernetes.io/docs/concepts/overview/. – Last modified January 03, 2023 at 10:07 AM PST

[Kubk] Kubernetes: *Pods*. https://kubernetes.io/docs/concepts/workloads/pods/. – Last modified August 24, 2023 at 6:38 AM PST

[Kubl] Kubernetes: *Resource metrics pipeline*. https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/. – Last modified August 02, 2023 at 10:04 AM PST

[Kubm] Kubernetes: *Service*. https://kubernetes.io/docs/concepts/services-networking/service/. – Last modified October 13, 2023 at 9:12 PM PST

[Kubn] Kubernetes: *StatefulSets*. https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/. – Last modified June 29, 2023 at 12:14 AM PST

[Kubo] Kubernetes: *Taints and Tolerations*. https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/. – Last modified October 09, 2023 at 8:04 PM PST

[Kubp] Kubernetes: *What Kubernetes is not*. https://kubernetes.io/docs/concepts/overview/#what-kubernetes-is-not. – Last modified July 12, 2023 at 1:25 AM PST

[Kubq] Kubernetes: *Why you need Kubernetes and what it can do*. https://kubernetes.io/docs/concepts/overview/#why-you-need-kubernetes-and-what-can-it-do. – Last modified July 12, 2023 at 1:25 AM PST

[Lina] Linkerd: *Architecture*. https://linkerd.io/2.14/reference/architecture/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linb] Linkerd: *Automatic mTLS*. https://linkerd.io/2.14/features/automatic-mtls/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linc] Linkerd: *Automatically Rotating Control Plane TLS Credentials*. https://linkerd.io/2.14/tasks/automatically-rotating-control-plane-tls-credentials/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Lind] Linkerd: *Automatically Rotating Control Plane TLS Credentials*. https://linkerd.io/2.14/tasks/automatically-rotating-control-plane-tls-credentials/#. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Line] Linkerd: *Circuit Breaking*. https://linkerd.io/2.14/reference/circuit-breaking/#probation-and-backoffs. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linf] Linkerd: *CLI*. https://linkerd.io/2.14/reference/cli/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Ling] Linkerd: *Extensions List*. https://linkerd.io/2.14/reference/extension-list/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linh] Linkerd: *Handling ingress traffic*. https://linkerd.io/2.14/tasks/using-ingress/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Lini] Linkerd: *HTTP, HTTP/2, and gRPC Proxying*. https://linkerd.io/2.14/features/http-grpc/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linj] Linkerd: *HTTPRoute*. https://linkerd.io/2.14/reference/httproute/#httproutetimeouts. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Link] Linkerd: *Injecting Faults*. https://linkerd.io/2.14/features/fault-injection/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linl] Linkerd: *Installing Linkerd*. https://linkerd.io/2.14/tasks/install/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linm] Linkerd: *Manually Rotating Control Plane TLS Credentials*. https://linkerd.io/2.14/tasks/manually-rotating-control-plane-tls-credentials/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linn] Linkerd: *Reference: Authorization Policy*. https://linkerd.io/2.14/reference/authorization-policy/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Lino] Linkerd: *Upgrading Linkerd*. https://linkerd.io/2.14/tasks/upgrade/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[Linp] Linkerd: *Using the Debug Sidecar*. https://linkerd.io/2.14/tasks/using-the-debug-container/. – Version: Linkerd 2.14; Accessed: 10.11.2023

[LLG+19] Li, Wubin ; Lemieux, Yves ; Gao, Jing ; Zhao, Zhuofeng ; Han, Yanbo: *Service Mesh: Challenges, State of the Art, and Future Research Opportunities*. http://dx.doi.org/10.1109/sose.2019.00026. Version: April 2019

[Mei22] Meinders, Katie: *CNCF Sees Record Kubernetes and Container Adoption in 2021 Cloud Native Survey*. https://www.cncf.io/announcements/2022/02/10/cncf-sees-record-kubernetes-and-container-adoption-in-2021-cloud-native-survey/. Version: Febuary 2022. – Accessed at 31.07.2023

[Mit90] Mitchell, R J. (Hrsg.): *Managing Complexity in Software Engineering*. Stevenage, England : Institution of Engineering and Technology, 1990 (Computing and Networks)

