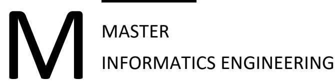


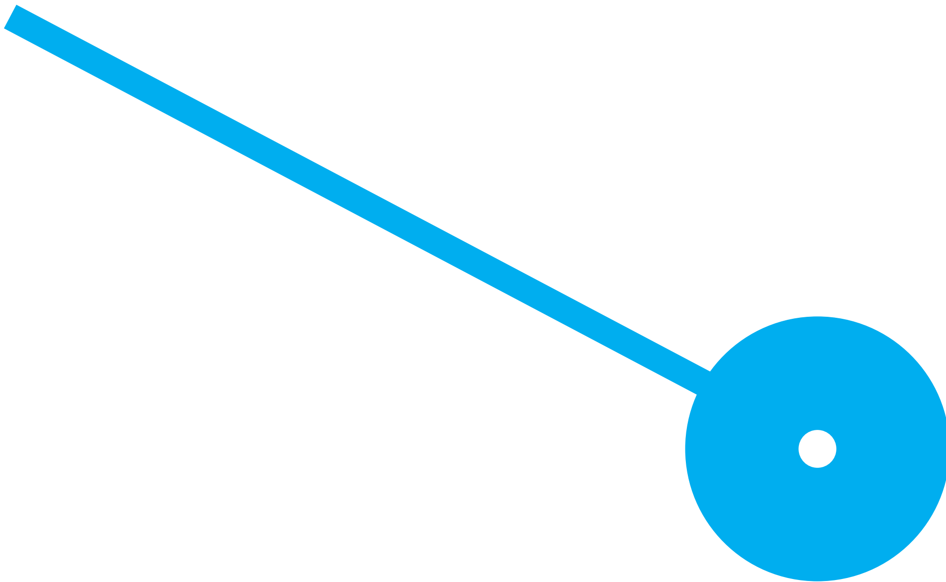
SCHOOL OF
MANAGEMENT AND
TECHNOLOGY
POLYTECHNIC OF
PORTO

M MASTER
INFORMATICS ENGINEERING

Microservices Orchestration vs. Choreography: A Comparison and Analysis

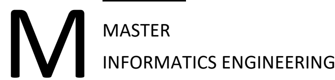
David Miguel Sousa Marques

OCTOBER/2024



P.PORTO

SCHOOL OF
MANAGEMENT AND
TECNOLOGY
POLYTECHNIC OF
PORTO



Microservices Orchestration vs. Choreography: A Comparison and Analysis

David Miguel Sousa Marques
8190565

Advisor(s)

PhD Ricardo Jorge da Silva Santos

Dissertation submitted in fulfilment of the requirements for the master's degree in informatics engineering in the School of Management and Technology of the Polytechnic of Porto.

OCTOBER/2024

Integrity Statement

I, **David Miguel Sousa Marques**, student no. **8190565**, of the Master's Degree in Informatics Engineering at the School of Management and Technology of the Polytechnic of Porto, declare that I have not plagiarised or self-plagiarised; therefore, the work titled "**Microservices Orchestration vs. Choreography: A Comparison and Analysis**" is original and of my own authorship, not having been used previously for any other purpose. I further declare that all sources used are cited in the text and in the final bibliography, according to the referencing rules adopted in the institution.

Acknowledgments

I would like to express my deepest gratitude to all those who have supported me throughout the journey of completing this thesis.

First and foremost, I am sincerely thankful to my advisor, Ricardo Santos, for the invaluable guidance, expertise, and constant encouragement. The insight and patience were instrumental in shaping this research. I am grateful for the time he invested in discussing ideas, reviewing drafts, and providing constructive feedback that greatly enhanced the quality of this work.

A special thanks to my colleagues and fellow researcher, Maria Tavares, for her collaboration and the stimulating discussions. Sharing this academic journey with her made the process much more enjoyable and fulfilling.

I owe a heartfelt thanks to my family, especially my parents, Pedro Marques and Fátima Sousa, for their unwavering support, love, and encouragement. Without their belief in me, this achievement would not have been possible.

To everyone who contributed to the completion of this thesis, both directly and indirectly, thank you from the bottom of my heart. This work is as much yours as it is mine.

> The key is not to see what no one has ever seen, but to think about something that everyone else sees in a way that no one has ever considered before.
>
> Arthur Schopenhauer

Abstract

This dissertation explores a critical evaluation of orchestration and choreography in microservices architecture, with particular attention to how these elements affect implementation complexity, latency, and resilience. Given the growing importance of microservices in modern software development, it is critical for developers and architects to comprehend these architectural principles. The study uses a mixed-methods approach to collect data on the efficacy of each approach in practical applications, including qualitative interviews with industry practitioners and the implementation of a solution based on a real-world scenario.

The results indicate that while orchestration enables more control over error management and process integrity, choreography provides improved scalability and service independence. The centralisation of orchestration can lead to weaknesses such the possibility of a single point of failure and, in some cases, a rise in latency. This present paper highlights how crucial it is to align architectural decisions with system specifications and provides an overview for visual decision-making that shows the considerations associated when deciding between orchestration and choreography.

Keywords: Microservices Architecture · Orchestration · Choreography · Analysis · Industry Insights · Development Practices

Resumo

Esta dissertação explora uma avaliação crítica da orquestração e da coreografia na arquitetura de microsserviços, com especial atenção à forma como estes elementos afectam a complexidade da implementação, a latência e a resiliência. Dada a importância crescente dos microsserviços no desenvolvimento moderno de software, é fundamental que os programadores e arquitectos compreendam estes princípios arquitectónicos. O estudo utiliza uma abordagem de métodos mistos para recolher dados sobre a eficácia de cada abordagem em aplicações práticas, incluindo entrevistas qualitativas com profissionais do sector e a implementação de uma solução baseada num cenário real.

Os resultados indicam que, embora a orquestração permite um maior controlo da gestão de erros e da integridade dos processos, a coreografia proporciona uma maior escalabilidade e independência do serviço. A centralização da orquestração pode criar vulnerabilidades, como o risco de um ponto único de falha, além de aumentar a latência em determinados cenários. O presente documento realça a importância de alinhar as decisões arquitetónicas com as especificações do sistema e oferece uma visão geral, facilitando a tomada de decisões com base nas considerações relativas à escolha entre orquestração e coreografia.

Keywords: Arquitetura de Microserviços · Orquestração · Coreografia · Análises · Perspetivas de Indústria · Práticas de Desenvolvimento

Contents

List of Figures viii

List of Tables ix

List of Listings ix

1 Introduction 1

2 Literature Review 2
- 2.1 State of Art . . . 2
  - 2.1.1 Microservices Archictetur . . . 3
  - 2.1.2 Design Patterns . . . 8
  - 2.1.3 Saga Pattern . . . 19
  - 2.1.4 Choreography in Microservices . . . 21
  - 2.1.5 Orchestration in Microservices . . . 23
  - 2.1.6 Comparative Analysis: Choreography vs. Orchestration . . . 26
  - 2.1.7 Practical Implementations and Case Studies . . . 27
- 2.2 Related Work . . . 31

3 Design and Specifications 33
- 3.1 Problem Overview . . . 33
- 3.2 Requirements . . . 34
  - 3.2.1 Functional Requirements . . . 34
  - 3.2.2 Non-functional Requirements . . . 35
- 3.3 Architecture . . . 35
  - 3.3.1 Technology Stack . . . 37
- 3.4 Prototype . . . 41
  - 3.4.1 Choreography Sagas . . . 45
  - 3.4.2 Orchestration Sagas . . . 49

