NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

RODRIGO CANELAS FÉLIX

Bachelor Degree in Computer Science and Engineering

METHODOLOGY FOR RECOMMENDATION OF MICROSERVICES ARCHITECTURES

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

METHODOLOGY FOR RECOMMENDATION OF MICROSERVICES ARCHITECTURES

RODRIGO CANELAS FÉLIX

Bachelor Degree in Computer Science and Engineering

Adviser: João Filipe Meneses Henriques
Technical Leader, Link Consulting

Co-adviser: Carla Maria Gonçalves Ferreira
Associate Professor, NOVA University Lisbon

Examination Committee

Chair: Jörg Matthias Knorr
Associate Professor, FCT-NOVA

Rapporteur: David Fernandes Semedo
Associate Professor, FCT-NOVA

Adviser: João Filipe Meneses Henriques
Technical Leader, Link Consulting

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2023

Methodology for Recommendation of Microservices Architectures

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.15) [34].

I dedicate this to my sister, who has always been my role model.

Acknowledgements

I'd like to express my sincere gratitude to my dissertation adviser, João Filipe Meneses Henriques. João, your support and availability during this thesis have been invaluable, I really appreciate your guidance and willingness to help, even when you were traveling. I also want to extend my heartfelt thanks to my co-adviser, Assistant Professor Carla Ferreira. Your guidance and honest feedback were truly exceptional, and I'm truly grateful for your support.

I'd like to acknowledge the NOVA School of Science and Technology, particularly the Department of Computer Science, for providing the resources and knowledge that enabled me to complete this dissertation. Special thanks to Link Consulting for giving me the opportunity to work on a thesis in an area I'm passionate about, with an adviser I highly respect.

I owe a great deal of thanks to my family. To my mother, your unconditional love and support allowed me to focus on my studies, I'm sorry I wasn't more present. To my sister, thank you for your advice, fun times and not-so-fun times, for being present when I couldn't, and for always having my back. To my girlfriend, thank you for your unwavering belief in me, for your continuous support, for traveling hours to be with me, for always putting up with me, thank you for every single moment we shared these last five years. To my "borrowed family", thank you for helping when I couldn't, thank you for all the smiles, the good environment, and for always making things easier.

A special mention goes to my dog, thank you for not letting me go nuts. Thank you for your companionship, for forcing me to take breaks, for all your hugs and love. Thank you for being my body pillow, my feet warmer, my vacuum cleaner. Without you, this journey wouldn't have been the same. To my friends from FCT, a huge thanks for putting up with me all these five years, I know it wasn't easy. Thank you for all the fun times and all the parties, but especially thank you for all the sad times when you were there to support me, all the nights studying, and even all the nights crying.

Abstract

Software architecture is a crucial element of software development, as it defines the elements and relationships of a system, guiding the development team and managing complexity as the system grows.

The increasing popularity of microservices architecture has brought a new challenge for developers in choosing the best microservices architecture for their specific needs. This kind of architecture involves breaking down a system into independent services, each focused on a specific business capability, establishing communication through lightweight mechanisms.

Choosing the right microservices architecture presents an interesting challenge because the architecture must consider various factors such as scalability, reliability, security, performance, and maintainability. This requires a detailed examination of the existing literature and best practices, as well as a practical analysis of real-world case studies.

To address this challenge, this thesis developed a methodology for determining a suitable microservices architecture based on specific requirements.

The methodology includes various requirements that were used to evaluate several architectural design alternatives. To improve its automation and make it easier to use, a recommendation system, that uses the methodology, was implemented in order to automatically consider user preferences and suggest the architecture.

The contribution of this thesis is an innovative solution that assists in determining an appropriate microservices architecture based on specific requirements. The developed solution does not only recommend a suitable architecture and its relevant components such as security and observability, but also provides a dynamically generated graphical overview of the resulting architecture.

Keywords: Methodology, Software Architecture, Microservices Architecture, Recommendation System, Requirements, Architectural Design Alternatives

Resumo

No desenvolvimento de software uma das componentes mais importantes é a arquitetura de software. Esta define os elementos e as relações de um sistema, orienta a equipa de desenvolvimento e ajuda a controlar a complexidade do sistema à medida que este vai progredindo.

A crescente popularidade dos microsserviços trouxe um novo desafio para os desenvolvedores de software na escolha da melhor arquitetura de microsserviços para as suas necessidades específicas. Este tipo de arquitetura envolve a divisão de um sistema em serviços independentes, cada um focado numa funcionalidade específica, estabelecendo comunicação através de mecanismos leves.

Escolher a arquitetura de microsserviços mais apropriada apresenta um desafio interessante, pois a arquitetura deve considerar vários fatores como escalabilidade, confiabilidade, segurança, desempenho e manutenibilidade. Esta escolha requer uma examinação da literatura existente e das melhores práticas, bem como uma análise prática de casos de estudo.

Para enfrentar este desafio, esta tese desenvolveu uma metodologia para determinar uma arquitetura de microsserviços adequada com base em requisitos específicos.

A metodologia inclui um conjunto de requisitos que foram utilizados para avaliar diferentes alternativas de design de arquiteturas. Posteriormente, para melhorar a sua automação e tornar a sua utilização mais simples, foi implementado um sistema de recomendação que, através do uso da metodologia, tem em consideração as preferências do utilizador e sugere a arquitetura.

A contribuição desta tese é uma solução inovadora que ajuda a determinar uma arquitetura de microsserviços apropriada com base nos seus requisitos específicos. O sistema de recomendação não recomenda apenas uma arquitetura adequada e os seus componentes relevantes, como segurança e observabilidade, mas também gera dinamicamente uma visualização gráfica da arquitetura final.

Palavras-chave: Metodologia, Arquitetura de Software, Arquitetura de Microserviços, Sistema de Recomendação, Requisitos, Alternativas de Design de Arquiteturas

Contents

List of Figures

Acronyms

ABAS Attribute-Based Architectural Styles *(p. 23)*

ATAM Architecture Tradeoff Analysis Method *(pp. vii, x, 23, 24, 58)*

GUI Graphical User Interface *(pp. 52, 54, 55, 58)*

JVM Java Virtual Machines *(p. 25)*

MVC Model-View-Controller *(pp. 9, 10)*

OCI Open Container Initiative *(p. 24)*

RSSE Recommendation Systems for Software Engineering *(p. 20)*

SAAM Software Architecture Analysis Method *(pp. vii, x, 22–24, 58)*

SOA Service-Oriented Architecture *(pp. 6, 8, 17)*

STS Secure Token Service *(p. 18)*

1 Introduction

1.1 Context

Developing a system without proper architectural planning is unacceptable nowadays, and this is especially true for microservices architectures. With this in mind, Link Consulting has proposed this thesis, which intends to offer guidelines for designing a microservices architecture.

Software architecture provides an organizational structuring of software systems [21], as it defines their elements and their relations [6]. It also provides a blueprint for the development team, guiding their decisions and helping them to manage complexity as the system grows. There are various architectural patterns and styles [44], each with its own strengths and weaknesses. The choice of architecture should be based on the specific needs and constraints of the system being built. Microservices architecture is one such pattern that has become increasingly popular in recent years, particularly in large and complex systems.

Microservices architecture is a modern software design pattern that structures an application as a collection of small, independent services where each service is designed to handle a specific business capability [14] and the services interact with each other using lightweight communication protocols. The goal of this architecture is to create a scalable, flexible, and efficient system that can easily adapt to changing business requirements.

Building a microservices architecture involves several key steps. First, the system must be broken down into a set of distinct services that represent the various functionalities (or business capabilities according to Fowler [14]), where each service should have a clear and focused scope. Then, the data and the communication between services must be established. This includes defining the type of communication [35], the data considerations [36] and protocols that will be used, as well as the security and authentication mechanisms to be used if needed. It is also important to consider what kind of infrastructure should be used to support all the needed features of the architecture. This includes decisions around deployment, management, and monitoring of services. Lastly, it is essential to have a solid plan for testing and maintaining the system, Fowler [14] recommends the "Design

for failure" approach. This may include setting up automated testing and continuous integration/continuous deployment processes, as well as monitoring and logging systems to help identify and resolve issues.

1.2 Motivation

With the increasing popularity of microservices architectures, developers started to face a new challenge: how to choose the best microservices architecture for their specific needs. With so many possibilities and characteristics to consider, it can be difficult to make an informed decision. The decision must consider various factors such as scalability, reliability and security, because they directly influence how well an application can adapt, perform, and endure in real-world scenarios.

To address this challenge, this thesis provided a methodology for choosing an adequate microservices architecture. The methodology takes into account the specific needs and constraints given by the developers and provides a suggestion based on the evaluation of the various options and making an informed decision.

The development of this thesis involved a thorough examination of existing literature and established best practices in the field of microservices architecture. The methodology was inspired by a review of the literature and commonly accepted best practices of microservices architecture, as well as a practical analysis of real-world case studies. Additionally, it benefits from the help of a recommendation system to automate its usage and provide some additional features.

1.3 Solution

This thesis's purpose was to develop a methodology to determine a suitable microservices architecture based on a set of requirements and recommend it to the user. The methodology includes a set of possible design alternatives for the system's architecture that helped to identify requirements related to each alternative.

To fulfill this thesis goal, we created matrices that classify the different design alternatives based on their level of support for each specific requirement. For example, in the matrices related to decisions concerning data, there is one in particular that focuses on the decision of whether to use one database per microservice or a common database across all microservices. This specific matrix takes into consideration requirements such as the need for isolated data schemas and the need for clear data ownership. The end goal of these matrices was to compare the requirements with the architectural design alternatives and provide the information needed as input for the methodology.

A recommendation system was developed as a complementary part of this methodology to automate the process of collecting user input and performing the calculations with the matrices. The recommendation system ended up not only deciding a suitable architecture (according to microservices orchestration), pointing out the relevant components

that the elements of the architecture will need to consider, but also providing a graphical overview of the architecture as a whole.

1.4 Contributions

This thesis made the following contributions:

Firstly, it developed a methodology for the evaluation and recommendation of microservices architecture design alternatives. This methodology covers a broad spectrum of design choices, providing users with a versatile decision-making framework.

Secondly, an innovative feature of this research is the introduction of evaluation matrices. These matrices served as a tool to correlate the requirements with the design alternatives, which provided a clear and organized way to assess the different available options and determine the best fit for a given context.

Thirdly, a user-friendly recommendation system has been crafted to use the methodology. It offers diverse interaction modes and stores the results locally, promoting transparency and adaptability to users' preferences.

Lastly, a dynamic graph visualization tool has been created, allowing the selected alternatives to be visualized. This visual approach helps to improve users' understanding of the connections and dependencies among architectural elements.

1.5 Document Structure

This thesis is organized into five main chapters, each serving a distinct purpose and contributing to the understanding of the subject matter:

- **Chapter 2** provides an overview of the main concepts of microservices, as well as a review of the background and related literature on the subject;
- **Chapter 3** explores the solution, describing the step-by-step process taken in its development, emphasizing all the critical processes that played a key role;
- **Chapter 4** offers a detailed overview of the solution's implementation. It highlights the steps undertaken to bring the solution from concept to reality;
- **Chapter 5** engages in an evaluation of the methodology, providing insights into the results and user feedback;
- **Chapter 6** encapsulates the accomplishments and contributions made throughout this thesis. It also offers a brief look into the future prospects and potential work that remains to be accomplished;

2

Background and Related Work

In this chapter, we will look over the main concepts and related literature on software architectures, microservices, and recommendation systems, to gain a deeper understanding of these fields.

We begin by exploring the fundamental principles of software architectures, where we'll examine various architectural patterns and their key features. Moving on, we turn our attention to microservices, covering a range of definitions and best practices for their orchestration. Our exploration extends to recommendation systems, where we will analyze their concepts and how can they aid developers. Next, we will study software architecture analysis, where we discuss its core ideology, as well as two key approaches. Moreover, we'll assess some technologies that could complement our thesis, namely some tools that could be used to develop and deploy the suggested microservices architecture.

To conclude, we will take a look at the technological decisions addressed in our thesis, specifically most of the considerations that were taken into account in our methodology.

2.1 Software Architectures

Since the purpose of this thesis was to build a methodology to recommend an architecture, we studied various aspects of software architectures and examined common design patterns and alternatives that proved themselves useful to guide our recommendations.

Software architecture is a crucial aspect of the development of large and complex systems as it provides the framework for organizing and structuring software components. Documenting an architecture effectively is just as vital as creating it, since that, an unclear or misconstrued architecture will fail to fulfill its purpose as a unified vision for system and software development [6].

Garlan and Shaw [15], defined a computing system's software architecture as the set of structures that allows for understanding the components, connections, and characteristics of the software within the system. To better understand what is software architecture perhaps is easier to reason about what software architects do. According to Krutchen [30], software architects are responsible for creating, fostering, and preserving the architecture

of software-intensive systems in which they are involved.

2.1.1 Architecture Patterns

We will examine some of the most widely recognized architectural patterns and their features, as described by Richards [44]. According to Richards, these patterns play a main role in shaping the fundamental features and behavior of an application.

Understanding architectural patterns is the key to making informed decisions about the design and development of software applications. Throughout this chapter, we will focus specifically on the microservices architecture pattern, as it holds the greatest relevance and significance to this thesis.

Layered Architecture

The usage of layered structures in software architecture has been a widely adopted pattern since its early development. Even today, it remains a common architectural style used across various industries [49].

The layered architecture pattern [44], also known as the n-tier architecture pattern, is a commonly used architecture pattern in Java Enterprise applications, often used by architects, designers, and developers. The pattern aligns with traditional IT communication structures in companies, this makes it a favored approach when developing business applications. In a layered architecture, components are grouped into horizontal layers that serve specific functions within the application. The exact number of layers is not prescribed, but the majority of architectures feature four: presentation, business, persistence, and database. However, there may be variations depending on the size of the application. Each layer has a distinct role and responsibility. For instance, the presentation layer handles the user interface and browser communication, while the business layer executes business logic. Layers form an abstraction of the work needed to satisfy a business request, so the layers don't have to be aware of one another.

The layered architecture has the benefit of separating component concerns. Each component within a layer only handles tasks related to that specific layer. This allows for a clear definition of roles and responsibilities, as well as simplified maintenance with defined interfaces and limited component scopes.

Event-Driven Architecture

The event-driven architecture [44] is a scalable, and adaptable pattern for both small and complex distributed applications. It is made up of decoupled components that process events asynchronously. Within this architectural pattern, there are two principal topologies: the mediator and the broker.

Mediator topology is a pattern in which an event mediator orchestrates multiple steps in a single event to process it. The components of this topology are a mediator, queues,

channels, and processors. The flow starts with a client sending an event to a queue and then to the mediator which then sends out processing events to channels that are received by processors. The processors contain the business logic to process the event.

In broker topology, the message flow is distributed across event processor components through a lightweight message broker. There are two main components: a broker component which contains event channels and event processor components. Every single event processor component is tasked with handling a specific event and then publishing a new subsequent event (or not). The broker topology is like a relay race where each event processor hands off the event once it has completed its processing. There is no central event mediator in this topology.

The event-driven pattern has several advantages [46], one of which is the ability to maintain data consistency across multiple services without relying on the use of complex distributed transactions. However, it's important to note that the programming model involved in this pattern can be more complex compared to other solutions.

Microkernel Architecture

The Microkernel Architecture pattern [44], also known as the plug-in architecture pattern, is a design approach for creating product-based applications. This pattern involves dividing the application logic into a main system and independent modules, resulting in a flexible and extensible design. The main system contains only the essential functionality, while the remaining modules add additional features.

This approach allows for feature separation and isolation. Many operating systems use this pattern, which is why it is referred to as the microkernel architecture. In business applications, the main system typically includes general business logic, while special cases and complex processing are handled by the modules.

Microservices Architecture

Microservices Architecture [44] is a popular alternative to other commonly used architectures such as monolithic and Service-Oriented Architecture (SOA), gaining widespread adoption. This pattern involves deploying individual components, often called microservices, as independent separate units for easier deployment and increased scalability.

In a microservices architecture, the central idea revolves around the notion of small decoupled service components that can communicate with each other through lightweight communication mechanisms [14]. A key challenge in this architecture lies in figuring out the appropriate way to orchestrate these services.

We will dive more into microservices and microservices architecture in section 2.2.

Space-Based Architecture

The space-based architecture [44], often referred to as the cloud architecture pattern, is a solution that improves scalability and concurrency in high-volume web-based business

applications. It gets rid of the dependence on a central database by adopting in-memory data grids and replicating the data across various processing units. This architecture consists of two components: processing units and virtualized middleware. Processing units contain application components, an in-memory data grid, and a replication engine, while the virtual middleware manages tasks related to maintenance and communications with components for data synchronization and request handling.

This pattern provides near-infinite scalability [44] inside the application and addresses variable scalability by being able to dynamically start-up or shut-down processing units as user load changes.

2.2 Microservices

In this section, we will immerse ourselves in the topic of microservices. Our focus will be on understanding the basics of what is a microservices architecture, as well as its various components and concepts. By exploring this topic, we gained a knowledge base that served as a foundation for the goal of this thesis.

Before diving into the technical definitions of what microservice architecture is, it is important to understand the background and evolution of this architectural style in recent years. In the past, the most commonly used architecture for server-side applications was known as monolithic architecture. This name is derived from the fact that all the application's logic, including the user interface, business logic, and data access, is encapsulated within a single codebase [19].

This approach to software development was popular because many of the languages used, such as Java, C/C++, and Python, provided their own abstractions for breaking down the complexity of programs into modules. These abstractions relied on the sharing of resources of the same machine [8], making it easy for developers to work with. This traditional approach to software development was widely adopted by large companies such as Amazon and eBay, as it provided ease of development, testing and deployment. However, as applications grew in complexity and size, the monolithic structure also grew, becoming a large and hard-to-manage piece of software [32]. The monolithic structure posed challenges in terms of scalability, maintainability, and flexibility, which led to the development of the microservice architecture as an alternative. In a microservice architecture, the application is composed of small, loosely coupled (not strongly connected) services, each with its own specific function [8]. These services (often called microservices) are then deployed independently of one another, allowing for greater flexibility and scalability in terms of development, testing and deployment. This architectural style is increasingly being adopted by companies as a way to overcome the challenges posed by monolithic architectures and to better adapt to the fast-paced and ever-changing demands of modern software development.

Nowadays the main approach when deploying microservices is through the usage of containers. Containers are a lightweight, portable way to package software, and

this technology is explained in greater detail in Containers. They are mainly used in microservices because teams can easily manage the dependencies and configurations of each microservice, ensuring that they will run consistently in any environment.

According to Fowler [14], there is no exact definition of a microservice, but there are certain characteristics that are often associated with them. These include decentralized control of languages and data, and a design that promotes flexibility, modularity and evolution of the system. Even if there is no precise definition, we can end up viewing microservices as a software architectural style that structures an application as a collection of small, independent services that can communicate with each other. These services are designed to be independently deployable, scalable, and maintainable [8], and they are typically developed and managed by small, cross-functional teams [14]. Each microservice should have a single responsibility in the sense that it does only one thing and one thing only [55], making them easier to understand.

There is a debate among experts in the field of software architecture about the nature of microservices. Some argue that microservices represent a distinct and unique architectural style, separate from other existing approaches. On the other hand, there are those who argue that microservices are simply an implementation approach to SOA. They contend that many SOA design patterns and recommended approaches can be discovered in the microservices literature, for example, the API Gateway Pattern has similar goals and motivations as those found in SOA-style ESBs [60]. Overall, both sides have slightly different points of view on what microservices are. Advocates of microservices argue that it is a new architectural style, while those in favor of SOA argue that microservices are simply a way to implement SOA [60].

2.2.1 Advantages of Microservices

As we moved towards making an architectural recommendation, it became important to understand why microservices stood out as our chosen option. Through examining the benefits that microservices offer, we hope to demonstrate why a microservices approach may be the best choice among available alternatives.

Microservices have become a popular trend in software development in recent years because they offer several advantages over traditional monolithic architecture [19] and for many, microservice architecture is becoming the default style for building enterprise applications [14]. One major benefit of microservices is that they can be independently deployed, meaning that only the updated portion of the application needs to be redeployed rather than the entire application. This benefit allows for the use of automated deployment techniques and faster deployment times (deployments can be made in seconds [8]).