[ML] Michael Landwehr, Robert B.: *IMS Intermodal Mobility Sales Platform (MaaS): High-Level Architecture*. https://intranet/display/IBDPMSP/High-Level+Architecture. – Accessed at 1.9.2023, Copy Available in Appendix with Figure 41

[Mora] Morgan, William: *linkerd2*. https://www.bestpractices.dev/en/projects/4629. – Version: 2021-03-17 15:58:46 UTC; Accessed: 10.11.2023

[Morb] Morgan, William: *Zero trust network security in Kubernetes with the service mesh*. https://buoyant.io/resources/zero-trust-in-kubernetes-with-linkerd. – Accessed: 10.11.2023

[NK00] Nystrom, Magnus ; Kaliski, Burt: *PKCS #10: Certification Request Syntax Specification Version 1.7*. RFC 2986. http://dx.doi.org/10.17487/RFC2986. Version: November 2000 (Request for Comments)

[Ope] OpenSSF: *OpenSSF Best Practices Badge Program*. https://www.bestpractices.dev/en. – Accessed: 1.11.2023

[Ped21] Pedraza, Alejandro: *Go directly to namespace jail: Locking down network traffic between Kubernetes namespaces*. https://buoyant.io/blog/locking-down-network-traffic-between-kubernetes-namespaces. Version: December 2021. – Accessed: 10.11.2023

[Pra23] Pramanick, Sumanta: *What is a Service Mesh, and why do you need it?* https://www.kellton.com/kellton-tech-blog/what-is-a-service-mesh-and-why-do-you-need-it. Version: Febuary 2023. – Accessed at 7.8.2023

[Pre21] Pretzer, Charles: *Protocol Detection and Opaque Ports in Linkerd*. https://linkerd.io/2021/02/23/protocol-detection-and-opaque-ports-in-linkerd/. Version: Febuary 2021. – Accessed: 10.11.2023

[Raj] Rajagopalan, Shriram: *Istio*. https://www.bestpractices.dev/en/projects/1395. – Version: 2022-03-31 03:26:47 UTC; Accessed: 10.11.2023

[Red22] RedHat: *What is container orchestration?* https://www.redhat.com/en/topics/containers/what-is-container-orchestration. Version: May 2022

[Riv92] Rivest, Ronald L.: *The MD5 Message-Digest Algorithm*. RFC 1321. http://dx.doi.org/10.17487/RFC1321. Version: April 1992 (Request for Comments)

[Ros20] Rosenthal, Casey: *Chaos Engineering*. 2020. – ISBN 1492043869

[SF23] Sanni, Michael ; Fernandez, Tomas: *The Service Mesh Landscape in 2023*. https://semaphoreci.com/blog/service-mesh-landscape. Version: June 2023. – Accessed at 7.8.2023

[SG18] Smith, Floyd ; Garrett, Owen: *What Is a Service Mesh?* https://www.nginx.com/blog/what-is-a-service-mesh/. Version: 2018. – Accessed at 7.8.2023

[Tig] Tigera: *Service Mesh: Benefits, Challenges, and 7 Key Concepts.* https://www.tigera.io/learn/guides/devsecops/service-mesh/. – Accessed at 7.8.2023

[Tiw17] Tiwari, Abhishek: *A sidecar for your service mesh.* https://www.abhishek-tiwari.com/a-sidecar-for-your-service-mesh/. Version: June 2017. – Accessed at 7.8.2023

[TSHJ12] Topalovic, Emin ; Saeta, Brennan ; Huang, Lin-Shung ; Jackson, Collin: Towards Short-Lived Certificates, 2012

[Wik] Wikipedia: *HAFAS — Wikipedia, die freie Enzyklopädie.* https://de.wikipedia.org/w/index.php?title=HAFAS&oldid=233503210. – [Online; Version 23. July 2023]

Appendix

The Appendix contains listings, graphics, and other documents relevant to the thesis but disrupt the reading flow. The physical Appendix of this thesis will only have graphics, documents, and selected listings. Access to digital artifacts referenced in this work, such as Kubernetes YAML configuration files, recorded network traffic, configuration files regarding the load test, and resulting raw data, is given via the *Electronic Appendix* included as a DVD. This separation makes it easier for readers to access the configuration files in a sensible and searchable format, utilizing their editor of choice for viewing. Raw data, such as the recorded network traffic or results from the load tests, are best inspected with specialized tools, and including them in the physical Appendix would complicate sensible access for readers.