4 Validation and Results 57
- 4.1 Methodology . . . 57
- 4.2 Evaluation of Approaches . . . 59
  - 4.2.1 Flexibility, Scalability and Service Independence . . . 60
  - 4.2.2 Monitoring and Management . . . 61
  - 4.2.3 Resilience and Failure Management . . . 62
  - 4.2.4 Latency . . . 63
  - 4.2.5 Implementation Complexity . . . 63

4.3 Validation Overview 65

5 Conclusion **70**

5.1 Future Work 71

6 Bibliography **72**

A HTTP Codes and Handling Exceptions **78**

List of Figures

2.1 Generic representation of microservices architecture style. Adapted from [1]. . . . 4
2.2 Representation of the Saga Pattern. Adapted from [2]. 19
2.3 Representation of the choreography saga model. Adapted from [2]. 21
2.4 Generic choreography workflow. Adapted from [3]. 22
2.5 Representation of the orchestration saga model. Adapted from [2]. 24
2.6 Generic orchestration workflow. Adapted from [3]. 25

3.1 Proposed Microservice Archicteture. Source: author. 36
3.2 Sequence diagram for the saga of creating a new user. Source: author. . . . 41
3.3 Sequence diagram for the saga of new client's order process. Source: author. 42
3.4 BPMN diagram for the user creation saga. Source: author. 50
3.5 Sequence diagram for the orchestrated saga of creating a new user. Source: author. 52
3.6 BPMN diagram for the order creation saga. Source: author. 53
3.7 Sequence diagram for the orchestrate saga of new client's order process. Source: author. 56

4.1 Answers from the survey on the familiarity with the microservices architecture of participants. Source: author. 59
4.2 Answers from the survey on the roles of participants. Source: author. . . . 59
4.3 Answers from the survey about the scalability in a production environment. Source: author. 61
4.4 Answers from the survey about the monitoring and management of microservices. Source: author. 62
4.5 Answers from the survey about the resilience and failure management of microservices. Source: author. 63
4.6 Answers from the survey on the familiarity with choreography approach tools. Source: author. 64
4.7 Answers from the survey on the familiarity with orchestration approach tools. Source: author. 65
4.8 A network graph to aid in decision-making for simpler saga when comparing choreography and orchestration. Source: author. 68
4.9 A network graph to aid in decision-making for complex saga when comparing choreography and orchestration. Source: author. 69

List of Tables

2.1 Analysis of distinct architectural styles and how they connect to microservices. Source: author. 6
2.2 Microservices Design Patterns. Source: author. 9
2.3 Comparative analysis of choreography and orchestration approaches in microservices architecture. Source: author. 26

3.1 Functional system requirements. Source: author. 34
3.2 Non-functional system requirements. Source: author. 35
3.3 System architectures modules. Source: author. 36

4.1 Summary of test scenarios. Source: author. 57

A.1 HTTP response codes and their corresponding error messages returned by services. Source: author. 78

List of Listings

3.1 A sample of an OpenAPI configuration for menu service. Source: author. . 43
3.2 A sample of a dish DTO in the menu service. Source: author. 44
3.3 A sample of a user entity using the Panache framework. Source: author. . 44
3.4 A sample of a menu repository using the Panache framework. Source: author. 45
3.5 A sample of the implementation of the user service in the choreography approach. Source: author. 46
3.6 A sample of the code validation and status update method. Source: author. 46
3.7 A sample of the menu for the day method. Source: author. 47
3.8 A sample of the implementation of the order service. Source: author. . . . 47
3.9 A sample of the implementation for checking the user's available funds. Source: author. 48
3.10 A sample of the code of the create user method in orchestration approach. Source: author. 49
3.11 A sample of the code of the create user listener. Source: author. 50
3.12 A sample of the code of the user activation request. Source: author. 51
3.13 A sample of the code of the create order method in orchestration approach. Source: author. 52
3.14 A sample of the code of the wallet deduct request. Source: author. 53
3.15 A sample of the code of the order tracking stage. Source: author. 54
3.16 A sample of the code of the insufficient funds path. Source: author. 55
A.1 Java implementation of the DataException class handling error codes and messages within the service. Source: author. 78
A.2 Java implementation of the ErrorCode enum mapping HTTP codes and messages to specific error conditions. Source: author. 79

Acronyms

API Application Programming Interface. 3, 11, 12, 17, 23, 26, 33, 35, 38, 40, 42–44, 50, 51, 53

AWS Amazon Web Services. 38

BCE Boundary-Control-Entity. 42

BPMN Business Process Model and Notation. viii, 39, 50, 51, 53

BSON Binary JSON. 40

CCP Common Closure Principle. 9

CI/CD Continuous Integration and Continuous Delivery. 40

CPU Central Processing Unit. 31, 32

DDD Domain Driven Design. 10

DMN Decision Model and Notation. 39

DNS Domain Name System. 17

DTO Data Transfer Object. x, 44

HTTP Hypertext Transfer Protocol. 6, 33, 47, 51, 53, 78

IT Information Technology. 5

JSON JavaScript Object Notation. 5

LLT Long Lived Transaction. 19

MSA Microservices Architecture. 1, 3, 8, 26, 27

MTTD Mean Time To Detect. 16

MTTR Mean Time To Repare. 16

MVC Model-View-Controller. 38

PaaS Platform as a Service. 4

REST Representational State Transfer. 3, 6

URI Uniform Resource Identifier. 11

WAN Wide Area Network. 22, 26

XML Extensible Markup Language. 5

Chapter 1

Introduction

Software development has experienced a major shift in recent years, with microservices being the most common approach to developing scalable and flexible systems. The term Microservices Architecture (MSA) has gained popularity as a way to describe a novel approach that structures systems as collections of deployable, autonomous applications. Development teams can quickly make changes because of this modularity, satisfying the market's increasing demand for agility.

Microservices come up to be a useful option for organisations trying to find immediate solutions to adapt to evolving market conditions. Application segmentation into separate components promotes innovation by enabling experimentation and user-need-driven adaptation, besides enabling frequent updates and continuous deployment.

There are two main approaches for integrating microservices: choreography and orchestration. Through orchestration, control is centralised in the custody of a leader who manages how many services interact. However, choreography enables direct communication across services by triggering coordination and actions using event-based techniques. Both approaches have important effects on a system's scalability, maintenance, and efficiency. The subtle distinctions between these approaches and how they affect distributed software development will be examined in this study.

The objective of the present work is to conduct a comprehensive investigation of orchestration and choreography approaches inside microservice environments, highlighting their benefits and drawbacks concerning scalability, maintenance, and performance. The study will concentrate on a pratical scenario, offering insightful information to help determine the best option to take for different contexts.

Our research's findings should deepen understanding of the decisions that influence microservices architecture and help professionals choose integration strategies that increase productivity, minimise costs, and encourage creativity.

Chapter 2

Literature Review

The progression of software architecture has resulted in numerous paradigms designed to enhance scalability, flexibility, and maintainability. Notably, microservice architecture has become a leading method, widely embraced across different sectors. This chapter seeks to offer a thorough examination of the literature surrounding microservices, concentrating on essential concepts and methodologies explored by both researchers and practitioners.

This section will examine the latest technical progress and current discussions within the research community, highlighting trends, identifying gaps, and pointing to potential future research areas. The goal of this chapter is to lay a robust groundwork for comprehending the present landscape of microservice architecture, thereby paving the way for more detailed analyses in the following chapters.