Monolithic structures, even if designed with good modularity, often become difficult to maintain over time. As a result, changes intended to affect only a single module may end up impacting other parts of the system. To alleviate the difficulties of maintaining monolithic structures, microservice architectures define clear boundaries between each service. This

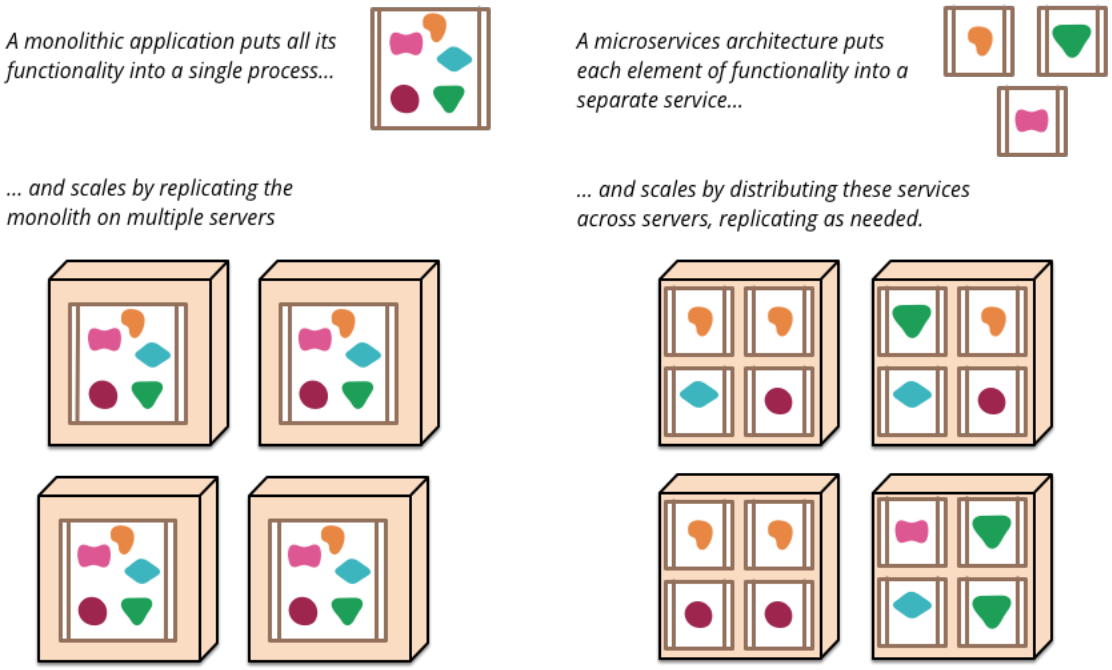


Figure 2.1: Scaling in Monoliths vs in Microservices (taken from [14])

helps to contain changes and minimize their impact on other parts of the system [14]. When it comes to scaling a system, monolithic architecture can be scaled horizontally by increasing the number of instances of the entire application and distributing the load across them using a load balancer [14].

On the other hand, the microservices architecture allows for a more fine-grained approach to scaling. Instead of scaling the entire application, individual microservices can be scaled horizontally, increasing the number of instances only for the specific services that require more resources (see figure 2.1). This allows for greater flexibility and efficient use of resources in a microservices-based system.

2.2.2 Design Patterns

In this section, we will discuss various design patterns that are commonly used in building software systems. We will cover patterns such as the Model-View-Controller (MVC) pattern, the "smart endpoints, dumb pipes" pattern, the Backends for Frontends pattern, the API Gateway pattern, and the Event-Driven pattern. These design patterns are widely adopted and we have found them to be very useful in real-world scenarios. We will examine the key concepts behind these patterns, as many of them served as a source of inspiration for the design alternatives that we will present in our recommended architecture.

Model-View-Controller

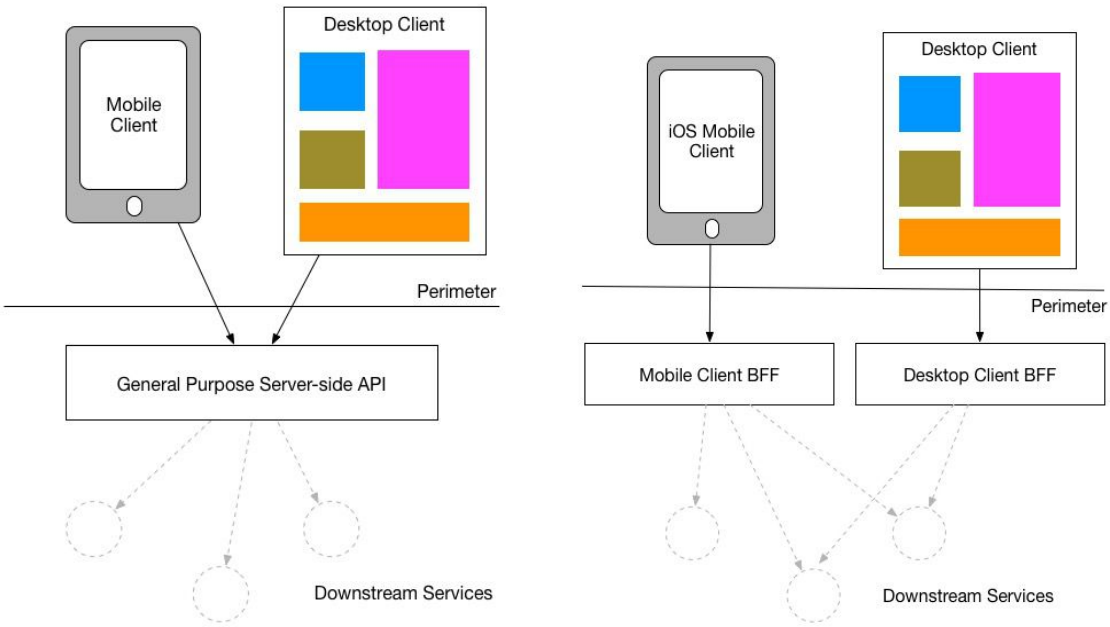
One common pattern when designing software is the MVC pattern. This pattern, which is widely used in traditional monolithic web applications, can also be applied to microservices. In this pattern, the model object encapsulates the information of the system, the view displays information to the user, and the controller implements actions [4].

Smart endpoints, dumb pipes

A pattern that is often used in microservices architecture is the "smart endpoints, dumb pipes" pattern. In this pattern, the services are designed to be as decoupled and as cohesive as possible [14]. Decoupled in the sense that they are not tightly connected nor dependent on each other, each service handles its own domain logic and cohesive in the sense that they should work together smoothly and naturally, they should integrate together in perfect harmony. The communication between services is then handled by simple RESTish protocols [14]. This pattern helps to minimize the complexity of inter-service communication and to ensure that each service can be developed, deployed, and scaled independently.

Backends for Frontends

Another pattern is Backends for Frontends which is a pattern that allows having different backends for different frontends. This pattern is useful when different types of clients or devices need to consume different subsets of an API.



(a) Backend with a single API

(b) One backend per user interface

Figure 2.2: Backend architecture comparison (adapted from [39])

Newman, [39] stated that to support multiple types of UI, it is common to start with a single, server-side API and gradually add more functionality as needed (figure 2.2a). Even so, mobile experiences often differ greatly from desktop web experiences, requiring different calls and displaying different data. To address this, the Backends for Frontends pattern can be used, where instead of a general-purpose API backend, there is a separate backend for each user experience (figure 2.2b). The client-side application is considered as two parts, an external client-side component, and a server-side component (that is the Backend for Frontend) inside the network. The Backend for Frontend is closely connected to a specific user experience and is usually managed by the same team as the user interface. This allows for a more efficient way to define and adjust the API as needed by the UI and it also simplifies the process of coordinating the release of both the client and server components [39].

API Gateway

Another very useful pattern for when different types of clients or devices need to consume different subsets of an API is the API Gateway pattern [45]. This pattern provides a single entry point (called API Gateway) for all external requests to the system and then proxies/routes the requests to the appropriate backend service based on the client's needs and requirements [45]. An API Gateway can also handle other traffic management tasks such as rate limiting and caching, and can even implement features to improve the system like security and Governance [27]. It may also be used as a layer between the client and the backend services, allowing for changes to the backend services to be made without affecting the client. This is especially useful for implementing microservices architectures, where different services are deployed independently and may have different versions.

Overall, the API Gateway pattern can help to improve the scalability, security, and maintainability of a system while also simplifying the process of routing requests to the appropriate backend service.

Event-Driven

The Event-Driven pattern is a design pattern that uses events to trigger actions and communicate between the services rather than making direct function calls, it is a common pattern in modern microservices applications [2]. This allows for a more decoupled and asynchronous communication between services, making the system more flexible and scalable, thus making it perfect for microservices.

One of the key benefits of the Event-Driven pattern is its ability to maintain data consistency across multiple services without using distributed transactions [46] and high levels of concurrency. Instead of having services constantly polling for changes, events are sent as soon as they occur, allowing for faster processing and reaction times. Additionally, this pattern allows for the easy integration of new services and components into the

system, as they can simply subscribe to the events they are interested in. A drawback is that the structure of the code and the logic of the system can be more complex.

2.3 Good Practices for Microservices Orchestraction

In this section, we will examine some good practices that are employed when building a microservices architecture. Exploring and understanding these practices was decisive in fulfilling the goal of this thesis, they laid the groundwork for the construction of our matrices. They helped lift alternatives for the evaluation matrices, and they were also particularly helpful in understanding the dependencies between alternatives used in the Dependency Matrices.

2.3.1 Dealing With Data in Microservices Architectures

When discussing data management in a microservices architecture, one question that may arise is whether or not to use a single, shared database. Nevertheless, it is generally considered best practice to avoid this approach in this architecture. As stated in [36], one of the foundational tenets of microservices is the idea of keeping data storage separate and isolated for each service. This means that each service has its own dedicated data store that cannot be directly accessed by other services.

Furthermore, there are several downsides to sharing a single database between microservices, as outlined in [51]. These include:

- **Limitations in scalability:** classical databases often only scale vertically, with an upper limit on the number of clients that can be served at the same time;
- **Potential availability issues:** classical databases may not be fully available, due to scheduled downtimes;
- **Mismatches between the database model and the needs of individual services:** the data model of the database may not fit the use cases of the microservices;
- Confusion over data ownership.

Dealing with data in microservices is a complex task. First, it is important to remember that each microservice should have a single responsibility. Thus each microservice should be responsible for a specific set of data, meaning that data should be divided among the different microservices based on their responsibilities. For example, a microservice that handles information about the users should only be responsible for storing and retrieving user data, and not data related to other entities.

According to Microsoft [36], the use of separate data stores for each service has several benefits, including:

- **Isolation of data schemas:** each service has its own private data store, avoiding unintentional coupling between services;
- **Limitation of the scope of change:** by separating each service's data, changes to the data schema need only to be coordinated within the service, rather than across all services that rely on the database;
- **Preservation of agility:** isolation of data stores allows for truly independent deployments;
- **Optimization of Data Storage:** a key benefit of having each microservice manage its own private data store is the ability to optimize data storage specific to the needs of that service. The data models, queries, or read/write patterns of each service would have less performance if a shared data store were used.

By using separate data stores for each service, we are leaning toward a technique called polyglot persistence [36]. This technique is described by Fowler as having "a variety of different data storage technologies for different kinds of data" [13]. Once the data has been divided among the microservices, it is important to ensure that the data is properly replicated. Replication not only increases reliability and improves performance but it allows for fail-overs [51]. This will ensure that the data is always available, even in the event of a failure.

Challenges for Data Management

Managing data in a distributed environment is often a demanding task.

A known challenge is having multiple copies of the same data scattered across different services [36]. For example, in an e-commerce platform, customer information may be stored in a service responsible for handling transactions, but it may also be replicated in another service that generates customer analytics. This can result in problems regarding data integrity and consistency. For instance, if the customer's address is updated in the transaction service, it may take some time for that change to propagate to the analytics service, which can create discrepancies in the customer data.

In traditional data modeling, it is a common practice to adhere to the principle of "one fact in one place" [36]. This means that each entity is represented only once in the schema, and other entities reference it instead of duplicating it. The "one fact in one place" principle simplifies updates and ensures data consistency. Yet, in a microservices architecture, data is stored in multiple locations, requiring consideration for the distribution of updates and management of consistency.

Approaches for Data Management

When managing data, every organization and situation is unique, so there is not a one-size-fits-all approach, it's impossible to provide a single method or strategy that can guarantee

success. Instead, there are some general guidelines made by Microsoft [36], that one should try to follow as much as possible:

- **Use eventual consistency when possible:** In some cases, strong consistency may be required, while in others, eventual consistency is sufficient. Determine the minimum level of consistency required for each part of your system;
- **Adopt strategies to maintain consistency of data across various services:** Ensure data remains consistent when multiple services are involved in a transaction with patterns like Compensating Transactions or Scheduler Agent Supervisor;
- **Limit data storage to what is necessary:** Each service should only store the data it needs to perform its specific functions. Avoid storing unnecessary information;
- **Ensure services are cohesive and loosely coupled:** Services that are closely connected and rely on each other heavily, consider either merging or refactoring them to reduce dependence;
- **Utilize an event-driven architecture:** Services can publish events when data changes, allowing other services to subscribe and react accordingly;
- **Use schemas to prevent tight coupling (JSON, Protobuf, etc.):** To prevent close interdependence between services, events that are published should be accompanied by a schema for serializing and deserializing purposes;
- **Scale events carefully:** To mitigate potential bottlenecks caused by a large volume of events, consider utilizing aggregation or batching as a means to ease the load.

2.3.2 Communication in Microservices Architectures

In a microservice architecture, communication between services is critical to the system's design. According to Fowler [14], Dragoni [8] and Microsoft [35], the communication between the services should be made using lightweight mechanisms, and the most popular protocols are asynchronous messaging used to propagate updates between microservices and HTTP resource API (mostly for querying).

In a monolithic application, different components communicate with each other by utilizing language-level methods or function calls within the same process. This communication can be tightly bound if objects are created using code, or can be loosely coupled by using dependency injection, where abstractions are referenced instead of concrete object instances [35]. In a microservice architecture, these language-level methods or function calls may still be used, but the main way to divide the software is by breaking it down into services and using communication mechanisms such as web service requests or remote procedure calls [14].

The biggest challenge in migrating to a microservices architecture is changing the way components communicate. Converting method calls to remote procedure calls can lead to

excessive and inefficient communication, which is not suitable for distributed systems [35]. When transitioning to microservices [35], one effective strategy is to maximize the isolation of business microservices. Asynchronous communication between internal microservices should be utilized, and fine-grained communication commonly used in intra-process communication between objects should be replaced with coarser-grained communication.

Fine-grained communication refers to a type of communication that involves exchanging small amounts of data frequently while coarser-grained communication, refers to communication that involves exchanging larger amounts of data less frequently. This replacement can be accomplished by grouping calls and returning aggregated results from multiple internal calls to the client.

The microservices community [14] encourages the usage of the previously mentioned pattern of smart endpoints and dumb pipes since this pattern aims to keep microservices as decoupled as possible but also as cohesive as possible within a single microservice.

Types of Communication

In a microservice architecture, the type of communication between client and server can vary depending on the scenario, such as whether it should be synchronous or asynchronous and if there is one receiver or multiple receivers [35]. Synchronous communication with a single client is common and can be done through a protocol like HTTP/HTTPS, but other options such as an asynchronous protocol like AMQP or a publish/subscribe for multiple receivers in an event-driven architecture may also be used.

Microsoft states that [35] to build a microservices application, the way in which the microservices are integrated is the most decisive part. The goal should be to reduce the amount of communication between the microservices, as the fewer interactions there are, the better. Naturally, some level of integration will still be required, as they need to work together. When it comes to communication between microservices, the critical rule is that it should be done asynchronously [35]. Synchronous communication between multiple microservices should be avoided whenever possible, even for queries. This is because if a call from one microservice to another is required to respond to a client application, the architecture may not be as resilient as it should be. The negative impact of synchronous communication in microservices can be significant. One of the main drawbacks is the creation of HTTP dependencies between microservices [35]. When HTTP request chains create extended request/response cycles, it not only undermines the autonomy of microservices, but their performance can also be negatively impacted if one of the links in the chain is not operating efficiently.

Furthermore, if multiple microservices rely on one another through synchronous dependencies such as query requests, the performance of client applications can be severely impacted, which can lead to a poor user experience.

2.3.3 Observability

According to my experience on the field, when it comes to building a successful microservices architecture, one important factor that a team must consider is its observability since it allows the identification and resolution of issues before they affect the system's performance. Observability has its roots in control system theory and according to Gopal [17], it measures how much one can infer from a system's internal state given its output.

In the context of cloud environments, the concept of observability retains its core definition, but it encompasses additional aspects. Specifically, it not only focuses on the extent to which one can infer about the system but also on the degree to which one can visualize it. The term is defined in [41] as "the ability to visualize and comprehend the behavior of a system by gathering, processing system data and presenting it in a suitable manner for the different type of users".

Logs, metrics, and tracing are considered the most crucial outputs (according to Gopal's terminology) for the observability of a distributed system, they are known as the three pillars of observability.

Next, we will be discussing three key techniques commonly used to increase observability in a system: monitoring, tracing and auditing.

Monitoring

According to IEEE, a software monitor is a tool that offers insight into the execution of another program executing concurrently [21]. IBM states that monitoring is used to measure the health of an application, such as creating an alert when disk usage is nearing capacity to prevent downtime. It is particularly useful for identifying long-term trends and alerting teams to potential issues [9].

The problem with monitoring is that it is limited in the sense that it can only detect known problems and requires knowledge of what metrics and logs to track.

Tracing

A trace is defined as a documented account of the execution of a computer program, including the sequence of executed instructions, and the names and values of variables [21].

In the world of distributed systems, distributed tracing, also called distributed request tracing, is an important tool for gaining observability. It is a type of correlated logging that helps to understand the behavior of individual services in a larger, interconnected system. By providing a clear picture of the job of each service, distributed tracing enables the monitoring of the performance of the services and the whole distributed system [40].

Auditing

Auditing refers to the process of examining a work product or set of work products to verify conformance with established specifications, standards, contractual agreements,

or other criteria [21]. In distributed environments, it is considered a valuable tool for gaining insights into the behavior of software components. For example, when testing new applications, a single execution trace can be analyzed to ensure that it meets the required specifications and standards [38].

Auditing is not only essential for quality assurance and risk management, but it can also be a legal requirement or a client requirement to ensure the protection of sensitive information and data privacy. It provides a means to assess the compliance of operations, identify potential issues, and ensure that standards are being followed.

2.3.4 Security

When addressing a microservices architecture, it is very important that security isn't taken lightly. In this section, we will explore some security concerns associated with microservices and discuss various aspects of securing microservices-based applications.

2.3.4.1 Attack Surface Expansion

As mentioned in section 2.2, microservices decompose applications into small, independent services that communicate over networks. Anelis et al. points out that this decomposition increases the attack surface compared to monolithic applications, as each microservice can potentially become a point of vulnerability [57].

2.3.4.2 Trust and Security Challenges

Microservices architecture presents significant trust and security challenges, inherited from SOA and distributed computation [7]. With its various components all interconnected and responsible for different aspects of the application, ensuring a cohesive and secure environment becomes of major importance. A breach in one microservice can potentially expose vulnerabilities in others, emphasizing the need for robust security measures.

2.3.4.3 Risk Assessment

Microservices often operate across multiple services or platforms, including public cloud environments, which introduces inherent security risks [7]. The complexity of development, challenges in monitoring, debugging, and the opacity of cloud service providers' infrastructures pose a significant "leap of faith".

2.3.4.4 Vendor Security and Cloud Services

Selecting a cloud service provider requires careful consideration of their security practices [7]. Regardless of the vendor's size or reputation, security should always be a top priority. Key questions should be raised regarding how vendors maintain trust and security within their services. This includes aspects such as authentication, data protection, and adherence to security best practices.

2.3.4.5 Authentication and Authorization Challenges

Microservices require robust authentication and authorization mechanisms and a decentralized microservices architecture presents challenges in managing authentication information. The question arises of where to store authentication data [57]: in a centralized authentication server or distributed across individual microservices. This decision impacts the maintenance and scalability of the authentication system. As Ahmadvand and Ibrahim [1] suggest, decomposition into microservices should respect security and scalability constraints.

2.3.5 Service Meshes

In Link Consulting, service meshes are included in most of the projects involving microservices, in this section, we will examine the main concepts related to them, clarifying their purpose and significance in the context of microservices.