The physcial Appendix contains the guidelines for the expert interview, a transcription of the interview, an architectural overview of Hacon's MaaS platform detailing the internals of HAFAS, and technical artifacts related to conducting the case study. This thesis is also available as a *Portable Document Format* (PDF) file in the *Electronic Appendix*. To ensure the completeness and correctness of the *Electronic Appendix*, `hashes.md5` MD5[1] hashes for all files[2]. These are useful for verifying the integrity of file content. The *master hash* for the `hashes.md5` file is given below.

`07e4bc2518da764bc9ea5ec28dca69a7`

If the DVD of the Electronic Appendix is not in a cover glued to the inside of the hardcover of the thesis, please get in touch with the author of the thesis immediately via the e-mail address given on page 2!

[1] [Riv92]

[2] Excluding the file `thesis.pdf` so the master hash of this page matches `hashes.md5`.

MaaS Platform Overview

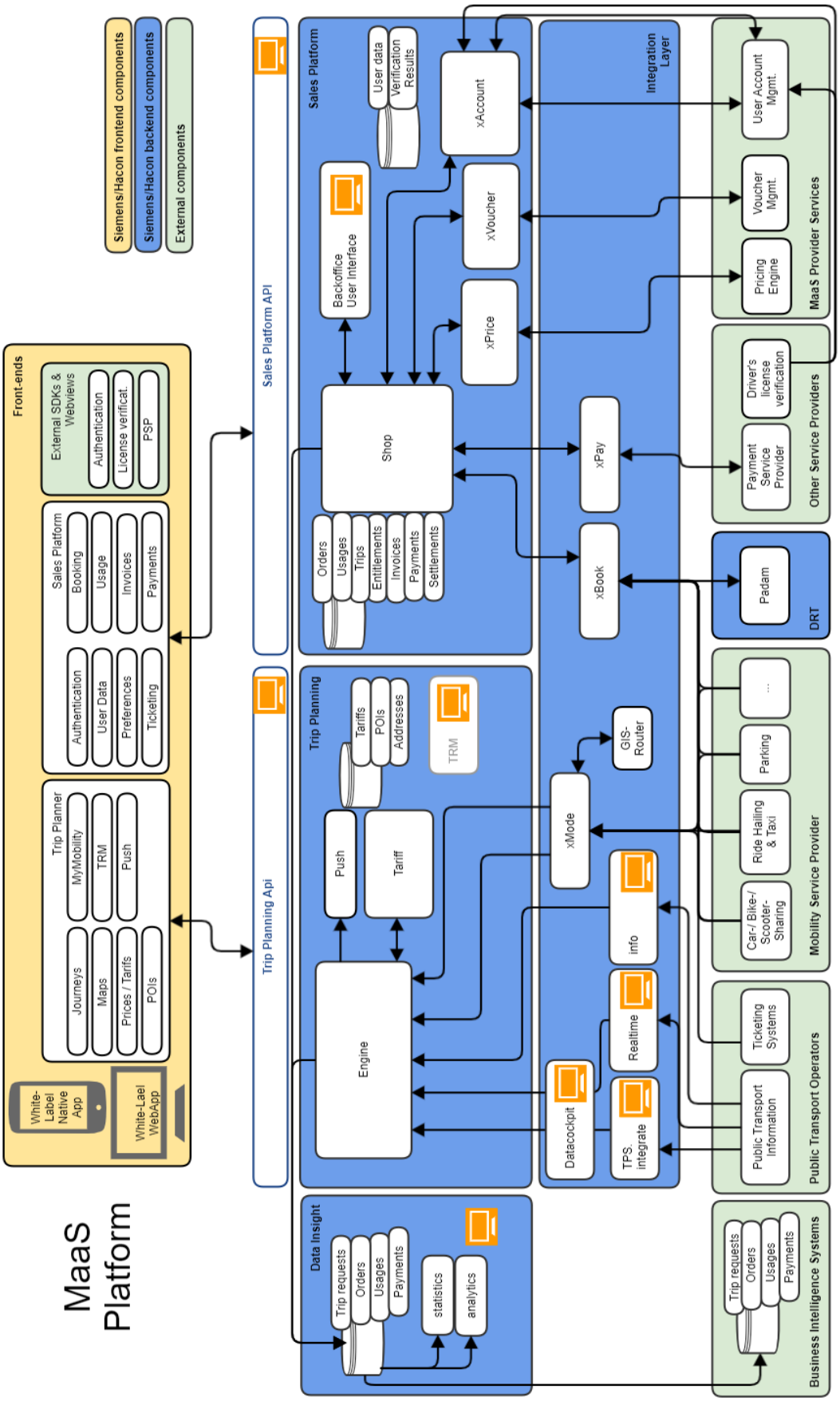

Figure 41: MaaS Platform Overview (Sourced from Hacon Intranet[ML])

Guidelines for the Expert Interview

Interview partner: **Fabian Korte**, *Cloud Architect* at *Hacon Ingenieurgesellschaft mbH*

The interview will be conducted using the questions listed below. The questions provide a loose framework and lead the direction that the interview will follow. As the interview is held via a Slack call, the resulting feedback from the answers of the interview partner might spawn new questions that are not listed in this guideline.

Fabian Korte is the cloud architect at Hacon. As a member of the department of hosting and operations, he is involved in all decisions regarding cloud architecture. He is a driving force behind the migration from HAFAS to Kubernetes and has yearlong experience with most of Hacon's productive systems. Because of his expertise, Fabian Korte has a detailed overview of the whole scope of a cloud-hosted HAFAS environment and has a lot of insight into how the involved services work together. Because of his qualifications, Fabian Korte has been chosen as the interview partner to conduct this expert interview with. The aim of this interview is to better understand the expectations that are set when introducing a service mesh to HAFAS and in which scenarios a service mesh would prove to be useful. Part of this interview is also trying to figure out potential requirements and criteria for selecting a service mesh project for the case study. Fabian Korte has agreed with having his name attached to the interview; it has been decided against anonymizing the interview as he qualifies for the distinct expert at Hacon for this domain.