2.1 State of Art

Microservice architecture has attracted significant interest from both academics and industry professionals due to its potential to revolutionise software development and deployment processes. This architectural approach involves decomposing complex applications into smaller, independently deployable services, each handling a particular set of business processes.

Given the complexity of microservices orchestration and choreography, a variety of approaches and strategies have been discussed in academic literature. The debate in the research community about microservices orchestration *versus* choreography has generated considerable enthusiasm, paralleling investigations of architectural techniques in software engineering. Researchers and practitioners explore the advantages, limitations, and ideal scenarios for using each strategy.

The surge in interest has led to an increase in the number of architectural patterns in software, necessitating some form of classification. Software architecture, which emerged in the 1980s, has evolved into a sophisticated field employing a variety of tools, notations, and approaches. Sharma [4] presents several architectural styles based on the type of application, such as shared memory, distributed systems, messaging, structure, adaptable systems, and modern systems.

For shared memory applications, relevant architectural styles include: (1) blackboard, characterised by an autonomous system of agents working together; (2) data-centered, which emphasises centralised data management; and (3) rule-based, which promotes the application of rule-based logic.

Distributed systems feature a variety of architectural styles, including: (4) client-server, which denotes a clear separation of duties between the client and server; (5) space-based architecture, which emphasises resource and space sharing; (6) peer-to-peer, encouraging equivalent node cooperation; (7) shared-nothing architecture, distinguished by the absence of shared resources; (8) broker, which facilitates component communication; (9) Representational State Transfer (REST), which uses simplified communication techniques; (10) service-oriented, based on the development of autonomous and interoperable services; and (11) microservices, which decompose an application into a collection of isolated and independent services.

For messaging applications, relevant styles include: (12) publish-subscribe, based on the selective distribution of information; (13) event-driven, which reacts to and triggers events; (14) asynchronous messaging, prioritising asynchronous communication; and (15) stream-based, using constant data streams to facilitate communication.

In the structural domain, several styles are pertinent: (16) component-based, which encourages modularity and reusability; (17) pipes and filters, which divide data flow into distinct stages; (18) monolithic, characterised by a single consolidated structure; (19) layered, which arranges components into different layers; and (20) call-and-return architecture, which separates the calling and returning components.

Adaptable systems include styles such as: (21) plugins, enabling modular expansions; (22) reflection, which allows dynamic system analysis and modification; and (23) microkernel, which emphasises a simple core with extensible functionalities.

Lastly, modern systems employ architectural styles such as: (24) architecture for big-data, suitable for the efficient processing of large volumes of data; (25) multi-tenancy architecture, supporting multiple users in isolation; and (26) architecture for grid computing, optimised for distributed processing.

Before diving into an analysis of these architectural styles, their applications, and their relation to microservices, acquiring a sophisticated understanding of microservices is essential. This sets the stage for a comprehensive analysis of the architectural styles and their relevance in the context of microservices (see Table 2.1).

2.1.1 Microservices Archicteture

The microservices architecture style consists of an ecosystem of independently deployed, single-purpose services that are typically accessed via an Application Programming Interface (API) gateway. According to Lewis and Fowler [5], the term "Microservices Architecture (MSA)" has gained popularity in recent years to describe a specific method of structuring software systems as independently deployable sets of services.

Based on Newman [6], the microservices paradigm involves developing software systems that are autonomously deployable and coherent, with each system performing an individual purpose. According to him, every microservice is a self-contained entity that can be deployed on its own as an independent operating system process or as an isolated service

on a Platform as a Service (PaaS). This crucial characteristic enables these services to develop independently, allowing for efficient deployment without requiring changes from users. The core of Newman's concept is this inherent autonomy and modularity, which transforms modern application development by promoting increased independence and self-sufficiency within the system.

Figure 2.1 illustrates the fundamental design of the microservices architecture style. User interface queries activate well-defined endpoints in an API gateway, which subsequently routes the user request to independently deployed services. Each service subsequently utilises its own data or requests access to data owned by other services. The main function of the API gateway is to mask the implementation and location of the corresponding services that map to the API gateway's endpoints. Additionally, the API gateway may perform infrastructure-related functions such as security and metrics collection.

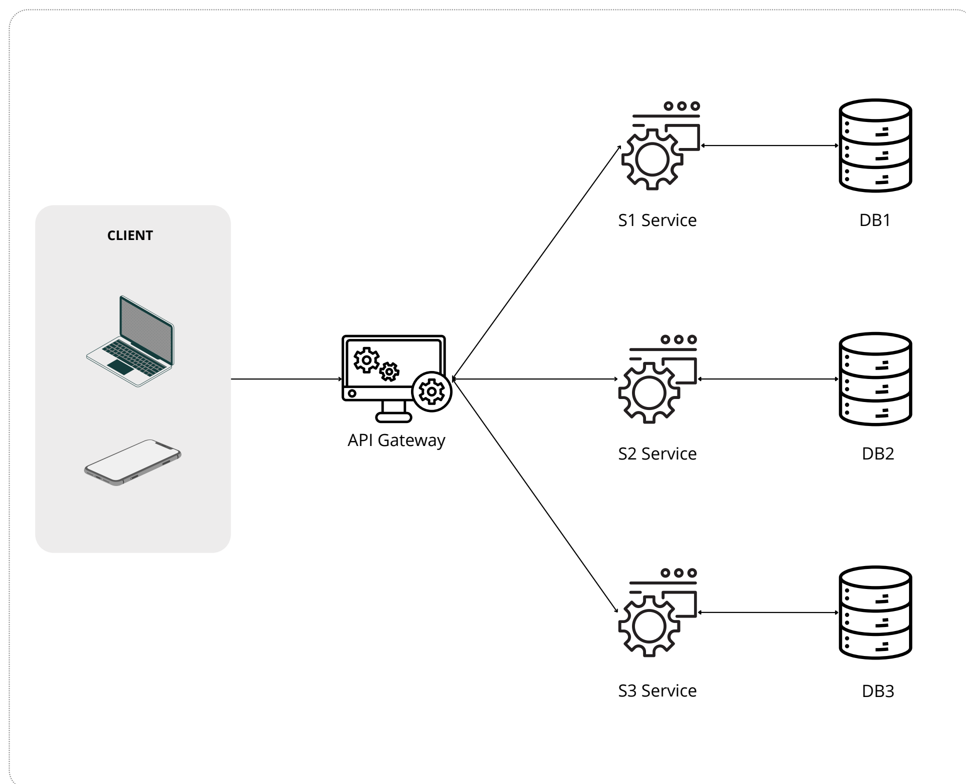


Figure 2.1: Generic representation of microservices architecture style. Adapted from [1].

Despite the illustration in Figure 2.1 showing each service linked to a different database, this is not always the case. The key point here is that data can only be accessed and updated by the service that owns the tables. Other services cannot access the tables directly; instead, they must request access to the data from the owning microservice. This concept is known as a *bounded context*, a term introduced by Eric Evans in his book [7]. Bounded context in microservices not only promotes architectural adaptability but also controls changes within the microservices ecosystem. When structural data changes, the bounded context requires only the service that owns the data to make the necessary adjustments.