2.3.5.1 The role of a Service Mesh

The rising popularity of microservices has brought forth a set of security challenges, as previously highlighted in section 2.3.4. These challenges primarily revolve around the intricacies of service-to-service communication. As Chandramouli aptly points out [5], these concerns stem from the inherently distributed nature of microservices. To address these security concerns, various solutions such as Secure Token Services Secure Token Service (STS), key management, encryption services for authentication and authorization and secure communication protocols come into the picture.

Moreover, microservices' ephemeral nature necessitates efficient service discovery mechanisms. Chandramouli [5] further suggests that "the service mesh is the best-known approach that can facilitate specification of these requirements".

2.3.5.2 Defining a Service Mesh

The definition of a service mesh is fairly straightforward. According to [52] a service mesh is essentially an infrastructure layer that consists of a collection of configurable proxies, which services can link to. Li [33] conceptually divides the service mesh's internal structure into two main components: the data plane and the control plane.

The data plane comprises a collection of intelligent proxies, typically deployed as sidecars alongside microservices. These proxies serve as intermediaries, controlling the communication between microservices. Their main functionalities encompass service discovery, health checking, routing, load balancing, authentication/authorization, and observability.

On the other hand, the control plane is tasked with managing and configuring these proxies to facilitate precise traffic routing and enforce security policies, by Li's words [33], "the control plane works as the brain of a service mesh".

2.3.5.3 Adoption of Service Meshes

Service meshes have assumed a pivotal role in the modern application landscape. Zhu et al. assert that service meshes are rapidly emerging as the default communication infrastructure for cloud-native applications [59]. This growing prominence is evidenced by a substantial increase in their in-production usage, with adoption rates surging by approximately 40-50% annually [12].

2.3.5.4 Downsides of Service Meshes

Despite their undeniable benefits, service meshes are not immune to certain drawbacks, with the most prominent being the introduction of overhead. The core architecture of service meshes mandates that all application traffic passes through sidecar proxies, which can inadvertently result in increased request latency and resource consumption. Zhu's investigation underscores the potential impact of this overhead. In certain scenarios, service meshes can introduce latency of several tens of milliseconds to individual requests, which may be unacceptable for latency-sensitive applications. Additionally, they can exert a substantial resource tax, consuming multiple virtual CPU cores even under moderate loads [59].

2.4 Recommendation Systems

With the objective of making better usage of our methodology, we decided to develop a recommendation system. In order to do this, an understanding of the fundamental concepts and theories of recommendation systems is crucial. Hence, in this section, we will examine the field of recommendation systems in detail to lay the foundation for our system.

A recommendation system is a tool that utilizes various computer science and engineering techniques to suggest items, information, or decisions to users based on their needs and preferences. The primary aim of these systems is to assist individuals in finding relevant information and making informed choices where they lack expertise or cannot consider all the available data [47]. The development of recommendation systems aimed to tackle two difficulties in their field that previous keyword-based information filtering systems could not [29]. Firstly, it addressed the issue of too many on-topic documents being picked with a keyword filter by means of human evaluation. Secondly, it tackled the challenge of filtering non-text documents considering only human preferences. These systems aid individuals in navigating a vast array of options by providing personalized suggestions for information, products, or people. These recommendations are built upon the system's understanding of one's preferences and current needs, as well as the aggregated preferences of the larger community of users [28].

Recommendation Systems for Software Engineering

Within the field of software engineering, Recommendation Systems for Software Engineering (RSSE) serve the same purpose as ordinary recommendation systems but with a specific focus on providing support to developers in their information-seeking activities related to code, APIs, and human expertise [47]. These systems have become essential in reducing the cognitive overload faced by developers and making software development more efficient and effective.

In recommendation systems architectures there are three key functionalities: data collection, recommendation engine, and user interface[47]. The data-collection mechanism collects development-process data and artifacts and stores them in a data model. The recommendation engine then analyzes this data model to generate personalized recommendations for developers. The user interface acts as the gateway for the recommendation cycle, triggering it and presenting the results to the developers.

2.4.1 Decision-Making Techniques

When it comes to software development, selecting the most appropriate architecture for a project is one of the most essential decisions that must be taken. Software engineering literature presents a variety of techniques for helping to make this choice, however, it fails to provide clear guidance on which technique is best suited for specific circumstances [11].

According to Falessi et al. [11], the quality of a decision-making technique can be judged by its ability to lead the user towards better alternatives while being user-friendly. A poor decision-making technique may result in difficulties in the development process and ultimately lead to bad choices being made. Misunderstandings caused by these poor techniques may result in the election of the wrong architectural alternative, which can lead to the dissatisfaction of the client and, in severe cases, a rework of the whole system. Thus, it is important to choose a decision-making technique that not only guides the user towards better alternatives but also minimizes the risk of misunderstandings.

According to Brooks [23], there is no magic solution that can effortlessly resolve all the difficulties in software engineering. Similarly, as noted by Falessi [11], the notion that a single solution cannot address all problems extends to the field of decision-making, where a one-size-fits-all approach is not feasible. In fact, to the best of his knowledge, no study has yet offered a systematic way to choose the best decision-making technique for making selections among design alternatives.

The research conducted by Falessi et al. [11] has made a significant contribution to the field of decision-making in software engineering and this contribution was of great use to this thesis. Their work has defined key concepts and provided a foundation for understanding the complexities involved in selecting the right decision-making technique, which helped design the recommendation system. In his research, Falessi [11] presented a Feature Tree Model (figure 2.3) that provides a characterization schema of decision-making techniques for resolving trade-offs in software architecture design. This figure serves as

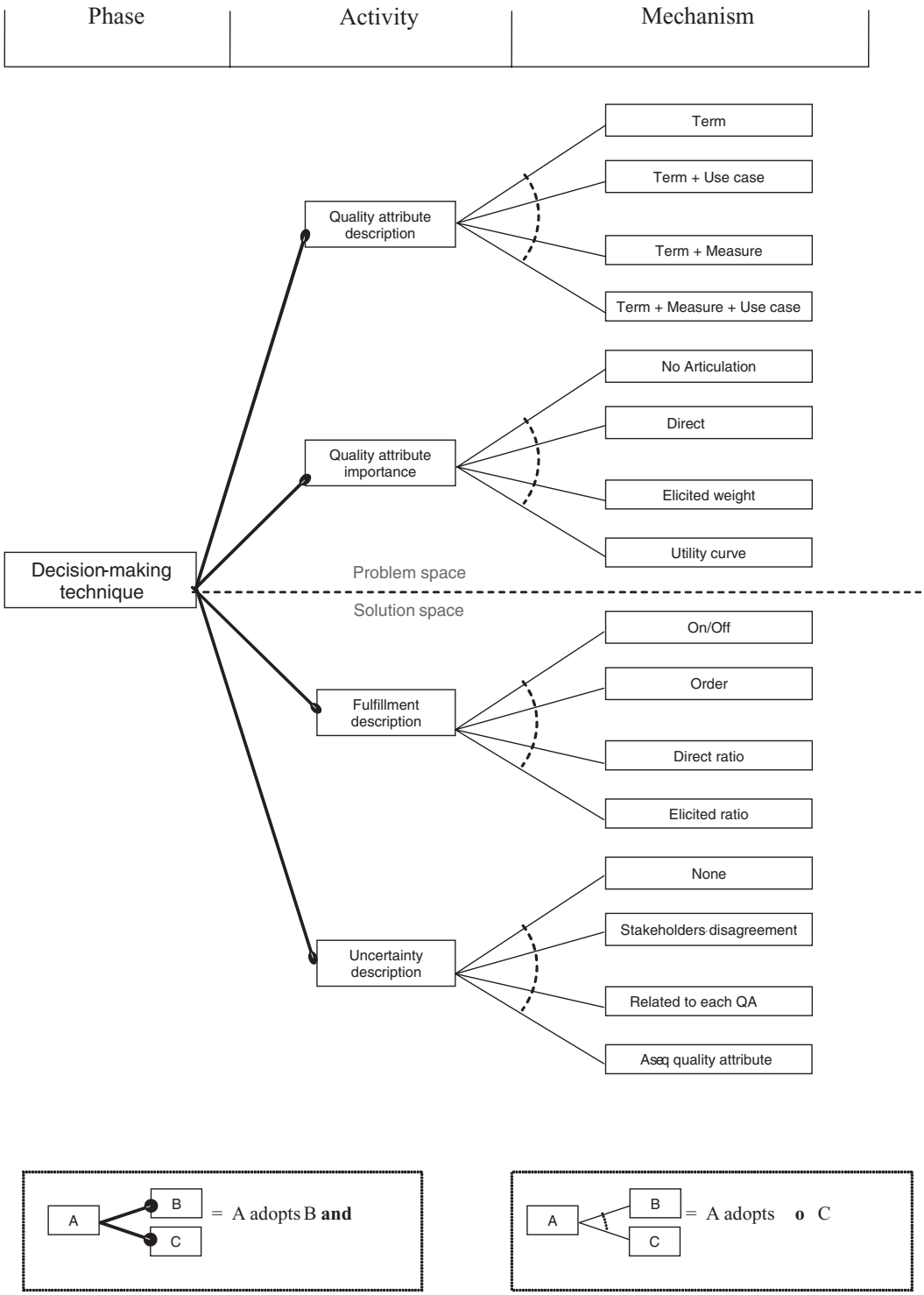


Figure 2.3: Feature tree model with a characterization schema of decision-making techniques (taken from [11])

a visual representation of some of the topics, especially the ones related to descriptions, that we had to be aware of when developing the recommendation system.

As we moved forward with the development of our recommendation system, we inspired ourselves in Falessi's work [11] and selected some of the mechanisms depicted in figure 2.3. Particularly, we used the "Term + Use Case" to describe the attributes. Falessi states [11] that in order to avoid misinterpretations there's a need to have a common and clear understanding of every attribute, even though we don't target attributes but requirements, in our point of view, the same logic should be applied. Regarding descriptions, instead of just opting for the requirement's term we opted to use both the term and the use case. Furthermore, we employed the "Elicited weight" method to assess the importance of attributes. We chose this technique because it illustrates well the attributes' importance while maintaining simplicity for future re-configurations.

2.5 Software Architecture Analysis

Designing a software architecture is a challenging task that requires consideration of multiple interdependent design decisions, related to a wide range of requirements concerns. Xu et al. [58], state that being able to understand the consequences of architectural design decisions is one of the most important aspects of this decision-making process.

Evaluating functional requirements for common services in a software system is straightforward as all stakeholders agree on how the system should function. However, assessing quality attributes can be more difficult as stakeholders often have differing opinions on what should take priority [58]. Software quality attributes include both the specific criteria of how the system is built and the qualitative constraints on performance or usability.

Finding a good solution for a high-quality software architecture involves exploring different options to meet the given requirements. With the help of software architecture analysis it is possible to understand if the solution found is an optimal one, that maximizes support for the quality attributes and fulfills all requirements.

2.5.1 Software Architecture Analysis Approaches

Some approaches that can help us evaluate our decision of which architecture to choose, based on the supported quality attributes, are the following:

Software Architecture Analysis Method (SAAM)

The SAAM [26] is a systematic approach for describing and analyzing software architectures. According to the SAAM methodology before analyzing any architecture there's a need to have a clear description of it, so it provides us with three perspectives for that purpose - functionality, structure, and allocation. After being able to analyze and describe the architecture, the main activities involved in the SAAM are characterizing a

canonical functional partitioning for the domain, mapping the functional partitioning into the architecture's structural decomposition, choosing a collection of quality attributes to assess the architecture, choosing a collection of specific tasks to test the chosen quality attributes, and evaluating the amount of support provided by each architecture for each task.

Architecture Tradeoff Analysis Method (ATAM)

The ATAM [25] is a risk identification method that detects areas of potential risk in a software architecture, it is influenced by not only the architectural styles and the quality attribute analysis communities but also SAAM. The ATAM is a nine-step process used to evaluate a system's architecture. It involves the generation of a utility tree, which is a hierarchical structure of quality attribute requirements, and the use of Attribute-Based Architectural Styles (ABAS)s to identify key architectural decisions. The output of the ATAM is a list of risks and non-risks, which can be problematic architectural decisions.

2.5.2 Evaluation of SAAM/ATAM

Xu et al. [58] made a survey of software architecture analysis approaches, where both SAAM and ATAM are evaluated as a group since they deliver similar ideas. In this evaluation, SAAM and ATAM share similar features in their evaluations. Both methods focus on identifying quality attributes based on stakeholders' needs and providing support for unquantifiable attributes. Yet, neither provides support for resolving conflicting goals among stakeholders.

Both methods evaluate the architecture after its development, modeling both architecture and quality attributes before the analysis. They both use human-based measurement and qualitative methods for evaluating and comparing design alternatives, but offer limited guidance for tradeoff analysis and relating quality attributes to architecture elements. Moreover, both have low automation and lack an information persistence mechanism.

Figure 2.4 represents the table provided in [58] for the graphical representation of the topics discussed above.

2.6 Technologies

In this section, we will explore some of the most commonly used technologies in Link Consulting, these technologies could be used to complement our thesis by serving as tools for developing and deploying the suggested microservices architecture. The seamless integration of these technologies and their widespread adoption in real-world scenarios has made them the preferred choices for numerous companies.

	Requirements Elicitation		
	Identifying QA	*Understanding Unquantifiable QA*	*Prioritizing Conflicting Goals*
SAAM/ ATAM	Stakeholder-based	Supported	Unsupported

	Architectural Design		
	Development Phase	*Modeled Artifacts*	*Identifying Design Decisions*
SAAM/ ATAM	Late Evaluation	Architecture (SAAM)	Unsupported
		Quality Attributes (ATAM)	

	Design Alternative Analysis			
	Design Alternative Analysis	*Design Alternatives Comparison*	*Tradeoff Analysis*	*Sensitivity*
SAAM/ ATAM	Human-based	Qualitative	Low	Sensitivity Points

	Overall Architectural Analysis		
	Context	*Mapping*	*QA Optimization*
SAAM/ ATAM	Independent	No Guidance	Unsupported

	Automation	
	Automated Tool	*Information Persistence*
SAAM/ ATAM	Low	Unsupported

Figure 2.4: Evaluating SAAM/ATAM (taken from [58])

Containers

Container technology can be used as a packaging and deployment method in the development of microservices.

Containers are packages of software that contain all of the necessary elements to run in any environment. Essentially, they are a way to virtualize the operating system, allowing applications to run anywhere, regardless of the underlying infrastructure [16]. This makes them a popular choice for companies and organizations that need to move quickly and deploy software efficiently, while also being able to handle large-scale workloads.

One of the key benefits of containerization is that it ensures consistency across different environments [16]. When an application is packaged into a container, it includes all the necessary dependencies and configurations, which eliminates the need to worry about the differences between the environments, making it easy to deploy and run.

They are becoming increasingly popular in software development, and currently, most microservices architectures use containers to deploy and manage their services. Two of the most popular container technologies are Docker and Open Container Initiative (OCI). Docker is an open-source platform that automates the deployment of applications inside containers, while OCI is an open-source standard for container images.

Orchestration platform

Regarding the choice of the platform for automating deployment, we usually go for Kubernetes. Kubernetes is a highly adopted orchestration platform, known for its open-source nature and its ability to automate the deployment, scaling, and management of containerized applications [31]. It provides an abstraction layer for the underlying infrastructure, making it easier to deploy and manage containers.

Some of the most important Kubernetes components mentioned by Gigi Sayfan in his book [50] are:

- **Cluster:** Collection of resources that Kubernetes uses to run the various workloads;
- **Nodes:** The physical or virtual machines that run the containerized applications, their job is to run pods;
- **Pods:** Kubernetes working unit, they are the smallest and simplest unit in the Kubernetes model and contain one or more containers. Typically, each microservice corresponds to a single pod in day-to-day usage;
- **Services:** An abstraction used to expose some functionality, they enclosure a set of Pods and a policy by which to access them. The routing to the microservices is done through services since they provide a stable IP address and DNS name;
- **Volumes:** A ephemeral way to store data in Kubernetes that allows data to be stored locally within a Pod. Volumes are where the local data of microservices is kept, allowing for shared access;
- **Ingress:** An external access point to microservices, ingress manages the complexities of exposing a service to outside the cluster, balancing traffic, handling SSL, and providing virtual hosting. An ingress is considered the entry point to a microservice.

Development frameworks

Another technology to consider when building microservices is the development framework. Within the company, two popular choices in this context are Spring and Quarkus.

Spring is a widely-used, open-source framework that provides a programming model for building java-based enterprise-grade applications [54]. It offers a wide range of features and tools that aid developers in building microservices.

Quarkus is a full-stack Kubernetes-native Java framework made for Java Virtual Machines (JVM)s and native compilation [20]. It is designed to optimize for resource efficiency and fast startup times, making it well-suited for cloud-native and containerized applications.

Database

Some microservices will need to store their data, in this scenario, according to the situation, the architectural design may indicate a relational database or a non-relational database. A commonly used relational database is PostgreSQL, which is an open-source object-relational database system that is known for its reliability, data integrity, and performance [42]. For situations where a non-relational database is more suitable, it is common to use MongoDB. MongoDB is a simple-to-use document database that is easily scalable.

Summing up, PostgreSQL is usually used when there's a need for support in complex data relationships and transactions, while MongoDB is more chosen when there's a need for a more flexible and scalable document-based data model.

Observability

In the context of observability, Usman has pointed out that, since microservices architectures are highly distributed and complex, traditional monitoring approaches are not sufficient to provide the level of visibility required to manage and troubleshoot these kinds of systems [56].

Consequently, when the requirement for observability arises in Link Consulting, Prometheus and Grafana are the preferred tools for the task. Prometheus [43] is an open-source monitoring and alerting system that is mainly used in cloud-native environments like Kubernetes. It includes a local time series database for storing metrics from various sources and provides a functional query language called PromQL for analyzing and visualizing the data [43]. Grafana [18], on the other hand, is an open-source platform for the visualization and analysis of metrics, logs, and traces. Grafana not only provides a wide range of visualization options to display metrics but also grants the capability for the creation of alerts [18].

Message broker

Lastly, when the architectural design recommends the usage of asynchronous communication the usual tools are either Solace [53] or Kafka [24] as messages brokers for intern communication. Both Solace and Kafka are popular message brokers that provide features such as publish-subscribe messaging, and they are widely used in distributed systems.

Service mesh

When a service mesh is required, Istio is the tool of choice, Istio is an open-source service mesh commonly used in microservices architectures. It effectively addresses the issues encountered by developers and operators in distributed setups [22]. Key features include:

- **Traffic Management:** Istio simplifies traffic routing, enhancing performance and enabling strategic service deployment. It provides service-level configurations, making tasks like A/B testing and canary deployments easy;

- **Security capabilities:** Istio provides security with identity verification, access controls, TLS encryption, and auditing tools. Its security model prioritizes in-depth defense for secure deployments in untrusted networks;

- **Observability:** Istio generates detailed telemetry for service interactions, aiding troubleshooting and performance optimization without requiring app changes. It offers comprehensive observability with metrics, traces, and access logs.

API Gateway

Whenever an API gateway is required, Kong is the preferred solution. Kong Gateway is an open-source, lightweight, fast, and flexible cloud-native API gateway [27]. Primary features include:

- Workflow automation and modern GitOps practices;

- Decentralization of applications/services and transitioning to microservices;

- Creation of a prospering API developer ecosystem;

- Identification API-related anomalies and threats;

- Security and governance over APIs/services, and improving API visibility across the entire organization.

2.7 Technological Decisions

This section explores some of the most widely considered technological decisions when architecting a microservices system. In this thesis, these choices are regarded as potential design alternatives that the system could recommend. Since we couldn't cover all possible decisions, we had to make choices regarding which ones to consider. Our selection was based on the prevalence of these decisions within Link Consulting. The seamless integration of these chosen strategies and their widespread use in real-world scenarios have established them as our preferred options.

Data Considerations

When dealing with data in a microservices architecture, several critical decisions come into play:

- **Database:** In Link Consulting one of the fundamental choices that is made in the beginning of every project revolves around the need for a database system. This decision is usually mostly based on the need for persistent data;

- **Database Architecture:** The architectural approach to databases is another critical decision. The most used database architectures in Link are One Database Per Microservice, One Physical Database for All Microservices with Different Schemas, or One Database Shared Across All Microservices. Each architecture has its unique strengths and challenges, as we discussed in 2.3.1;

- **Database Model:** The choice between a relational or non-relational database model is usually a hard one considering that each has its own advantages and disadvantages. Even according to experts such as Sahatqija [48] there is no absolute answer when trying to figure out which is the best option, the best is to consider some requirements such as scalability, flexibility and performance.