Q1: What are your main expectations for introducing a service mesh to HAFAS?

Q2: Which scenarios do you see in HAFAS that warrant introducing a service mesh to it?

Q3: Do you think that there are features of a service mesh that cannot be used with HAFAS at the moment?

Q4: How do you feel about proprietary service mesh projects?

Q5: How do you feel about open-source service mesh projects?

Q6: What would you see as the main requirements for a service mesh at Hacon?

Q7: Do you see a service mesh as mainly a tool to solve a specific requirement, or are you interested in its broad range of features?

Q8: Do you see a future at Hacon where a service mesh is actively used to separate the concerns of business logic and network functionality?

Q9: What kind of resource overhead and latency would be acceptable for you in order to think about using a service mesh in a productive environment?

Transcript of the Expert Interview

00:00:00 - 00:00:04 ***Marc Herschel***: What are your main expectations for introducing a service mesh to HAFAS?

00:00:08 - 00:01:52 ***Fabian Korte***: Let me tell you a little about the history. So currently we are about moving more and more of our systems to Kubernetes. And on the other hand, we are moving more and more systems to public clouds. In most cases, this is AWS. And we get rising security requirements because of that. So customers ask for encryption in transit of all data, for example. And that's something we do not do by default in all parts of our systems. And that is exactly a point where it is expected that a service mesh would help us. So encryption of traffic in the inter-component communication and then also in a co-hosted environment better isolation of the systems from each other. We hope that service mesh capabilities would ease the implementation of the features. Of course, it's something that we could also implement on component level, but we would hope that a service mesh would deliver the same features without too much overhead.

00:01:53 - 00:02:16 ***Marc Herschel***: Thank you for your answer. So what I'm reading out is that you are mainly interested in security features of the service mesh. What do you think about the traffic management or monitoring and observability features? Do you see that as interesting or as a priority?

