FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

# Service Mesh Design Patterns

**João Tiago Duarte Maia**


Master in Software Engineering

Supervisor: Prof. Filipe Figueiredo Correia

July 28, 2022

# Service Mesh Design Patterns

**João Tiago Duarte Maia**

Master in Software Engineering

July 28, 2022

# Abstract

As the benefits and applicability of microservice architectures become better understood by the software industry, and this architecture becomes increasingly more adopted for building stable, independent and scalable cloud applications, a new set of concerns have alerted developers regarding communication between the different microservices.

A service mesh tries to address this issue by creating a clear separation of concerns between application logic and the infrastructure needed for the communication between the different services. This is accomplished by abstracting the cross-cutting concerns related to communication out of the internal services making it possible to be reused by the different services.

Existing literature describes a SERVICE MESH pattern and a SIDECAR pattern. This paper leans on these patterns and proposes six new patterns found by observing, what is commonly called good practices. The six patterns are SERVICE MESH, SHARED COMMUNICATION LIBRARY, NODE AGENT, SIDECAR, SERVICE MESH TEAM and MULTI-CLUSTER SERVICE MESH. In the second half of the thesis, it is conducted an experiment on two different design approaches of the service mesh using an open-source microservices demo project and hosting it on the Google Cloud Platform. This experiment is conducted in order to validate the accuracy of the consequences defined in the patterns.

**Keywords**: Service Mesh, Design Patterns, Architectural Patterns, Microservices

# Resumo

À medida que os benefícios e a aplicabilidade da arquitetura de microsserviços se tornam melhor compreendidos pela indústria de software, e essa arquitetura se torna cada vez mais usada para a construção de aplicações na cloud, independentes e escaláveis, um novo conjunto de preocupações tem alertado os engenheiros de software quanto à comunicação entre os diferentes micro serviços.

Uma service mesh tenta resolver este problema criando uma clara separação de preocupações entre a lógica da aplicação e a lógica de infraestrutura necessária para a comunicação entre os diferentes serviços. Isto é conseguido abstraindo as preocupações transversais relacionadas com a comunicação para fora dos serviços internos tornando possível a sua reutilização pelos diferentes serviços.

A literatura existente descreve o padrão SERVICE MESH e o padrão SIDECAR. Esta tese apoia-se nestes padrões e propõe seis novos padrões encontrados ao observar o que normalmente se chama de boas práticas. Os seis padrões são SERVICE MESH, SHARED COMMUNICATION LIBRARY, NODE AGENT, SIDECAR, SERVICE MESH TEAM e MULTI-CLUSTER SERVICE MESH. Na segunda metade da tese é demonstrada uma experiência de duas abordagens de design diferente de service mesh usando um projeto de demonstração de microsserviços de código aberto e hospedando-o no Google Cloud Platform.

**Keywords**: Service Mesh, Design Patterns, Architectural Patterns, Microservices

# Acknowledgements

First of all I'd like to thank my supervisor Professor Filipe Alexandre Pais de Figueiredo Correia for all his help, patience, availability and enthusiasm. I could always count on him whenever I ran into a problem or had a question about my research or writing at any time of the day.

Thanks to all my friends and colleagues that helped and encouraged me along the way. Special thanks to my friend Xavier Cravo for his help during the whole process and by keeping me motivated. I'd also like to express my sincere gratitude to all the wonderful faculty at FEUP that have helped me learn over this two years.

Would like to thank my girlfriend for all the love, the laughs, the friendship and also for being able to deal with me in the good but also the bad days that had to be lived in order to finish this thesis.

Finally, I must express my very profound gratitude to my parents, my sister and my other family members for providing me with unfailing support throughout my years of study and through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you all.

Tiago Maia

*"The only impossible journey is the one you never begin"*

Tony Robbins

# Contents

# List of Figures

# List of Tables

# Abbreviations and Symbols

| | |
|---|---|
| GCP | Google Cloud Platform |
| GKE | Google Kubernetes Engine |
| CPU | Central Processing Unit |
| GiB | Gibibyte |

# Chapter 1

# Introduction

## 1.1 Context

It is hard to argue against the value of existing design and architectural patterns documented in the software design literature. They help developers from all software domains to improve their work with cataloged good solutions for commonly recurring problems. Nevertheless, the documentation of such good solutions using patterns has yet to cover the challenges and solutions faced by all software developers anywhere. In particular, new architectural tactics and practices keep emerging, supported by the increasing maturity of the technologies for managing and orchestrating microservices systems. One of such emerging solutions is the *service mesh*.

Service meshes are proposed with the premise of bridging some of the liabilities of the microservices when the microservices architecture pattern is increasingly being used among companies [44]. The service mesh creates an independent and standardized way to control and measure the traffic inside a cluster of microservices by removing all the communication logic between the different services that are not directly related to the business logic (cross-cutting concerns related to communication) out of the specific services. These services will instead use the new capabilities supplied by the service mesh enabling developers to have better control over the entire application thus improving reliability, security and visibility. However, compared to other technologies present in the software industry, we currently have a clear lack of documented design and architectural patterns for the service mesh.

## 1.2 Motivation

Documenting good practices as patterns is a way to synthesize design knowledge. Thus, in this recent topic, documenting new patterns is of great importance as can later on guide developers that want to adopt a service mesh in their applications. Since the design and architectural decisions have a long-lasting influence throughout the life cycle of a development project, having more knowledge when migrating into a new architectural style, such as the service mesh, can make

the difference between a correct implementation that adds value to the product or a new layer of complexity with zero added value.

Service meshes are on the rise as shown in a survey conducted by the CNCF [9], Cloud Native Computing Foundation. In the survey, it was concluded that among the two hundred and fifty members of the CNCF and Kubernetes communities that participated in this survey, a vast majority were already using service meshes in production (60%), some were evaluating their use of it (19%), other were starting to implement it (10%). Only a few answered that they were not using a service mesh (9%) showcasing that this technology is essential for many developers.

## 1.3 Objectives

The purpose of this thesis is to **create new design patterns for the service mesh** and to **validate the created patterns**.

The first point is achieved by analyzing existent literature related to patterns around the service mesh and by observing the, what is commonly called, good practices around the subject. These new patterns should guide software developers and software architects wanting to gain a better understanding of the trade-offs and approaches to creating a service mesh. In other words, they can support developers to correctly understand their own context and where the service mesh can add value to that specific application context. They may, for example, allow developers to pick a specific service mesh architecture over others making it possible to save CPU and memory in the cluster based on that choice.

The second point requires trying to validate the value of the patterns by executing an empirical study (an experiment) on two of the design implementations of a service mesh. The experiment will be composed of 3 distinct cases, an initial microservices project without a service mesh as a baseline and two other cases taking the baseline project and each one having a different service meshes implementation. Then, these three different cases will be measured in their native state, without receiving traffic, and in a stress situation by injecting traffic into the cluster that holds the application. The results of the experiment will help to validate some consequences suggested by the patterns, helping to prove the veracity and value of the patterns.

## 1.4 Contributions

The main contributions of this work are:

- **State-of-the-Art Review** - in which we provide a comprehensive study of documentation in published works related to service meshes and tools that implement service meshes, as well as an analysis in empirical studies with patterns.
- **Documentation of new Patterns** - based on the documentation analyzed we propose six design patterns around the service mesh.

- **Controlled Experiment** - an experiment of an implementation of two of the possible design patterns for the service mesh (SIDECAR and NODE AGENT) using an open-source microservices application and hosting it on the Google Cloud Platform.

## 1.5 Document Structure

Besides the introduction, this dissertation contains six more chapters and it is divided into the following order.

- **Background** (*cf* Chapter 2) - in the background chapter, we introduce multiples terms that are going to be mentioned throughout this thesis and give a bit of context behind them.
- **State of the Art** (*cf* Chapter 3) - in the state of the art chapter, we summarize the documents analyzed and the information extracted from it.
- **Problem Statement** (*cf* Chapter 4) - in the problem statement chapter, we present the problem that this thesis will try to fix and the research questions associated with this work.
- **Patterns** (*cf* Chapter 5) - in the patterns chapter, we present the patterns that we documented as well as the relationship between each other and the target audience for the patterns.
- **Empirical Study** (*cf* Chapter 6) - in the empirical study chapter, we demonstrate and explain the controlled experiment that we conducted to help to validate our patterns.
- **Conclusion** (*cf* Chapter 7) - in the conclusion chapter, we summarize the work that has been done in this thesis and discuss the possible future work.

# Chapter 2

# Background

In this chapter, we describe and contextualize the topics required to fully comprehend this document. We begin by giving an idea of what a design pattern is and what are its main components. This will be a piece of important information to be able to easily analyze the patterns that are documented later on, in the Patterns Chapter (Chapter 4). Then we will make a brief analysis of some terms related to microservices and service meshes to give a summarized overview of the topics involved in this area.

## 2.1 Design Patterns

The concept of design pattern surged in a book written by Christopher Alexander named "A Pattern Language" [5], wherein for daily problems of the modern society, like the way towns should be decomposed or where the hospital should be situated, the author presents a network of solutions throughout the book with always a systematic way of describing the context, the problem and the solution. Eventually, this way of decomposing a problem and creating a solution was later adapted to the world of software architecture and software engineering. One great example of design patterns in software engineering that the community greatly respects and admires is the book "Gang of Four" by Gamma et al. [17].

A pattern is, hence, defined as a proven solution to a recurring design problem. The pattern should be defined in an agnostic way of the technologies and programming languages so it can be re-used in multiple cases as long as the context of the specific problem is the same. Design patterns consist of a set of re-usable structures that can solve the contextual problem in a simplified way and brings developers closer to each other since it creates specific vocabulary around the software design community.

Software design patterns tend to follow a specific structure that, although not mandatory, helps other developers to read and understand more easily due to familiarity with the structure. Authors usually will define their patterns with the following properties:

- **Context:**

Context is where the situation is explained to the readers. Answers questions like "Where is the problem occurring?", "How did we get there?", "What are the characteristics and variables that are important?". The same problem may have different solutions for different contexts.

– **Problem:**

The problem is where the problem that the pattern is trying to solve is explained. Answer questions like "If we do not fix this situation what happens?", "What are the forces associated with the problem?" "Why is this problem hard to solve?".

– **Solution:**

The solution is where it shows the actions needed to solve the specific problem in the given context. Answers questions like "How do we fix the problem?"

– **Consequences:**

The consequences are not as standard as the previous properties and it explains to the reader what are the benefits and drawbacks associated with implementing the solution discussed previously. Answers questions like "What does the application gain if it implements the solution?" and "what does the application loses if it implements the solution?"

In this paper, the patterns that we are going to document later on are going to follow this approach.

## 2.2 Containers

Containers are packaged applications that work as standard units of software and can run in any computing environment, due to the fact that all its code and dependencies are compacted inside itself. The recent container term, or container image that is the file that defines the settings of the container, contains everything needed to run the application such as application code, runtime, system tools, system libraries and settings as refers Docker, one of the most used container tools [12]. However, this was not always the case, since the technology dates back to 1979 with the Unix version 7, bringing the primordial concept of isolating a process by restricting an application's access to a specific directory. Time passed and the systems evolved and so did the concept of a container, as in 2001 we were introduced to the Linux VServer [47]. In this system, it was already defined the concept of partitioning inside of a computer and partitions of resources. Nevertheless, only in 2008 did the concept of containers stabilized with the LXC (LinuxX Containers) [27] wherein the isolation of the usage of CPU and memory was already possible and it worked on the kernel level. Finally, in 2013 Docker emerged with its build based on the LXC technology and with its accessible GUI made container technology, popular among developers.

## 2.3 Microservices

A new era of building applications has arrived, the Microservices era.

Dr. Peter Rogers used the term "micro-web-services" in 2005 during a presentation about cloud computing [37]. The term evolved through time and in 2011 the term "microservices" debuted at a conference of software architects [14].

Microservices architecture is an architectural pattern of designing and building software applications that consist of structuring the application as a set of individual services that are loosely coupled, independently deployable, scalable and highly maintainable. The Microservices architecture enables developers to rapidly and reliably deploy their changes in a specific service without the need to affect the entire application since these services, or modules, are simply independent computing units.

The practical concept of microservices, the individual decoupled modules as single units, is not entirely new as Gregory Young mentions in his talk at the μCon2016 when he compares microservices foundations with the original object orientation model from the 1970s written by Alan Kay. The architectural pattern has gained a lot of popularity in recent years since it solves a lot of the problems we face today such as high availability, multiple platforms, high scalability, speed and many others. *MarketsandMarkets* published a study [29] where they have predicted that the size of the global microservices architecture market will increase with a compound annual growth rate of 21.37% from 2019 to 2026 showing that this architectural pattern is indeed increasing in use in the global market.

Not surprisingly, the microservices architecture is not a silver bullet for all the software engineering problems. The pattern has some known drawbacks associated with it, such as the complexities related to communication, security, monitoring and debugging.

Communication is a problem in the sense that we need to establish communication protocols such as network protocols, APIs and interfaces for the different services to communicate between themselves. Security was always a constant problem in the software world and when the communication between services starts to wander through the web it gets worse as it gets more vulnerable to attacks. Monitoring is very important to guarantee the best performance of the different services and in a group, with observability, it enables an easier and faster debugging of any problem that can arise.

## 2.4 Cross-cutting Concerns

These known complexities associated with microservices are frequently called cross-cutting concerns since these concerns affect the entire application in a transversely way through all the services. The concerns range through different software areas, yet the cross-cutting concerns that connect directly with the core concept of the service mesh are the Communication cross-cutting concerns. These concerns are directly connected to the way that the services communicate with

each other. However, the communication topic can be divided into other subtopics such as security, resilience, observability, service discovery and routing since all these subtopics are directly connected with the way services communicate.

## 2.5 Service Mesh

A service mesh in its simplest definition is a **configurable infrastructure layer that encapsulates cross-cutting concerns into independent modules that can be used by the different services.**

Let's take some time to decompose and digest the description mentioned above. Microservices applications have cross-cutting concerns (communication, security, service discovery, observability etc) as mentioned in the previous sections. These cross-cutting concerns need to be implemented in all the different services by the developers and in many cases, different services use different programming languages which increases, even more, the group of tools and frameworks the developers need to implement and maintain. The developers instead of being focused on implementing business logic and increasing the value of their application are losing time with these transversely complexities associated with all the services in their application.

Microservices architecture came with the premise that developers would gain the agility to have more governance power over their services and would increase their productivity, but instead, are now losing time to matters that should not be their concern in the first place. The idea of the service mesh to fix this problem is to create a new **infrastructure layer** that can abstract all the cross-cutting concerns out of the individual services. This means that the services can remove all the implementation logic of the cross-cutting concerns out of their code and can use the new layer. This new layer enables developers to control and monitor the service-to-service communication, as it also enables a single stable point of change in case of the evolution of the configurations associated with the cross-cutting concerns. The concrete implementations and ideas behind different types of service meshes will be discussed further ahead in this article since they are part of the crucial subject of the thesis. Service meshes can be found in the bigger tech companies with a lot of microservices to take full advantage of the concept and enable all its potential. Some examples of those companies are Airbnb [2], Spotify, Trivago, Yahoo and many others [20].

## 2.6 Control Plane

The control plane is one of the core elements of a service mesh and acts as the brain for controlling the different services running inside the service mesh. This module can be abstracted as a set of different network communication rules to be applied on the data plane. When interacting with a service mesh the developer passes rules to the control plane, in the form of YAML configuration files, and the job of the control plane is to apply those rules to the data plane and ensure that they are being respected. Different implementations of a service mesh have different components inside, the commonly called, control plane, so it is ineffective to try to explain the concrete components as the answer would be "it depends".

## 2.7 Data Plane

The data plane is another one of the core elements of a service mesh. It is responsible for processing the data requests, flowing in the service mesh network, and using the configurations defined by the control plane, in the communications between the different services. Depending on the implementation of the service mesh, the data plane can be the group of sidecar proxies used by the SIDECAR service mesh or a single agent used by the NODE AGENT service mesh (these implementations will be discussed and documented further ahead in the Pattern Chapter (*cf* Chapter 5).

# Chapter 3

# State of the Art

In this chapter, we will analyze papers related to service mesh patterns, practices, adoptions and tools.