Cache Considerations

Caching data is one of the main strategies for optimizing system performance, but it requires careful consideration. The decision to implement **caching mechanisms** involves weighing the trade-offs between improved performance and increased costs. Erman [10] states that usually, in order to make benefit of using a cache, costs must be fairly high from a financial point of view. Nevertheless, sometimes the increase in performance is a decisive factor on its own.

API Considerations

APIs are the communication channels that enable microservices to interact with each other and the rest of the system, making the selection of the appropriate API type crucial. The decision to use either **synchronous** or **asynchronous APIs** affects system responsiveness, since in synchronous communication the client might even block while it waits, while the same doesn't happen in asynchronous communication [3].

Although Microsoft advises not to use synchronous communication when interacting between multiple microservices [35], this choice must be evaluated in terms of complexity, fault tolerance, and other project-specific requirements.

API Gateway Considerations

API gateways play a pivotal role in managing API requests and ensuring system security and observability. When it comes to managing API requests, the decision to utilize an **API gateway** holds significant importance. This choice is influenced by several key factors, including the requirement for service discovery, load balancing, monitoring, and security. An API gateway typically encompasses these functionalities, as it serves as a centralized point of entry [37].

Service Mesh Considerations

Service meshes are increasingly adopted for enhanced microservices communication and security. The decision of whether to implement or not a **service mesh** is an important one since the adoption of a service mesh can greatly influence service discovery, load balancing, fault tolerance, traffic monitoring, circuit breaking, and authentication and access control [33].

Observability Considerations

Observability is crucial for understanding system health and performance. Deciding whether to implement **observability** features is usually not an easy task since even though it is not mandatory for the functionality of the system, most of the time is vital for its good behavior. As highlighted in the observability section 2.3.3, implementing observability allows for the identification and resolution of issues before they affect the system. Typically, at Link Consulting, we consider introducing observability features when there is a need to monitor the system's health, and the client is comfortable with the associated costs.

Security Considerations

Security is of great importance in a microservices architecture, and making the right security decisions is imperative:

- **Security:** Deciding on the need for implementing security measures is a foundational choice. Similar to observability, at Link Consulting, this decision mainly relies on the client's preferences and their willingness to accommodate the associated costs. Nevertheless, we always aim to uphold security best practices;
- **Security Implementation:** The implementation of security within a microservices architecture requires careful consideration and one must decide the best methods and best locations to do it. As mentioned in the security section 2.3.4, the question of where to take care of authentication is a very common one, usually the options are: in a centralized authentication server or distributed across individual microservices.

3

Solution Design

The following chapter outlines the methodology that was developed as the solution of this thesis.

3.1 Description

The goal of this thesis was to develop a methodology that based on a proposed set of requirements, recommends which kind of microservices architecture should be used, what components it should have and some of the orchestration details. Specifically, the output of the methodology includes how the architecture should be organized, including considerations for microservices' handling of persistent data, while also covering topics like whether the database should be a relational database or a non-relational database and whether the communication should be synchronous or asynchronous.

The developed solution defined possible design alternatives that were the key to identifying sets of requirements that correlate with them. This correlation between the design alternatives and the requirements was thus taken into consideration by a recommendation system, that was built, such that it could be able to question the user about which requirements are necessary for the problem at hand and suggest a microservices architecture based on that.

3.2 Design Alternatives and Requirements Identification

The first step to achieve our goal was the identification and profound examination of various design alternatives in microservices architecture, then, the next step was the identification of requirements that correlate with them.

These steps formed the foundation for us to select the most appropriate design alternatives based on a set of requirements. For instance, one design alternative considered in the methodology revolves around whether an architecture requires a database. If a database is needed, another alternative explored is whether individual databases should be used for each microservice or if a shared database should serve all microservices. The

requirements aligned with these design alternatives were the need for persistent data in the former case and the need for isolated schemas (among other considerations) in the latter.

3.3 Evaluation Matrices

Requirements		Design Alternatives	
		Database Required	No Database Required
Data Persistency	Yes	5	0
	No	0	5

Figure 3.1: Example of an evaluation matrix populated with values from 0 to 5

Once the identification and investigation of the various alternatives and requirements essential to achieving our objective were concluded, the subsequent phase involved crafting separate matrices for distinct sets of alternatives. For example, three matrices were crafted for data alternatives, one matrix for communication alternatives, two matrices for security alternatives, among others.

These matrices, named evaluation matrices, were constructed using the technological decisions explored in section 2.7, alongside the correlations mentioned before, between the requirements and the potential design alternatives. To elaborate further, consider the matrix specifically focused on evaluating the necessity of incorporating a database, a graphical example of this matrix can be seen in figure 3.1. In this case, the matrix was crafted using the requirement "Data Persistency" as its main and only requirement, and the associated design alternatives encompass the inclusion or exclusion of a database.

Dependency Matrices

Dependencies		
Sheet Name	Expected result	Alternatives to be ignored
Database	Database Required	All

Figure 3.2: Example of the dependency matrix regarding the choice of the database data model

As the complexity of crafting the evaluation matrices grew apparent, we recognized the need for dependency matrices. These matrices were designed to ensure that certain evaluations depended on the outcomes of others, enhancing the precision and context-awareness of our recommendation system. For instance, the choice of the data model is only considered if the previously discussed matrix indicates the necessity of a database, otherwise choosing a database model wouldn't be appropriate. A graphical example of this dependency matrix can be seen in figure 3.2

3.4 Populating the Matrices

With the evaluation matrices defined, the next step was to assign reasonable values that aligned the user's answer for each requirement with a design alternative. For instance, let's revisit the matrix focusing on the necessity of incorporating a database. As can be seen in figure 3.1, one design alternative involved not using a database, while another included the usage of a database. The requirements for this table were just the need for data persistency and so the options available were "Yes" or "No", to support or neglect the requirement. To quantify these options, a classification system was employed. The "Yes" option was assigned the highest score on the classification scale, denoted by the value 5, when addressing the "Database Required" alternative. Conversely, it received the lowest score, represented by the value 0, when considering the "No Database Required" alternative. This configuration indicated strong backing for the data persistency requirement. On the other hand, the "No" option was assigned a maximum classification score of 5 when related to the "No Database Required" alternative, and a minimum score of 0 when connected to the "Database Required" alternative. This distribution reflected limited endorsement of the data persistency requirement. It's worth noting that this matrix is one of the simplest matrices in our methodology, many of them include intermediate values, not limited to just "5" or "0".

This kind of classification system using these values we called "scores" was inspired by the technique first introduced in section 2.4.1 as the "Elicited weights" classification mechanism.

3.5 Description Matrices

Requirements		Descriptions
Data Persistency	Yes	The system requires persistent storage of data over time. This ensures that data can be stored and accessed across microservices, allowing for data sharing and long-term data retention.
	No	There is no need to keep persistent data over time since the microservices are designed to operate independently and do not rely on sharing or persisting data across different services.

Figure 3.3: Example of a description matrix describing the two options available

In the previous sections, we introduced the development of the evaluation matrices aimed at assessing the correlation between design alternatives and requirements. Building upon these evaluation matrices, we extended our analysis by creating an additional set of matrices referred to as description matrices. These matrices were designed to provide a description of each individual requirement identified during the initial phases of our study. For each evaluation matrix, a corresponding description matrix was developed. Each row in a description matrix was dedicated to elucidating the specific meaning of a particular requirement allowing the users to make informed choices when considering whether to incorporate or exclude each requirement.

The assignment of descriptions to requirements was inspired by another technique introduced in section 2.4.1 named the "Term + Use Case" description mechanism.

A graphical representation of the description matrix corresponding to the previously displayed evaluation matrix is depicted in figure 3.3. This matrix elucidates the meaning of the two options available, facilitating the user's comprehension of the requirement.

3.6 Recommendation System

The final step involved the development of a recommendation system, this system was developed to ease the interaction of the users with the methodology by automating most of the process. As stated in section 2.4, these kinds of systems play a crucial role in helping developers with their decisions and, in this thesis we opted to leverage a recommendation system by having him consider the matrices we've constructed and the input provided by users, ultimately offering recommendations for suitable architectural choices.

Initially, the recommendation system presents users with the defined requirements, alongside descriptive information sourced from the matrices, it then allows users to select the chosen option for each requirement. Once the user's choices are made, the recommendation system employs these selections, combines them with the matrix values, and evaluates the available design alternatives to ascertain the most optimal architectural choice. For each requirement, the recommendation system examines the values within the matrices, calculates scores for each alternative, and identifies the one with the highest total score. The identified alternative is most likely the one that aligns most closely with the user's preferences.

However, the recommendation system's functionality extends beyond this decision-making process. It additionally generates a visual graph that dynamically represents all chosen design alternatives and their relationships. This graph helps to provide a clear visualization of the different options and their connections, making it considerably easier for the users to understand the relationships among the various architectural alternatives.

At its core, the recommendation system prompts the user with "questions", pinpoints the most fitting architecture, and offers a graphical overview of the selected alternatives.

3.7 Architectures

As mentioned in the previous sections, the goal of the developed methodology was to suggest a suitable architecture for a given problem, and for that, we needed to consider a range of different design alternatives that best fit the problem at hand and then choose an architectural design that integrates them well. This process was challenging, as it required defining and combining a large number of different alternatives that often conflicted with each other.

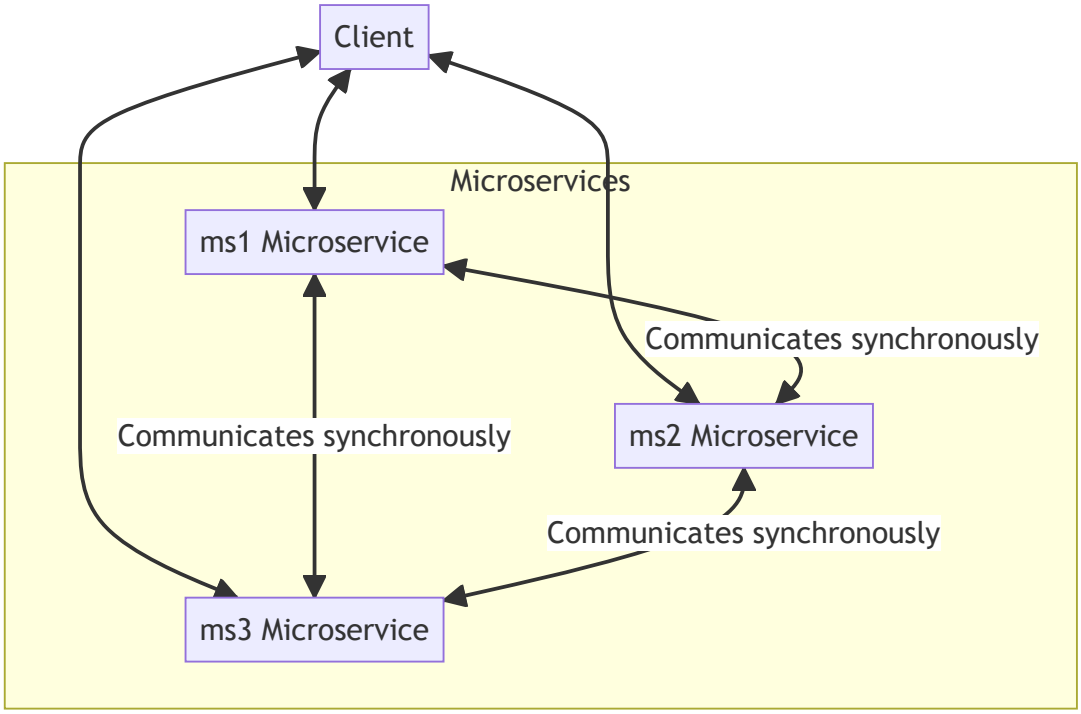


Figure 3.4: Three microservices communicating with each other synchronously

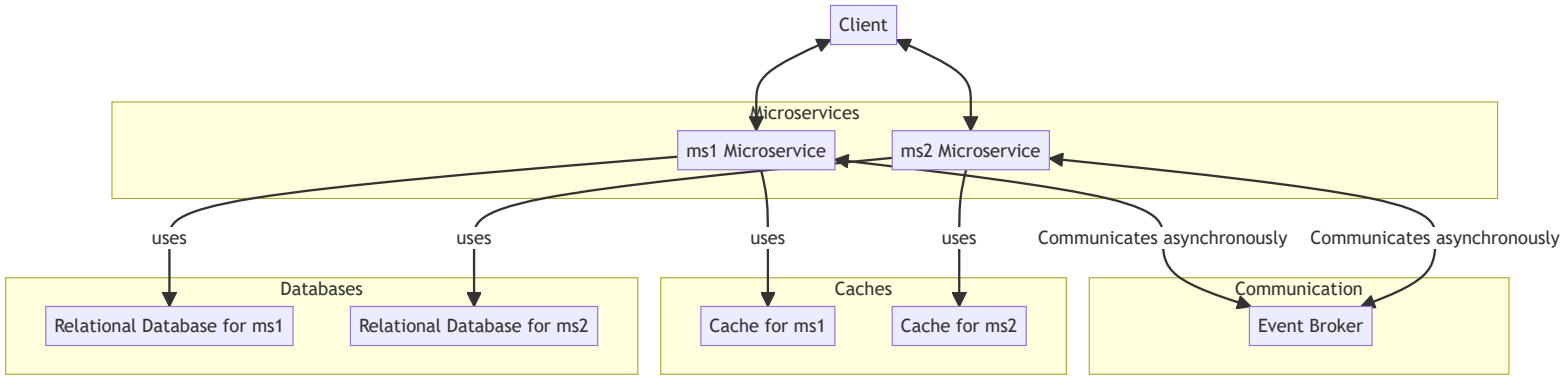


Figure 3.5: A microservice architecture using an event broker to communicate, each microservice with its own database and cache

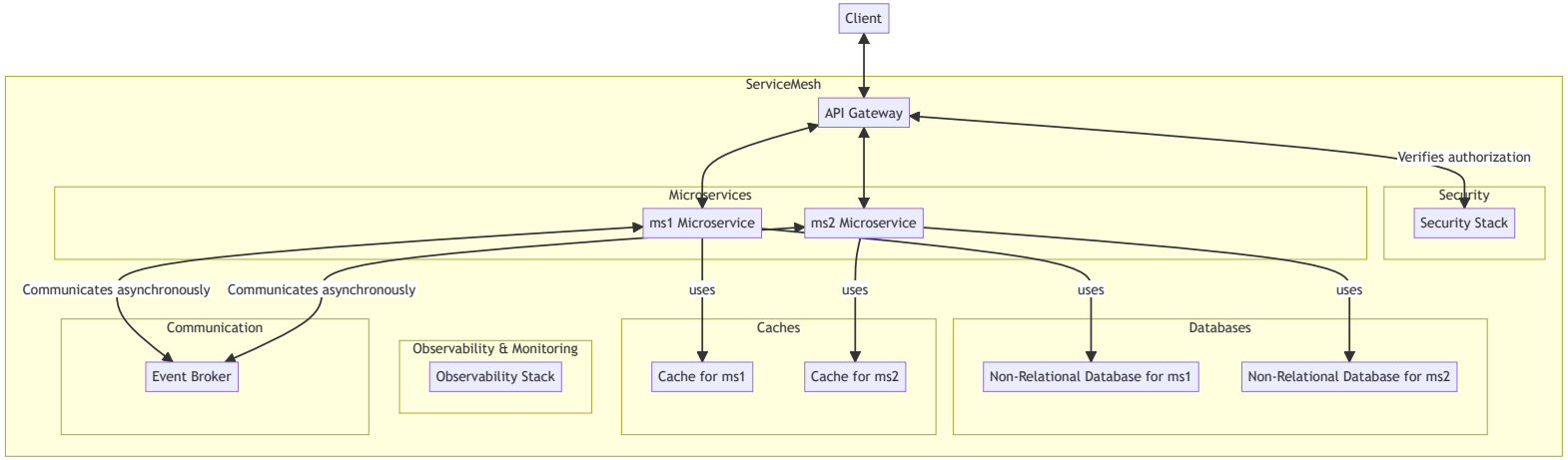


Figure 3.6: A microservice architecture with an event broker, an API gateway, both security and observability stacks, and a database and a cache for each microservice

Next, we will provide some examples of some final architectural outputs given by the methodology:

- An architecture composed of three microservices interacting with each other without the need for additional components. This architectural design emerged from a scenario where data persistence is not required. An example is presented in figure 3.4;
- An architecture composed of two microservices, each one with its own relational database and its own cache, and the communication between them is made asynchronously using an event broker. This architectural design originates from contexts where not only data persistency is needed but ACID transactions too. An example can be seen in figure 3.5;
- An architecture composed of two microservices each one with its own non-relational database and its own cache, the communication with each other is made asynchronously via an event broker. There is an API gateway for managing external interactions and a service mesh for managing internal communication. Additionally, a security stack ensures system protection, while an observability stack monitors operational performance. This architectural design arises from scenarios demanding data persistency, efficient data retrieval, effective external and internal communication management, security measures, and system monitoring. An illustrative example is provided in figure 3.6.

It's important to note that these are just a few examples of the methodology's outputs, many other potential architectures were also thoroughly explored.

4

Implementation

In this chapter, we will provide a detailed explanation of the steps taken during the implementation of this thesis, outlining the process followed to achieve the goals defined in the chapter 3.

4.1 Design Alternatives and Requirements Identification

In the process of identifying design alternatives and their corresponding requirements for microservices architecture, a systematic approach was taken to establish a meaningful correlation between each alternative and the specific requirements it could address. This correlation was achieved through an analysis of existing literature, insights from industry best practices, discussions with experts from Link Consulting, and my own perspective.

The goal was to create a structured and informed understanding of how each design alternative aligns with each distinct requirement based on their inherent characteristics, advantages, and limitations.

4.1.1 Literature Review and Industry Insights

The process began by exploring a wide range of sources, including research papers, articles, case studies, documentation and projects developed in Link Consulting related to microservices architecture. This literature review enabled the identification of various design alternatives that have been proposed or implemented in different contexts. Nonetheless, we focused primarily on design alternatives commonly employed within Link. Concurrently, insights from industry best practices and experiences shared by me and my co-workers provided practical perspectives on the viability and challenges associated with these alternatives.

4.1.2 Alternative Profiling

Each identified design alternative was profiled, detailing its key attributes, benefits, drawbacks, and scenarios in which it is most suitable. This profiling involved understanding the architectural principles, communication patterns, data management strategies, security

mechanisms, scalability approaches, and observability techniques associated with each alternative.

4.1.3 Requirements Elicitation

Leveraging the insights gathered from the alternative profiling, the benefits and drawbacks associated with each design alternative were systematically taken into account and used to find requirements that each alternative could address. For example, if an alternative has demonstrated strengths in horizontal scalability, it gave origin to requirements related to performance and scalability. Similarly, if an alternative has introduced complexities in inter-service communication, it would bring requirements involving communication patterns and reliability into the picture.

4.2 Evaluation Matrices

The creation of evaluation matrices was a key part of the implementation phase. These matrices were designed to establish a clear mapping between the identified design alternatives and their corresponding requirements. Each matrix was dedicated to a specific set of design alternatives. For instance, within the context of data management, three matrices were formulated: one to determine the need for a database, another to assess the most appropriate database architecture, and a third to explore the most suitable database model. Additional details regarding these matrices will be provided in section 4.2.1.

To build the evaluation matrices, we opted to use spreadsheet software within a workbook structure. This approach offered not only ease of modification for internal weights but also the flexibility to incorporate new requirements, alternatives, and even additional matrices as needed.

Dependency Matrices

As the complexity of the evaluation matrices grew apparent during their formation, a significant observation emerged: certain matrices should only come into consideration based on the outcomes of others. To illustrate, consider this scenario involving security where we introduced two evaluation matrices related to security: one determined the necessity for security measures, while the other pinpointed where security should be implemented within the architecture. While the presence of security might necessitate implementing a security stack, the specific authorization mechanisms might be distinct. This determination, whether through an external tool or internal programming, could influence the overall architecture.

Consequently, it became imperative to establish a dependency framework to address this interdependence of matrices. This dependency structure ensured that the recommendation system could generate more precise and context-aware suggestions. Specifically, the evaluation matrix focused on security implementation would only be considered if

Dependencies		
Sheet Name	Expected result	Alternatives to be ignored
Security	Security Required	All
APIGateway	API Gateway Required	Security inside the API Gateway

Figure 4.1: Example of the matrix regarding security implementation dependencies

the preceding matrix indicating the need for security yielded a positive result. Furthermore, within the dependency matrix, we provided the flexibility to ignore not only entire matrices, as seen in the case of security, but also specific alternatives within a matrix. For example, as depicted in figure 4.1, the alternative "Security inside the API Gateway" would be ignored if the recommendation didn't contemplate the usage of an API gateway.