Advantages and Disadvantages

Now that we've outlined the structure of microservices, let's examine their benefits and drawbacks in detail. The potential benefits of the microservices approach are extensive and impact a diverse range of industries, from logistics to Information Technology (IT) and more. Authors like [5, 6, 8] highlight the following benefits:

- The adoption of a system containing multiple cooperating services enables the integration of various technologies in each component. This adaptability is useful when trying to improve particular features of the system, such as performance. A multiservice system provides numerous separate environments to test and integrate new technologies, unlike monolithic applications where adopting a new database, programming language, or framework can have unintended consequences.

- In the face of component failures, the system can identify and address issues without affecting overall functionality. Although deploying a monolithic system across multiple machines can reduce the probability of failure, microservices empower the creation of highly resilient systems that can handle complete service failures, adjusting their operations dynamically.

- The ability to modify and deploy individual services independently of the rest of the system facilitates faster code deployment. In the event of an issue, it can be quickly pinpointed to a specific service, facilitating quick reversion as opposed to a monolithic application, which has a significant effect and high risk because every small modification necessitates the deployment of the entire software.

- Microservices enable a more precise alignment of architecture with organisational structure. This helps minimise the number of individuals working on a single codebase, optimising team size for enhanced productivity.

- Separate testing is possible on modified and impacted components of the architecture, allowing for more accurate evaluation of changes. This approach also enables one to modify the scope of testing in accordance with the magnitude of changes.

Like any architectural strategy, microservices come with their own set of challenges and considerations. It is crucial to carefully balance any potential impacts with the benefits. Several authors, including [5, 8], have identified notable disadvantages that warrant thoughtful consideration:

- Programming distributed systems is more difficult since remote calls are error-prone and slow.

- Ensuring strong consistency in distributed systems is extremely difficult, making ultimate consistency a shared responsibility.

- Managing a variety of services that are frequently redeployed requires a skilled operations team.

- Due to network latency, communication across a network in a microservices architecture may result in performance reductions compared to in-memory call mechanisms.

- The prevalent use of Extensible Markup Language (XML) and JavaScript Object Notation (JSON) as primary data-interchange formats in microservices necessitates additional security measures for data encryption and authentication with third-party

services. Consequently, integrating authentication methods with external services and guaranteeing the safe storage of sent data poses additional challenges.

- The diverse technologies employed in microservices generally employ distinct methods for delineating contracts related to service composition, which can result in ambiguous situations and clumsy client development.

2.1.1.1 Architectural Styles and Their Impact on Microservices

Now that we have a solid knowledge of microservice architecture, we can focus on examining how different architectural styles relate to and influence the microservices ecosystem. The purpose of this research is to clarify the different methods for software system structuring and how they affect the creation, implementation, and scalability of microservices.

Table 2.1 offers a thorough analysis of the most popular architectural styles, revealing their applications and meaning within the microservices paradigm.

Table 2.1: Analysis of distinct architectural styles and how they connect to microservices. Source: author.

Architectural style	When to use	When not to use	Impact on Microservices
Client-Serve	When there must be a clear division of responsibility between the client and server. When centralised control is feasible and moderate scalability is required.	In distributed data applications or those requiring high scalability. In low-latency, real-time applications.	Helps define clear service boundaries, but could be too rigid for highly dynamic systems.
Space-Based	When extremely high levels of concurrent scalability or elasticity are required. In systems needing high performance and responsiveness, using in-memory caching for faster data access.	For applications with large transactional data volumes due to memory constraints. When there are tight budget and time constraints due to technical complexity. When faster adaptability to change is necessary.	Supports microservices by enhancing scalability and performance, critical for handling large-scale distributed systems.
Representational State Transfer	For simple, stateless HTTP communication between clients and servers. In resource-centric applications with loose component coupling.	In applications requiring tight component coupling or complex operations. In highly interactive, real-time applications.	Commonly used in microservices for its simplicity and stateless nature, facilitating independent service communication.

Service-Oriented	In large, complex systems requiring loosely coupled modular services. For applications integrating multiple languages and technologies.	In small-scale applications with simpler architectures. For projects with simple requirements where it may be overly complex.	Provides a foundation for microservices by promoting modular, loosely coupled services.
Publish-Subscribe	In systems requiring high performance, scalability, and fault tolerance. In business transactions that react to systemic and external triggers.	Where most processing is user data requests or simple CRUD tasks. In applications requiring data consistency and simultaneous processing.	Enhances microservices by enabling asynchronous communication and decoupling of services.
Event-Driven	In systems needing fault tolerance, scalability, and high performance. For handling business transactions responding to internal and external events. In applications managing complex, non-deterministic workflows.	Where user data requests or basic CRUD activities predominate. When synchronous processing and high data consistency are required. When precise control over event timing and workflow is necessary.	Supports microservices by promoting loose coupling and scalability through event-based communication.
Component-Based	For developing reusable and modular components. In large systems with independently developed and maintained components.	In smaller applications with simpler architectures. In applications with overheads related to managing component interactions and dependencies.	Facilitates microservices by allowing independent development and deployment of components.
Pipes and Filters	In applications with specific data processing flows divided into small segments. For creating modular and reusable filters for data processing.	In applications involving multiple delicate stages. In applications with linear data processing workflows, as it may add overhead.	Supports microservices by promoting a modular approach to processing data streams.
Monolithic	In larger projects with simpler architectural designs. Where quick development and implementation are primary goals.	In large-scale applications with complex and changing needs. Where maintenance and scalability challenges arise as the application grows.	Less suitable for microservices due to lack of modularity and scalability. Often a starting point before transitioning to microservices.

Layered	For projects with substantial budget or scheduling limitations. For those preferring monolithic development without distributed systems' challenges. For enabling component separation within specific application levels.	In projects with high performance, fault tolerance, flexibility, and scalability requirements. When most changes affect multiple layers and occur at the domain level. When changes impact several layers, potentially involving multiple teams.	Can be restrictive for microservices due to layered dependencies. Useful for structuring service interactions.
Microkernel	As a base for product-based or custom applications with planned extensions. In applications with multiple configurations based on client environments or deployment strategies. For easy understanding and cost-effectiveness. In situations with limited resources and time constraints.	In highly elastic and scalable systems, as the core system is a bottleneck processing all requests. In projects requiring high fault tolerance. Where most changes are to the core system without using plugins for other functionalities.	Useful for microservices by providing a modular core with extensible plugins, facilitating adaptability.

The widespread adoption of MSA has certainly transformed software development, offering unmatched flexibility and scalability. Nevertheless, this architectural approach has its disadvantages, particularly related to database design and transaction management. We will explore these challenges more comprehensively later in the discussion, with a particular emphasis on database design and transaction management.

2.1.2 Design Patterns

As we move forward, it's important to note that the prevalence of microservices-based architectures in today's software development has increased the demand for established design patterns to handle the particular difficulties posed by this architectural style. Effective solutions to frequent challenges that occur when breaking down monolithic applications into smaller independent services can be found in microservices architecture patterns.

There are several benefits to the microservices architectural style, including enhanced fault tolerance, scalability, and flexibility. However it also brings with it additional challenges related to data management, observability, cross-cutting issues, and service-to-service communication. These challenges can be mitigated by employing microservice design patterns, which encourage best practices to create highly effective, maintainable, and reliable microservice-based systems.

Several of the microservice design pattern types are displayed in Table 2.2.

Table 2.2: Microservices Design Patterns. Source: author.