## 3.1 Introduction

In this chapter, we present a thorough overview of previously published work related to the design of service meshes. It is worth mentioning that the topic does not have a lot of literature documented since it is a very recent topic with a lot of investigation work to be done yet. Nevertheless, we decided to divide this work into three different sections. Section 3.4 describes literature related to already documented design and architectural patterns around service mesh and some practices associated with it. Section 3.5 analyzes case studies, related to adopting a service mesh in different contexts, where it shows the different aspects that make companies want to adopt a service mesh. For the last section, Section 3.6, due to the lack of variety of articles and information, we decided to analyze service mesh tools as a source of information, and compare them to see what are the major differences between them.

## 3.2 State of the Art review Goals

Given the previous problem statement, we present the following three main questions hovering in our minds when we started the review.

- **What are the documented factors that influence the adoption of a service mesh?**

  Consider: Which are the factors that lead developers to choose a specific implementation of a service mesh? Does a specific condition in the context lead developers to choose one implementation over the other? Which are the conditions that lead to a change?

- **What are the patterns already documented related to service mesh?**

Consider: Does already exist any type of documented pattern related to service mesh? If so, the person or entity who documented it is trustworthy and was it proved the usability of the pattern?

– **What practices for adopting a service mesh have not been documented as software patterns?**

Consider: What are the patterns not yet documented ? Where can my thesis add value to the community?

## 3.3 Methodology

To answer all these questions we conducted a direct research on the literature related with the topic of service mesh and more precisely a research on service mesh design patterns and practices associated with implementing it. First it was made a quick analysis to all the articles that we could find related with service mesh. Since the topic is very recent the service mesh documentation is very scarce and rare. Nevertheless the research query that we used was the following:

```
({Service Mesh} AND {Design Patterns}) OR ({Service Mesh} AND Pra-
ctices)
```

Eventually, this query needed to be optimized since the documentation that was retrieved was not enough. So the final query included an extra term to retrieve more articles that although more generic on the service mesh subject could still add value to the paper if properly analyzed and filtered.

```
({Service Mesh} AND {Design Patterns}) OR ({Service Mesh} AND Pra -
ctices) OR {Service Mesh}
```

This research query was used in three of the most famous scholarly search engines (Google scholar, Scopus and Inspec).

The criteria for inclusion of articles in this review was based on the articles having service mesh practices, service mesh patterns or service mesh adoptions as it's core theme. The criteria for exclusion of articles was based on the article not having a clear research topic, not having clear research methodology or not framing correctly into the thesis topic.

## 3.4 Best Practices and Patterns

Many practices for developing cloud-native applications have been so far described, and a relevant number of patterns has been written to document them [35, 36, 7, 42, 43, 39, 41, 38, 4, 11, 1].

Our analysis of current literature showed few resources addressing specifically service mesh practices, which we attribute mostly to the close relationship with topics that are still undergoing a lot of change, as is the case for containers and microservices. Since the market is very competitive

concerning microservices, the tools that implement service mesh have come to evolve their own architectures over time. One good argument in favour of this theory is the fact that in a recent paper by El Malki et al [13] written in 2019 where it mentions that *Consul*, one of the pioneer service mesh tools, uses a *none sidecar* approach regarding the service mesh implementation. However, in the writing time of this document, the tool uses a sidecar approach in its service mesh implementation as shown clearly in one of Consul's website tutorials demonstrated in Figure 3.1 present on Consul's website[1]. If we analyze Consuls' GitHub page [18], since it is an open-source project, we can clearly see that the project is constantly being updated and changed.

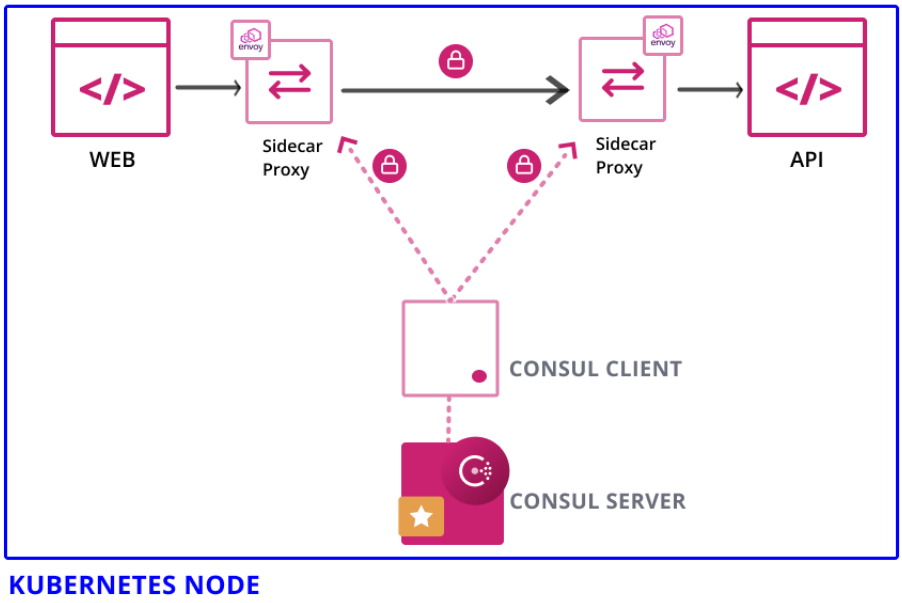


Figure 3.1: One of the images presented in a Consul tutorial in their website showing the use of the sidecar

Nevertheless, El Malki et al mentions interesting practices related to a service mesh by analyzing different tools and their respective documentation, articles and grey literature. All of the practices mentioned in the article hit the foregrounds of the service mesh and mention the advantages and disadvantages of each approach. Next, we will quickly summarise the five best practices mentioned in the article.

– **Cross-Service Communication**

  In this section, the authors mention how we can choose from a sidecar approach where all the communication passes through the sidecar before it gets to the service (service proxy) or we could use an API-based communication where all the requests would hit the API and then would be redirected to the right services. The sidecar approach would allow an easier implementation of resiliency and security techniques, but on the other hand, the API approach

[1] More information about the image on https://learn.hashicorp.com/tutorials/consul/service-mesh

would benefit from the simplicity of its implementation and would reduce communication overhead.

- **Ingress Communication**

  Although the communication between services is important, the communication from the exterior to the interior of the cluster is of the most importance as well. In this aspect, the authors explained that the addition of a front proxy, which is also known as an ingress proxy, enables the protection of the cluster from malicious or overloaded traffic. This new proxy can then be transformed into an API gateway and can then obtain new functionalities, if desired, as the authors exemplify by saying "It can provide proxy tasks such as load balancing and multi-protocol support at the perimeter of the service mesh".

- **Traffic control**

  Regarding traffic control, the three main approaches suggested are to use the previously mentioned front proxy to intercept incoming traffic and redirect it with load balancing features or to use a specific control plane implementation. This control plane implementation can be done in two ways, a centralized control plane where the control plane is a central component that controls the traffic of the entire service mesh and configures and manages the sidecars. The other option is the distributed control plane where each service of the service mesh has a cache that is updated independently from the other services and enables the configuration of the sidecars.

- **Service Mesh Expansion**

  In this topic, the authors describe the option to scale the service mesh and the solution presented is the multi-cluster support. This solution enables a single control plane to configure and control multiple service mesh (multiple clusters). This could simplify the process of controlling multiple service meshes as we only have to adjust a single control plane, but on the other hand, it creates a single point of failure and the possibility of a bottleneck.

- **Central Services and Proxy Tasks**

  As for the last topic, the article debates the division of tasks between the service proxies (the sidecar proxy) and the central service (the control plane). Since a lot of functionalities related to the concept of a service mesh can be implemented in either the central plane or the service proxies, the authors balance through the different subjects by debating the pros and cons of choosing where to distribute the tasks.

These types of best practices are a good source of knowledge to create new patterns since we know that these practices are used to solve a recurring problems.

Regarding documented patterns, we could find an article that describes very well the service mesh as a pattern written by Dobaj et al. [11]. In the article, the authors mention the service

mesh with a sidecar implementation where they explain the context, problem, forces and solution. Other patterns related to service mesh were found in grey literature as the sidecar pattern defined by Microsoft on their website [32] and the service mesh pattern defined by Chris Richardson, a great reference in the microservices industry, in his website [34] having almost no information detailed.

## 3.5 Case studies

Implementing a service mesh is being sold by the service mesh tools industry as an easy task but sometimes is not as easy as it may seem. An article named "Research on Enterprise Service Governance Based on Service Mesh" [45] written in the year 2020, introduces to the readers discussions related to issues of the service mesh. Some of those discussions almost force us to remember that each case or company is a single and unique case and that is almost impossible to arrange a solution to all the possible combinations of scenarios when the article says:

"Cannot support all requested communication protocols. Enterprise is utilizing many protocols in existing applications, such as HTTP/rest, Dubbo, RPC, web service, socket, etc. Unfortunately, none of the above platforms can support all these protocols well."

Clearly stating that currently, we do not have a single tool that supports all the communication protocols, since some tools are better in some protocols and other tools are better in other protocols.

Resource utilization and performance related to necessary memory to allocate the service mesh is another big drawback especially when the project to configure and control is composed of many microservices stated by the authors.

While the authors stress some drawbacks, they clearly understood the advantages of the service mesh and implemented their own version of the service mesh based on Istio with some nuances to what they called the ESMP, enterprise service mesh platform, and explained that "we propose one ESMP that can be implemented in enterprise production environment by largely revising and customizing the components based on Istio". Based on their experience with the implementation the authors created four design principles to follow when migrating an application to a service mesh architecture. Those principles are :

- Minimum cost
- Compatibility with the application technology stack
- High adaptability
- High availability

By observing this list we can see that the authors want to state a clear point. The service mesh should serve the application and not the opposite. Strict principles that recommend a correct choice of service mesh implementation around the needs and characteristics of the application.

One of the great features that the service mesh brings to the table and makes companies want to adopt this new architectural pattern is the powerful observability and debugging features. When adopting the microservices architecture the developers on one hand gain scalability, maintainability and availability. On the other hand, they lose observability of operations since the requests are now dispersed through multiple microservices that are divided into individual modules over the cluster of the application. Complicating more the equation, the traditional monitoring tools available in the market today cannot solve the situation as mentioned by Nunes et al. [33]. The authors made a case study for their company SigSaude, a medical application for managing clinics, since they faced the observability problem when transitioning to the microservices architecture. In the article, the authors mention a list of requirements that the application needed to improve its observability capabilities such as execution time metrics, traceability in the flow of requests, success rate metrics, alerts related to average request time and others. All these requirements were easy to achieve with the default microservices architecture and the article goes on to explain how the use of Istio enabled the application to gain access to all the necessary metrics except for the alerts. Istio has one component named Mixer that receives the traffic intercepted by all the sidecars associated with each service and then works together with tools such as Grafana and Kiali that have already easy guides to connect with Istio service mesh, without having to code or change the internal services and enable the visualization of all those metrics collected. In Figure 3.2 we can see the overview of how Mixer from Istio works shown in Istio's website [22].

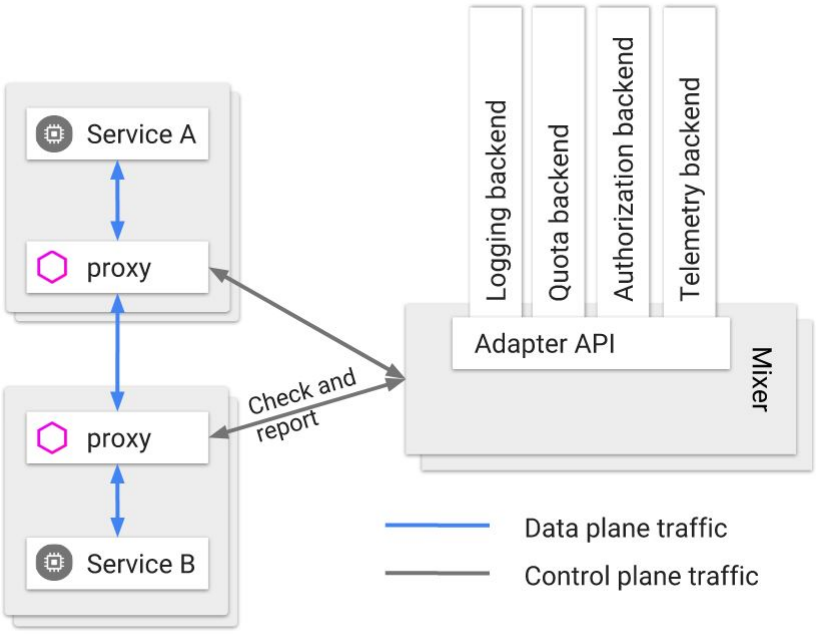


Figure 3.2: Overview of the architecture of the Mixer component from Istio

These new observability features help developers to debug and solve problems much faster as they have access to metrics, logs and distributed tracings, which enables the quick identification of the service that is causing issues.

A great way to test the service mesh capabilities is to take the already implemented security features in the service mesh and enhance them to a superior level. NIST, National Institute of Standards and Technology, is a government agency of the United States of America that, in August of 2021, published an article [8] where they create a guide for implementing attribute-based access control in a service mesh. Attribute-based access control, ABAC, as explained in the article "ABAC is defined as an access control method where subject requests to perform operations on objects are granted or denied based on assigned attributes of the subject". A service mesh by default has a lot of security features possible to be implemented in the application like authentication and authorization, mutual TLS, role-based access control and ingress and egress control. All the mentioned features work together to create what is commonly called the zero-trust security model [30]. Nevertheless, we can take the security even further by changing the role-based access control to attribute-based access control, which is a stronger version as it has a greater number of possible control variables. The authors in the article explain that the proxies in the service mesh, either the ingress, egress or the sidecar, can play the role of Policy Enforcement Points (PEPs), since they intercept all the requests that flow through the cluster, the PEPs responsibilities are to "enforce access control policies by performing the allow/deny actions based on the verdict provided by PDP [Policy Decision Point]". This enables the proxies to authorize or deny requests based on authentication entities that validate the authenticity of the client trying to use the application increasing the overall security of the services.

Another great research work was made by Mohammadali Akbarisamani in his paper [3], wherein it searches for the potential advantages of adding a service mesh to an application in the 5G core network. The author starts by analyzing the three most popular tools(Istio, Linkerd and Consul) and eventually chooses Istio due to its big range of functionalities. The implementation of the features of Istio impresses the author as he states "Istio by itself is a capable service mesh framework and it could be a usable solution even in terms of 5GC as it is being consequently developed and introduces new functionality needed for the 5G rollout and newer evolutions". Adds even further that Istio has a scalable architecture that can support different needs and even mentions that the load balancing options for the resilience available through the proxies are of great importance. However, Mohammadali also notes that Istio has some downsides according to his view, the high dependency on the infrastructure (Kubernetes) and the complexity of the configuration. At the end of the article, the author concludes that the applicability of Istio still requires more investigation but he is happy with the results of the latency added ( 0.5ms per sidecar) and believes that Istio is a high potential framework to be used in the 5G context.

Considering the fact that service mesh is a very recent topic, there is a lot of room for innovation and to improve even further what is already implemented by the service mesh tools. Li et al. went ahead and created an article [26] wherein they explain the current problem of applying configurations in a service mesh. Currently, if a user wants to customize the behaviour of the sidecars, he must create management policies and apply them in the control plane. Those policies will later be converted into configurations and applied in the sidecars through the data plane management API. Ideally, this would work without any problem. However, this is not always the case and can

be due to three major factors as suggested by the author. The administrator's intentions may not be configured correctly. The configured policies may not be the actual behaviour of the sidecars due to conversion errors, misconfigurations or the policies will not be installed correctly. The data plane policies may not be enforced properly because of errors in the data plane or runtime failures. Taking into consideration these issues, the authors created a hypothesis of a possible tool, what they called the MeshScope, for inspecting the configurations in the sidecars. MeshScope can make policy verification in two ways. Static policy verification is done by comparing the policies defined in the control plane and the policies applied in the data plane on the sidecars. The other way to verify is the dynamic policy verification, this one uses fake requests to the services with specific headers, tokens or authorizations to seek that the policies applied on the sidecars are being used correctly.

## 3.6 Tools

It is possible that for a recent topic there is not a lot of information documented. However, for that reason, a great source of knowledge is to analyze the tools that implement the technology in question. Regarding service mesh, we currently have some tools for implementing it. For this analysis, we have divided the tools into three categories for the sake of better explaining the choice of the tools to analyze.

- **Public Service Mesh Tools:**

  The Public tools are open-source tools, this means that all the code used to build the tool is widely available to everyone to analyze, use, change and even distribute.