By integrating these dependency matrices alongside the evaluation matrices, we effectively created a framework that allowed us to encapsulate relationships and conditional dependencies between different aspects of microservices architecture. Incorporating dependency matrices ended up enhancing the methodology's accuracy and also contributing to its adaptability, making it so that the alternatives suggested were more coherent with each other.

4.2.1 Developed Evaluation Matrices

As stated before, we employed a series of matrices to enable the evaluation and recommendation of design alternatives and each matrix was crafted to address a specific set of alternatives. We have previously showcased one of the developed matrices in figure 3.1, offering a visual representation of our matrix-based approach. Now, we will provide an overview of the matrices used and their significance within our methodology.

Database Matrix:

- **Purpose:** The Database Matrix focused on the perseverance of data, it aided in the evaluation of design alternatives concerning whether or not to have a database.

- **Importance:** Considerations on the perseverance of data are essential for the success of microservices. This matrix helped assess the alignment of design alternatives with the requirement "Data Persistency".

Database Architecture Matrix:

- **Purpose:** The Database Architecture Matrix focused on the architectural considerations associated with databases. It helped in evaluating the suitability of three architectural approaches, such as "One Database Per Microservice", "One Physical Database for All Microservices with Different Schemas", or "One Database Shared Across All Microservices".

- **Importance:** The chosen database architecture can exert a profound influence on system performance and long-term maintenance. This matrix took into account factors like data schema isolation, scalability requirements, and maintenance costs.

Database Model Matrix:

- **Purpose:** The Database Model Matrix aimed to guide decisions regarding the type of database model, whether relational or non-relational.

- **Importance:** Choosing the right database model is crucial for data consistency and query performance. This matrix allowed us to evaluate how well each model aligned with requirements like the need for data consistency and scalability.

Cache Matrix:

- **Purpose:** The Cache Matrix addressed the need for caching data and its impact on system performance and scalability.

- **Importance:** Caching can significantly improve response times, but it comes with trade-offs. This matrix helped assess whether caching was appropriate based on factors like performance, scalability, cost, and others.

API Type Matrix:

- **Purpose:** This matrix considered the type of APIs used, whether synchronous or asynchronous, and their implications.

- **Importance:** API design affects system complexity and responsiveness. This matrix allowed us to evaluate the API alternatives in terms of complexity, scalability, fault tolerance, and more.

API Gateway Matrix:

- **Purpose:** The API Gateway Matrix focused on whether to use or not to use API gateways for managing API requests.

- **Importance:** API gateways play a crucial role in request management, security, and observability. This matrix helped assess their suitability based on factors like rate limiting, load balancing, and security.

Service Mesh Matrix:

- **Purpose:** This matrix explored the adoption of a service mesh for traffic management, improved security, and improved observability between microservices.

- **Importance:** Service meshes enhance microservices communication and security but introduce complexity. This matrix aided in evaluating their suitability based on factors like traffic management, security, and scalability, and potential overheads.

Observability Matrix:

- **Purpose:** The Observability Matrix focused on choosing whether to implement observability through the usage of either just an observability stack, both the stack and other tools already in use like a service mesh or an API gateway, or opting not to implement a stack at all.

- **Importance:** Observability is vital for understanding system health. This matrix helped assess the necessity to introduce observability features in the system using requirements such as the need for system measurement, costs, and prioritization of systems health.

Security Matrix:

- **Purpose:** This matrix was designed to assess the need for implementing security measures.

- **Importance:** Security is paramount in a microservices architecture. This matrix helped in the decision of whether to implement security or not, taking into consideration factors such as sensitive data handling, authentication, authorization, and secure communication.

Security Implementation Matrix:

- **Purpose:** The Security Implementation Matrix assesses the methods and locations for implementing security within a microservices architecture, focusing on authorization verification and the decision of whether it should be centralized or distributed among the microservices.

- **Importance:** Proper security implementation is crucial for safeguarding microservices. This matrix aids in evaluating the alignment of security measures with specific requirements, including scalability, simplicity, the need for unconventional security measures, and simplification of security maintenance.

It is crucial to note that, as stated before, these matrices are not fixed. They can be customized with different requirements, and they even support the addition of new matrices to adapt to specific project needs and preferences.

4.3 Populating the Matrices

Populating the matrices involved assigning weights to each requirement-design alternative pair. These weights represented how well a particular design alternative satisfied a given requirement. To determine these weights, we devised a classification system that allowed for a quantitative assessment.

The classification system ranged from 0 to 5, with 5 indicating strong alignment with a requirement and 0 indicating poor alignment. This step required careful consideration and was based on a combination of research, expert opinions, and my own analysis. It's important to note that these weight assignments are fully configurable, offering users the flexibility to fine-tune them based on specific project needs and preferences. This characteristic is particularly valuable, as it allows for potential enhancements through various means, such as leveraging advanced techniques like machine learning and artificial intelligence for data training.

By utilizing such technologies, users can not only customize the recommendation process but also potentially optimize it further, ensuring that it aligns with the details of their particular project, enhancing both performance and accuracy.

4.4 Description Matrices

Description matrices were developed as a tool to ensure the understanding and consideration of each requirement identified earlier in the process. These matrices serve as a major aid for users, enabling them to grasp the significance and implications of each requirement before making critical decisions.

For each distinct evaluation matrix, we've diligently developed a corresponding description matrix and within each matrix, every row contains an informative breakdown of a specific requirement. This detailing empowers users to make well-informed choices that align with their project's unique context and objectives. It's essential to note that these matrices have also been designed to be dynamic and highly configurable. By understanding that every project presents its own nuances and challenges we have made it so that users have the flexibility to create and modify all the necessary descriptions in order to match their specific project context, ensuring that the recommendation process remains as adaptable and as accurate as possible.

To enhance user efficiency and minimize their workload, we have ensured that each description is uniquely tailored to its respective requirement. This means that if a description has already been crafted for one requirement, there's no need to duplicate it, users can simply leave it blank, saving time and effort while maintaining the clarity and integrity of the matrices.

4.5 Recommendation System

In this section, we will probe deeper into the recommendation system, a critical component of our thesis, the component that is responsible for automating the methodology.

4.5.1 Matrices Interpreter Tool

At the heart of our recommendation system, we've developed the matrices interpreter tool. This JavaScript-based application plays a pivotal role in processing the matrices mentioned earlier. To achieve this, the tool leverages various essential tools and packages:

- **"xlsx" Library:** We utilize the "xlsx" library for efficient manipulation of the Excel workbooks that contain our matrices. This library simplifies the extraction of data from Excel sheets, allowing us to smoothly process them;
- **"prompt-sync" Package:** For user interaction through the command line, we employ the "prompt-sync" package. This choice enhances the user experience, enabling real-time responses to system queries;
- **"commander" Module:** To handle various operational modes and commands, we rely on the "commander" module. This versatile tool simplifies the parsing of command-line arguments, facilitating a more intuitive user interface.

We chose these components to ensure the tool's efficiency, user-friendliness, and adaptability to different user preferences.

4.5.2 Modes of Operation

Our recommendation system offers users multiple modes of operation, catering to their diverse needs:

- **Normal Mode:** In this mode, users are guided step-by-step through the questions. They are presented with the defined requirements and descriptive information associated with each matrix, ensuring a clear understanding of the options;
- **Summary Mode:** To expedite the recommendation, users can opt for the summary mode using the "-r" or "–resume" flag. In this mode, the system recalls previously provided answers from the "answers.json" file, allowing users to review and modify their selections and proceed swiftly without reanswering questions;
- **Help Commands:** Users can leverage built-in help commands provided by "commander" to navigate the recommendation system effectively. These commands provide guidance and explanations for various functionalities, enhancing user interaction.

4.5.3 The Recommendation Process

Serving as the backbone of our recommendation system, the matrices interpreter tool processes all acquired information, including user selections and matrix values, to provide the recommendations. The underlying algorithm calculates scores for each design alternative, taking into account the user's input, the weights assigned to each requirement in the matrices and the interdependencies specified in the dependency matrices. The option with the highest total score in each matrix is identified as the optimal choice for each set of alternatives.

4.5.4 Output and Examples

The recommendation system generates several important outputs to assist users in making informed decisions:

- **answers.json:** This file records the user's responses to each question posed during the decision-making process. Users have the option to modify this file and rerun the program in summary mode to avoid reanswering questions, offering flexibility and convenience;
- **designAlternatives.json:** This file compiles a list of considered design alternatives and their corresponding scores. It provides transparency into the decision-making criteria, allowing users to understand how each alternative was evaluated;
- **results.json:** Encapsulating the final design alternatives chosen by the recommendation system, this file provides users with concrete architectural recommendations based on their input and the established criteria.

All these outputs are stored in an auto-generated folder called "data" present in the project's main directory.

4.5.5 The Decision-Making Algorithm

Understanding how our system determines the chosen alternatives is essential for the users, allowing them to fine-tune it. The algorithm follows a structured process:

- **Loading and Organizing:** The algorithm begins by loading and organizing the workbook, aligning the description matrices with the evaluation matrices;
- **User Interaction:** The system prompts the user and collects information on the desired project requirements while always presenting descriptive information for context, ensuring that the recommendations are aligned with the user's specific needs;

- **Scoring and Selection:** Based on the user's responses, the system computes scores for each design alternative, summing values row by row. The algorithm takes into account the dependency matrices to ensure that recommendations are context-aware and aligned with interdependencies between different microservices alternatives. This iterative process is applied to all matrices, and for each matrix, the alternative with the highest cumulative score emerges as the system's recommendation.

4.5.6 Dynamic Graph Construction

In addition to providing recommendations, our system takes a significant step forward in enhancing user comprehension by dynamically generating visual graphs using Mermaid.js. These dynamic graphs serve as invaluable tools for users to explore and understand the intricate interconnections and relationships between various architectural design alternatives, making the recommendation process more intuitive and insightful.

Our dynamic graph construction process begins by loading the relevant data from the "results.json" file, which contains the outcomes of various design choices. It then prompts the user for essential input, such as the number of microservices they plan to have and the names of each microservice.

The generated graph is composed of different components, each representing a chosen alternative:

- **Microservices:** These are at the core of the architecture, and each microservice is depicted as a distinct node within the graph. The relationships and dependencies between microservices are visually represented, allowing users to grasp how data flows and interactions occur;

- **API Gateway (optional):** If an API Gateway is part of the chosen design, it is integrated into the graph, showing how it connects with clients and various microservices. This visualization aids in understanding how API Gateway impacts the overall communication flow;

- **Security Stack (optional):** For architectures with security considerations, a Security Stack subgraph is included. It showcases the security implementation and how microservices or the API Gateway interact with it to ensure proper authorization and protection;

- **Event Broker (optional):** In scenarios where asynchronous APIs are employed, an Event Broker subgraph illustrates how microservices communicate asynchronously, highlighting the event-driven nature of the architecture;

- **Caches (optional):** When caching mechanisms are utilized, the graph visualizes cache instances and their associations with specific microservices or as a global cache;

- **Databases (optional):** If databases are part of the design, the graph portrays the relationships between microservices and their corresponding databases, demonstrating the data storage and retrieval mechanisms;
- **Observability and Monitoring (optional):** For architectures with observability features, an Observability Stack subgraph is included. It provides insights into monitoring and logging capabilities, illustrating the integration of these features into the architecture;
- **Service Mesh (optional):** In architectures employing a service mesh, a dedicated subgraph showcases the interaction between the service mesh and the microservices.

Once all components are integrated into the graph, the resulting visualization is saved as "graph.mmd" and further converted into both PNG and SVG formats for easy accessibility.

5

Evaluation

5.1 Evaluation Objectives

In this chapter, we will evaluate the accuracy of our methodology by evaluating the results obtained from the implementation of our tool, as well as its usability. This evaluation aims to assess the effectiveness and practicality of the developed microservices architecture recommendation system.

Evaluating Accuracy

To evaluate the accuracy of the architectural recommendations provided by the system, we analyzed how well the recommendations were aligned with what experts in microservices architecture expected.

Assessing Usability

To assess the usability of the recommendation system, we examined how easily users used the tool, the problems found, and if there were any errors during the tests. This step also helped validate the clarity of the instructions provided to users.

5.2 Testing Methodology

In this section, we describe in detail the methodology employed in evaluating the recommendation system's accuracy and usability. The testing process involved a group of participants with expertise in microservices architecture, who engaged in user tests to simulate real-world scenarios.

5.2.1 Sample Characterization

To conduct the evaluation, a sample of seven participants was selected. The participants were chosen based on their experience and expertise in the field of microservices, particularly their experience in making architectural decisions. The sample included colleagues

from my workplace who possess extensive knowledge and experience in microservices architecture. Specifically, six of them have worked or are currently working as architecture leads on various projects within the organization. Additionally, one colleague who is conducting research on microservices, particularly focusing on migration decisions, was included in the sample.

5.2.2 User Tests with Domain Experts

The testing methodology primarily involved a single procedure: the **User Tests with Domain Experts**. In this procedure, all participants from the sample were invited to engage in user tests, simulating real-world scenarios where the users would utilize the recommendation system to make informed choices about microservices architecture.

To enhance the accuracy of the evaluation, some testers were encouraged to use actual project data from their respective teams. They were asked to compare the recommended architecture with the implemented architecture and provide feedback on the alignment or disparities. This approach allowed us to assess the system's accuracy and its practical applicability in specific project contexts.

5.2.3 Preparations for Testing

In the preparation for the evaluation process, several steps were taken to ensure the assessment of our tool. This section outlines the preparations made, including user selection, technical accommodations, and the development of feedback collection methods.

- **User Selection**: To ensure the quality of the evaluation, users with substantial expertise in the field were meticulously chosen to serve as test users for the tool. Their extensive experience in microservices architecture made them well-suited for evaluating the recommendation system;
- **Docker Image and Docker Compose**: To accommodate users who either did not have Node.js installed or preferred using Docker, a Docker image and Docker Compose configuration were thoughtfully prepared. This approach aimed to ensure that all users could easily access and use the tool without technical barriers;
- **Feedback Collection Form and Questions**: A feedback collection form was developed to gather insights from the participants regarding their experience with the recommendation system, a copy of the form is present in annex I. This form included specific questions designed to solicit user opinions and feedback on various aspects of the recommendation system;
- **User Guide**: To facilitate user interaction and understanding of the tool, an instructional guide on how to use the program was provided to users at the beginning of the feedback collection form. This guide served as a reference point to help users utilize the recommendation system.

5.3 Results and Evaluation

In this section, we analyze the user responses collected during the evaluation of the recommendation system. The feedback and insights provided by the users are crucial for assessing the system's performance and identifying areas for improvement.

5.3.1 Survey Questions and User Responses

Q1 **Did the results align with your expectations?**: Users were asked if the results of the recommendation system aligned with their expectations. The responses to this Yes/No question helped to provide insights into user satisfaction. Figure 5.1 illustrates the distribution of responses, with all respondents providing a positive "Yes" response, indicating that 100% of the users were in agreement;

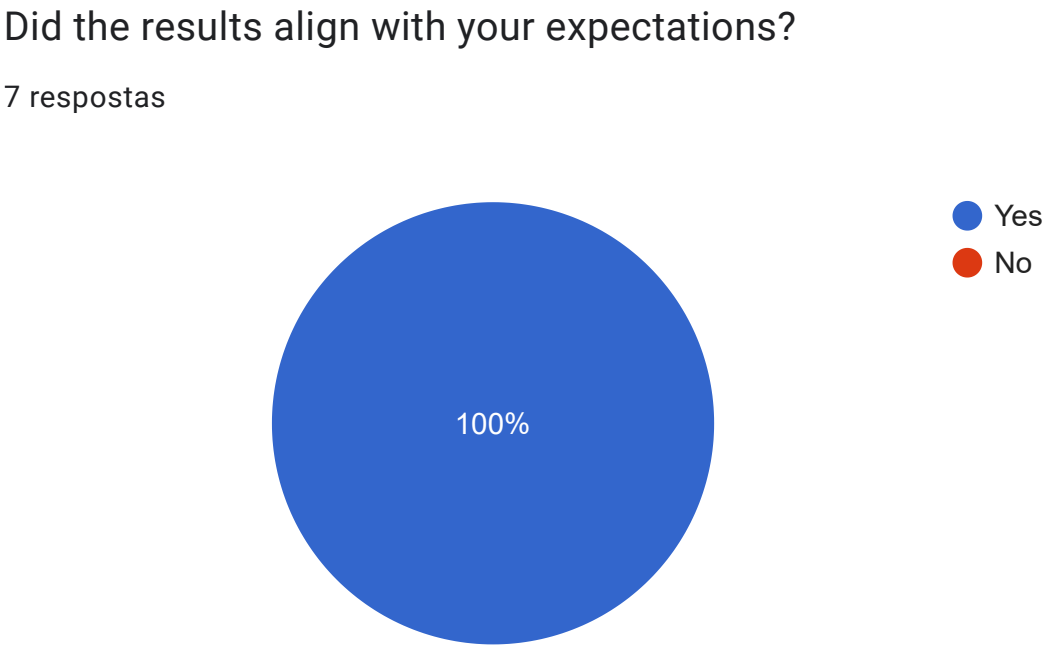


Figure 5.1: Question 1 responses distributions

Q2 **On a scale from 1 to 5, how satisfied were you with the recommendation results?**: To gauge user satisfaction more precisely, users were asked to rate their satisfaction with the recommendation results on a scale from 1 to 5. This Likert scale question had a mean satisfaction score of approximately 4.43, a mode of 4, and a standard deviation of approximately 0.49. The distribution of responses is visualized in figure 5.2;

Q3 **Were there any specific alternatives where you felt the recommendation was particularly effective?**: Users were queried about the areas where they found the recommendations on the alternatives to be particularly effective. The two areas with the highest percentages of positive feedback were APIGateway and Observability

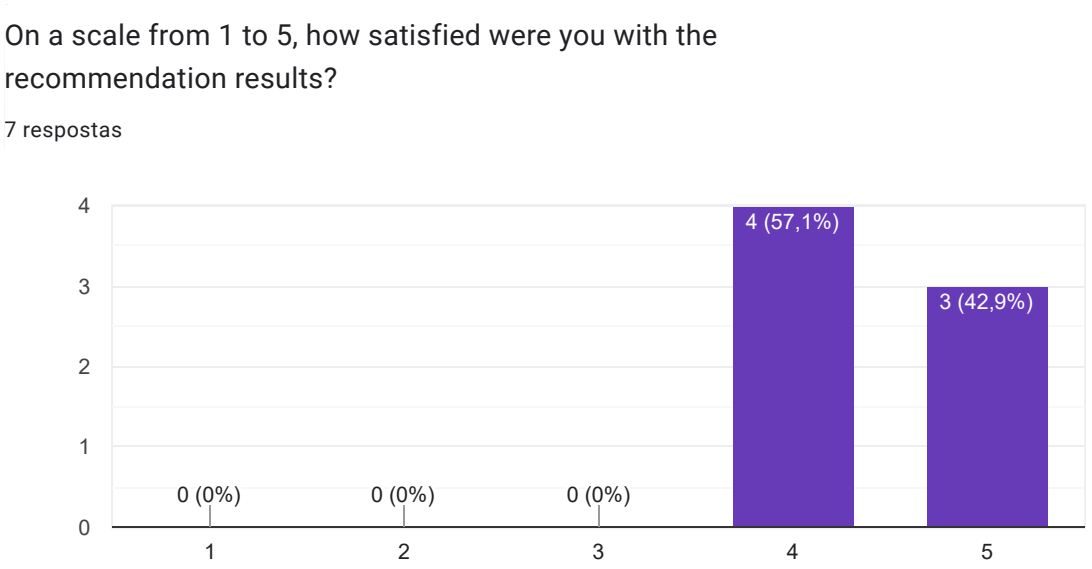


Figure 5.2: Question 2 responses distributions

both with 85.7% of the users' votes. The distribution of responses is depicted in figure 5.3;

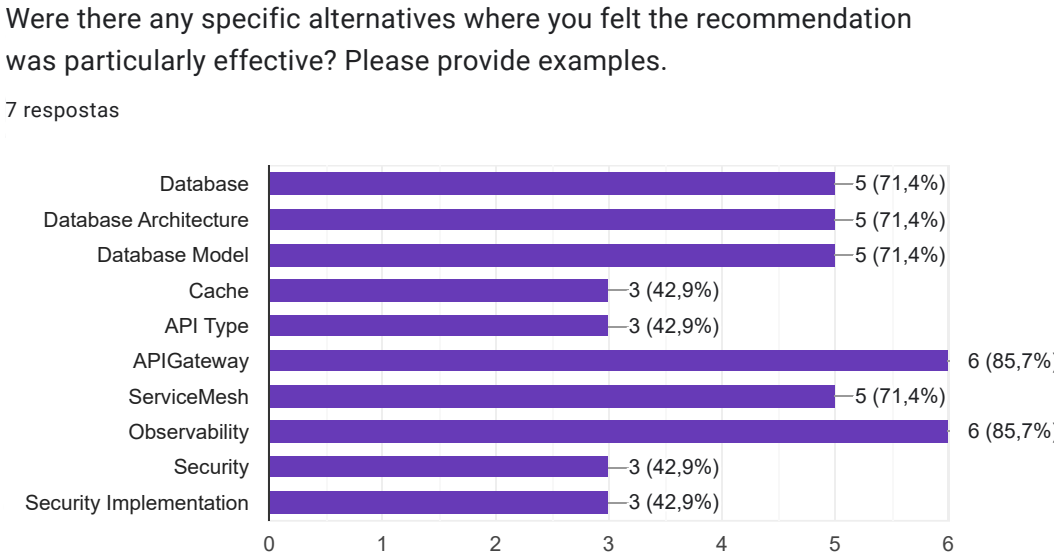


Figure 5.3: Question 3 responses distributions

Q4 **Were there any specific alternatives where you felt the recommendation provided less satisfactory results?**: Similarly, users were asked about areas where they felt the recommendations on the alternatives were less satisfactory. The four areas that most users found in need of improvement were Database Model, Cache, API Type, and Service Mesh all with 28.6% of the users' votes. The distribution of responses is presented in figure 5.4;

Were there any specific alternatives where you felt the recommendation provided less satisfactory results? Please provide examples.