Type	Description	Advantages	Disvantages
Decomposition Patterns	Based on business capabilities, subdomains, or the strangler pattern, these designs provide guidelines on how to logically break down a monolithic application into more manageable and independent microservice.	Enhanced scalability, flexibility and fault tolerance	Increasingly difficult to coordinate and manage several services
Integration Patterns	The challenges of service-to-service communication are addressed by integration patterns like the API Gateway pattern, which provides clients with a single point of access to multiple microservices.	Simplified client interaction Centralized cross-cutting concerns management	Potential performance bottleneck Single point of failure if not designed properly
Database Patterns	Database patterns, such as the Database per Service and Shared Database per Service patterns, are useful in determining the most appropriate data management approach for microservices.	Improved autonomy and scalability for individual services Reduced complexity for shared data	Potential data consistency issues Tight coupling between services
Observability Patterns	Observability patterns, which include Distributed Tracing and Health Check patterns, focus on tracking, recording, and monitoring microservices to guarantee their functionality and health.	Improved visibility into system behavior and performance	Additional complexity in implementing and managing observability tooling
Cross-Cutting Concern Patterns	Cross-cutting concern patterns cover shared needs, such as Blue-Green Deployment, Circuit Breaker, and Service Discovery, that cut across several microservices.	Consistent implementation of common requirements Improved reliability and resilience	Potential for increased architectural complexity

2.1.2.1 Decomposition Patterns

Beginning with decomposition patterns, these offer instructions on how to divide a monolithic application into more manageable and independent microservices. Some of the most prominent patterns of decomposition are:

2.1.2.1.1 Decompose by Business Capability

One architectural approach that groups microservices according to the business processes they perform is called Decomposing by Business Capability. By aligning microservices with business demands, this method ensures that every service reflects a specific company feature. Rather than focusing on how an organization completes tasks, business capabilities are stable entities that describe what an organization does to create value.

There are several benefits when employing the Decomposition by Business Capability Pattern; due to the tendency of business capabilities to be consistent over the years, it promotes a stable design where development teams become independent, cross-functional groups that prioritise providing commercial value over just technical features. By adhering to the Common Closure Principle (CCP), which states that modifications to one business capability should be isolated within its associated service, this approach improves

cohesion and loose coupling of services. Each service covers a small range of closely linked operations, and teams can also individually design, implement, and test their services, which encourages agility and minimises dependency on other teams [9, 10].

However, there are concerns with this pattern. Identifying and defining an organisation's business skills can be challenging and exhausting, requiring a thorough understanding of its knowledge, processes, and organisational structure. If business capabilities overlap or are not clearly defined, there is a risk of tight coupling between microservices, complicating the replacement or modification of individual services without impacting the entire system. Furthermore, it might be difficult to coordinate several microservices to serve a single business activity, especially if the capability requires multiple stages or interfaces with other systems [9, 10].

Despite these obstacles, a well-implemented decomposition by business capacity strongly integrates software development with business goals, making sure that vital business operations receive priority and improving the user experience.

2.1.2.1.2 Decompose by Subdomain

The Decomposition by Subdomain Pattern employs Domain Driven Design (DDD) to break down a monolithic application into smaller, more manageable microservices. In this approach, the business domain is separated into multiple subdomains, each of which corresponds to a distinct component of the organisation. These subdomains fall into three categories: generic, supporting, and core.

Generic subdomains address common business tasks that are frequently handled by standard solutions. Supporting subdomains connect to business functions but are not distinctive differentiators. Core subdomains are essential to the business's unique value proposition.

This design has several significant advantages. Microservices aligned with business subdomains make the architecture flexible and stable in the face of business changes, as subdomains are generally stable entities. Development environments become more responsive and agile when teams are cross-functional and independent, focussing on delivering business value rather than technical features. The resulting coherent and loosely connected services enhance the system's scalability, maintainability, and robustness. Each service's separate operation simplifies scaling and upgrading without affecting other system components.

The Decomposition by Subdomain Pattern has significant drawbacks despite its benefits. It requires in-depth knowledge of the business, its organisational structure, and its domain model to identify suitable subdomains; also, the iterative nature of this approach could call for close cooperation between development teams and business stakeholders. Moreover, the development of serveral microservices could delay service discovery and integration, consequently raising operational complexity.

2.1.2.1.3 Strangler Pattern

An effective approach to controlling the gradual transformation of a monolithic application into a microservices architecture is the Strangler Pattern. In this pattern, there are two main categories of services: those that provide new features and those that replicate the

monolith's existing capabilities. Introducing new functionalities with microservices is one of the best ways to demonstrate their real benefits to the organisation. The main goal of this pattern is to mitigate the risks and complexity involved in a complete system overhaul by progressively replacing the monolith by methodically separating its components into microservices.

As a general rule, the Strangler Pattern is applied in stages. The monolithic application is first merged with by a new parallel application. The "*transform*" stage of this process entails gradually developing microservices to handle certain monolithic features. In the "*coexist*" stage, the new and the old applications run concurrently in the same Uniform Resource Identifier (URI) space; the microservices get traffic from the monolith and are gradually redirected to them once they are operational. Finally, in the "*remove*" stage, the application completely switches to the microservices architecture, and the outdated monolithic components are systematically discontinued. This gradual migration minimises user disruption and allows for regular feature releases [11, 12].

There are several benefits to applying the Strangler Pattern, especially for legacy systems in active use. By allowing the old and new systems to operate simultaneously, it facilitates a smooth transition, ensuring that any issues with the new microservices can be resolved without disrupting the entire application. This design is flexible, allowing for front-end improvements as well as back-end restructuring to accommodate microservices; but careful preparation and execution are essential to its success, especially to ensure that both systems have access to the resources they require and that the transfer will not negatively impact user experience [11, 12].

2.1.2.2 Integration Patterns

The challenges of communicating between different services are addressed through integration patterns. Some of the most prominent integration patterns are

2.1.2.2.1 Gateway Pattern

One of the most important design solutions in the world of microservices is the API Gateway Pattern, which addresses the need for server-side aggregation strategies when building end-user applications [13]. By acting as a single point of entry, this pattern optimises system efficiency and simplifies client communications by decreasing the number of client requests, routing them to the appropriate microservices, gathering the required information, and providing a unified response to consumers [13, 14]. It facilitates communication and enhances performance by accomplishing this by encapsulating every detail of the microservices.

Functionally, the gateway serves as the system's central hub, routing, and processing requests to the relevant microservices. It goes beyond basic routing by launching several microservices, combining their output, and offering customised APIs for every customer. Since the Gateway has a broad understanding of the service locations, it can also manage load balancing, monitoring, and static response processing [13]. While it does not focus heavily on publishing or managing services, it creates customised APIs for different platforms, enhancing client application communication and allowing microservices to evolve independently.

One of the many benefits of the Gateway Pattern is its simplicity of growing through customised APIs, which makes it easier to add new functionalities. Focused on market needs, it adjusts services in response to changing customer demands without requiring a complete system overhaul, ensuring backward compatibility and minimising disruption to current users [13].

However, there are some potential drawbacks as well, for instance, the gateway becoming an obstruction if it is not designed effectively; the necessity of numerous interfaces increases implementation complexity; and the need to thoughtfully consider Application Programming Interface reuse when multiple clients share the same API. Additionally, as the number of microservices increases, scalability issues may arise, necessitating more effective routing techniques and dynamic configuration management [13].

2.1.2.2.2 Aggregator Pattern

The Aggregator Pattern is a crucial design technique, particularly useful when a client needs to quickly retrieve data from multiple services. The client submits a single request to an aggregator service, which then collaborates with multiple providers to collect and aggregate the necessary data, rather than sending individual requests to each service [15, 16]. By centralising the data retrieval process, this design substantially decreases the complexity of managing various service requests, consequently streamlining client-side implementation.