- **Private Service Mesh Tools:**

  The Private tools are the tools that we currently do not have access to its source code and thus are the ones harder to analyze.

- **Service Mesh Tools Based of Others:**

  In this last section, tools that are based on other tools, for lack of a better name, are tools that use the open-source tools, mentioned above, to recycle their code and add new functionalities to eventually sell them.

To get a better picture of the tools to analyze, we have consulted the Cloud Native Computing Foundation (CNCF) [16]. The CNCF can be used as a trustful source of knowledge since it is a foundation well known (CNCF is part of the nonprofit Linux Foundation) and respected with an active community of developers in the field of cloud software engineering. The main objective of the CNCF is to make cloud native computing ubiquitous by joining different tools from different fields of cloud computing together and creating common interfaces so that the developers in the end can take advantage of this common way of "doing things". After taking into consideration the

service meshes mentioned in the CNCF it was decided that we were going to analyze five open-source tools. The choice of going for five open-source tools was based on the fact that the private tools are harder to analyze due to their lack of architectural explanations and scarce documentation of implementations. The tools based on other tools usually are very similar to their predecessors, so the changes are not relevant. Before the analysis, we want to mention some service mesh tools that are not going to enter this analysis but still deserve a word of recognition such as AWS App Mesh the service mesh from the Amazon, OpenShift Service Mesh the service mesh from RedHat and AspenMesh. This analysis did not include non-functional features from the service meshes since we concluded that we would not be able to mine patterns based on the non-functional features.

The open-source tools we decided to analyze are:

- **Istio** , it is the most adopted service mesh in the market, based on a survey made by the CNCF in 2020 [15], and has become the face of Service mesh. It was initiated by Google, IBM and Lyft which has helped the fame of the tool [21].

- **Linkerd** , was the first service mesh. It was initiated by Buoyant and it is considered an ultra light service mesh, due to the fact that it only works under Kubernetes infrastructure in its most recent, v2, version [25].

- **Consul** , HashiCorp's Consul has been well known for its service discovery solution, but it has been transformed into a complete service mesh with the addition of Envoy proxy, enabling the usage of the sidecar pattern [10].

- **Traefik Mesh** , formerly known as Maesh, it is the service mesh based on the cloud-native API Gateway Traefik Mesh [31].

- **Kuma** , similar to Traefik Mesh, Kuma is a service mesh based on a API Gateway, this time on the Kong API Gateway [24].

To compare the five service meshes, we are going to base ourselves on the core concepts and premise of the service mesh's ideology (Communication, Security, Monitoring, Resilience).

- Communication

  Regarding communication protocols, all of the service meshes support TCP, both HTTP protocols and gRPC. The usage of the sidecar pattern is a certainty in the majority of service meshes, however, the Traefik Mesh is one of the exceptions having a single proxy agent per node and not per service. Related to routing features, all of the service meshes support load balancing wherein some meshes have multiple functions of load balancing and Consul is the one with the widest choice of options. Still, on the routing subtopic, Traefik Mesh and Linkerd get at a disadvantage compared with the others due to not letting do a header

and path-based traffic splits and only percentage-based traffic splits as the others allow both functionalities.

– Security

In the security area mTLS, mutual transport layer security, is a security protocol very important and all the service meshes allow it besides Traefik Mesh which only works with TLS, a simpler version of the mTLS. Service meshes that work with mTLS have the security protocol enabled by default with the exception of Kuma which needs to be configured first.

– Monitoring

As for monitoring, all the meshes except Traefik Mesh have a Dashboard to consult their service mesh, even for Istio and Kuma that can have a dashboard with the help of third-party tools such as Kiali for Istio and Grafana for Kuma. Grafana is a very powerful and popular monitoring tool, and all of the meshes already come with an integration pre-configured excepting Consul. Another very powerful monitoring tool is Prometheus and in this case, all of the services come with an integration already configured. For distributed tracing, only Consul does not come with an integrated pre-configuration for Jaeger, a great monitoring tool for back-end tracing.

– Resilience

For the resilience patterns, such as circuit breaking, retry and timeout all of the service meshes implement them. However, if we want to configure different values for retry and timeout in different endpoints, Kuma and Traefik Mesh do not allow it. In terms of injection tests, Istio and Kuma lead the group of service meshes since they allow both fault injection and delay injection in comparison with the others.

– Extras

An important point to take into consideration is the service Proxy that each service mesh uses. For the majority of the list, it is being used Envoy with the exception of Linkerd and Traefik Mesh, which both use their own version of the service proxy. Those being respectively Linkerd2-proxy and Traefik Mesh proxy. One way to save resources is to have a single service mesh to control multiple clusters of applications like most of them do except for Traefik Mesh. It is worth mentioning that all five service meshes have support for Kubernetes integration.

All the information mentioned above can be found summarized in Table 3.1.

Regarding the architectures of the tools, we currently can divide them into two different categories. Service Mesh tools that use the sidecar pattern, as shown in Figure 3.3, and the tools that follow a non-sidecar approach, as shown in Figure 3.4 as explains the team behind AspenMesh [6]. For ease of discussion and explanation, we named the approaches, Service Mesh by Sidecar

| | Istio | Linkerd | Consul | Traefik Mesh | Kuma |
|---|---|---|---|---|---|
| TCP | ● | ● | ● | ● | ● |
| HTTP1 / HTTP2 | ● | ● | ● | ● | ● |
| gRPC | ● | ● | ● | ● | ● |
| Sidecar based | ● | ● | ● | | ● |
| Load balancing | ● | ● | ● | ● | ● |
| Header/path traffic split | ● | | ● | | ● |
| Percentage traffic split | ● | ● | ● | ● | ● |
| mTLS | ● | ● | ● | | ● |
| Internal dashboard | ● | ● | ● | | ● |
| Grafana integration | ● | ● | | ● | ● |
| Prometheus integration | ● | ● | ● | ● | ● |
| Jaeger integration | ● | ● | | ● | ● |
| Circuit breaking | ● | ● | ● | ● | ● |
| Timeout / Retry | ● | ● | ● | ● | ● |
| Diff endpoints resilience config | ● | ● | ● | | |
| Fault injection | ● | | | | ● |
| Delay injection | ● | | | | ● |
| Multi cluster | ● | ● | ● | | ● |

Table 3.1: Summary of the features of the service mesh tools

and Service Mesh by Node Agent, respectively. The service mesh by sidecar, injects next to each microservice a service, the so-called sidecar, that acts as a personal proxy for the service, overseeing all the communications in and out of the service. On the other hand, the service mesh by node agent creates a single agent per node (machine) that acts as a proxy, overseeing all the traffic of the node.

The tools analyzed before that use the Service Mesh by Sidecar architecture are Istio, Linkerd, Consul and Kuma. Out of the five tools, currently, only Traefik Mesh uses the Service Mesh by Node Agent approach.

Based on the analysis presented above, it is not possible to conclude what is the best service mesh, as the service meshes are not comparable to that level. On one hand, we have the Traefik Mesh or Consul, with clearly fewer features compared with the other service meshes, like Istio, but on the other hand, it makes it easier to implement and configure them. When deciding the service mesh to implement, it is important to understand the features present in the service mesh that the application requires, and not only choose the service mesh with more functionalities, as it tends to get more complex to configure.

## 3.7 Discussion

We have presented an exploration of the research literature related to service mesh patterns and service mesh in its general concept. Due to the lack of information documented in the research

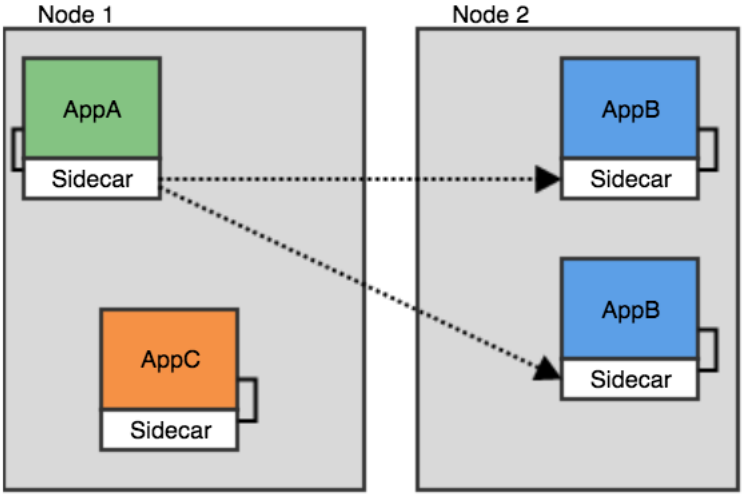


Figure 3.3: Service Mesh by Sidecar Architecture presented by AspenMesh

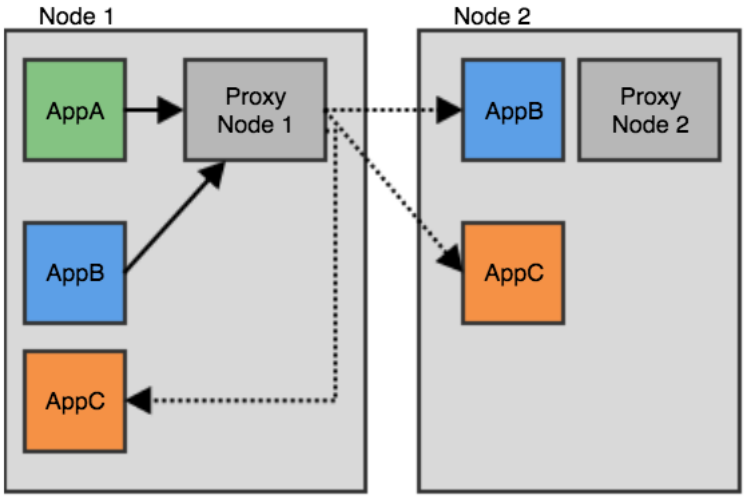


Figure 3.4: Service Mesh by Node Agent Architecture presented by AspenMesh

literature, we included it in our research service mesh tools. This allowed us to add a new perspective on the different implementations and features that each service mesh has to offer to the developers when they want to adopt a service mesh in their application.

Some companies have been taking a step in favour of the service mesh architectural style and taking advantage of its functionalities as we can observe from the case studies. Those companies explain what were the forces that drove them to take the action to change. After analysing the state of the art, we found one article for each one of the three main functionalities that the service mesh provides. Those being security, resiliency and observability. Answering the question "What are the documented factors that influence the adoption of a service mesh?" it is hard to have a clear conclusion based on the articles analyzed since each one of them supports a different functionality.

Service mesh is a very recent topic, so it still lacks documentation and investigation on many

levels. Researchers tend to approach more case studies for particular companies with a specific context, and the counterpart articles with empirical studies based on patterns seem to not exist (based on our research). Nevertheless, there are already documented patterns as we saw. The service mesh pattern and the sidecar pattern as mentioned previously in this chapter answer "What are the patterns already documented related to service mesh?". To the best of our knowledge, there is no empirical study which focuses on the design and architectural patterns of the service mesh and their consequences when applied, both positive and negative. As a result, it leaves a lot of room for future pattern documentation work to add value to the software community.

Answering the last research question, "What best practices for adopting a service mesh have not been documented as software patterns?" we have created some patterns (*cf* Chapter 5) on some of the practices mentioned in the literature that we have reviewed. Not all of the practices were transformed into patterns since it is not possible to always translate a good practice into a design or architectural pattern.

# Chapter 4

# Problem Statement

In this chapter we start by describing, with more detail, the problem that exists regarding the current lack of documented patterns for service meshes designs and their alternatives (*cf* Section 4.1). Then, we present the thesis research questions ( (*cf* Section 4.2). Finally, we will analyze the methodology associated to answer the research questions (*cf* Section 4.3).

## 4.1 Scope

Design patterns are ordinarily characterized as reusable answers for recurrent design problems. They help developers facing all kinds of problem domains to improve their work with cataloged good solutions for commonly recurring problems. It also creates a common vocabulary that facilitates interactions between developers, making it easier to discuss the solution design for a specific problem.

It is hard to argue against the value of existing design and architectural patterns documented in the software design literature. Nevertheless, the documentation of such good solutions using patterns has yet to cover the challenges and solutions faced by all software developers anywhere. In particular, new architectural tactics and practices keep emerging, supported by the increasing maturity of the technologies for managing and orchestrating microservices systems. One of such emerging solutions is the *service mesh*.

Throughout the previous chapter, we have explored the state of the art related to the documentation of patterns for the service mesh. Two patterns were found, SERVICE MESH and SIDECAR. However, none of the found patterns had all the elements that we sought and so in Section 3.7, the discussion reached the conclusion of plain deficiency of documented solutions for the different possible contexts where a service mesh could be applied and its different design solutions.

## 4.2 Research Questions

Based on the lack of documented solutions for the different service mesh designs we created the following research questions in order to tackle this issue:

- **RQ1** "To what extent can we document new design patterns for the service mesh or improve even further the ones that already exist?" We aim to create design patterns for the service mesh and contribute with actionable knowledge in the field.
- **RQ2** "To what extent did our patterns accurately captured the consequences of their solutions?" Although in our patterns we presented multiple consequences that could affect the application, for this thesis we are going to focus on the CPU and memory usage when the patterns are applied. To answer this question we will perform an experiment to measure the resource utilization of CPU and memory in two different design approaches for the service mesh.

## 4.3 Methodology

With this work, we aim to improve the overall understanding of service meshes, in terms of their functionalities and design, by creating new design patterns.

For that reason, we started with a **state of the art review** (*cf* Chapter 3) in order to understand the current scenario of the patterns for the service mesh.

During this review, we compare the existing patterns already documented and draw conclusions regarding their most crucial problems, since those can help us improve our own patterns later on. We also analyse the different case studies and tools that implement a service mesh. Then, based on the previous analysis of the literature we describe six **patterns** for the service mesh (*cf.* Chapter 5).

Based on the new design patterns documented, we conduct an **empirical study** to validate some consequences of the new summarized knowledge. This step of validation is essential since a pattern in its essence is only a theory. This theory, although it was created based on observing good practices has a problem as Khols mentions in his paper [23] saying "the process of pattern mining is similar to scientific discovery" and so they need to be evaluated following a scientific methodology like any other theory as the author continues to explain "Patterns are not tried-and-true per se, just like theories they have to be subjected to empirical tests".

Therefore, we draw inspiration from other works that empirically evaluate patterns [40, 46] and design a case study to evaluate our own service mesh patterns.

In this study, we created three different clusters and implemented the same application in each cluster with a different service mesh (being one of those clusters the baseline without a service mesh applied). This experiment allows us to measure the different values of CPU, memory and network used in those three different contexts. These metrics, collected in our clusters, help answer the research questions for this thesis and support the value of the patterns created.

# Chapter 5

# Patterns

As we have previously covered, part of the goals of this thesis is to create new design patterns for the service mesh, making it easier to choose between different ways to implement one and giving a better overview of the main consequences of adopting this new technology. This is very important to guide new developers to have a better grasp of the service meshes since the design and architectural decisions have a long-lasting influence throughout the life cycle of a development project. In this chapter we will review the goal that is going to be tackled in this chapter (*cf* Section 5.1), we will visit the methodology used to write the patterns (*cf* Section 5.2) as well as an overview of the patterns (*cf* Section 5.3) and finally we will describe each pattern individually.

## 5.1 Goals

The goal of this chapter is to present the results obtained from the pattern mining process that occurred during the state of the art review (*cf.* Chapter 3) and to answer the first research question presented in the problem statement (*cf.* Chapter 4).

- **RQ1** "To what extent can we document new design patterns for the service mesh or improve even further the ones that already exist?"

## 5.2 Methodology

The patterns that are going to be presented in this document were mined by analyzing the literature related to Service Mesh and by analyzing open-source service mesh tools (such as Istio, Linkerd, Consul, Kuma and Traefik Mesh). Based on the analysis, we identified what are the most recurring practices associated with solving a specific problem and then documented those same practices in the form of patterns.

## 5.3 About the Patterns

These patterns are targeted at software developers and software architects wanting to gain a better understanding of the trade-offs and approaches to creating a service mesh. In other words, they can support developers to correctly understand their own context and where the service mesh can add value to that specific application context. They may, for example, allow developers to pick a specific service mesh architecture over another, making it possible to save CPU and memory in the cluster based on that choice. Summarized information about the logical relation between the different patterns can be seen in Figure 5.1.