00:02:19 - 00:03:38 ***Fabian Korte***: I mean, for observability. I do not see too many use cases currently because we have our own monitoring stack already, which also monitors traffic within the system. And also we have metrics from the components that deliver enough data for the inter component communication. But for traffic management, there could be interesting scenarios, for example, for blue-green deployments or cannery deployments and also for example, traffic mirroring for debugging purposes. But that's something I think we can not only solve. Or if you would like to introduce it to, to HAFAS, then we could not solve it solely on the network level because we have also data dependencies. And this makes the pure blue-green deployment hard to achieve. So it's interesting to look at these features, but it wouldn't be our main use case or main interest.

00:03:40 - 00:04:18 ***Marc Herschel***: Thank you. So. In question two, we are discussing which scenarios do you see in HAFAS that warrant introducing a service mesh to it? I think this can kind of be answered already in a way that you are at first seeing mTLs encryption as an interesting scenario, and secondly, possibly because of encryption. You are also seeing an interest in using the certificate management and rotating features that a service mesh provides.

00:04:18 - 00:04:41 ***Fabian Korte***: Yes. I mean the main use case we want to have or we need to make sure that the solution does not come with a lot of overhead. So, of course, we would need to have a solution that allows us to also manage or to be manageable without too many resources.

00:04:43 - 00:04:50 ***Marc Herschel***: How do you feel about the features of circuit breaking that some service measures provide?

00:04:54 - 00:06:07 ***Fabian Korte***: That's something that is an interesting feature, but it wouldn't be an unique selling point for us. I mean, we have components, central

components that are used for delivering specific features. And if these components are getting blocked for by any circumstances or that they are not running anymore, and they take too much time to answer. This can block certain parts of the system. And then it might be helpful to shut down the functionality for a while basically. So that's a use case where circuit breaking might be an option, but such functionality is already partially implemented in our components itself. So some might already be an overlap in functionality. So, it's not something we would solely achieve by introducing a service mesh.

00:06:08 - 00:06:34 ***Marc Herschel***: All right. Thank you. And the last feature that I would ask you about is fault injection, which is provided by some service mesh projects to debug and test the robustness of an application by injecting faulty requests on a service mesh level.

00:06:36 - 00:07:09 ***Fabian Korte***: Yes. I mean, my perspective is currently from the hosting and operations perspective. That's a scenario or an environment where we wouldn't use fault injection features. But that's something that could be definitely interesting for our development teams and testing environments. So, for testing and debugging environments, I guess it could also make good use of fault injection features.

00:07:09 - 00:07:35 ***Marc Herschel***: All right. Thank you for your answer. I would say we skip over question three as this already has some overlap with some of the answers you've provided before. When we are looking at service mesh projects and Hacon. How would you feel about proprietary or open-source service mesh projects? Is this an important criterion for selecting a service mesh project?

00:07:39 - 00:08:41 ***Fabian Korte***: So in general we thrive to stay provider or cloud provider independent. And we wouldn't pick a service mesh where we would need to fear some kind of provider lock-in. On the other hand, we need to consider the overhead. We would have to introduce a service mesh and the costs. So I mean, since we are hosting a lot of our systems in AWS, also, we might consider using a native solution like the app mesh. So, in general, we are open in both directions. It depends on the efforts and the costs in the end.

00:08:42 - 00:09:55 ***Marc Herschel***: All right. Thank you. Looking at the requirements on a service mesh, I would guess a requirement is that it A) Spans and covers all the scenarios that Hacon is interested in. And B) are there any concerns on the maturity level of a service mesh project, as they are a lot of service mesh projects in the wild in various states of implementation, and would you see a requirement is also that the project is popular, which means there is a large amount on resources available online when you are seeking help with working with the project?

00:10:00 - 00:10:50 ***Fabian Korte***: Yes. I mean, that's definitely an important point regarding open-source projects. So we wouldn't pick a solution or an implementation that is used only by a small user basis, but it has to have a certain degree of maturity and also a certain degree of user adoption. If you went for an open-source project for proprietary services, I would assume that they have a maturity level that is high enough so that we can use them. But yes, for open source projects, it's always the question of where to get the support and if they are or will be correctly maintained in the future.

00:10:51 - 00:11:42 ***Marc Herschel***: Yeah, I can understand that. The CNCF has some good resources on that, and some open source service measures such as Linkerd also have paid support, apparently. Would you see a service mesh as mainly a tool to solve a specific requirement, or are you interested in its broad range of features? I think this question we can skip because previous answer of you has already included that you are mainly interested in the security features of a service mesh. Going on to question eight, would you see a future at Hacon on where service mesh is actively used to separate the concerns of business logic and network functionality?