To deal with client requests concurrently and efficiently manage interactions with many services, the aggregator service itself has to be carefully designed; scalability, fault tolerance, and resilience must all be carefully taken into account. Although the pattern optimises speed and simplifies client interactions, it requires robust architectural decisions to mitigate the risks related to centralised aggregation [16].

The main advantages of the Aggregation pattern include reducing network chatter and increasing performance by consolidating various service answers into a single cohesive response for the client; this aggregation increases the efficiency of service interactions and simplifies client-side logic. The approach supports quicker microservice scalability and encourages modular design by offering a single endpoint for similar features. Additionally, it enables organisations to maximise resource use by reducing redundant data transfer and enhancing service orchestration [15, 16].

The pattern also presents some drawbacks. It may add complexity to system architecture and maintenance, especially when it comes to making sure that the aggregator service is stable. A fault isolation strategy and thorough monitoring are required since dependence on multiple services creates potential points of failure. To properly employ the Aggregation Pattern in microservices systems, careful use case analysis must be combined with proactive architectural design and execution [15, 16].

2.1.2.2.3 The Chain of Responsibility Pattern

The Chain of Responsibility design pattern offers a structured approach to handling requests by chaining multiple processing objects. Each handler in the chain can process a request or pass it to the next handler in the sequence [17, 18]. This pattern is particularly useful for scenarios where it is unknown at compile time which specific handler will pro-

cess a request, or where different handlers might respond to the request based on runtime conditions. For example, in an image editing application [14], the Chain of Responsibility pattern was used to sequentially resize and transform images. The Adjuster microservice analysed the image size and made the necessary adjustments before passing the data to the Converter microservice for monochrome conversion. This sequential handling enables modular and flexible processing without requiring a strong connection between the request's source and receivers.

The Chain of Responsibility pattern's ability to dynamically distribute processing responsibilities for requests based on runtime conditions is one of its main benefits; because new handlers may be introduced or current ones can be updated without impacting the client who submits the request, this flexibility encourages modularity and extensibility in the software. Additionally, by splitting the senders and the receivers, this design makes system architecture evolution and maintenance easier. One potential drawback, though, is making sure that every request is finally processed somewhere along the chain; if a request fails to be processed by a handler, it may result in unresolved requests, which may need to be carefully planned and designed to prevent [17].

This pattern succeeds in practical applications by providing a smooth transfer of responsibilities across different processing objects, such as sophisticated workflow management tools or library cataloging systems. Also it is a useful addition to the collection of design patterns for current software development since it allows developers to develop scalable systems where the processing of requests may dynamically adjust to changing demands or conditions during runtime [17]. The modularity and flexibility of microservices architectures are enhanced by this structured approach to processing requests through a series of processing objects, which ensures that roles are clearly defined and effectively addressed.

2.1.2.3 Database Patterns

Database standards are another crucial factor that influences which data management strategy is best for microservices. Some of the most prominent patterns of the database are:

2.1.2.3.1 Database per Service Pattern

A key design pattern in microservices architecture is the database per service pattern, which allows each microservice to only access its own private database [13]. The maintenance of distinct databases for every microservice makes this method especially beneficial in facilitating migration from monolithic applications to microservice-based systems. There are several approaches to implement this pattern such as (1) private tables per service, (2) schema per service, and (3) database server per service [19, 20].

1. In the private tables per service approach, each service has a set of tables accessed exclusively by that service. This approach is supported by several database platforms, including MySQL, PostgreSQL, and HyperSQL.
2. The schema per service approach provides a private database schema to every service; however, this choice is also constrained by the features of specific database platforms.

3. The database server per service approach is the most isolated solution, in which every service runs its own database server; the service and its database can simultaneously scale independently inside a database cluster due to this approach, ensuring the highest level of independence and scalability.

This pattern's primary benefits include enhanced security, independent development, and scalability. Scalability can be achieved by individually scaling the database clusters as needed, as each service has its own dedicated database; this encourages independent and parallel development by allowing development teams to operate independently on their services without interfering with others when database schema changes are required. Additionally, the possibility of data tampering by other services is minimised when a single microservice is the only one with database access [13, 21].

The database per service pattern comes with drawbacks too. The difficulty of establishing transactions across many services, each with its own database, is a significant drawback. Furthermore, the demand for multiple database connections and queries might make querying data across various services inefficient. Supervising several relational and non-relational databases contributes to the administrative complexity [16, 21].

Despite these challenges, by allowing each microservice to separately store and access data from its own data store, this design strengthens loose coupling—a fundamental aspect of microservices architecture.

2.1.2.3.2 Shared Database Pattern

The Shared Database Pattern involves multiple microservices that access a single shared database. This approach allows leveraging existing database schemas without significant modifications, which is particularly useful for migrating monolithic applications to a microservices architecture [13]. The principal advantage lies in its simplicity of use and the ability to use recognised ACID[1] transactions to ensure data consistency. This can be extremely important in preserving the system's integrity before and following the migration.

There are a number of disadvantages to this pattern, despite its familiarity and simplicity of usage. Developers must coordinate schema changes across multiple services due to the inherent development-time coupling, which is a significant drawback. Moreover, runtime coupling might happen when several services communicate with the same database, which could cause interference and performance issues, and the flexibility and scalability of the system may be restricted if a single database is incapable of effectively satisfying all the storage and access requirements of every service [20, 22].

Adopting the Shared Database Pattern requires careful consideration, especially when it comes to ensuring that database modifications are backward compatible and evaluating the architecture to prevent hot tables. This technique can be helpful when substantial refactoring of the current codebase is not desired or when the operation of a single database is appreciated. It is also suitable in situations where maintaining data consistency via ACID transactions is crucial, and interdependencies among microservices make

[1] ACID stands for Atomicity, Consistency, Isolation, and Durability. These standards support data reliability and integrity and constitute the basis of database systems transaction management.

the database per service pattern difficult to implement [23]. In the end, even if this pattern is feasible in some situations, it might not be consistent with the fundamental microservices concept of loose coupling.

2.1.2.4 Observability Patterns

In contrast, observability standards focus on tracking, documenting, and maintaining an eye on microservices to make sure they are operating properly. Examples that are relevant are as follows:

2.1.2.4.1 Log Aggregation Pattern

Applications that employ a microservices architecture often consist of many service instances executed on different servers, each producing its own log files, making it challenging to track and understand software responses for particular requests. Effective debugging and monitoring may be hampered by the dispersed nature of logging. The log aggregation approach addresses this issue by consolidating logs from every service instance into a single cohesive platform. This centralised logging service collects, stores and normalises the logs of each microservice, allowing for a complete search, analysis, and alert configuration [12, 24].

This pattern is implemented by configuring a logging structure that routes log data, from multiple service instances, to a centralised logging server. The logs are consolidated on this server, providing a simple search and analysis. Developers can identify problems faster and comprehend the flow of requests between different services by combining logs. In addition, automated alerts can also be set up to warn the team in the event that specific patterns or error messages appear in the logs, which helps maintain system dependability and promote proactive response to problems [12, 24].

In a microservices ecosystem, the Log Aggregation pattern enhances observability and simplifies debugging. Teams can view the entire stack in the production environment using a centralized log repository, making it easier to correlate log data with individual service instances and operational events. Organizations can ensure accurate monitoring, effective incident management, and continuous improvement of the microservices architecture by implementing the Log Aggregation pattern [12, 24].