The patterns that we are going to document are:

- SERVICE MESH - Addresses the need for a clear separation between the business logic and the logic needed for the communication between the different services.
- SHARED COMMUNICATION LIBRARY - One possible design alternative for the SERVICE MESH pattern where a library is created to be reused in the different services and it can work without a container infrastructure coordination.
- NODE AGENT - One of the possible designs for the *Service Mesh* Pattern where a single module is created in the node and acts as a proxy for all the services inside of that node.
- SIDECAR - One of the possible designs for the *Service Mesh* Pattern where a sidecar module is deployed next to each service, acting as an individual proxy for that specific service.
- SERVICE MESH TEAM - Adjusts the way the teams in a company are structured by creating a new team that promotes and actively develops different aspects of the service mesh keeping a smooth experience between the different teams using the service mesh.
- MULTI-CLUSTER SERVICE MESH - A way to save resources and improve the resilience of the service mesh by controlling different clusters with a single service mesh.

These patterns, as described in the next few sections, have also been published in the proceedings of the EuroPLoP 2022 conference [28].

## 5.4 Service Mesh

When architecting a system based on microservices with the objective of having a clear separation of concerns ...

... we sometimes realise the existence of cross-cutting concerns related to communication between the different services (*e.g.*, security, resilience, observability, and routing). Such concerns can be responsible for the tangling and scattering of logic across the different services.

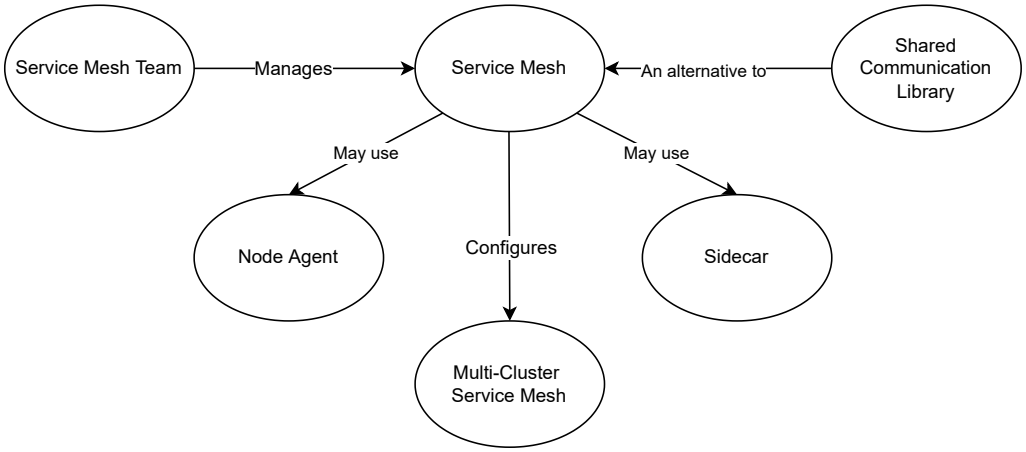


Figure 5.1: The patterns that are going to be documented in this article and their relation between each other.

### 5.4.1 Problem

Cross-cutting concerns related to communication often lead to the duplication of code and a tangling division of responsibilities between infrastructure logic and business logic.

The Figure 5.2 illustrates a scenario in which the same general communication concerns exist in different services of a system.

During the lifetime of a system, different concerns addressed by developers during the implementation may need to be rethought and adapted. Such changes may stem from policies changing within the company, updates to external libraries, the evolution of the programming language, etc. These changes may imply changes in different services when cross-cutting concerns are at stake.

Creating flexible modular software in a microservices architecture is not always trivial when concerns related to communication are at stake. On the one hand, creating reusable decoupled modules that implement these needed concerns is not straightforward since many of them cross-cut most of the services that compose the system. Ideally, the new module(s) will have all the methods and configurations needed to be used in all the services, and that by itself requires the creation of new abstractions, so they can be universally used by the application. In contrast, the system has changes between the services that, although minor, are decisive for the correct functioning of the services, so leaving some space for the services to implement their own configurations that require more flexibility is a must. This balance between encapsulation of cross-cutting concerns and flexibility of services is difficult to achieve. On the other hand, creating these modules allows a single point of change when the code of cross-cutting concerns needs to be updated, enables the elimination of duplicate or very similar code between services and the redundancy of implementing cross-cutting logic every time a new service is created.

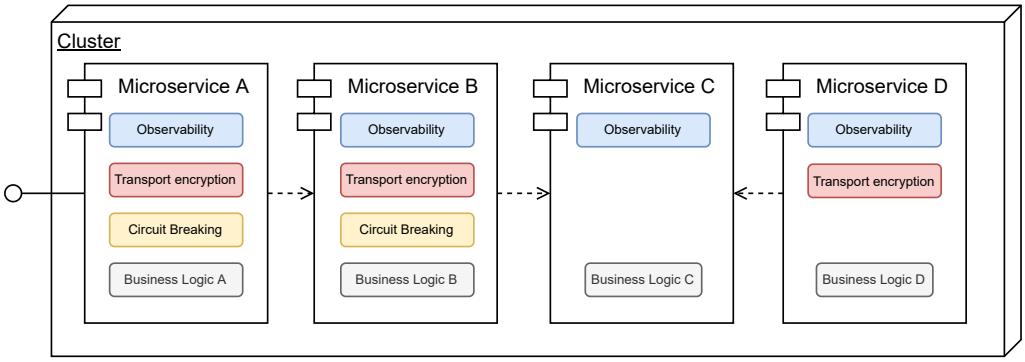


Figure 5.2: A microservice architecture without the use of a SERVICE MESH. Each service implements its own business logic, but also some communication-related concerns that cross-cut the system as a whole.

### 5.4.2 Solution

**Encapsulate cross-cutting concerns related to communication into independent modules that can be used by the different services.**

Analyze each service that implements the cross-cutting concerns and start to create abstractions of what needs to be implemented for the creation of the new modules to be shared with all services. These modules as mentioned in the problem statement need to be generic enough (for example the module(s) can't have any type of domain model reference) so they can be reused. Move the implementation logic governing service-to-service communication and cross-cutting concerns out of individual services to the new modules created. As shown in the Figure 5.3.

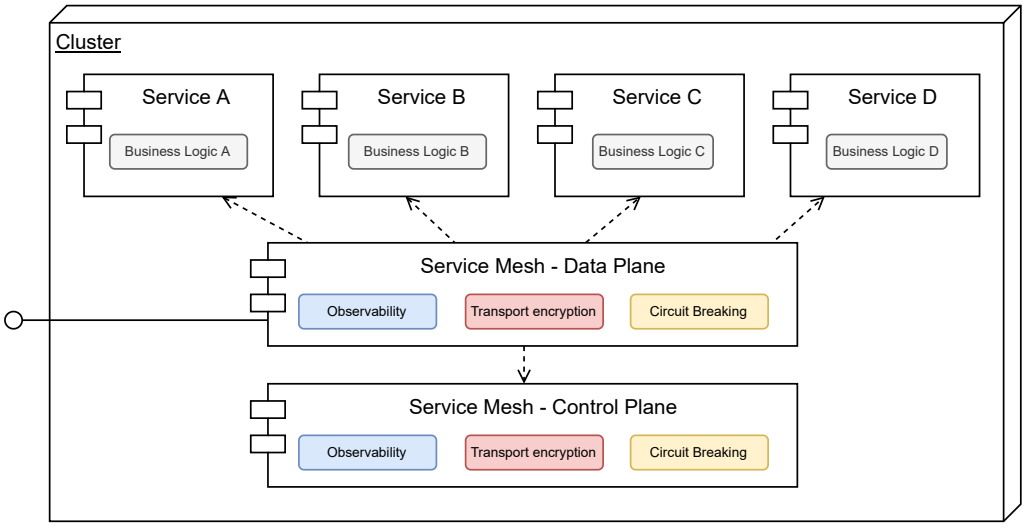


Figure 5.3: A microservices architecture with a SERVICE MESH implemented. The two main components, the data plane and the control plane, are responsible for managing the cross-cutting concerns for all the services.

Depending on the specific details of the context and problem, a service mesh can be implemented in different ways:

- **Service Mesh using a node agent** - Used in applications where the cost-effectiveness of resource use is one of the most important factors.
- **Service Mesh using a sidecar** - Used in applications where services are very different from each other and different cross-cutting configurations for almost every service would benefit the application.

### 5.4.3 Consequences

This pattern has the following positive consequences:

- Easiness to change cross-cutting concerns since now the cross-cutting concerns are compact in a single place.
- Cleaner code inside the services due to elimination of duplicated code in the different services.

This pattern has the following negative consequences:

- The effort to implement the new modules.
- Technology's immaturity related to the containers since that technology is very recent with constant evolution and upgrades.
- Single point of failure.
- New dependency added.

### 5.4.4 Related Patterns

This pattern has the following related patterns:

- MICROSERVICES - to apply the patterns in this paper we are always taking into account the fact that the application is using the microservice architecture.
- SHARED COMMUNICATION LIBRARY - a possible alternative to the SERVICE MESH pattern that accomplishes the same objective without the need for a containerization infrastructure underneath.
- NODE AGENT - a way to use the service mesh, the NODE AGENT creates a data plane based on a single agent per node.
- SIDECAR - a way to use the service mesh, the SIDECAR creates a data plane based on deploying proxies next to each service.

### 5.4.5 Known Uses

Known uses for the Service Mesh pattern can be found in all the service mesh tools that exist such as Istio, Linkerd, Consul, AWS App Mesh, etc. in their Service Mesh implementations.

## 5.5 Shared communication library

When architecting a system based on microservices with the objective of having a clear separation of concerns ...

... and the system consists of a single-language microservices application and the team(s) owns most of the services.

### 5.5.1 Problem

**Cross-cutting concerns often lead to the duplication of code and a bad division of responsibilities between infrastructure logic and business logic.**

The Figure 5.4 illustrates a scenario in which the same general communication concerns exist in different services of a system.

A Service Mesh allows addressing cross-cutting concerns in service-based systems. When the application is written in a single programming language, the effort to implement a new module that can serve all the services of the application reduces since it only needs to be created for that particular language. The system should be easy to deploy and orchestrate since it's a simple system so creating a single module that all the services can use seems like a good strategy. On the other hand, it creates a bottleneck dependency in the services to the new module since all the services are using it, so if for some reason it's badly implemented or needs changes it will affect the entire application. Another problem is the big effort to balance the features inside the new module since this module needs to be abstract enough so all services can use it, but it also needs to be flexible so the services can implement their own preferences if needed.

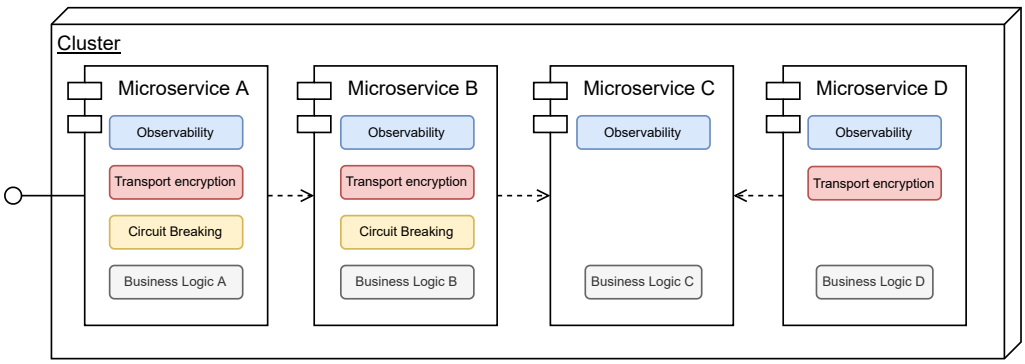


Figure 5.4: A microservice architecture system without the use of a SHARED COMMUNICATION LIBRARY. Each service implements its own business logic, but also some communication-related concerns that cross-cut the system as a whole.

### 5.5.2 Solution

**Encapsulate cross-cutting concerns related to communication into libraries that can be used by the different services.**

Create library(s) that have all the cross-cutting features needed in the application and use them internally in your services, by making explicit calls in the service code to the functions in those libraries, as shown in the Figure 5.5. According to the needs of the application, either have a single library that encapsulates all the cross-cutting concerns or have multiple libraries. This decision of dividing concerns into multiple libraries should be based on the specific code that the library is encapsulating. For example, if some service needs to abstract a cross-cutting concern that only affects that service, and perhaps tends to change a lot, then a new library should be created for that case. Each implementation case is a different one, but some rules of thumb can be followed to make sure that the library implementation is speeding up development, and not creating social tensions between teams. Libraries should not be created for business logic or domain-specific code, since it is only applicable to each service. Code and methods that change frequently are not a good idea to have in a library since it forces all the services to update the libraries even if they do not need it, and this usually leads to errors or side effects. "Prefer duplication over the wrong abstraction" [1] because creating the wrong abstractions tends to create more problems in the future when the code needs to be updated.

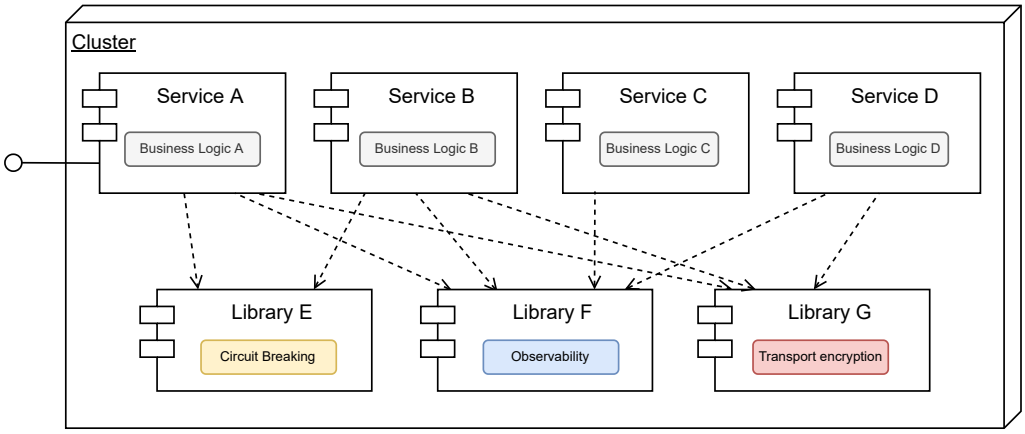


Figure 5.5: A microservice architecture system with a SHARED COMMUNICATION LIBRARY. Here each service can re-utilize the logic provided by the different libraries in their internal logic without having to implement it from scratch.

### 5.5.3 Consequences

This pattern has the following positive consequences:

- The underlying infrastructure where the service is being run, the container runner (like Kubernetes), does not have to know about the libraries, so it does not require any type of cooperation.

[1] Sandi Metz, "The Wrong Abstraction". 2016. https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

- Small trust boundary, since the service only has to call the library in its own process and not a remote service somewhere in the network, the library code only has the same privileges as the service calling it.
- Very simple to implement compared to other service meshes architectures.
- Removed the coupling in the services from their internal implementation of the cross-cutting concerns since that implementation was removed.

This pattern has the following negative consequences:

- Added a dependency in the services to the library.
- Efforts to implement the library will escalate as the number of languages increases in the application since the methods will need to be replicated for each language.
- Can only be implemented in the services that the team(s) own(s) since it needs a change in the internal code of the service.
- Requires more cooperation between teams to ensure proper functionality between all the services that want to use the library.
- Hard to maintain and update the library due to the cooperation needed between teams.

### 5.5.4 Related Patterns

This pattern has the following related patterns:

- MICROSERVICES - to apply the patterns in this paper we are always taking into account the fact that the application is using the microservice architecture.
- SERVICE MESH - a possible alternative to the SHARED COMMUNICATION LIBRARY pattern that accomplishes the same objective with the need for a containerization infrastructure underneath but is agnostic of the software language used in the different services.
- NODE AGENT - a way to use the service mesh, the NODE AGENT creates a data plane based on a single agent per node.
- SIDECAR - a way to use the service mesh, the SIDECAR creates a data plane based on deploying proxies next to each service.

### 5.5.5 Known Uses

Known uses for the SHARED COMMUNICATION LIBRARY pattern can be found for example in the way Netflix tackled the cross-cutting concerns problem, commonly called the Netflix OSS (open-source software). The Netflix OSS is a group of open-source software that also includes the

group of libraries created by Netflix to solve distributed-systems problems at scale. Examples of those libraries are the Ribbon library used for load balancing, the Karyon for service discovery, the Feign for HTTP client binding and many others [2]. Another example can be found in the library made by Twitter named Finagle [3] used to construct high-concurrency servers.