7 respostas

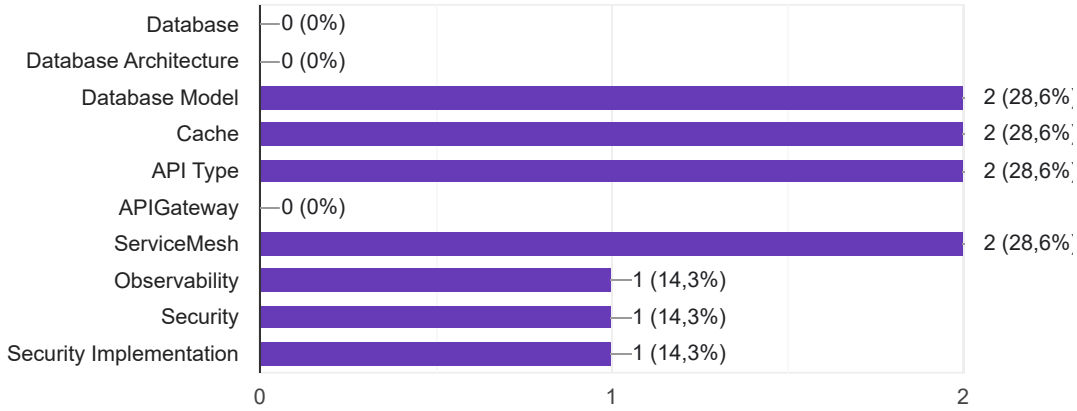


Figure 5.4: Question 4 responses distributions

Q5 **Were there any aspects that you found to be insufficiently explored or developed?**: Users were prompted to identify the aspects they found to be insufficiently explored or developed. The two areas most selected by users were Service Mesh (42.9%), and Cache (28.6%). The distribution of responses can be seen in figure 5.5;

Were there any aspects that you found to be insufficiently explored or developed? If so, please specify which topics and what you would like to see improved.

7 respostas

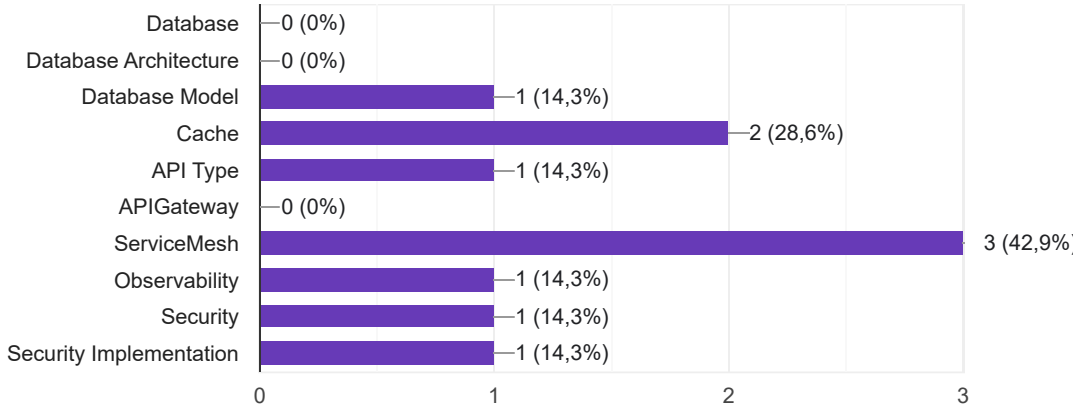


Figure 5.5: Question 5 responses distributions

Q6 **Do you have suggestions for any changes or enhancements to the recommendation algorithm or the way it is presented that could make it more useful or user-friendly?**: This open-answer question allowed users to provide suggestions for

changes or enhancements to the recommendation algorithm or its presentation;

Some of the user suggestions are:

- **Topic Skipping**: Allow users to skip certain topics if they are not relevant or of interest to their system;
- **Include Technology References**: Incorporate references to the technologies and platforms that are aligned with the suggested results;
- **User-Friendly Interface**: While not the primary focus, consider improving the presentation interface to make it more user-friendly and intuitive;
- **Progress Tracking**: Include a progress bar or a display indicating the number of questions the user has answered and how many are remaining (e.g., "10 of 20 questions answered");
- **Consider varying priorities**: Recognize that certain questions may apply differently to specific microservices within the system. For instance, some microservices may prioritize performance, while others may not.

Q7 **Were there any challenges or difficulties you encountered while using the tool?**: This open-answer question prompted users to share any challenges or difficulties they encountered while using the tool;

Some reported challenges include:

- **Repetitive Questions**: In the same context, there were many look-a-like questions;
- **Interface Usability**: The lack of a Graphical User Interface (GUI);
- **Visualizing Impacts**: Certain questions were hard to visualize in terms of their consequences when answering yes or no.

Q8 **Do you have any additional comments or feedback that could help improve the multi-sheet alternative selector?**: Users had the opportunity to offer any additional comments or feedback that could contribute to the improvement of the multi-sheet alternative selector.

A selection of user comments and feedback are:

- **Real-time Persistent Question Storage**: The tool could save previously answered questions. This would enable users to continue where they left off, even if they quit in the middle;
- **Architecture Purpose**: The tool could inquire about the purpose of the microservices to provide more context-aware suggestions.

5.3.2 Analysis of User Responses

5.3.2.1 Accuracy and relevance of recommendations

The analysis of user responses indicates that the majority of participants found the architectural recommendations provided by the system to be accurate and relevant to their specific project contexts. This alignment with user expectations was confirmed by all participants in Question Q1.

The Likert scale question (Question Q2) revealed that users rated their satisfaction with the recommendation results relatively high, with a mean satisfaction score of approximately 4.43 and a standard deviation relatively small (0.49) which implies that the variation among the answers was minimal. This high mean score with such a low standard variation, suggests that users generally found the recommendations valuable and in line with their expectations.

Users highlighted specific areas (Question Q3) where they found the recommendations to be particularly effective. Alternatives related to APIGateway and Observability received the highest percentages of positive feedback. This indicates that the system excelled in providing recommendations for these specific architectural decisions.

While the system demonstrated effectiveness in many cases, some areas (Question Q4) received lower satisfaction ratings from users. Despite receiving fewer votes, alternatives related to Database Model, Cache, API Type, and Service Mesh were identified as the ones where users felt the recommendations could be improved. It's worth mentioning that even though these recommendations were voted as the ones with "less satisfactory results", the majority of users did not necessarily find them unsatisfactory, as each of these four areas just received 28.6% of the votes. These findings lead to the conclusion that the evaluation matrices corresponding to these areas are the ones that would benefit the most from the refinement of weights through the utilization of AI techniques, mentioned in Fine-Tuning Using AI.

Users also identified some alternatives that they found to be insufficiently explored or developed (Question Q5). Alternatives related to Service Mesh and Cache were the most chosen as underexplored, this could mean that these alternatives are the ones that would be more positively impacted by further investigation and the creation of additional matrices to improve the recommendation system's coverage and effectiveness, mentioned in Expansion of Design Alternatives.

5.3.2.2 Usability of the tool

User feedback suggests that the tool's usability was generally well-received. Only one user expressed the desire for more context about the application in the survey and suggested making it clearer that the "summary mode" is optional. Apart from this, no other participants reported difficulties in using the tool or felt that it lacked clear instructions. This overall positive feedback suggests that the tool is user-friendly. Nonetheless, suggestions

for further improving usability were provided in response to questions Q6 and Q7.

Some suggestions pertained to the perceived repetitiveness of certain questions, making the process a bit exhausting. This issue could be easily addressed, as all matrices are fully customizable, and all requirements (each requirement leads to one question) can be edited or even deleted. However, in our opinion, it's better to empower users who feel this way to take control of this customization. It's important to note that this customization allows users to strike a balance between requirements quantity and context awareness. Deleting requirements excessively may result in a less context-aware system, potentially leading to less accurate recommendations.

Another significant suggestion was related to the difficulty in visualizing the consequences of certain questions. To address this, we have already made some revisions to enhance the clarity and informativeness of the description matrices. Regarding the absence of a GUI and the opportunity to skip certain topics, we believe that these suggestions should both be contemplated in our system and they will be addressed as a future work prospect in Graphical User Interface.

5.3.2.3 Feedback on the improvement possibilities

User feedback has provided insights into potential enhancements for the recommendation system. Participants shared suggestions and ideas for improving the system's functionality in questions Q6 and Q8.

One of the suggestions was the incorporation of references to specific technologies within the recommendations. While some preliminary investigation work has been conducted, particularly in the development of matrices related to both relational and non-relational databases, the extensive nature of this task prevented its full implementation within the scope of this thesis. Nevertheless, this idea is acknowledged as a promising avenue for future development, as outlined in Specific Matrices with Examples.

Another suggestion was to inquire users about the purpose of each microservice to provide more context-aware recommendations. This recommendation aligns with the future work prospect of Fine-Tuning Using AI. By gathering information about the purpose of each microservice from users, we could improve our suggestions for specific project contexts.

Concerning the suggestions for real-time persistent question storage and the inclusion of progress-tracking indicators. These features could significantly improve the user experience when answering the questions about the requirements. These suggestions will be considered for future development, under the wing of the GUI discussed above.

Lastly, there was a valuable suggestion regarding the varying priorities of different microservices. This suggestion draws attention to the fact that different microservices within the system may have distinct priorities, this is a suggestion that will be taken into consideration as a future prospect in Microservices Priorities.

5.3.3 Limitations

During the testing phase, certain limitations were encountered, including:

- **Participant Selection**: The need for participants with substantial experience in microservices architecture, particularly those accustomed to making architectural decisions, limited the size of the participant pool;
- **Tool Requirements**: Participants were required to have Node.js installed or use Docker to run the tool, which may have posed a limitation for some users.

5.4 Summary and Findings

In this chapter, we conducted an evaluation of the microservices architecture recommendation system, aiming to assess the accuracy of the architectural recommendations provided and the system's usability. The evaluation process involved a group of domain experts who engaged in user tests to simulate real-world scenarios.

5.4.1 User Satisfaction and Expectations

The findings revealed that the architectural recommendations generated by the system aligned well with the expectations of the users. All participants unanimously agreed that the results were in line with what they anticipated. Additionally, when asked to rate their satisfaction on a scale from 1 to 5, all the users provided high satisfaction scores, indicating that they found the recommendations valuable and relevant to their specific project contexts.

5.4.2 Effectiveness and Improvement Opportunities

While the system demonstrated effectiveness in providing recommendations for various architectural decisions, some areas have received lower satisfaction ratings from users, leading to certain alternatives being identified as areas where recommendations could be improved. Users provided valuable suggestions for addressing this topic, including the incorporation of specific technology references, the clarification of the purpose of each microservice, and advanced features such as progress tracking indicators and real-time persistent question storage.

5.4.3 Usability and User-Friendliness

Overall, users found the tool to be user-friendly and did not report difficulties in using it. However, some participants provided feedback regarding the repetitive and exhaustive nature of certain questions and the absence of a GUI, suggesting opportunities for improvement in terms of usability.

6

Conclusion

In this chapter, we will conclude by reflecting on the accomplishments of this thesis and discuss possible future work for upgrading our methodology and our tool.

6.1 Accomplishments

The primary objective of this thesis was to develop a methodology for the evaluation and recommendation of microservices architecture design alternatives. In this regard, we have achieved several significant accomplishments.

Development of a Methodology

A major milestone of this research has been the creation of a simple-to-use methodology. Our methodology addresses a wide spectrum of design alternatives within the microservices architecture, encompassing everything from database choices to security measures. This kind of approach provides users with the tools needed to make informed choices about their architectural designs.

Matrices-based Evaluation System

Our thesis has introduced an innovative approach by implementing evaluation matrices. These matrices offer a systematic and quantifiable method for assessing various design alternatives. This approach ensures that recommendations are grounded in well-defined criteria, granting the flexibility to adapt to the specific context of different projects.

User-Friendly Recommendation System

We have developed a user-friendly recommendation system that accommodates diverse user needs. Users can interact with the methodology in various modes, including a summary mode for streamlined usage. The system's outputs, such as "answers.json" and "results.json", are all stored locally which provides transparency and flexibility throughout the recommendation system's usage.

Dynamic Graph Visualization

A standout accomplishment of this work is the dynamic graph generation included in the recommendation system. This visualization tool illustrates what would have been just textual alternatives, allowing users to visually grasp the relationships and dependencies between different architectural components. It provides better user understanding and promotes more insightful decisions.

Interdependency Handling

To improve the accuracy of our recommendations, we have incorporated dependency matrices. These matrices consider the interdependencies between various design alternatives, enhancing the coherence of the suggested architectures. This ensures that the recommendations align with real-world architectural complexities.

Customizability

Our methodology is designed to be highly customizable. Users have the flexibility to fine-tune weights, incorporate new alternatives, and create new matrices. This adaptability ensures that the methodology remains relevant in many projects, as they can all fine-tune it. Moreover, it safeguards against the methodology becoming outdated over time.

6.2 Future Work

While we have made significant strides in this thesis, there are promising directions for future work that can further enhance the impact and applicability of our research.

Fine-Tuning Using AI

Exploring advanced techniques such as machine learning and neural networks to fine-tune the assigned weights for requirements. This could result in more accurate recommendations based on real-world data.

Expansion of Design Alternatives

Continuously incorporating new design alternatives to our existing matrices is essential to make sure our methodology doesn't get outdated. The technology landscape is dynamic, with new architectural patterns and tools emerging regularly, keeping our methodology up-to-date is essential to maintain its utility.

Specific Matrices with Examples

When discussing the testers' feedback in chapter 5, one suggestion was the incorporation of references to specific technologies, extending our matrices to include specific examples of tools and technologies for each design alternative would be a valuable endeavor. For instance, creating matrices that compare specific database systems like MongoDB, PostgreSQL, etc. It's worth noting that we have already begun this effort by working on matrices for specific databases. However, progressing further in this direction has proven to be a challenging task. The complexity arises from the extensive investigative work required, encompassing detailed research into the features and capabilities of each database system, as well as the analysis of their dependencies, new requirements, and the creation of descriptions for the new requirements to facilitate user understanding.

Graph Generator Maintenance

As the matrices and design alternatives are expanded, it is crucial to keep updating the dynamic graph generator. This visualization tool should evolve alongside our methodology to ensure users have access to up-to-date insights.

In-Depth Evaluation Methods

Applying more in-depth evaluation methods like Software Architecture Analysis Method (SAAM) and Architecture Tradeoff Analysis Method (ATAM) can provide valuable insights into the application of our methodology. Exploring these methods can help to further validate their effectiveness beyond what has already been evaluated.

Graphical User Interface

As mentioned in chapter 5, creating a GUI that's easy to use and visually attractive would make our methodology more user-friendly and encourage more people to use it. Some potential features to consider include real-time persistent question storage and progress-tracking indicators. Implementing these technologies would ensure a smooth user experience and encourage a wider audience to use our recommendation system.

Microservices Priorities

Building on the feedback from chapter 5, it would be beneficial to incorporate the ability to accommodate different priorities for different microservices within the same architecture. This feature would recognize that requirements may vary from one microservice to another, and the recommendations should consider these differences. For example, one microservice may be recommended to use a relational database, while another may be recommended a non-relational database.

6.3 Remarks

In summary, this thesis has successfully introduced a methodology for evaluating and recommending microservices architecture design alternatives. We have also created tools that interact with it, including all the matrices and the recommendation system with its dynamic graph visualization, to facilitate its usage and reconfigurability.

Looking forward, there are exciting prospects for future work. Exploring advanced techniques like machine learning, expanding our matrices, upgrading the dynamic graph generator, and enhancing the user interface will further enhance the utility and accessibility of our methodology.

In conclusion, this work serves as a strong foundation for improving architectural design decisions, and we anticipate its continued growth and hope for an impact in the field of microservices architecture.

Bibliography

[1] M. Ahmadvand and A. Ibrahim. "Requirements Reconciliation for Scalable and Secure Microservice (De)composition". In: *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*. IEEE Computer Society, 2016, pp. 68–73. DOI: 10.1109/REW.2016.026. URL: https://doi.org/10.1109/REW.2016.026 (cit. on p. 18).

[2] Amazon. *Event-Driven Architecture*. URL: https://aws.amazon.com/event-driven-architecture/ (visited on 2023-02-07) (cit. on p. 11).

[3] K. Bakshi. "Microservices-based software architecture and approaches". In: *IEEE Aerospace Conference Proceedings* (2017-06). ISSN: 1095323X. DOI: 10.1109/AERO.2017.7943959 (cit. on p. 28).

[4] J. Bucanek. "Model-View-Controller Pattern". In: *Learn Objective-C for Java Developers*. Berkeley, CA: Apress, 2009, pp. 353–402. ISBN: 978-1-4302-2370-2. DOI: 10.1007/978-1-4302-2370-2_20. URL: https://doi.org/10.1007/978-1-4302-2370-2_20 (cit. on p. 10).

[5] R. Chandramouli, Z. Butcher, et al. "Building secure microservices-based applications using service-mesh architecture". In: *NIST Special Publication* 800 (2020), 204A (cit. on p. 18).

[6] P. C. Clements et al. "Documenting Software Architectures: Views and Beyond". In: *Proceedings of the 25th International Conference on Software Engineering, May 3-10, 2003, Portland, Oregon, USA*. Ed. by L. A. Clarke, L. Dillon, and W. F. Tichy. IEEE Computer Society, 2003, pp. 740–741. DOI: 10.1109/ICSE.2003.1201264. URL: https://doi.org/10.1109/ICSE.2003.1201264 (cit. on pp. 1, 4).

[7] N. M. Coelho, M. C. Cunha, and L. G. Ferreira. "Security in Microservices Architectures". In: *CENTERIS 2020 - International Conference on ENTERprise Information Systems / ProjMAN 2020 - International Conference on Project MANagement / HCist 2020 - International Conference on Health and Social Care Information Systems and Technologies*

2020, Vilamoura, Portugal. Ed. by M. M. Cruz-Cunha et al. Vol. 181. Procedia Computer Science. Elsevier, 2020, pp. 1225–1236. DOI: 10.1016/j.procs.2021.01.320. URL: https://doi.org/10.1016/j.procs.2021.01.320 (cit. on p. 17).

[8] N. Dragoni et al. "Microservices: yesterday, today, and tomorrow". In: *CoRR* abs/1606.04036 (2016). arXiv: 1606.04036. URL: http://arxiv.org/abs/1606.04036 (cit. on pp. 7, 8, 14).