2.1.2.4.2 Distributed Tracing Pattern

In a microservices architecture, monitoring and debugging requests can become very complicated since they frequently navigate across several services. A client request may be handled by different tasks across many services, resulting in dispersed log entries and fragmented visibility. It may be challenging to detect problems and comprehend how requests navigate the system due to its complexity. The Distributed Tracing Pattern addresses this challenge by assigning each request a unique identity and monitoring it throughout its life cycle across different services. This approach helps developers identify delay triggers, track the time spent by each service, and gain insight into how services interact with each other [12, 25].

The Distributed Tracing Pattern is implemented by configuring services to produce trace data and associate distinct identifiers with every request. A unique trace ID is assigned to each new request as it joins the system, and this trace ID is shared with all involved services; i.e., each service transmits its trace data to a centralised tracing service, along with pertinent metadata (such as timestamps and operation details). By collecting all of the data, the centralised service provides an extensive representation of the request's path through the system. Developers can monitor how services interact, estimate how long each task takes, and identify performance issues by analysing these records [12,25,26].

Distributed Tracing Pattern has a few benefits. It provides clear insight into request management, allowing for faster issue detection and a significant reduction in Mean Time To Repare (MTTR) and Mean Time To Detect (MTTD). Distributed Tracing solutions can be easily incorporated into various cloud-native environments and support a wide range of programming languages and applications. Additionally, it improves teamwork by clarifying which team is responsible for addressing particular errors [26].

However there are drawbacks as well, such the need for manual instrumentation and possible gaps in front-end analysis if certain components are left out. Also, due to sampling constraints, not all requests are captured, which may result in the omission of crucial information [26].

2.1.2.4.3 Performance Metrics Pattern

The Performance Metrics Pattern is essential to preserving visibility and control over the health and performance of the system in a microservices architecture. This approach involves collecting data from each service on particular tasks, such as latency, CPU usage, and memory consumption. The data is then aggregated into an integrated metrics service that offers comprehensive reporting and alerting features [12, 24]. By using a centralised technique, developers can easily identify performance limitations and other issues by continuously monitoring the microservices infrastructure's overall performance.

Two models are usually used in the Performance Metrics pattern implementation: push and pull. In the pull approach, the metrics service periodically collects data from each service, while in the push model, services actively send their metrics data to the central metrics service. Organizations can ensure efficient and effective performance management by customizing their metrics gathering and monitoring processes to match their specific demands and infrastructure by selecting the appropriate model [12, 27].

This pattern has advantages and drawbacks. Integrating metrics collection into each microservice can add complexity to the codebase by combining monitoring code and business logic. On the other hand, microservices-based application performance and dependability can be significantly improved by employing insights obtained from centralised statistics, such as detecting service latency or resource usage behaviours [27].

2.1.2.5 Cross-Cutting Concern Pattern

Lastly, shared needs that are present in different microservices are addressed by the cross-cutting concern patterns. Examples that are pertinent are as follows:

2.1.2.5.1 Service Discovery Pattern

With the goal to enable dynamic communication between multiple instances of the same microservice running across different virtualized containers, Service Discovery is a crucial component of microservices architecture. It ensures that clients may quickly identify and connect with the correct instances, although microservice instances can shift dynamically. Service discovery can be divided into two categories: client and server [13].

The client is responsible for selecting one of the available services by querying a service registry, a database containing the network locations of every available service instance. The client uses load balancing to choose an instance to send a request. The main benefit of this pattern is its simplicity and ease of development, as clients are already aware of service instance locations and can connect without server complications. However, a significant drawback is the high coupling between the client and the service registry, which can lead to increased maintenance challenges and potential failure points [13].

On the other hand, on the server, clients submit requests through a load balancer, which retrieves information from the service registry and redirects the request to an available instance. This approach removes the demand for an intermediary layer such as an API Gateway by allowing direct communication between clients and services. To ensure that the dynamic location of every microservice instance is preserved, the service registry works similarly to a Domain Name System (DNS) server; upon receiving a service request from a client, the registry confirms the service's availability and provides the necessary connection data [11, 13].

This pattern offers benefits such as improved health management, easier maintenance, greater fault tolerance, and simplified communication. It also facilitates development and migration by promoting the reimplementation and dynamic registration of services. However, it faces challenges like the need for robust interface design, increased complexity due to the service registry layer, and the inherent difficulty of monitoring distributed systems [13].

2.1.2.5.2 Circuit Breaker Pattern

In microservices architecture, the Circuit Breaker Pattern is a reliability design pattern that helps manage service failures by preventing failed services from being invoked again. It works similarly to an electrical circuit breaker: when a service experiences a predetermined number of consecutive failures, any future attempts to contact the service are halted for a specific timeout duration. Any attempt to use the service during this timeout is instantly rejected; a finite number of test requests are permitted to flow through the circuit breaker once the timeout finishes. The circuit breaker returns to normal working if these queries are successful; otherwise, the timeout period is reset and failures persist [12, 28].

This design offers several advantages. Firstly, it prevents the system from failing as a whole by assisting services in handling the failure of other services they rely on. It saves resources, including threads and connections, that might otherwise be wasted waiting for failed service answers by immediately rejecting calls to the broken service. Furthermore, it strengthens system resilience by separating failures and offering a way to progressively recover from them. It also improves system availability and stability by preventing the exhaustion of network resources and mitigating the impact of a failed service on the

remaining system components [12, 29].

The Circuit Breaker Pattern presents challenges, such as determining appropriate timeout values and failure thresholds. Incorrect settings can result in false positives, causing the circuit breaker to trip inappropriately, or false negatives, preventing it from tripping despite continuous failures [28]. These scenarios can negatively impact system reliability and performance. Additionally, this pattern adds complexity to the system by requiring processes to manage the timeout duration, identify problems, trip the circuit, and reset the circuit when the service restarts.

The Circuit Breaker Pattern can be especially helpful in practice when synchronous calls are required to access resources such as databases or third-party services, where errors might have a significantly impact on system performance. Circuit Breakers can be integrated into systems to mitigate the probability of prolonged failures and to expedite recovery, thereby enhancing the overall resilience and robustness of the microservices architecture [12, 29].

2.1.2.5.3 Blue-Green Deployment Pattern

The Blue-Green Deployment Pattern is a planned method for managing software releases that aims to minimise delays and facilitate smooth rollbacks. Using this pattern, two production environments, labelled blue and green, are maintained as similarly as possible. One environment is always active and handles all production traffic, while the latest version is extensively tested in the inactive environment (green) before being released. Production traffic migrates from the blue environment to the green environment once the green environment's performance has been validated. The transition occurs without disruption due to a straightforward router adjustment; traffic can be quickly switched back to the blue environment if complications arise, enabling rapid rollback and reducing the risk of prolonged failure [12, 30–32].

There are several significant benefits. A key advantage is achieving minimal downtime during deployments, which is essential to maintain operational continuity and ensure high user satisfaction. This design also simplifies rollback operations; the impact on clients is minimised when a failure occurs, as reverting to the previous version (blue environment) is simple and quick to accomplish [30–32]. The separation between the two environments allows thorough evaluation of the new release in a production-like setting, ensuring that the live environment remains unaffected. By isolating the new release, any problems can be identified and resolved before they impact end users.