00:11:47 - 00:12:14 ***Fabian Korte***: Maybe you can elaborate a little on the question or what I mean, how far I understand some, or our goal would be to have a better isolation of traffic and also a little better control about the traffic by introducing a service mesh. So what else do you see as a separate concern in the question?

00:12:14 - 00:12:54 ***Marc Herschel***: Yes. This question targets specific features were changes to the business logic have to be made, such as also distributed tracing where the service mesh is incorporated, to a point at Hacon where some business logic functionalities are outsourced to the service mesh, such as retry logic or rate limiting.

00:12:55 - 00:13:50 ***Fabian Korte***: So from my point of view. I wouldn't expect that. It still comes like that we would deeply integrate with a specific service mesh, actually, because we still have requirements on our software that the software or the components itself are also or can be hosted in different environments. These include Kubernetes-based environments, but also bare metal environments and environments based on, virtual machines. So, I wouldn't expect that we make any changes to the software that would require the software to run inside a service mesh. So yes, we will try to stay independent of that.

00:13:50 - 00:14:19 ***Marc Herschel***: All right. Thank you for the clarification. Lastly, introducing a service mesh always has a resource overhead. And it might also impact latency which most service mesh projects claim is not true. But well it always depends on the practical case at the end. What kind of resource overhead or latency would be acceptable for you in order to think about using a service mesh in a productive environment at some point?

00:14:23 - 00:15:11 ***Fabian Korte***: So from the performance point of view, I would say, well, my gut feeling would be to set like an overhead of 5 to 10% regarding latency and costs might be acceptable, but beyond that... I think it's getting harder to have a good cost-benefit ratio. So yes, we can accept some overhead regarding the operations and also regarding the costs of running the system. But yeah, I would say they need to be limited up to 10%.

00:15:11 - 00:15:45 ***Marc Herschel***: Yeah, that sounds sensible. I mean, it has to be justified that the overhead is not too large. Well, I am thanking you for your time to conduct this interview and give me a more precise insight into the motivation of introducing a service mesh at Hacon.

Kubernetes YAML Configuration Files

All Kubernetes configuration files are accessible in the *Electronic Appendix* under the path `kubernetes-config`. Deploying HAFAS-based workloads is impossible as any non-Hacon internal cluster will lack the required pull secret to access the Docker images. The configuration files, however, still result as digital artifacts of the thesis and are thus contained in the *Electronic Appendix*. Table 20 below lists the YAML content of the specific folders in detail.

To protect company secrets and allow public release of this work all sensitive data has been replaced with <CENSORED>!

Folder	Content
api-server	Configmap, Deployment, Ingress, Service
cert-management	Trust anchor CRD, Intermediate Certificate CRD
grafana	Prometheus Authorization Policy CRD, Configmap, Deployment, Ingress
hafasengine	Configmap, Deployment, Service
hamm	Configmap, Deployment, Ingress, Service
hdc	Configmap, Ingress, Service, StatefulSet
jmeter	Deployment[3]
linkerd-viz	Ingress
postgres	Service, StatefulSet

Table 20: Kubernetes YAML configuration files in the digital Appendix

Setting up Grafana and Exposing the Linkerd Dashboard

To import Linkerd's Prometheus metrics into Grafana the following approach has been followed:

- Install `grafana` from the helm repository `https://grafana.github.io/helm-charts` in the `linkerd-viz` namespace.
- Apply the authorization policy in the `grafana` folder of the digital Appendix in order to allow access to Prometheus from Grafana.
- Hook Grafana with Linkerd's *viz* extension via the command `linkerd viz install -set grafana.url=linkerd-viz.grafana:3000 | kubectl apply -f -`.
- Create the *Ingress* `grafana` to make it web-accessible.

To expose the Linkerd Dashboard provided by the *viz* extension to the web, the *Ingress* in the folder `linkerd-viz` of the *Electronic Appendix* has been created in the `linkerd-viz` namespace.

[3]The resources required for building the docker container are available under the folder `jmeter-docker` of the *Electronic Appendix*.