[9] I. C. Education. *Observability vs. Monitoring: What's the Difference?* 2022-08. URL: https://www.ibm.com/cloud/blog/observability-vs-monitoring (visited on 2023-01-29) (cit. on p. 16).

[10] J. Erman et al. "To Cache or Not to Cache: The 3G Case". In: *IEEE Internet Comput.* 15.2 (2011), pp. 27–34. DOI: 10.1109/MIC.2010.154. URL: https://doi.org/10.1109/MIC.2010.154 (cit. on p. 28).

[11] D. Falessi et al. "Decision-making techniques for software architecture design: A comparative survey". In: *ACM Comput. Surv.* 43.4 (2011), 33:1–33:28. DOI: 10.1145/1978802.1978812. URL: https://doi.org/10.1145/1978802.1978812 (cit. on pp. 20–22).

[12] C. N. C. Foundation. *CNCF SURVEY 2020*. Cloud Native Computing Foundation (CNCF). 2020. URL: https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf (cit. on p. 19).

[13] M. Fowler. *PolyglotPersistence*. 2011-11. URL: https://martinfowler.com/bliki/PolyglotPersistence.html (visited on 2023-01-21) (cit. on p. 13).

[14] M. Fowler and J. Lewis. *Microservices*. 2014-03. URL: https://martinfowler.com/articles/microservices.html (visited on 2023-01-05) (cit. on pp. 1, 6, 8–10, 14, 15).

[15] D. Garlan and M. Shaw. *Software Architecture*. 2008 (cit. on p. 4).

[16] Google. *What are containers? | Google Cloud*. URL: https://cloud.google.com/learn/what-are-containers (visited on 2023-01-15) (cit. on p. 24).

[17] M. Gopal. *Modern control system theory*. 1993. URL: https://www.google.com/books?hl=pt-PT&lr=&id=busgee_YN3oC&oi=fnd&pg=PR7&dq=Gopal,+M.:+Modern+Control+System+Theory+DOI&ots=SfvM7JUmxW&sig=GbikKC59BZ2k3ruT_eX1u8KJWGw (cit. on p. 16).

[18] Grafana. *Grafana documentation*. URL: https://grafana.com/docs/grafana/latest/introduction/ (visited on 2023-01-27) (cit. on p. 26).

[19] L. H. *Why Microservices Matter*. 2015-01. URL: https://blog.heroku.com/why_microservices_matter (visited on 2023-01-09) (cit. on pp. 7, 8).

[20] R. Hat. *What is Quarkus?* URL: https://www.redhat.com/en/topics/cloud-native-apps/what-is-quarkus (visited on 2023-09-22) (cit. on p. 25).

[21] *IEEE Standard Glossary of Software Engineering Terminology*. 1990 (cit. on pp. 1, 16, 17).

[22] Istio. *The Istio service mesh*. URL: https://istio.io/latest/about/service-mesh/ (visited on 2023-09-22) (cit. on p. 26).

[23] F. P. B. Jr. "No Silver Bullet - Essence and Accidents of Software Engineering". In: *Computer* 20.4 (1987), pp. 10–19. DOI: 10.1109/MC.1987.1663532. URL: https://doi.org/10.1109/MC.1987.1663532 (cit. on p. 20).

[24] A. Kafka. *Apache Kafka*. URL: https://kafka.apache.org/documentation/ (visited on 2023-02-07) (cit. on p. 26).

[25] R. Kazman, M. Klein, and P. Clements. *ATAM: Method for Architecture Evaluation*. 2000 (cit. on p. 23).

[26] R. Kazman et al. "SAAM: A Method for Analyzing the Properties of Software Architectures". In: *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994*. Ed. by B. Fadini, L. J. Osterweil, and A. van Lamsweerde. IEEE Computer Society / ACM Press, 1994, pp. 81–90. URL: http://portal.acm.org/citation.cfm?id=257734.257746 (cit. on p. 22).

[27] Kong. *Kong Gateway - v3.1.x | Kong Docs*. URL: https://docs.konghq.com/gateway/latest/ (visited on 2023-02-06) (cit. on pp. 11, 27).

[28] J. A. Konstan. "Introduction to recommender systems". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008, Vancouver, BC, Canada, June 10-12, 2008*. Ed. by J. T. Wang. ACM, 2008, pp. 1373–1374. DOI: 10.1145/1376616.1376776. URL: https://doi.org/10.1145/1376616.1376776 (cit. on p. 19).

[29] J. A. Konstan. "Introduction to recommender systems: Algorithms and Evaluation". In: *ACM Trans. Inf. Syst.* 22.1 (2004), pp. 1–4. DOI: 10.1145/963770.963771. URL: https://doi.org/10.1145/963770.963771 (cit. on p. 19).

[30] P. Kruchten. "What do software architects really do?" In: *J. Syst. Softw.* 81.12 (2008), pp. 2413–2416. DOI: 10.1016/j.jss.2008.08.025. URL: https://doi.org/10.1016/j.jss.2008.08.025 (cit. on p. 4).

[31] Kubernetes. *Kubernetes*. URL: https://kubernetes.io/ (visited on 2023-01-15) (cit. on p. 25).

[32] L. D. Lauretis. "From Monolithic Architecture to Microservices Architecture". In: *IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops 2019, Berlin, Germany, October 27-30, 2019*. Ed. by K. Wolter et al. IEEE, 2019, pp. 93–96. DOI: 10.1109/ISSREW.2019.00050. URL: https://doi.org/10.1109/ISSREW.2019.00050 (cit. on p. 7).

[33] W. Li et al. "Service Mesh: Challenges, State of the Art, and Future Research Opportunities". In: *13th IEEE International Conference on Service-Oriented System Engineering, SOSE 2019, San Francisco, CA, USA, April 4-9, 2019*. IEEE, 2019. DOI: 10.1109/SOSE.2019.00026. URL: https://doi.org/10.1109/SOSE.2019.00026 (cit. on pp. 18, 29).

[34] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[35] Microsoft. *Communication in a microservice architecture*. 2022-04. URL: https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture#asynchronous-microservice-integration-enforces-microservices-autonomy (visited on 2023-01-22) (cit. on pp. 1, 14, 15, 28).

[36] Microsoft. *Data considerations for microservices*. 2022-12. URL: https://learn.microsoft.com/en-us/azure/architecture/microservices/design/data-considerations (visited on 2023-01-21) (cit. on pp. 1, 12–14).

[37] F. Montesi and J. Weber. "Circuit Breakers, Discovery, and API Gateways in Microservices". In: *CoRR* abs/1609.05830 (2016). arXiv: 1609.05830. URL: http://arxiv.org/abs/1609.05830 (cit. on p. 28).

[38] A. Mounji et al. "Distributed audit trail analysis". In: *1995 Symposium on Network and Distributed System Security, (S)NDSS '95, San Diego, California, USA, February 16-17, 1995*. Ed. by J. T. Ellis, D. M. Balenson, and R. W. Shirey. IEEE Computer Society, 1995, pp. 102–113. DOI: 10.1109/NDSS.1995.390641. URL: https://doi.org/10.1109/NDSS.1995.390641 (cit. on p. 17).

[39] S. Newman. *Backends For Frontends*. 2015-11. URL: https://samnewman.io/patterns/architectural/bff/ (visited on 2023-01-15) (cit. on pp. 10, 11).

[40] A. Parker et al. *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020 (cit. on p. 16).

[41] R. Picoreti et al. "Multilevel Observability in Cloud Orchestration". In: *2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress, DASC/PiCom/DataCom/CyberSciTech 2018, Athens, Greece, August 12-15, 2018*. IEEE Computer Society, 2018, pp. 776–784. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00134. URL: https://doi.org/10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00134 (cit. on p. 16).

[42] PostgreSQL. *PostgreSQL: The world's most advanced open source database*. URL: https://www.postgresql.org/ (visited on 2023-01-15) (cit. on p. 26).

[43] Prometheus. *Prometheus*. URL: https://prometheus.io/docs/introduction/overview/ (visited on 2023-01-27) (cit. on p. 26).

[44] M. Richards. "Software Architecture Patterns". In: (2015). URL: http://oreilly.com/safari (cit. on pp. 1, 5–7).

[45] C. Richardson. *API gateway pattern*. URL: https://microservices.io/patterns/apigateway.html (visited on 2023-02-06) (cit. on p. 11).

[46] C. Richardson. *Event-driven architecture*. URL: https://microservices.io/patterns/data/event-driven-architecture.html (visited on 2023-01-15) (cit. on pp. 6, 11).

[47] M. P. Robillard, R. J. Walker, and T. Zimmermann. "Recommendation Systems for Software Engineering". In: *IEEE Softw.* 27.4 (2010), pp. 80–86. DOI: 10.1109/MS.2009.161. URL: https://doi.org/10.1109/MS.2009.161 (cit. on pp. 19, 20).

[48] K. Sahatqija et al. "Comparison between relational and NOSQL databases". In: *41st International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2018, Opatija, Croatia, May 21-25, 2018*. Ed. by K. Skala et al. IEEE, 2018, pp. 216–221. DOI: 10.23919/MIPRO.2018.8400041. URL: https://doi.org/10.23919/MIPRO.2018.8400041 (cit. on p. 28).

[49] J. Savolainen and V. Myllärniemi. "Layered architecture revisited - Comparison of research and practice". In: *Joint Working IEEE/IFIP Conference on Software Architecture 2009 and European Conference on Software Architecture 2009, WICSA/ECSA 2009, Cambridge, UK, 14-17 September 2009*. IEEE Computer Society, 2009, pp. 317–320. DOI: 10.1109/WICSA.2009.5290685. URL: https://doi.org/10.1109/WICSA.2009.5290685 (cit. on p. 5).

[50] G. Sayfan. *Mastering kubernetes*. Packt Publishing Ltd, 2017 (cit. on p. 25).

[51] D. Schmitz. *Dealing with data in microservice architectures*. 2021-05. URL: https://dev.to/koenighotze/dealing-with-data-in-microservice-architectures-part-3-replication-4h7b (visited on 2023-01-21) (cit. on pp. 12, 13).

[52] M. R. S. Sedghpour, C. Klein, and J. Tordsson. "Service mesh circuit breaker: From panic button to performance management tool". In: *HAOC '21: Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems, Virtual Event, United Kingdom, April 26, 2021*. ACM, 2021, pp. 4–10. DOI: 10.1145/3447851.3458740. URL: https://doi.org/10.1145/3447851.3458740 (cit. on p. 18).

[53] Solace. *Editions of PubSub+ Event Broker: Software*. URL: https://docs.solace.com/Software-Broker/SW-Broker-Set-Up/Setting-Up-SW-Brokers.htm (visited on 2023-02-07) (cit. on p. 26).

[54] Spring. *Spring Framework*. URL: https://spring.io/projects/spring-framework (visited on 2023-01-15) (cit. on p. 25).

[55] J. Thones. "Microservices". In: *IEEE Softw.* 32.1 (2015), p. 116. DOI: 10.1109/MS.2015.11. URL: https://doi.org/10.1109/MS.2015.11 (cit. on p. 8).

[56] M. Usman et al. "A Survey on Observability of Distributed Edge & Container-Based Microservices". In: *IEEE Access* 10 (2022), pp. 86904–86919. DOI: 10.1109/ACCESS.2022.3193102. URL: https://doi.org/10.1109/ACCESS.2022.3193102 (cit. on p. 26).

[57] A. P. Vale et al. "Security in microservice-based systems: A Multivocal literature review". In: *Comput. Secur.* 103 (2021), p. 102200. DOI: 10.1016/j.cose.2021.102200. URL: https://doi.org/10.1016/j.cose.2021.102200 (cit. on pp. 17, 18).

[58] L. Xu, D. J. Richardson, and H. Ziv. *A Survey of Software Architecture Decision-Making Techniques*. Institute for Software Research, 2007. URL: www.isr.uci.edu (cit. on pp. 22–24).

[59] X. Zhu et al. "Dissecting Service Mesh Overheads". In: *CoRR* abs/2207.00592 (2022). DOI: 10.48550/arXiv.2207.00592. arXiv: 2207.00592. URL: https://doi.org/10.48550/arXiv.2207.00592 (cit. on p. 19).

[60] O. Zimmermann. "Microservices tenets". In: *Comput. Sci. Res. Dev.* 32.3-4 (2017), pp. 301–310. DOI: 10.1007/s00450-016-0337-0. URL: https://doi.org/10.1007/s00450-016-0337-0 (cit. on p. 8).

A

Recommendation system usage example

In this appendix, we will walk you through the process of downloading, installing, and using our recommendation tool to suggest a microservices architecture for your project.

Our tool is designed to simplify the decision-making process by evaluating various design alternatives and providing you with recommendations based on your selected requirements.

A.1 Downloading and Exploring the Tool

To get started, follow these steps:

1. **Download the Tool:** You can download our tool from the following link: https://drive.google.com/file/d/1lDydRyYm07rI7DQMa_lEp7HBRwGBZzPH/view

2. **Unzip the Tool:** After downloading, locate the downloaded folder and unzip it to your desired location.

3. **Explore the Contents:** Open the unzipped folder and explore its contents. You will find an xlsx file containing the developed matrices. This xlsx file allows you to view and modify the matrices according to your project's specific requirements. Feel free to make changes as needed.

A.2 Installation and Usage

A.2.1 Using Node

1. **Install Dependencies:** Open your terminal and navigate to the tool's directory. Run the command `npm install` to install the required dependencies.

2. **Running in Normal Mode:** To start the program in normal mode, run the command `node index.js start`.

3. **Running in Summary Mode:** To run the program in summary mode (which allows you to review and modify previous answers), use the command `node index.js start -r`.

4. **Generate Dynamic Graph:** If you want to dynamically generate a graph representing the suggested architecture, run `node index.js graph`.

A.2.2 Using Docker

1. **Running in Normal Mode:** To start the program in normal mode using Docker, run `docker-compose run start`.

2. **Running in Summary Mode:** To run the program in summary mode (which uses the answers from a previous normal mode run), execute `docker-compose run resume`.

3. **Generate Dynamic Graph:** To dynamically generate a graph representing the suggested architecture using Docker, use `docker-compose run graph`.

A.3 Understanding the Outputs

After running the tool, you will find several important output files and folders in a directory named "data" within the tool's main directory. Here's what each file contains:

- **answers.json:** This file records your responses to the questions asked during the decision-making process. You can modify this file if you want to change your answers and rerun the program in summary mode.

- **designAlternatives.json:** This file lists the considered design alternatives and their corresponding scores, giving you insight into the decision-making criteria.

- **results.json:** This file contains the final design alternatives recommended by the system based on your input and the established criteria.

A.4 Using the Dynamic Graph

One of the unique features of our tool is the dynamic graph that visually represents the recommended microservices architecture. The graph provides a clear and intuitive understanding of how various architectural components interact. Here's how to use it:

1. Run the command to generate the dynamic graph: `node index.js graph` (Node) or `docker-compose run graph` (Docker).

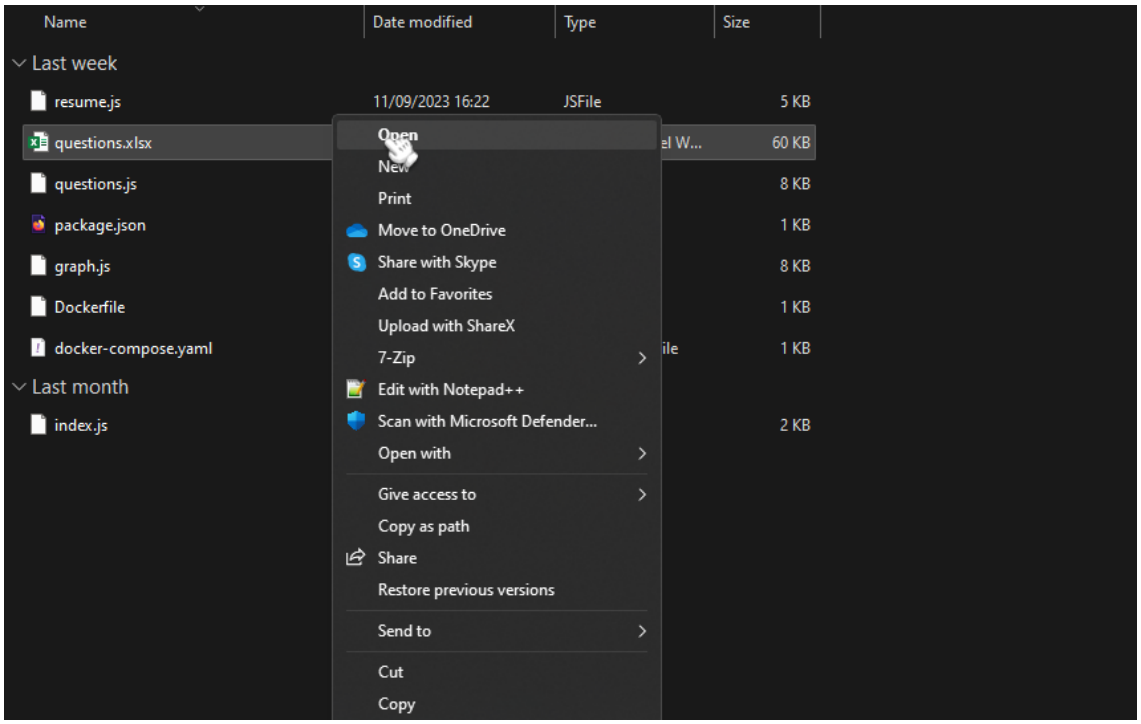
2. The generated graph will be saved as "graph.mmd" and automatically converted into both PNG and SVG formats. You can find these files in the "data" directory.

3. Open the generated PNG or SVG file to explore the recommended architecture. The graph visually depicts microservices, API gateways, security stacks, event brokers, caches, databases, observability features, and service meshes, if applicable.

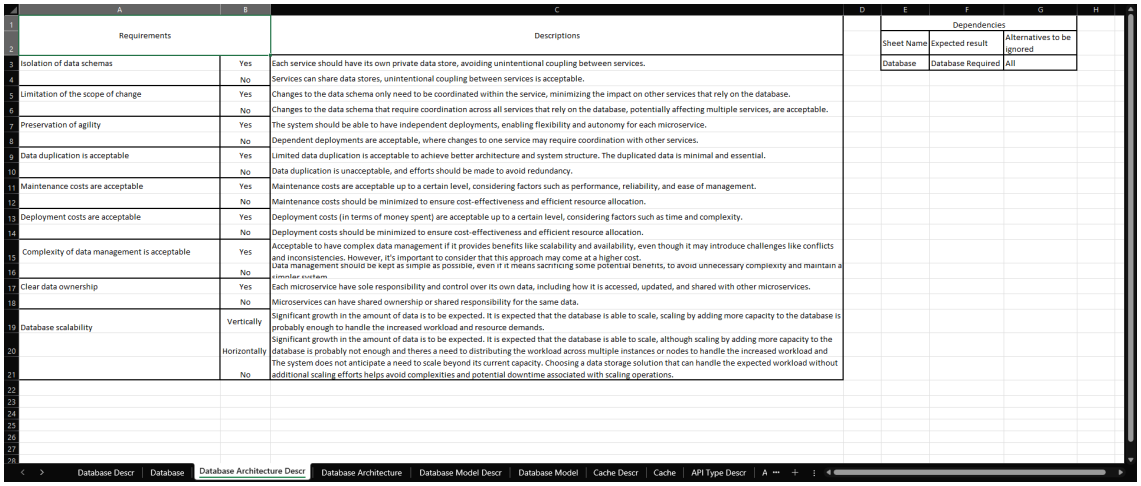
A.5 Examples

To provide a clearer understanding, let's walk through a few examples of using our recommendation tool:

Modifying the Matrices



(a) Opening the xlsx file containing the matrices



Requirements		Descriptions
Isolation of data schemas	Yes	Each service should have its own private data store, avoiding unintentional coupling between services.
	No	Services can share data stores, unintentional coupling between services is acceptable.
Limitation of the scope of change	Yes	Changes to the data schema only need to be coordinated within the service, minimizing the impact on other services that rely on the database.
	No	Changes to the data schema that require coordination across all services that rely on the database, potentially affecting multiple services, are acceptable.
Preservation of agility	Yes	The system should be able to have independent deployments, enabling flexibility and autonomy for each microservice.
	No	Dependent deployments are acceptable, where changes to one service may require coordination with other services.
Data duplication is acceptable	Yes	Limited data duplication is acceptable to achieve better architecture and system structure. The duplicated data is minimal and essential.
	No	Data duplication is unacceptable, and efforts should be made to avoid redundancy.
Maintenance costs are acceptable	Yes	Maintenance costs are acceptable up to a certain level, considering factors such as performance, reliability, and ease of management.
	No	Maintenance costs should be minimized to ensure cost-effectiveness and efficient resource allocation.
Deployment costs are acceptable	Yes	Deployment costs (in terms of money spent) are acceptable up to a certain level, considering factors such as time and complexity.
	No	Deployment costs should be minimized to ensure cost-effectiveness and efficient resource allocation.
Complexity of data management is acceptable	Yes	Acceptable to have complex data management if it provides benefits like scalability and availability, even though it may introduce challenges like conflicts and inconsistencies. However, it's important to consider that this approach may come at a higher cost.
	No	Data management should be kept as simple as possible, even if it means sacrificing some potential benefits, to avoid unnecessary complexity and maintain a
Clear data ownership	Yes	Each microservice have sole responsibility and control over its own data, including how it is accessed, updated, and shared with other microservices.
	No	Microservices can have shared ownership or shared responsibility for the same data.
Database scalability	Vertically	Significant growth in the amount of data is to be expected. It is expected that the database is able to scale, scaling by adding more capacity to the database is probably enough to handle the increased workload and resource demands.
	Horizontally	Significant growth in the amount of data is to be expected. It is expected that the database is able to scale, although scaling by adding more capacity to the database is probably not enough and theres a need to distributing the workload across multiple instances or nodes to handle the increased workload and
	No	The system does not anticipate a need to scale beyond its current capacity. Choosing a data storage solution that can handle the expected workload without additional scaling efforts helps avoid complexities and potential downtime associated with scaling operations.