## 5.6 Node Agent

When architecting a system based on SERVICE MESH pattern...

...and the system consists of a multi-language microservices application where the cost-effectiveness of resource use is one of the most important factors.

### 5.6.1 Problem

**Cross-cutting concerns often lead to the duplication of code and a bad division of responsibilities between infrastructure logic and business logic.**

The Figure 5.6 illustrates a scenario in which the same general communication concerns exist in different services of a system.

A Service Mesh allows addressing cross-cutting concerns in service-based systems. When the application is written in multiple coding languages, the possibility of using a library to implement the service mesh tends to seem quite troublesome since it would need multiple libraries for the multiple languages. Therefore, the urge to create a new type of module arose. With the world of containers growing each day, the ideal scenario would be to create a new module that would be deployed as a container on the same node as the services. Regarding the structure and the way that the module is deployed is where things get tricky. On one hand, this module should be easy to implement and deploy, to make sure that the developers can use it without concerns, and it should also be lightweight since machine resource availability should not be a reason to not use a service mesh. On the other hand, having a new module for each service would cost more resources but would allow the full independence of the services regarding cross-cutting configurations and network configurations (network namespaces, routing tables).

### 5.6.2 Solution

**Encapsulate cross-cutting concerns related to communication into a Node Agent that acts as a proxy for all the services that are running on the same working node.**

With this solution, the main goal is to save resources in the worker node by only having a single node agent per worker node. Ultimately this means that all the containers deployed on that node are going to work in coordination with the node agent. This agent is going to work as a proxy and is responsible for supervising all the communications in and out of the services, to apply the cross-cutting concerns defined for the services. This approach already requires some

[2] https://netflixtechblog.com/evolution-of-open-source-at-netflix-d05c1c788429
[3] https://twitter.github.io/finagle/

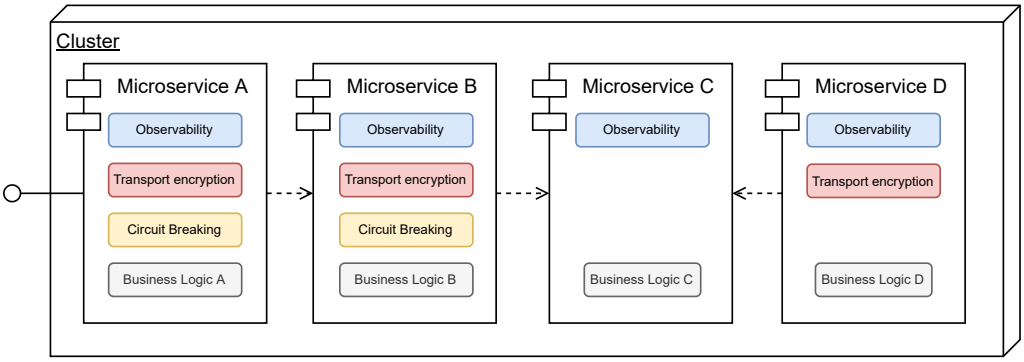


Figure 5.6: A microservice architecture system without the use of a NODE AGENT. Each service implements its own business logic, but also some communication-related concerns that cross-cut the system as a whole

cooperation from the infrastructure, since this model will not work without some coordination regarding resource sharing between services (memory to buffer data, configurations to apply to each node). The Figure 5.7 represents the overview architecture when the pattern is applied.

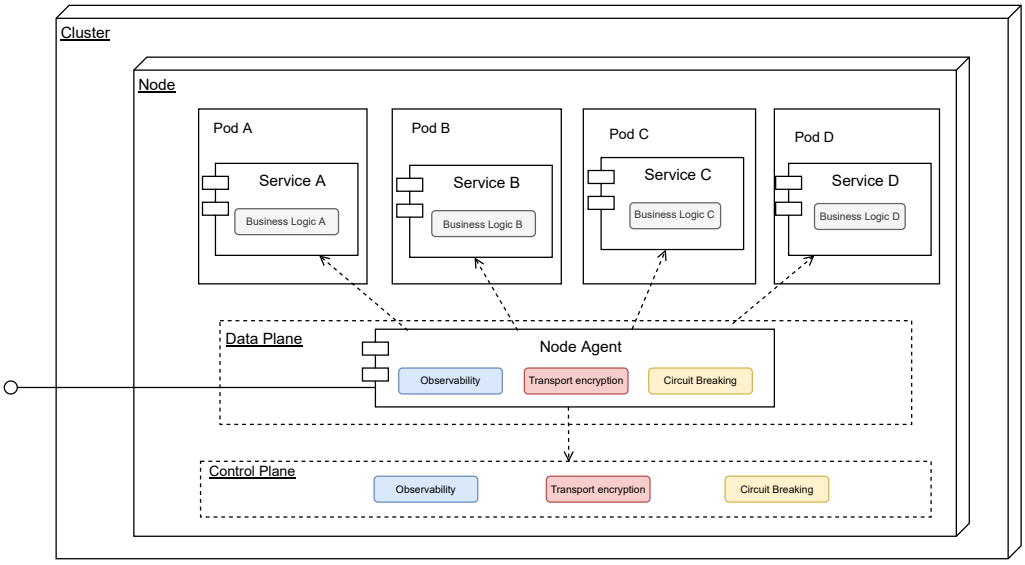


Figure 5.7: A microservices architecture system with a NODE AGENT pattern applied. Here we can see the node agent acting as a proxy for all the traffic inside the node of a given cluster.

### 5.6.3 Consequences

This pattern has the following positive consequences:

- Saving Resources (CPU and memory) compared with the SIDECAR pattern since we only implement a single agent per node.
- Teams don't need to own the services to implement it.

- Easier to manage and maintain compared to a library.
- Easier to implement compared with the sidecar approach since there is only one proxy to configure.

This pattern has the following negative consequences:

- Needs to share configurations in the node to all containers.
- Needs more coordination between different teams to organize configurations for different services.
- Needs cooperation with the container infrastructure since the node agent requires to be deployed on the node where the services are running.

### 5.6.4 Related Patterns

This pattern has the following related patterns:

- MICROSERVICES - to apply the patterns in this paper we are always taking into account the fact that the application is using the microservice architecture.
- SIDECAR - a way to use the service mesh as the node agent, the SIDECAR creates a data plane based on deploying proxies next to each service.

### 5.6.5 Known Uses

Known uses for the NODE AGENT pattern can be found for example in the earlier version of the Linkerd service mesh (prior to version 2.x) and in the Traefik Mesh service mesh.

## 5.7 Sidecar

When architecting a system based on SERVICE MESH pattern...

...and the system consists of a multi-language microservices application where the different services would benefit from having different configurations applied to their cross-cutting concerns related to communication.

### 5.7.1 Problem

**Cross-cutting concerns often lead to the duplication of code and a bad division of responsibilities between infrastructure logic and business logic.**

A Service Mesh allows addressing cross-cutting concerns in service-based systems. When the application is written in multiple coding languages, the possibility of using a library to implement the service mesh tends to seem quite troublesome since it would need multiple libraries for the multiple languages. Therefore the urge to create a new type of module arose. With the world

of containers growing each day, the ideal scenario would be to create a new module that would be deployed as a container on the same node as the services. Regarding the structure and the way that the module is deployed, is where things get tricky. On one hand, this module should be easy to implement and deploy to make sure that the developers can use it without concerns, and it should also be lightweight since machine resource availability should not be a reason to not use a service mesh. On the other hand, having a new module for each service would cost more resources but would allow the full independence of the services regarding cross-cutting configurations and network configurations (network namespaces, routing tables).

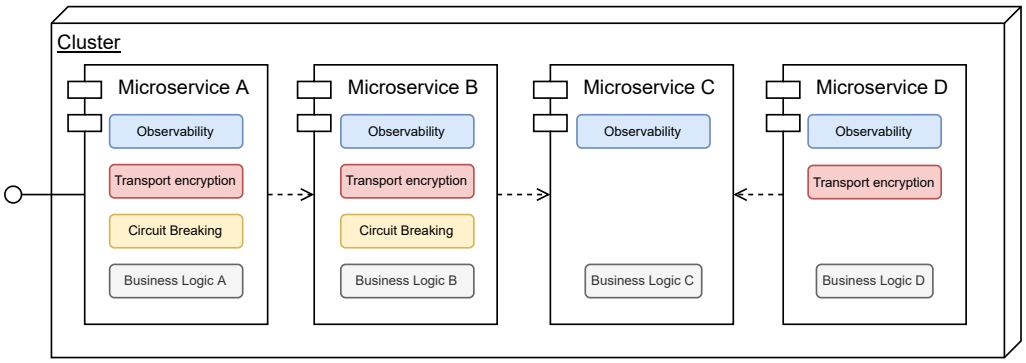


Figure 5.8: A microservice architecture system without the use of a SIDECAR. Each service implements its business logic, but also some communication-related concerns that cross-cut the system as a whole

### 5.7.2 Solution

**Encapsulate cross-cutting concerns related to communication into a new module that runs alongside each of the services of the application.**

Instantiate a new container (the sidecar) for each of your services. These new containers will live in the same pods as the services. The new modules will act as a personal proxy for each service providing all the cross-cutting concerns needed since all the communication to the service must now pass through the sidecar. As the sidecar lives in the same pod as the service it can access all the resources from the service without any significant latency between communications (this enables to monitor the resources being used by the service and the sidecar)

### 5.7.3 Consequences

This pattern has the following positive consequences:

- Independence of configurations for each service (very customizable).
- Safety of the services increases since the sidecar acts as a proxy even within internal application communication creating another security barrier.

This pattern has the following negative consequences:

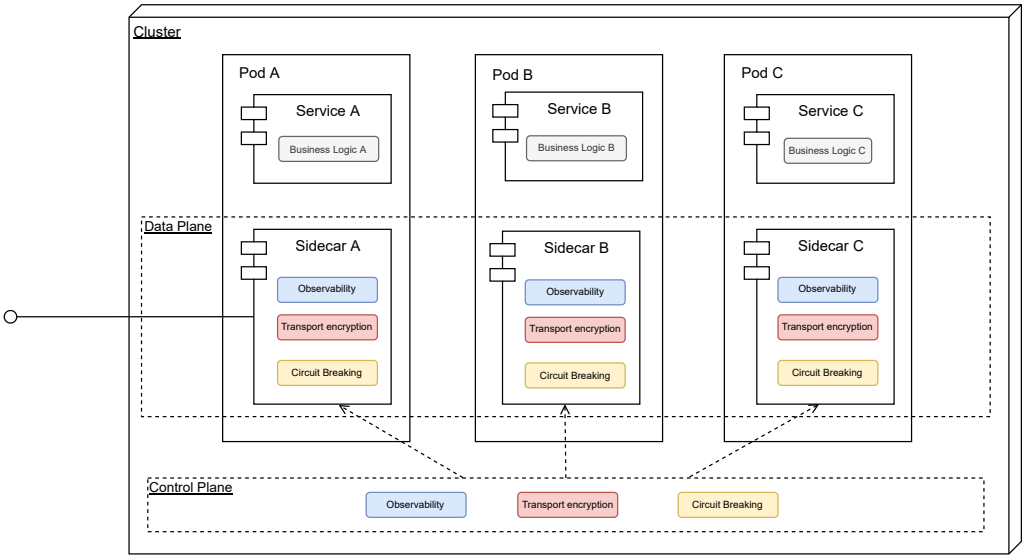


Figure 5.9: A microservices architecture system with a SIDECAR pattern applied. Each service on its own pod has a new container, the sidecar, running alongside and acting as a proxy for the traffic of that service. In this example, Pod A is also acting as an Ingress/Gateway of the application for the simplicity of the chart.

- More complex to implement compared with the other architectural approaches.
- Dependency of coordination and cooperation with container orchestration infrastructure (such as Kubernetes).
- Needs to allocate more resources (CPU and Memory) in the cluster to be used on the sidecars.

### 5.7.4 Related Patterns

This pattern has the following related patterns:

- MICROSERVICES - to apply the patterns in this paper we are always taking into account the fact that the application is using the microservice architecture.
- NODE AGENT - a way to use the service mesh as the sidecar, the NODE AGENT creates a data plane based on deploying an agent per node.

### 5.7.5 Known Uses

Known uses for the SIDECAR pattern can be found for example in the majority of the service meshes available such as Istio, Linkerd (version 2.x), Consul, AWS and many others.

## 5.8 Service Mesh Team

When architecting a system based on microservices with the concern of having a clear separation of concerns ...

... and after taking into consideration the advantages of the service mesh pattern and implementing it, questions may appear regarding the structure of the company after such change.

### 5.8.1 Problem

**The teams are not enough qualified to solve all the problems and bugs that appear related to the Service Mesh, also teams are dependent of others teams to take actions regarding the service mesh.**

Having a service mesh can help fix a lot of troubles since we successfully encapsulated cross-cutting concerns out of the services into the new modules. However, by doing so a new problem arises. Now the application is running a service mesh, but to keep running it smoothly it's not an easy task because the emerging technology demands a lot of knowledge and expertise. On one hand, leaving the developers the task of also controlling the service mesh on top of their own services can accumulate a lot of stress and overwork. Nevertheless, it ensures that the service mesh works with the best configurations since the developers are the ones with more know-how about their services. On the other hand, having separate people with expertise on the subject working on the service mesh can enhance the speed to troubleshoot and improve the service mesh, but it also requires a lot of cooperation between the developers and the experts which can lead to delays in development.

### 5.8.2 Solution

**Create a new team inside the company whose main goal is to guide actions related to the service mesh and remove the dependencies between teams.**

The new team should be able to guide other teams to apply the best configurations in the Service Mesh for their services, answer their doubts and make sure that everything works as intended. This new team should remove the work to implement the cross-cutting concerns from the developers, so they can focus on implementing the business logic. One thing to keep in mind is that every service is different from the other. Consequently, the service mesh team's objective should never be to replace the developer's work related to analyzing and understanding the requirements and configurations for their services regarding the cross-cutting concerns, but to better support their work. By having this new team handle this new piece of the infrastructure, development teams will not need to depend directly on each other to decide on the best service mesh setup, and can instead rely on the service mesh team to understand the needs of all the teams and take actions related to the service mesh.

### 5.8.3 Consequences

This pattern has the following positive consequences:

- Help developers to simplify the process of implementing service mesh.
- Developers can focus their full time on business logic.
- Faster troubleshooting and bug fixing.
- Better structure of the company, well-defined roles and responsibilities between developers.
- Remove dependency between teams of developers since they can now depend only on the service mesh team.

This pattern has the following negative consequences:

- Need more developers for the new team, this can be achieved by hiring more developers or other teams need to lose some developers.
- Effort in cooperation between developers and the service mesh team.
- Dependency added between teams and the service mesh team.

### 5.8.4 Related Patterns

This pattern has the following related patterns:

- MICROSERVICES - to apply the patterns in this paper we are always taking into account the fact that the application is using the microservice architecture.
- SERVICE MESH - to be able to use this pattern it is implicit that a service mesh is already being used.

### 5.8.5 Known Uses

We can find references for the SERVICE MESH TEAMS in numerous resources available on the Web, but very few of significant details. Notwithstanding, one of the companies know for using a SERVICE MESH and for having a SERVICE MESH TEAM is Salesforce [19].

## 5.9 Multi-Cluster Service Mesh

You have a critical microservices application running with a service mesh deployed across different clusters, and you want to improve even further the application resilience.

### 5.9.1 Problem

**In the case of a cluster or control plane degradation, user workloads may suffer an outage thus making the application not available**.

A SERVICE MESH allows addressing cross-cutting concerns in service-based systems, thus improving the division of responsibilities between the cross-cutting concerns and the business logic of each service.

When architecting critical microservices applications, the SERVICE MESH can help with the high availability needed for those types of software, with features like load balancing, constant monitoring of the traffic, automatic restart of defectuous containers, secure communication channels and many others. Example of an implementation of a service mesh can be seen in Figure 5.10. Nevertheless, all these features act at the application level, but we can still have problems at the infrastructure level related to the control plane or the cluster. On one hand, we need to have mechanisms to ensure that, even if a part of the infrastructure supporting the application fails the application can still be available. On the other hand, creating new high availability mechanisms can be very troublesome, time-consuming and even add a new complexity layer.