Linkerd's CLI Proxy Verification Output

```
kubernetes-api
--------------
OK: can initialize the client
OK: can query the Kubernetes API

kubernetes-version
------------------
OK: is running the minimum Kubernetes API version

linkerd-existence
-----------------
OK: 'linkerd-config' config map exists
OK: heartbeat ServiceAccount exist
OK: control plane replica sets are ready
OK: no unschedulable pods
OK: control plane pods are ready
OK: cluster networks contains all pods
OK: cluster networks contains all services

linkerd-config
--------------
OK: control plane Namespace exists
OK: control plane ClusterRoles exist
OK: control plane ClusterRoleBindings exist
OK: control plane ServiceAccounts exist
OK: control plane CustomResourceDefinitions exist
OK: control plane MutatingWebhookConfigurations exist
OK: control plane ValidatingWebhookConfigurations exist
OK: proxy-init container runs as root user if docker container
   runtime is used

linkerd-identity
----------------
OK: certificate config is valid
OK: trust anchors are using supported crypto algorithm
OK: trust anchors are within their validity period
OK: trust anchors are valid for at least 60 days
OK: issuer cert is using supported crypto algorithm
OK: issuer cert is within its validity period
OK: issuer cert is valid for at least 60 days
OK: issuer cert is issued by the trust anchor

linkerd-webhooks-and-apisvc-tls
-------------------------------
OK: proxy-injector webhook has valid cert
OK: proxy-injector cert is valid for at least 60 days
OK: sp-validator webhook has valid cert
OK: sp-validator cert is valid for at least 60 days
OK: policy-validator webhook has valid cert
OK: policy-validator cert is valid for at least 60 days
```

```
50
51 linkerd-identity-data-plane
52 ----------------------------
53 OK: data plane proxies certificate match CA
54
55 linkerd-version
56 ---------------
57 OK: can determine the latest version
58 OK: cli is up-to-date
59
60 linkerd-control-plane-proxy
61 ---------------------------
62 OK: control plane proxies are healthy
63 OK: control plane proxies are up-to-date
64 OK: control plane proxies and cli versions match
65
66 linkerd-data-plane
67 ------------------
68 OK: data plane namespace exists
69 OK: data plane proxies are ready
70 OK: data plane is up-to-date
71 OK: data plane and cli versions match
72 OK: data plane pod labels are configured correctly
73 OK: data plane service labels are configured correctly
74 OK: data plane service annotations are configured correctly
75 OK: opaque ports are properly annotated
76
77 linkerd-viz
78 -----------
79 OK: linkerd-viz Namespace exists
80 OK: can initialize the client
81 OK: linkerd-viz ClusterRoles exist
82 OK: linkerd-viz ClusterRoleBindings exist
83 OK: tap API server has valid cert
84 OK: tap API server cert is valid for at least 60 days
85 OK: tap API service is running
86 OK: linkerd-viz pods are injected
87 OK: viz extension pods are running
88 OK: viz extension proxies are healthy
89 OK: viz extension proxies are up-to-date
90 OK: viz extension proxies and cli versions match
91 OK: prometheus is installed and configured correctly
92 OK: viz extension self-check
93
94 linkerd-viz-data-plane
95 ----------------------
96 OK: data plane namespace exists
97 OK: prometheus is authorized to scrape data plane pods
98 OK: data plane proxy metrics are present in Prometheus
99
100 Status check results are OK
```

Listing 7: Console output of running `linkerd check -proxy` command

Insights on the Execution of Load Tests

Figure 42 shows the three extra nodes provided for both load tests. The API server and five engine replicas are distributed evenly among the nodes. Figure 43 shows the execution of a latency load test targeting the *Trip Search* capabilities of the API server. Four latency load tests have been conducted in total to collect baseline and Linkerd scenario data for the two request types.