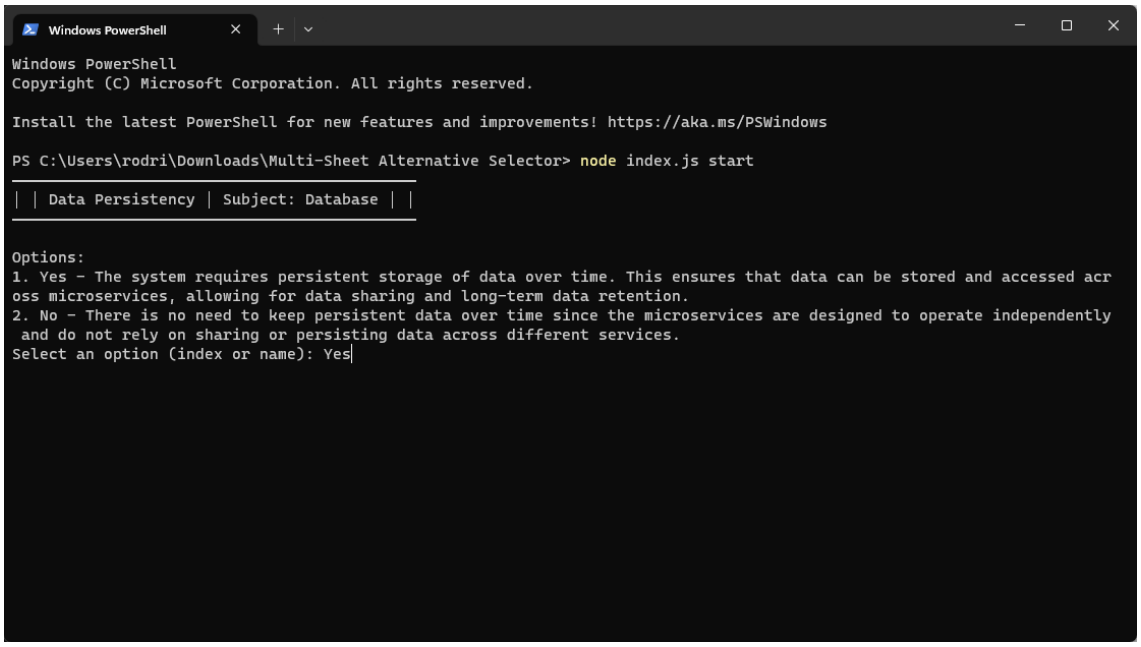
Dependencies		
Sheet Name	Expected result	Alternatives to be ignored
Database	Database Required	All

Database Descr | Database | Database Architecture Descr | Database Architecture | Database Model Descr | Database Model | Cache Descr | Cache | API Type Descr

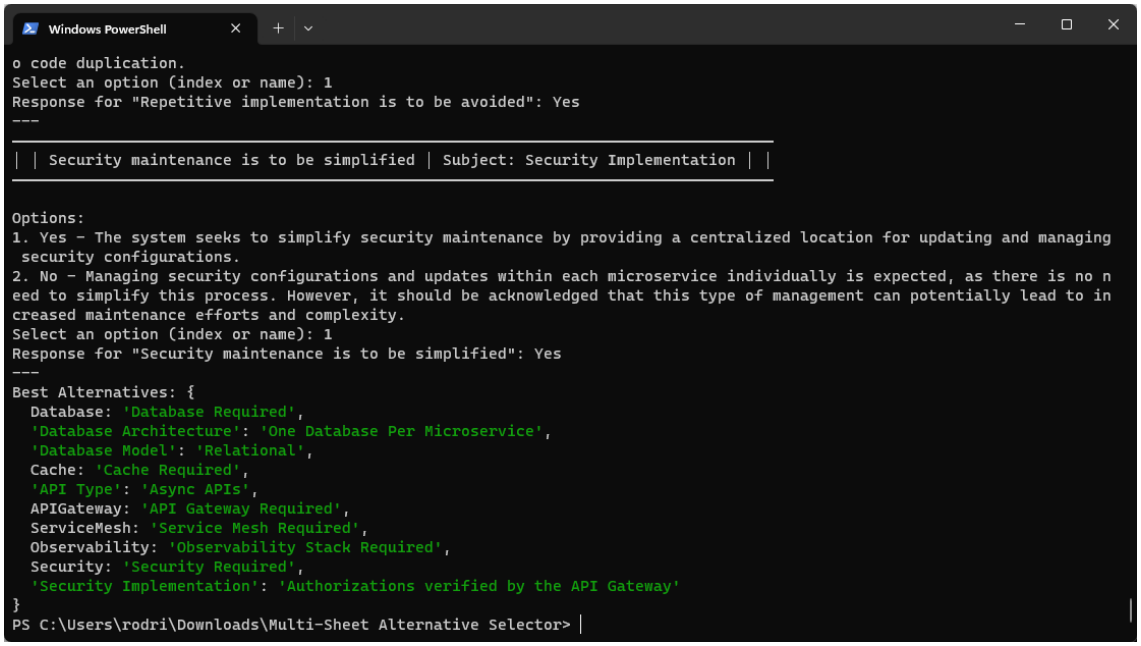
(b) Configuring the description matrices

Figure A.1: Modifying the Matrices

Running the Tool in Normal Mode



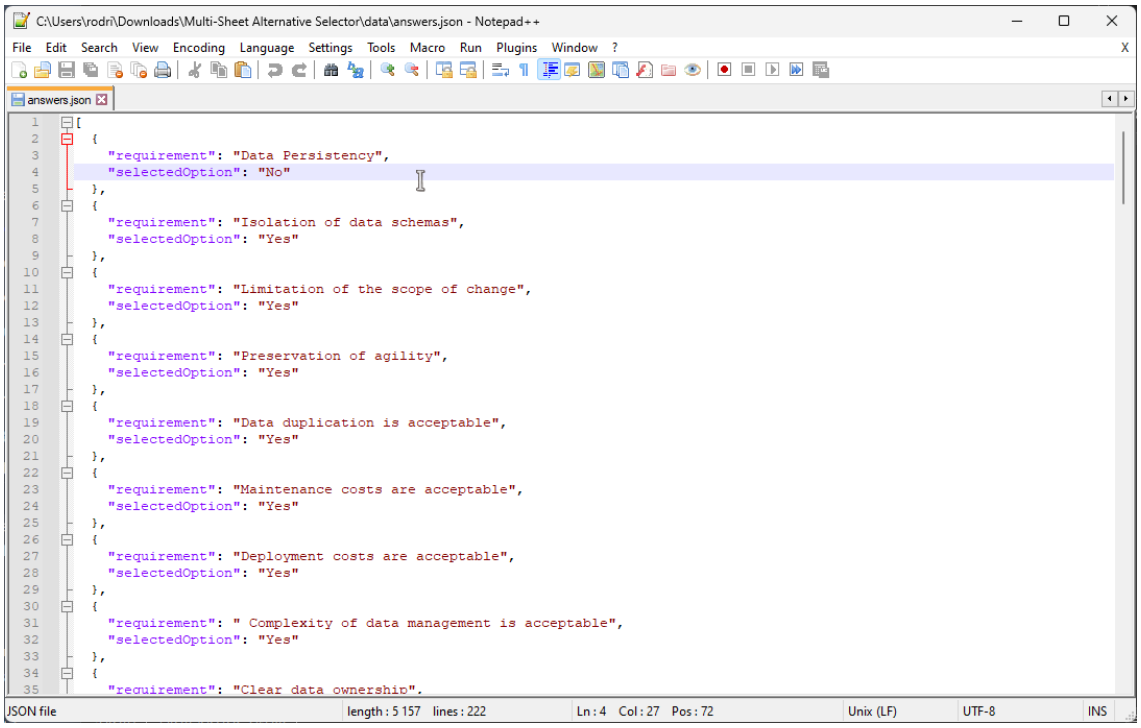
(a) Running the Tool in Normal Mode and answering the questions



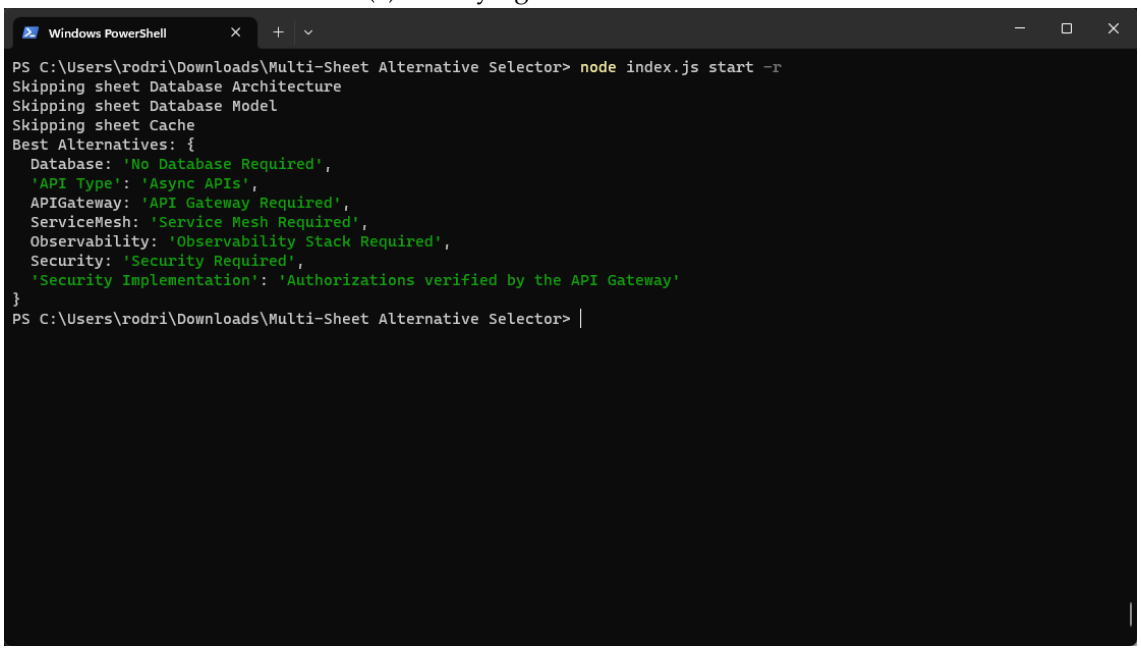
(b) Finishing answering the questions and receiving the recommended alternatives

Figure A.2: Running in Normal Mode

Running the Tool in Summary Mode



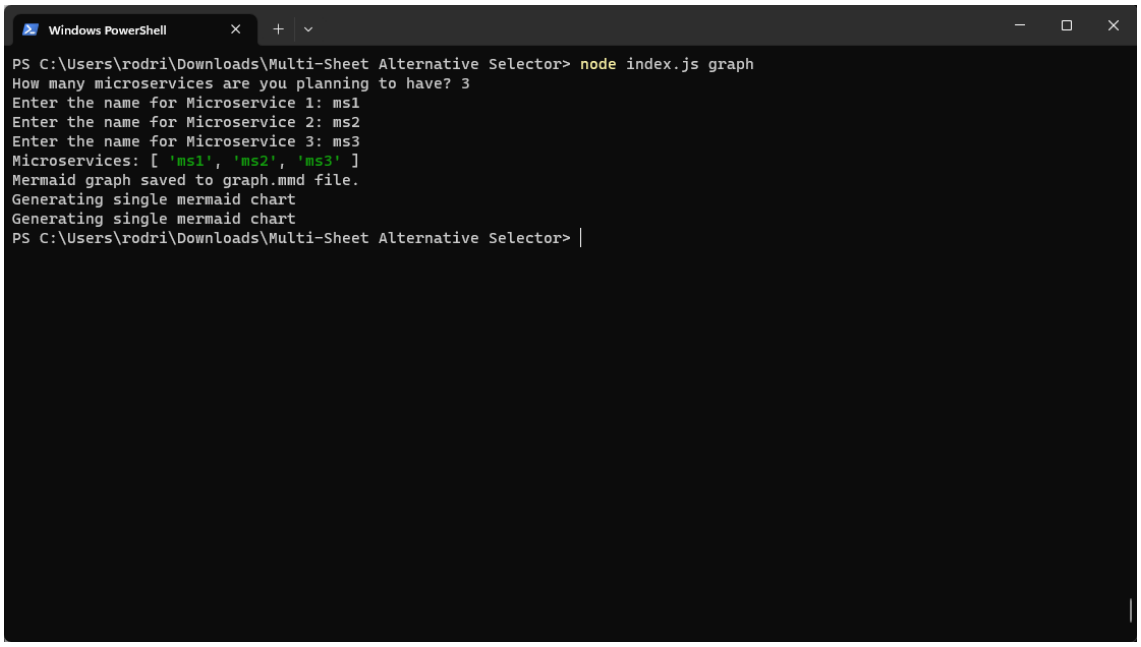
(a) Modifying some of the answers



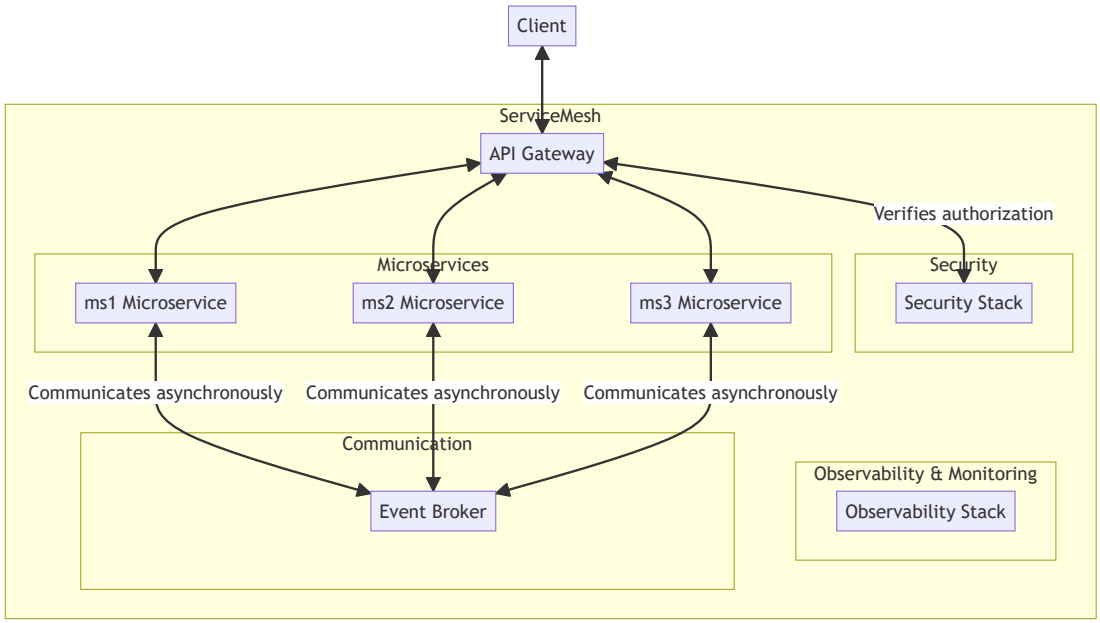
(b) Running the Tool in Summary Mode

Figure A.3: Running in Summary Mode

Generating the Dynamic Graph



(a) Modifying some of the answers



(b) Running the Tool in Summary Mode

Figure A.4: Generating the Dynamic Graph

I

Multi-Sheet Alternative Selector Feedback Survey

This annex contains the Multi-Sheet Alternative Selector Feedback Survey form that was presented to the users.

Multi-Sheet Alternative Selector Feedback Survey

This survey is designed to gather insights on your satisfaction with the recommendations provided by our thesis project.
Your responses will help us understand how well the project met your expectations and identify areas for improvement.
Your input is instrumental in enhancing the project's effectiveness, and we greatly appreciate your participation.

To download the program:
https://drive.google.com/file/d/1lDydRyYm07rI7DQMa_lEp7HBRwGBZzPH/view?usp=sharing

Steps to start running the program:

Using Node:
1 - Unzip it
2 - Run "npm install" to install the dependencies
3 - Run "node index.js start" to run the program in the "normal mode"
4 - Run "node index.js start -r" to run the program in the "summary mode"
5 - Run "node index.js graph" to dynamically generate a graph that represents the suggested architecture

Using Docker:
1 - Unzip it
2 - Run "docker-compose run start" to run the program in the "normal mode"
3 - Run "docker-compose run resume " to run the program in the "summary mode"
4 - Run "docker-compose run graph" to dynamically generate a graph that represents the suggested architecture

You can check the generated graph and the json file with the results (amongst other generated files) in the folder called "data".

Note: You can use the "summary mode" to run the project using the answers.json file generated after running in the "normal mode" for the first time. This is useful if you want to modify one of the answers and run the program again without having to answer all of the questions again.

Thank you for taking the time to provide your feedback!

Inicie sessão no Google para guardar o seu progresso. Saiba mais

* Indica uma pergunta obrigatória

Figure I.1: Survey Form - Page 1

Name: *

Sua resposta

Name of the project that you are involved into: *

Sua resposta

Did the results align with your expectations? *

- ◯ Yes
- ◯ No

On a scale from 1 to 5, how satisfied were you with the recommendation results? *

	1	2	3	4	5	
Very Dissatisfied	◯	◯	◯	◯	◯	Very Satisfied

Figure I.2: Survey Form - Page 2

Were there any specific alternatives where you felt the recommendation was particularly effective? Please provide examples. *

- [ ] Database
- [ ] Database Architecture
- [ ] Database Model
- [ ] Cache
- [ ] API Type
- [ ] APIGateway
- [ ] ServiceMesh
- [ ] Observability
- [ ] Security
- [ ] Security Implementation

Were there any specific alternatives where you felt the recommendation provided less satisfactory results? Please provide examples. *

- [ ] Database
- [ ] Database Architecture
- [ ] Database Model
- [ ] Cache
- [ ] API Type
- [ ] APIGateway
- [ ] ServiceMesh
- [ ] Observability
- [ ] Security
- [ ] Security Implementation

Figure I.3: Survey Form - Page 3

Were there any aspects that you found to be insufficiently explored or developed? *
If so, please specify which topics and what you would like to see improved.

- [ ] Database
- [ ] Database Architecture
- [ ] Database Model
- [ ] Cache
- [ ] API Type
- [ ] APIGateway
- [ ] ServiceMesh
- [ ] Observability
- [ ] Security
- [ ] Security Implementation

Do you have suggestions for any changes or enhancements to the recommendation algorithm or the way it is presented that could make it more useful or user-friendly?

Sua resposta

Were there any challenges or difficulties you encountered while using the tool? If so, please describe them.

Sua resposta

Do you have any additional comments or feedback that could help improve the multi-sheet alternative selector?

Sua resposta

Figure I.4: Survey Form - Page 4