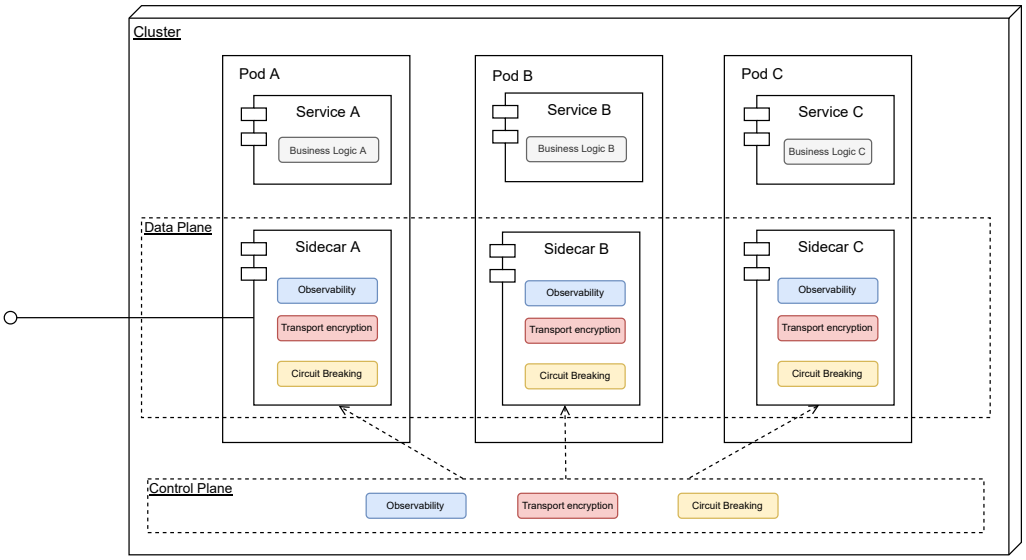


Figure 5.10: A microservices architecture system with a SERVICE MESH applied, more precisely a service mesh using the SIDECAR pattern.

### 5.9.2 Solution

**Make the service mesh deploy one control plane in each cluster where the application is running.**

Deploying one control plane in each cluster will enable the service mesh to control the application in a more stable and secure way gaining new resilience features. Since we have multiple control planes in different clusters it is possible now to shift traffic between clusters when needed and maintain the advantages of the SERVICE MESH. This way the MULTI-CLUSTER SERVICE

MESH can prevent failures when one cluster is down or allow for a zero-downtime upgrade of the Kubernetes infrastructure when needed by shifting traffic from one cluster to another. An example of the MULTI-CLUSTER SERVICE MESH can be seen in Figure 5.11.

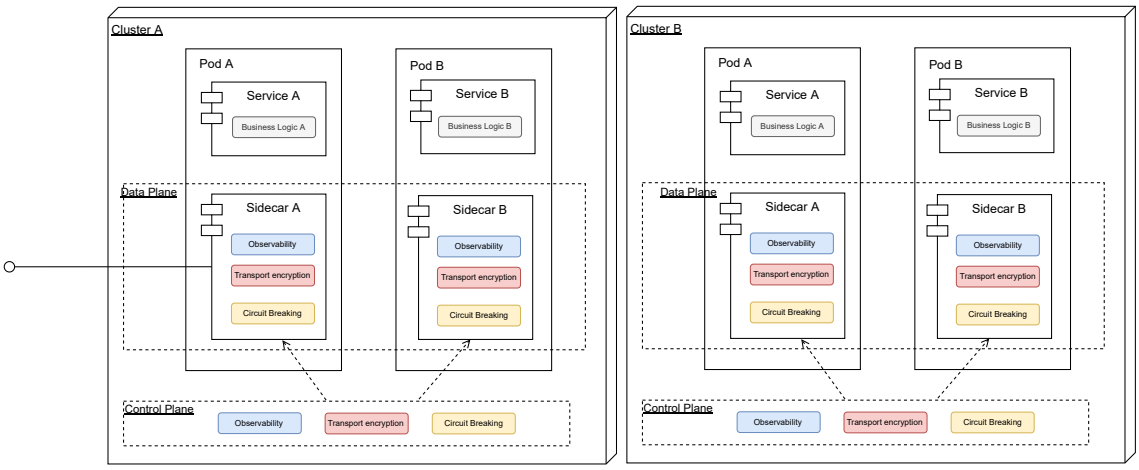


Figure 5.11: A microservices architecture system with a MULTI-CLUSTER SERVICE MESH pattern applied. Here we can see that the two clusters have each one a control plane.

### 5.9.3 Consequences

This pattern has the following positive consequences:

- Service level isolation by being able to deploy some services to a specific cluster, and if a call is made from another cluster it will fail the DNS lookup.
- Possibility of making changes to a specific cluster without affecting the others.
- Enables disaster recovery and targeting traffic away from unhealthy clusters.
- High availability in the case of one cluster needed to update the Kubernetes infrastructure we can use another cluster to keep the application running.

This pattern has the following negative consequences:

- More complexity due to having to configure and maintain multiple control planes.
- More resources needed to be allocated for the control planes in the different clusters.

### 5.9.4 Related Patterns

This pattern has the following related patterns:

- MICROSERVICES - to apply the patterns in this paper we are always taking into account the fact that the application is using the microservice architecture.
- SERVICE MESH - to be able to use this pattern it is implicit that a service mesh is already being used.

### 5.9.5 Known Uses

Known uses for the MULTI-CLUSTER SERVICE MESH pattern can be found for example in some of the service meshes available such as Istio, Linkerd, Consul and Kuma.

## 5.10 Discussion

During this chapter we were able to present six design patterns to answer the first research question presented in Section 5.1. The patterns presented consisted in a **problem statement**, a **solution** for that problem, the **consequences** of implementing the solution, **related patterns** and the **known uses**.

The mining pattern process was presented in Section 5.2, and the target audience for the patterns as well as the relationship between one another was explained in Section 5.3.

Now that we have our patterns documented, we can try to answer the first research question of this thesis.

- **RQ1** "To what extent can we document new design patterns for the service mesh or improve even further the ones that already exist?" - As we mentioned previously we were able to document six design patterns. Four of those considered new design patterns (NODE AGENT, SHARED COMMUNICATION LIBRARY, SERVICE MESH TEAM and MULTI-CLUSTER SERVICE MESH) and two of those considered improvements of previous already documented patterns (SERVICE MESH and SIDECAR). This process of documenting design patterns was not a straight path, since some initial possibilities of patterns were discarded midway because not all good solutions for a problem can be considered a software pattern. We have concluded that for the literature analyzed these were the possible patterns to be documented, nevertheless, major work is still in demand in this area as we did not cover all types of literature, especially, grey literature.

# Chapter 6

# Empirical Study

Throughout this chapter, we will describe the controlled experiment used to evaluate some aspects of the patterns presented in Chapter 5. This controlled experiment consists of the deployment of a microservices open-source project in the Google Cloud Platform and the implementation of two different service mesh designs, to ensure that both the design patterns associated with the service mesh, NODE AGENT and SIDECAR, would be covered.

The chapter is divided in the following sections: on the **goals** we will explain the objective of this experiment and what carried us into executing it (*cf* Section 6.1), on **methodology** we summarize the different steps required to mount the experiment (*cf* Section 6.2), the **replication** section illustrates the practical part of the experiment explaining the order of actions and files used so that anyone can replicate it (*cf* Section 6.3). The **results** show the summarized view of the values measured in the different approaches (*cf* Section 6.4) and in the end we have the **threat to validity** that discuss what could have spoiled our experiment (*cf* Section 6.5) and in the **discussion** we present the possible conclusions of the data analyzed and answer the research question (*cf* Section 6.6).

## 6.1 Goals

The goal of this experiment is to evaluate our patterns created in Chapter 5 and to answer the second research question presented in Chapter 4. Namely, the research question that is going to be answered in this chapter is:

- **RQ2** "To which extent did our patterns accurately captured the consequences of their solutions?" Although in our patterns we presented multiple consequences that could affect the application, for this thesis we are going to focus on the CPU and memory usage when the patterns are applied. We will perform an experiment to measure the resource utilization of CPU and memory in three different design approaches for the service mesh.

We decided to focus our experiment on the SIDECAR and NODE AGENT since those two patterns are the concrete implementations of a service mesh out of all the patterns that we documented in Chapter 4.

## 6.2 Methodology

In the preparation and execution of our experiment, we had to go through different phases and so we will be introducing different aspects that helped us to create this experiment. Questions such as "What is the microservices application used?", "What are the service meshes used?", "What was the infrastructure used to assemble this experiment?" and some others, should be answered during the following sections. The first section, **project selection** (*cf* Chapter 6.2.1), explains the criteria for the selection of the project and presents the chosen project that was necessary to use as a baseline and to deploy the service meshes on top of. The next section, **infrastructure** (*cf* Chapter 6.2.2), explains the infrastructure that was required to be mounted to hold the application, the next three section , **baseline** (*cf* Chapter 6.2.3), **sidecar implementation** (*cf* Chapter 6.2.4) and **node agent implementation** (*cf* Chapter 6.2.5) resume the steps taken to be able to mount the three different contexts inside the infrastructure, the last two section **dependent variables** (*cf* Chapter 6.2.6) and **injecting traffic** (*cf* Chapter 6.2.7) explain what were the dependent variables monitored and the approach chosen to inject different levels of traffic to monitor those same variables . An in-depth description of these steps can be found in the following sections. The Figure 6.1 gives an overview of the order of actions taken to mount this experiment.

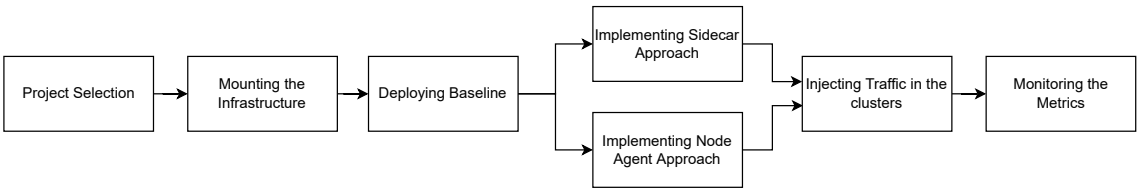


Figure 6.1: Overview of the steps present in the methodology process for the controlled experiment

### 6.2.1 Project selection

Based on the goal defined in the previous section, we want to measure the CPU and memory usage of a microservices application with two different service mesh approaches to be able to answer the research question. Consequently, to be able to make this experiment we decided to choose a microservice application that needed to answer a few specific criteria:

- **Open-source** - The application should be open-source to ensure that any person can replicate this experiment with their own preferences and settings.
- **Multiple Microservices** - The application should be a microservices application with different services having each one its functionality. It is different to compare a microservices

application with two different microservices, but being replicated ten times each service than to compare an application with ten different services, each one replicated two times, since the complexity of the communication between the microservices would increase (security certificates, DNS discovery etc).

- **Multiple Languages** - The application should have services using different languages to illustrate better the context where the service mesh should be implemented.
- **Agnostic to Service Meshes** - The application should not be using a service mesh, meaning that the application should not have been built for the purpose of showcasing a specific service mesh as this would remove the unbiased factor.

Taking into account the criteria above defined, by running a search on Google, we found on GitHub a microservices application that fitted the criteria. We decided to use the Sock Shop application. This e-commerce microservices open-source application was developed by WeaveWorks[1] and is "intended to aid the demonstration and testing of microservice and cloud native technologies" as mentioned in the GitHub repository of the project[2]. The application is composed of eight microservices (front-end, order, payment, user, catalogue, cart, shipping and queue-master) one of them being the API that receives the traffic (front-end) and redirects requests to specific services as needed. The application uses four different programming languages (Go, Java, JavaScript and C#), three main frameworks (Spring Boot, Go Kit and Node.js), two databases (Mongo and MySQL) and one message broker (RabbitMQ). An architecture overview of the application can be found in Figure 6.2. The repository of the service had the application already containerized and ready to be deployed in a Kubernetes cluster.

### 6.2.2 Infrastructure

Since we wanted to deploy the microservices application we had to choose a cloud infrastructure, in our case a cluster, to hold our application and the service meshes. We knew that to implement a service mesh we required a container orchestration tool managing the microservices, hence we opted to use Kubernetes[3], as it is currently considered the de facto standard for container orchestrators and the container orchestrator that service meshes tools are supporting the most, as showed in Section 3.6 (when we concluded that all the service meshes analyzed supported Kubernetes). There were a few options regarding how we could implement a Kubernetes cluster. Initially, we tried to use Minikube[4] as it enables a quick set up of a local Kubernetes cluster in our machine, but we had to change our approach since it started to show gaps in crucial features (specifically with mapping ports from the cluster in our virtual machine to the browser in our Windows machine). Thus, we decided to use the Google Cloud Platform (GCP). GCP comes with a lot of cloud

[1]More information about the company can be found at `https://www.weave.works/`
[2]Repository of the project `https://microservices-demo.github.io/`
[3]More information about Kubernetes at `https://kubernetes.io`
[4]More information about Minikube at `https://minikube.sigs.k8s.io`

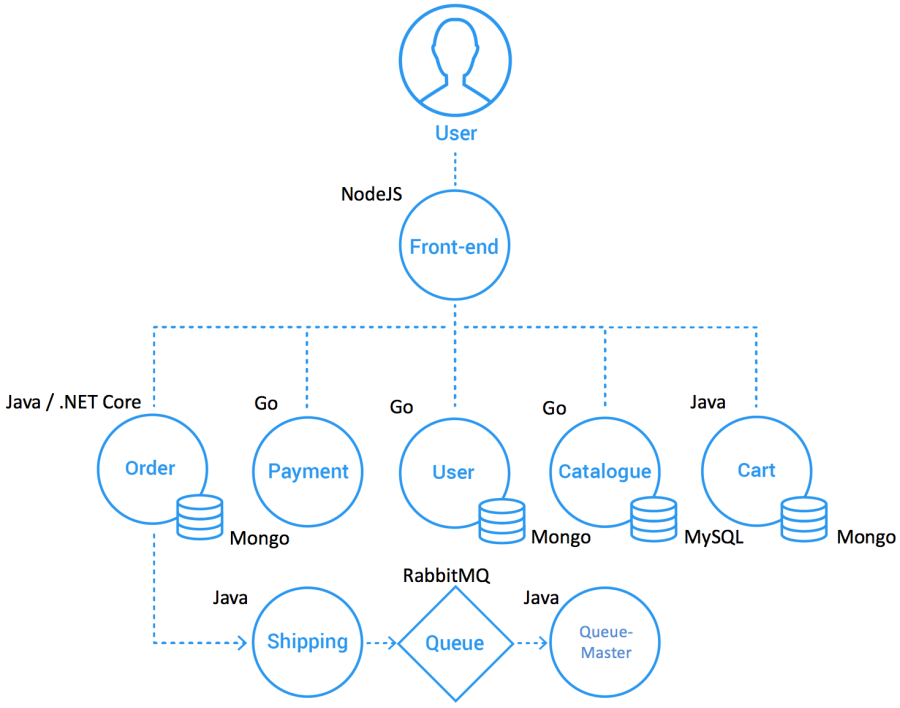


Figure 6.2: Sock-shop architecture overview, source GitHub repository of the project

tools and one of those tools is the Google Kubernetes Engine (GKE) which we used to hold our microservices application.

### 6.2.3 Baseline

Since the objective of this controlled experiment was to evaluate and compare the sock-shop application, running with two different service mesh designs and without a service mesh app, we needed to have a baseline for the comparison. We used as a baseline the sock-shop application in its "natural" state, to which we compared implementations of the SIDECAR and NODE AGENT service mesh patterns. By accessing the GKE console we were able to connect to our cluster and deploy the different parts of the application and the required Kubernetes components so that the services could communicate with each other. To manage a Kubernetes cluster, we use YAML configuration files with a specific schema defined for each type of resource.

### 6.2.4 Sidecar Implementation

According to the pattern SIDECAR, described in Section 5.7, this design of a service mesh, deploys a control plane and for the data plane, it deploys a new container next to each container that is running a microservice of the application, the so-called sidecar.

We decided to choose the Istio service mesh for the implementation of the SIDECAR since it is one of the multiple service meshes that use the sidecar design. Istio deploys the istiod component that acts as the control plane for the service mesh, controlling the traffic inside the cluster. For the

data plane it uses, by default, Envoy proxies as sidecars to "Encapsulate cross-cutting concerns related to communication into a new module that runs alongside each of the services of the application" as mentioned in the pattern solution. Figure 6.3 shows a simplified overview of the final architecture (it does not include all microservices for clarity and simplicity of the image).