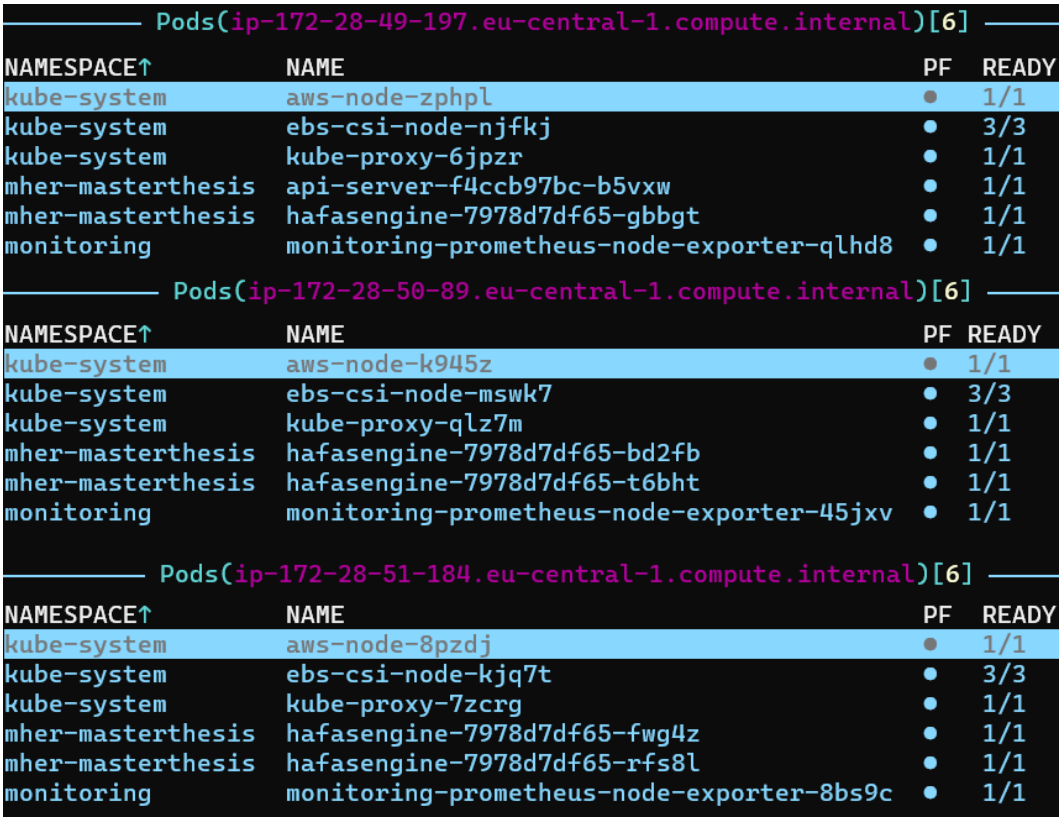

Figure 42: Pre-baseline latency load test: Involved pods isolated from other workloads to three dedicated nodes

```
== DOWNLOADING TEST PLAN @ https://bashupload.com/qVsr2/Test_plan_2.jmx ==
Connecting to bashupload.com (116.203.186.178:443)
saving to 'runtest.jmx'
runtest.jmx          100% |*****************************************************************
'runtest.jmx' saved
== DOWNLOAD TEST OK => STARTING LOAD TEST !!! ==
Creating summariser <summary>
Created the tree successfully using runtest.jmx
Starting standalone test @ November 26, 2023 6:28:55 AM CET (1700976535955)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message on port 4445
summary +     31 in 00:00:03 =    9.2/s Avg:    37 Min:    34 Max:   123 Err:     0 (0.00%)
summary +    300 in 00:00:30 =   10.0/s Avg:    35 Min:    33 Max:    54 Err:     0 (0.00%)
summary =    331 in 00:00:33 =    9.9/s Avg:    35 Min:    33 Max:   123 Err:     0 (0.00%)
summary +    300 in 00:00:30 =   10.0/s Avg:    34 Min:    32 Max:    45 Err:     0 (0.00%)
summary =    631 in 00:01:03 =   10.0/s Avg:    35 Min:    32 Max:   123 Err:     0 (0.00%)
summary +    300 in 00:00:30 =   10.0/s Avg:    34 Min:    32 Max:    40 Err:     0 (0.00%)
summary =    931 in 00:01:33 =   10.0/s Avg:    34 Min:    32 Max:   123 Err:     0 (0.00%)
```

Figure 43: Load test targeting the *Trip Search* capabilities of the API server running on the *Apache JMeter* docker container