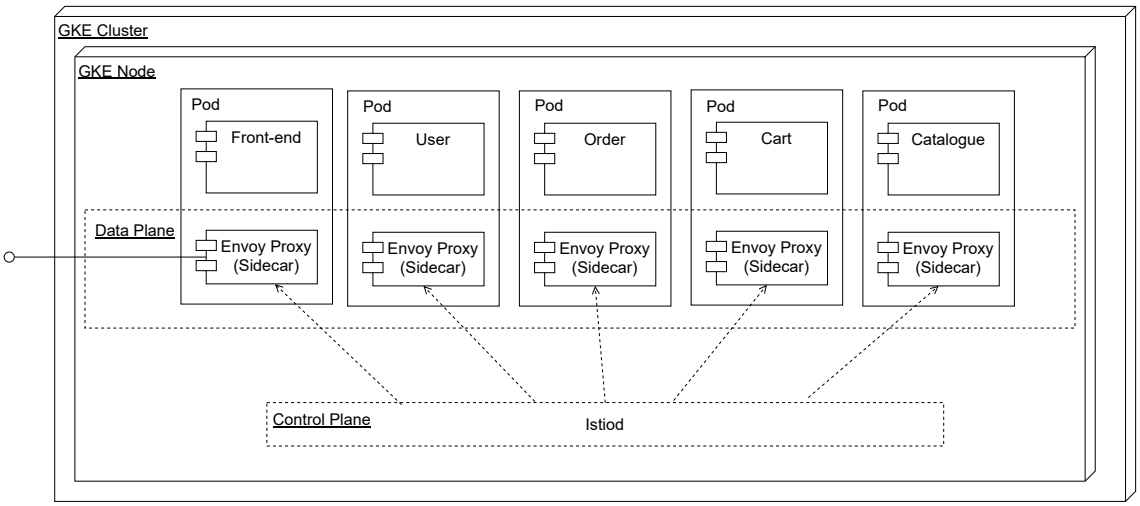


Figure 6.3: Sock-shop architecture simplified overview using Istio as a service mesh

### 6.2.5 Node Agent Implementation

According to the pattern NODE AGENT, described in Chapter 5 and Section 5.6, this design of a service mesh deploys a control plane, and for the data plane, it deploys a single agent running on each node that the cluster has.

Traefik Mesh was the choice for the NODE AGENT implementation since it is the only open-source service mesh tool that currently uses the node agent approach. Traefik Mesh deploys the Traefik control plane component, that act as the control plane for the service mesh, controlling the traffic inside the cluster. For the data plane, it uses the Traefik Proxy as the node agent to "Encapsulate cross-cutting concerns related to communication into a Node Agent that acts as a proxy for all the services that are running on the same working node." as mentioned in the pattern solution. Figure 6.4 shows an overview of the final architecture (it does not include all microservices for clarity and simplicity of the image).

### 6.2.6 Dependent Variables

One of the elements presented in the patterns documented in Chapter 5 is the consequences of the patterns. In this experiment we wanted to validate that the consequences mentioned in the pattern SIDECAR and the pattern NODE AGENT related to CPU and memory usage could be validated, our dependent variables (in order to answer our research question). Thus, for this experiment, we wanted to observe, measure and compare the memory and CPU with different load tests. We ended up using one of the tools available in the GCP called Monitoring. This tool acts as a central point

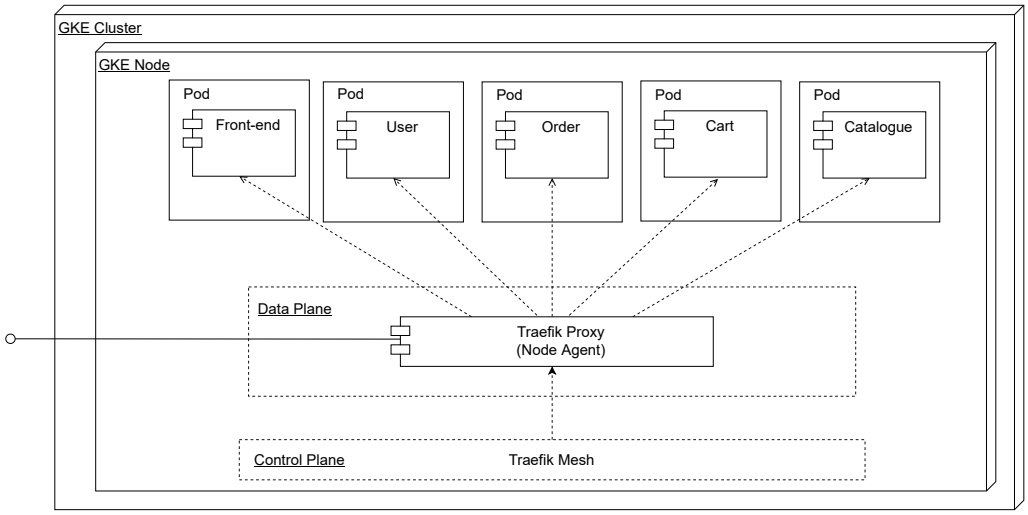


Figure 6.4: Sock-shop architecture simplified overview using Traefik Mesh as a service mesh

for aggregating logs, metrics and alarms that are present throughout our entire GCP project, and so we were able to collect metrics related to the GKE (that can be translated into metrics for our Kubernetes cluster). An example of the monitoring dashboard mentioned previously can be found in Figure 6.5. Inside the Monitoring tool, we were able to create a custom dashboard with our own metrics of choice. To do this we went to the monitoring section in GCP, added a new dashboard and then added new graphics by selecting the type of graph that we wanted, the variable to monitor and applied a filter to represent the sum of the node values.

For the CPU usage, the metric used is CPUs and it is explained in the Kubernetes documentation[5] as "CPU is reported as the average core usage measured in cpu units. One cpu, in Kubernetes, is equivalent to 1 vCPU/Core for cloud providers, and 1 hyper-thread on bare-metal Intel processors." As for the memory usage, the documentation explains "Memory is reported as the working set, measured in bytes, at the instant the metric was collected.".

### 6.2.7 Injecting Traffic

To be able to compare the three implementations, we decided that we should measure the cluster resource usage when dealing with different levels of traffic entering the application. To simulate traffic with different intensities entering our cluster and using the sock shop, we used an external framework named Locust[6] to inject the traffic in a controlled way by making requests to the different services. The different phases of traffic intensity were divided as follows:

- **No traffic** - this is the default state of the application in the cluster without receiving any type of traffic.

[5]More information on metrics for Kubernetes can be found here https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/

[6]More information about locust can be found on its website at https://locust.io/

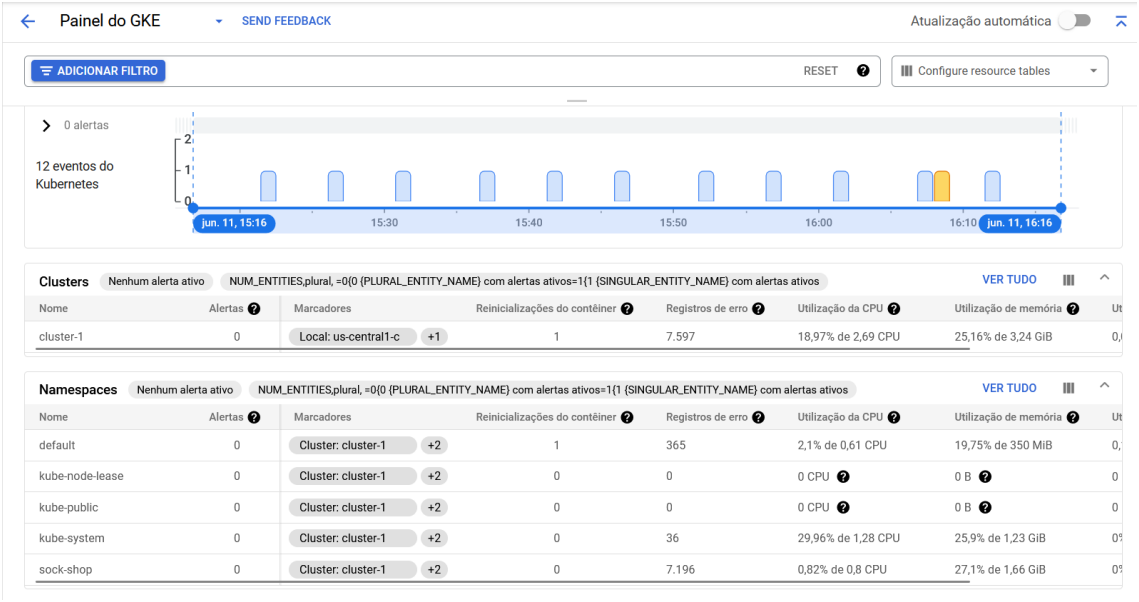


Figure 6.5: Monitoring panel example for the Google Kubernetes Engine cluster in the Google Cloud Platform

- **Medium Intensity traffic** - this is the state of the application in the cluster when it is being injected traffic with the configurations `-clients 20 -hatch-rate 1`
- **High Intensity traffic** - this is the state of the application in the cluster when it is being injected traffic with the configurations `-clients 50 -hatch-rate 1`

As some of the terms mentioned above may not be clear, we will try to explain them. The variable "clients" is referring to the maximum number of active users making requests at the same time to the application. The variable "hatch-rate" is the number of users created per second starting at 0, also known as the ramp-up period.

## 6.3 Replication

In order to replicate all the experiments described in this chapter, we offer some instructions about the usage of the study's materials. This type of information is useful since it is important to give the tools to other developers so that they can individually replicate this experiment to confirm the data presented.

All the documents and files mentioned throughout this chapter are available in a GitHub repository[7]. This repository will hold all the files and instructions necessary to replicate this experiment as explained in Section 6.3.

Below, we explain the steps taken and the files used to execute this experiment to help to replicate this experiment. To begin to replicate this experiment we need to mount the infrastructure.

[7]https://github.com/tiago-maia/ServiceMesh-Implementations

- Create a new Google Cloud Platform project
- Enable Google Kubernetes Engine in the project - and so we created our first Kubernetes cluster with three nodes (default behaviour when creating a new cluster to have higher availability)
- Create an admin RBAC[8] and apply to console user - to ensure that we could have full access over our cluster
- Repeat the initial three steps for each project needed. One for the sidecar approach, one for the node agent approach and one for the baseline.

After having the three different clusters, each one for a different implementation, we deployed the sock shop microservices application in each cluster individually. We will demonstrate the types of Kubernetes objects used for the deployment of the sock-shop (all the files necessary to deploy the application can be found in the repository mentioned above in the following path "ServiceMesh-Implementations/sock-shop/deploy/kubernetes/manifests/")

- **kubernetes deployment** - configures the container(s) image(s) to be used in the pod and configures the number of replicas of that same pod to be deployed.
- **kubernetes service** - configures an abstraction of a network service that is required for the automatic service discovery, enabling traffic to reach the pods.
- **kubernetes namespace** - configures a mechanism to isolate groups of resources within a cluster.

To deploy the application we simply used the command `kubectl apply -f /manifests` that orders Kubernetes to apply in the cluster all the files inside the specified folder. We used this command in the three different clusters to ensure that in all the clusters the sock shop application was running.

As for the implementation of Istio in the GKE cluster, it was straightforward. We started by downloading the most recent Istio version, in the time of writing this article it was the version 1.13.4, then we added the istioctl to our PATH and we proceeded to install Istio on our cluster running the following command `istioctl install -set profile=demo -y`. After having Istio installed in our cluster we needed to notify it of the namespace that contained the resources to add to the service mesh. This is done by applying a label to the desired namespace `kubectl label namespace sock-shop istio-injection=enabled`, this way Istio knows that deployments that happen in that namespace are to be added to the service mesh alongside a sidecar next to each service. Hence, after we created the sock-shop namespace we changed the label and deployed the sock shop application.

[8] RBAC - Role-Based Access Control

Compared to Istio's installation Traefik Mesh had some nuances that needed to be tackled. First, for the installation, it was required to use Helm (a package manager for Kubernetes resources) with the version v3 to obtain the package and to install Traefik Mesh in the cluster. After installing the Traefik Mesh, we found out that the service mesh only worked with CoreDNS and the GKE cluster uses KubeDNS, so we had to change the DNS server. Being finished with the changes, Traefik Mesh was ready to be used and so with a simple change of the hostname, we could reach our cluster through the service mesh. For example, to reach the service front-end before the Traefik Mesh we would "curl frontend.sock-shop.svc.cluster.local" and now with Traefik we would use "curl frontend.sock-shop.traefik.mesh".

## 6.4 Results

In this section, we are going to present the results obtained through our experiment. Taking into account that the experiment had three test treatments (no service mesh, SIDECAR and NODE AGENT) and each test treatment was exposed to three different experimental conditions (no traffic, medium traffic and high traffic) and this experiment was executed for7 two dependent variables we ended up with eighteen sets of values of our analysis. For simplicity, we are going to describe the results and show the values in box plots to help visualise the results. Nevertheless, all the original graphs from the GCP monitoring and the CSV values are included in our replication package under the TestResults folder.

### 6.4.1 CPU Usage

For the CPU measuring, we used the graph of **CPU usage time** in the GCP monitoring dashboard. In the GCP documentation, this data is described as "Cumulative CPU usage on all cores used on the node in seconds".

This variable allows us to measure the CPU being used in real-time in all our nodes. As our services can be deployed in any node, since when we deployed our application we didn't specify a node, we applied a filter to sum the values of the three nodes into a single value getting the result of CPU usage for our cluster.

The time interval of each traffic injection intensity in each implementation was of ten minutes starting to count after the ramp-up finished and measured every ten seconds leading to sixty data points.

Initially with no traffic being injected the sidecar approach values of CPU usage in the cluster rounds the 0.5 CPUs, the node agent approach rounds the 0.25 CPUs and the baseline averages 0.21 CPUs as shown in the Figure 6.6.

After we initialized the traffic injection with middle intensity the sidecar approach values of CPU usage in the cluster got up to around 2.3 CPUs, the node agent approach increased to 1.8 CPUs and the baseline to 1.79 CPUs as shown in the Figure 6.6.

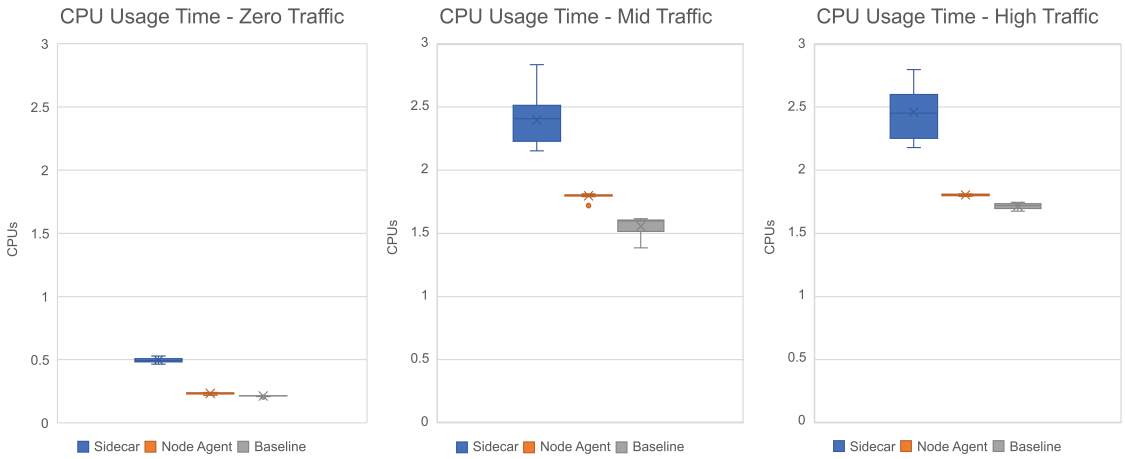


Figure 6.6: CPU usage in GKE cluster with different levels of traffic being injected

By increasing the injection of traffic to high the values increased a little more with the sidecar approach of CPU usage in the cluster got up to around 2.5 CPUs, the node agent approach stayed around the 1.8 CPUs and the baseline also stayed around 1.79 CPUs as shown in the Figure 6.6.

### 6.4.2 Memory Usage

For the memory measuring, we used the graph of **Memory usage** in the GCP monitoring dashboard. In the GCP documentation, this data is described as "Cumulative memory bytes used by the node". This variable allows us to measure the memory being used in real-time in all our nodes. As it happened for the measure of CPU, we also applied here a filter to join the values of memory into a single value for better visualization of the data.

The time interval of each traffic injection intensity in each implementation was of ten minutes starting to count after the ramp-up finished and measured every ten seconds leading to sixty data points.

Initially with no traffic being injected the sidecar approach values of memory usage in the cluster rounds the 6.73 GiB, the node agent approach spent around 5.83 GiB and the baseline averages 5.60 GiB as shown in the Figure 6.7.

After we initialized the traffic injection with middle intensity the sidecar approach values of memory usage in the cluster got an increase in value but also in the interval of values ranging from 7.50 GiB up to 8.51 GiB, the node agent approach increased to 5.9 GiB and the baseline had a small increase to averaging 5.75 CPUs as shown in the Figure 6.7.

By increasing the injection of traffic to high the values increased a little more with the sidecar approach of CPU usage in the cluster got up to averaging 8.07 GiB, the node agent approach averaging 6 GiB and the baseline had almost no increase with the values averaging 5.8 GiB as shown in the Figure 6.7.


Figure 6.7: Memory usage in GKE cluster with different levels of traffic being injected

## 6.5 Threats to Validity

During the design of the controlled experiment, we thought about the experimental conditions that could cause deviations in the results compromising the validity of our approach. Since we want to obtain the most precise and unbiased results possible for our research questions, some precautions were taken to discard these threats.

- **Reproducibility of the infrastructure** - a local implementations of Kubernetes clusters using of a personal machine could make it impossible to replicate for other developers (due to lack of resources in their personal machine) and so we used the google cloud platform and the google Kubernetes engine
- **Project** - some applications can be biased in favour of a specific service mesh and so we tried to mitigate this issue opting to use a generic micro-services project that is used for demonstration purposes.
- **Human Error** - both implementations of the service mesh were executed by us, meaning that is always possible that we introduced errors or distortions of values.
- **Representativeness** - we are making conclusions on our implementations of the SIDECAR and the NODE AGENT only based on the tools that we implemented, meaning that we could have implemented the SIDECAR with multiple tools instead of Istio and perhaps we would have different results.

## 6.6 Discussion

The experiment was designed to be able to compare the two different designs of the service mesh against the baseline case, the case with no service mesh, to try to compare them and to answer

the second research question mentioned in Chapter 4. After analyzing the results obtained in Section 6.4 we can distinctly notice the difference in the usage of the resources in the cluster by the different approaches. The SIDECAR approach, for which we have used Istio, was the one that spent the most amount of resources, and the approach that had the wider group of values for each testing phase, clearly seen in the box plot graphs both in CPU and in memory. The node agent, for which we have used the Traefik Mesh, and the baseline had very similar results but the node agent still uses a bit more in all the tests for CPU and memory. While still receiving no traffic at all, we can observe a clear difference between the SIDECAR approach and the others leading to the conclusion that this approach and the NODE AGENT approach both have a constant extra value of resources required for implementing their approaches and a possible variable value according to the traffic conditions.

Now that we have collected the results for the different service mesh designs, we can try to answer the second research question of this thesis.

- **RQ2** "To which extent did our patterns accurately captured the consequences of their solutions?" As we have said in previous sections, although in our patterns we presented multiple consequences that could affect the application, we limited the scope of our analysis in the context of this thesis to the CPU and memory usage implied by the patterns. Based on the experiment we can confirm our assumption that one of the consequences in our patterns SIDECAR and NODE AGENT is, indeed, an increase in CPU and memory usage compared with the baseline that did not use a service mesh. On one hand, the sidecar consumes more than the node agent since that approach deploys a new container next to each service requiring more CPU and memory for the sidecars and the control plane. On the other hand, the node agent only needs to deploy a single agent per node, thus spending fewer resources than the sidecar approach, and although it has higher values than the baseline the difference is relatively small.

# Chapter 7

# Conclusions

This chapter provides a quick overview of this thesis. We start by summarising the conclusions of this work (*cf* Section 7.1), then we present its main contributions (*cf* Section 7.2) and, finally, we explore the possible paths that could be followed as future work (*cf* Section 7.3).

## 7.1 Summary

After doing a literature review on the service mesh presented in Chapter 3, specifically, documented patterns for the service mesh, empirical studies of implementations of service meshes and analyzing the different approaches already documented, it was possible to assess the problem statement defined in Chapter 4. In Chapter 5 we documented six design patterns, of those being, four entirely new patterns (NODE AGENT, SHARED COMMUNICATION LIBRARY, SERVICE MESH TEAM and MULTI-CLUSTER SERVICE MESH) and two that improve upon previously documented patterns (SERVICE MESH and SIDECAR). Having the patterns ready, we wanted to show, to a certain extent, that the consequences defined in the patterns were accurate. Consequently, in Chapter 6 we performed an experiment and compared two implementations of service meshes, the SIDECAR and the NODE AGENT. Based on that experiment, we were able to confirm our assumption that using a service mesh requires more CPU and memory compared to an application running without a service mesh.

With this dissertation, we believe that we made our contributions to this field with conclusions that can help developers guide their implementation of a service mesh. This thesis took one more step towards the documentation of patterns for the emerging technology that are the service meshes.

## 7.2 Main Contributions

This dissertation had the following main contributions:

- **State-of-the-Art Review** - We provided a comprehensive study of documentation in published works related to service meshes and tools that implement service meshes, as well as an analysis of empirical studies with patterns.
- **Documentation of new Patterns** - based on the documentation analyzed we documented and proposed six design patterns around the service mesh. Four new design patterns (SHARED COMMUNICATION LIBRARY, NODE AGENT, MULTI-CLUSTER SERVICE MESH and SERVICE MESH TEAM) and two refactors of previous documented patterns (SERVICE MESH and SIDECAR).
- **Controlled Experiment** - We created a controlled experiment of an implementation of two of the possible design patterns for the service mesh, SIDECAR and NODE AGENT. Achieved by using an open-source microservices application and hosting it on the Google Cloud Platform, and comparing the resource usage of both implementations against a baseline without a service mesh.

## 7.3 Future Work

Two main aspects can help discuss this work's future steps: the patterns and the empirical studies.

**Patterns**

- **Pattern Mining** - Due to the fact that service mesh is a recent topic there is still a lot of information coming out every day to be analyzed and perhaps more patterns to be documented. Some sources of information that could lead to new patterns can be the analysis of grey literature or the analysis of new service mesh tools that this thesis did not cover. A possible new topic for a pattern is a service mesh running in the operating system's kernel developed by eBPF[1].

**Empirical studies**

- **Controlled Experiment in Academic Environment** - An idea for a study that would complement this thesis would be a controlled experiment in an academic environment to evaluate the efficiency of a service mesh. The idea would be to create two teams of students, both using the same microservices application, but one team would implement the cross-cutting concerns with a service mesh, and the other team would need to implement the cross-cutting concerns in a traditional way. With this experiment, we could measure the efficiency of a service mesh in terms of effort and cost. The suggestion of the experience being executed in an academic environment is due to the fact that it would be very hard to execute this in a professional environment due to the costs associated.

[1] More information about the topic can be found on their website https://ebpf.io/what-is-ebpf

- **Case Study** - Since we provided some new patterns, and not all of those patterns could be validated through our empirical study, it would be interesting to perform case studies of implementations of the other patterns. For example, the SERVICE MESH TEAM or MULTI-CLUSTER SERVICE MESH in a real case scenario and observe the results obtained in the end. What would be the consequences and the conclusions taken from the implementation of the patterns? How was the change viewed by the developers?

- **Cross-cutting concerns** - Although the experience conducted in this experiment had a fairly good project that answered a lot of our initial requirements, the project was missing the implementations of cross-cutting concerns that would be beneficial for the experiment. These cross-cutting concerns would lead to an investigation of the possible advantages of implementing cross-cutting concerns using a service mesh by comparing the number of lines of necessary code to implement the required features. A suggestion for future work would be to perform a similar experiment like this one, but with a project that has cross-cutting concerns implemented.

- **Service Mesh Team** - This thesis only covered a small portion of the consequences mentioned throughout all the patterns documented. One idea to validate another consequence would be to validate the consequences of the SERVICE MESH TEAM pattern. It would be interesting to have in a real case scenario a team of developers that used a service mesh but had the help of professionals in the area and another team of developers that would try to do it all by themselves. At the end of the experiment, we could do interviews or questionnaires with the developers and try to summarize the answers and compare them to the consequences of the pattern. Would the developers prefer to work alone? Was it faster to debug issues by communicating with the service mesh team or did it never help?

- **Costs** - Another step that could be taken would be to measure the costs associated with implementing a service mesh in a cloud infrastructure like the Google Cloud Platform. After implementing the service mesh we could measure the extra cost and calculate the difference between an implementation without a service mesh versus one with a service mesh. What would be the monthly value increase? Do we save money if we use more nodes with fewer resources or fewer nodes with more resources? This could be even extended to a study that would try to calculate the advantages of big companies using a service mesh against small companies using a service mesh. Since the effort to implement a service mesh is constant do big companies save more money than small companies in proportion?

- **Usage of Resources** - Although in this study we were able to conclude that the SIDECAR uses more resources than the NODE AGENT we still lack concrete conclusions related to the resource usage of each one. A suggestion would be to implement an experiment similar to the one executed in this thesis and perform more test cases with different values of traffic being injected. When the traffic increases does the difference in resource usage between the

NODE AGENT and the SIDECAR diminish or increase? Do big companies benefit from using one service mesh approach versus the other?

# References


[1] Cloud design patterns - Azure Architecture Center, 2021.

[2] Airbnb. Taming service-oriented architecture using a data-oriented service mesh. https://medium.com/airbnb-engineering/taming-service-oriented-architecture-using-a-data-oriented-service-mesh-da7

[3] Mohammadali Akbarisamani. Service based architecture with service mesh platform in the context of 5g core. 11 2019.

[4] Carlos Albuquerque, Kadu Barral, Filipe Correia, and Kyle Brown. Proactive monitoring design patterns for cloud applications. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*, EuroPLoP '22, New York, NY, USA, 2022. Association for Computing Machinery.

[5] Christopher Alexander. *A Pattern Language*. Oxford University Press, 1977.

[6] AspenMesh. Service meshes. https://aspenmesh.io/service-mesh-architectures/.

[7] Kyle Brown, Bobby Woolf, Joseph Yoder, Cees De Groot, Chris Hay, and Ian J. Mitchell. Patterns for Developers and Architects building for the cloud, 2021.

[8] Ramaswamy Chandramouli, Zack Butcher, and Aradhna Chetal. Attribute-based access control for microservices-based applications using a service mesh. page 33, 8 2021.

[9] CNCF. Service meshes are on the rise — but greater understanding and experience are required, may 2022. Available at https://www.cncf.io/wp-content/uploads/2022/05/CNCF_Service_Mesh_MicroSurvey_Final.pdf.

[10] Consul. Main page. https://www.hashicorp.com/products/consul.

[11] Jürgen Dobaj, Markus Schuss, Michael Krisper, Carlo Alberto Boano, and Georg Macher. Dependable mesh networking patterns. pages 1–14, 07 2019.

[12] Docker. Use containers to build, share and run your applications.

[13] Amine El Malki and Uwe Zdun. Guiding architectural decision making on service mesh based microservice architectures. In Tomas Bures, Laurence Duchien, and Paola Inverardi, editors, *Software Architecture*. Springer International Publishing, 2019.

[14] Keith D. Foote. A brief history of microservices. https://www.dataversity.net/a-brief-history-of-microservices/#.

[15] Cloud Native Computing Foundation. 2020 survey. https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[16] Cloud Native Computing Foundation. Orchestration & management - service mesh. https://landscape.cncf.io/card-mode?category=service-mesh&grouping=category.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Abstraction and reuse of object-oriented design*. In European Conference on Object-Oriented Programming, 1993.

[18] Hashicorp. Consul versioning page. https://github.com/hashicorp/consul/releases.

[19] Mark Interrante. Salesforce Goes Big With Containers Service Meshes — Joins the CNCF.

[20] Istio. Istio case studies. https://istio.io/latest/about/case-studies/.

[21] Istio. Main page. https://istio.io/.

[22] Istio. Mixer configuration model. https://istio.io/v1.4/docs/reference/config/policy-and-telemetry/mixer-overview/.

[23] Christian Kohls and Stefanie Panke. Is that true...? thoughts on the epistemology of patterns. In *Proceedings of the 16th Conference on Pattern Languages of Programs*, PLoP '09, New York, NY, USA, 2009. Association for Computing Machinery.

[24] Kuma. Main page. https://kuma.io/.

[25] Linkerd. Main page. https://linkerd.io/.

[26] Xing Li†‡, Xiao Wang‡, and Yan Chen. Meshscope: A bottom-up approach for configuration inspection in service mesh. 8 2020.

[27] LXC. What's lxc. https://linuxcontainers.org/lxc/introduction/.

[28] Tiago Maia and Filipe Correia. Service mesh patterns. In *Proceedings of the 27th European Conference on Pattern Languages of Programs*, EuroPLoP '22, New York, NY, USA, 2022. Association for Computing Machinery.

[29] Markets and Markets. Cloud microservices market by component (platform and services), deployment mode (public cloud, private cloud, and hybrid cloud), organization size (large enterprises and smes), vertical, and region - global forecast to 2023. https://www.marketsandmarkets.com/Market-Reports/cloud-microservices-market-60685450.html.

[30] Aspen Mesh. Service mesh: Adopting a zero-trust approach to security for containerized applications. https://aspenmesh.io/service-mesh-zero-trust-security/.

[31] Traefik Mesh. Main page. https://traefik.io/traefik-mesh/.

[32] Microsoft. Sidecar pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/sidecar.

[33] Jacyana Suassuna Nunes, Silvio Costa Sampaio, Ramon Santos Malaquias, and Itamir de Morais Barroca Filho. Deploying the observability of the sigsaude system using service mesh. In *2020 20th International Conference on Computational Science and Its Applications (ICCSA)*, 2020.

[34] Chris Richardson. Pattern: Service mesh. https://microservices.io/patterns/deployment/service-mesh.html.

[35] Chris Richardson. *Microservices patterns: with examples in Java*. Manning Publications Co., Shelter Island, NY, October 2018.

[36] Chris Richardson. A pattern language for microservices, 2021.

[37] Peter Rodgers. Service-oriented development on netkernel- patterns, processes & products to reduce system complexity. https://web.archive.org/web/20180520124343/http://www.cloudcomputingexpo.com/node/80883.

[38] Tiago Boldt Sousa, Filipe Figueiredo Correia, and Hugo Sereno Ferreira. Patterns for software orchestration on the cloud. In *Proceedings of the 22nd Conference on Pattern Languages of Programs*, PLoP '15, USA, 2015. The Hillside Group.

[39] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. Overview of a pattern language for engineering software for the cloud. In *Proceedings of the 25th Conference on Pattern Languages of Programs*, PLoP '18, USA, 2018. The Hillside Group.

[40] Tiago Boldt Sousa, Hugo Sereno Ferreira, and Filipe Figueiredo Correia. A survey on the adoption of patterns for engineering software for the cloud. *IEEE Transactions on Software Engineering*, 48(6):2128–2140, 2022.

[41] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Messaging systems and logging. In *Proceedings of the 22nd European Conference on Pattern Languages of Programs*, EuroPLoP '17, New York, NY, USA, 2017. Association for Computing Machinery.

[42] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: Automated recovery and scheduler. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Tiago Boldt Sousa, Hugo Sereno Ferreira, Filipe Figueiredo Correia, and Ademar Aguiar. Engineering software for the cloud: External monitoring and failure injection. In *Proceedings of the 23rd European Conference on Pattern Languages of Programs*, EuroPLoP '18, New York, NY, USA, 2018. Association for Computing Machinery.

[44] Statista. Usage of microservices within organizations. https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/.

[45] Yuchuan Tian, Haitian Hao, Xiaojian Xu, and Bing Liang. Research on enterprise service governance based on service mesh. *Journal of Physics: Conference Series*, 1673(1):012003, 11 2020.

[46] Guilherme Vale, Filipe Figueiredo Correia, Eduardo Martins Guerra, Thatiane de Oliveira Rosa, Jonas Fritzsch, and Justus Bogner. Designing microservice systems using patterns: An empirical study on quality trade-offs. In *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, pages 69–79, 2022.

[47] Linux VServer. Welcome to linux-vserver. http://www.linux-vserver.org/Welcome_to_Linux-VServer.org.