Nevertheless, implementing Blue-Green Deployment presents challenges: one primary disadvantage is the additional complexity and cost of maintaining two identical production environments. This requirement can consume a lot of resources, particularly in conventional on-premises configurations. A robust infrastructure is also needed to efficiently handle traffic routing and switching between environments. Moreover, database improvements might be particularly challenging because they must be compatible with both application versions; to address this, database improvements should be tested and deployed independently to ensure that they do not interfere with application functionality [30, 32].

Despite these barriers, cloud computing has allowed organisations to dynamically allocate and reallocate resources as needed, making Blue-Green Deployments more practical and affordable [12].

2.1.3 Saga Pattern

For microservices systems, the Saga Pattern offers a reliable method to handle distributed transactions. Instead of relying on a single central coordinator, the Saga Pattern splits transactions into smaller, more manageable stages, each linked to a unique microservice.

In 1987, a work aimed at supporting Long Lived Transactions (LLTs) introduced the concept of Saga Pattern [33]. Sagas have been adapted more recently to address consistency problems in modern distributed systems, as demonstrated in a 2015 study on distributed sagas [34]. Since then, the Saga Pattern has evolved to better address cases where data consistency across multiple microservices needs to be maintained.

A saga consists of a sequence of local transactions, each of which triggers an event to initiate the next transaction that modifies its database. To reverse the modifications made by previous transactions, compensatory transactions are executed if a local transaction fails [35, 36].

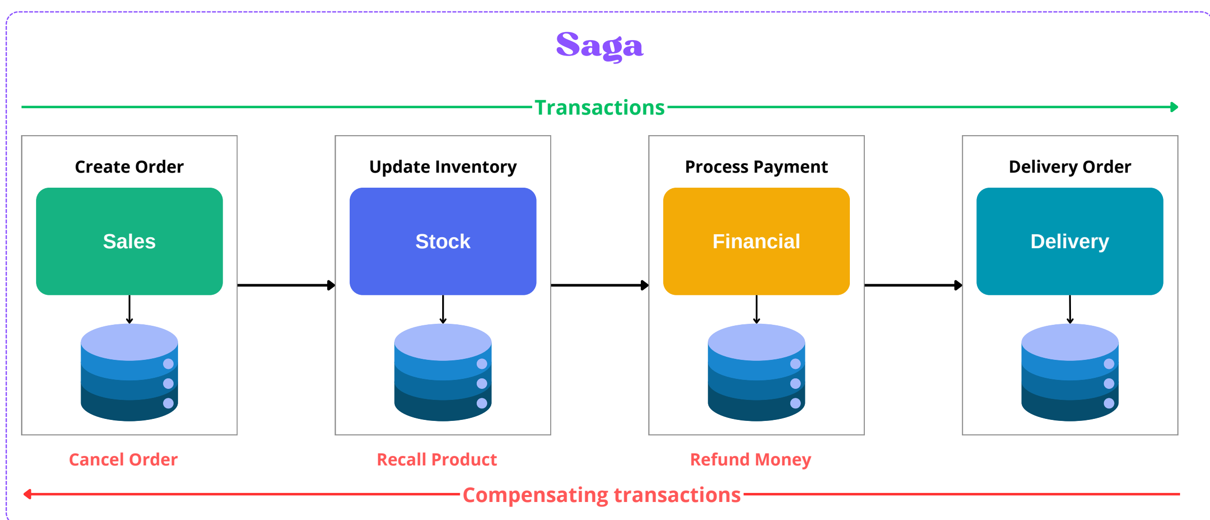


Figure 2.2: Representation of the Saga Pattern. Adapted from [2].

Figure 2.2 illustrates an example of an e-commerce scenario in which a client creates an order to demonstrate how this pattern is applied. An interesting feature is that the order status changes to *cancelled* instead of being deleted if a failure occurs and the compensatory transaction initiates a new event or modifies the previous state. This illustrates that the Saga Pattern results in either decisive success or failure.

There are two main techniques for coordinating *sagas*: choreography (see Section 2.1.4) and orchestration (see Section 2.1.5). Generally, there is no central coordinator in choreographed sagas; instead, each service publishes domain events that trigger further transactions in other services. Although this decentralised strategy supports scalability and loose coupling, it can be challenging to manage and understand. Updates by a single service may impact other services, creating unwanted dependencies and consequently making system maintenance more difficult.

In contrast, in orchestrated sagas, a central coordinator (or *orchestrator*) controls the sequence in which the transactions are executed. The saga flow remains transparent and simple to manage by the orchestrator, which provides centralised management by

notifying all involved parties of the next transaction to be executed. The centralised nature of this approach simplifies monitoring and debugging.

Advantages and Disadvantages

There are several notable benefits of employing the Saga Pattern in distributed transaction management. Firstly, it avoids the need for distributed transactions, which are sometimes challenging and costly to construct, and enables applications to preserve data consistency across multiple services. By segmenting the transaction into smaller, more manageable stages, the Saga Pattern ensures that each microservice can function independently while still contributing to the overall transaction [37].

Another significant advantage is fault tolerance. Systems adopting the Saga Pattern can recover smoothly from errors while preserving data integrity through the implementation of compensatory transactions. This is especially important in distributed environments where errors are inevitable and the system's consistency and reliability depend on the ability to properly undo changes.

Furthermore, the Saga Pattern encourages loose coupling and scalability, especially in choreographed sagas. Such systems allow for a more responsive and flexible design, as services can scale horizontally and independently without interfering with each other. Since changes to one service have minimal impact on others, this decoupling also simplifies the evolution of services.

However, the Saga Pattern also presents several challenges that need careful consideration. One of the main limitations is implementation complexity; developing compensating transactions and managing the compensation logic can be challenging and error-prone, especially in large distributed systems. The system's complexity increases as each service has to handle not only its primary function but also the logic required to reverse its actions if necessary.

Another significant issue is the lack of automatic isolation. Because sagas cannot guarantee automatic isolation like ACID transactions, data anomalies may arise when multiple sagas are executed concurrently. Therefore, developers must take extra precautions when designing systems to handle these anomalies and ensure data consistency [35].

Tracking and debugging are another challenge, particularly in choreographed sagas without a central control node. Since these systems are decentralised, tracking the progress of transactions and identifying the root cause of failures can be difficult. Efficient logging and monitoring systems are essential for managing and analysing these distributed transactions.

Best Practices

To successfully implement the Saga Pattern in a practical manner, consider these best practices:

1. **Establish clear boundaries**: Ensure that all microservices in a saga have clear responsibilities and constraints.
2. **Develop strong compensations**: Implement robust compensatory transactions to handle failures smoothly.

3. **Monitor and Log transactions**: Establish comprehensive monitoring and log to track the progress of sagas and effectively identify issues.

4. **Employ independent operations**: Design local transactions to be independent to manage retries without causing unintended side effects.

2.1.4 Choreography in Microservices

The choreography approach describes a decentralised mechanism for coordinating interactions between services where each microservice reacts independently to events and state changes. This approach uses event-driven communication, where individual microservices broadcast events to a message broker from which other microservices receive notifications and take appropriate action.

In the context of choreography in sagas, when a service in the saga concludes a task, it broadcasts an event that triggers the next service. These services must be capable of restarting operations by triggering events in reverse sequence in case of a failure [3,38,39].

Following the example illustrated in Figure 2.2, we will now examine the choreography model presented in Figure 2.3. This model adapts the e-commerce scenario to demonstrate how services interact with each other in a decentralised way within the context of the Saga Pattern.

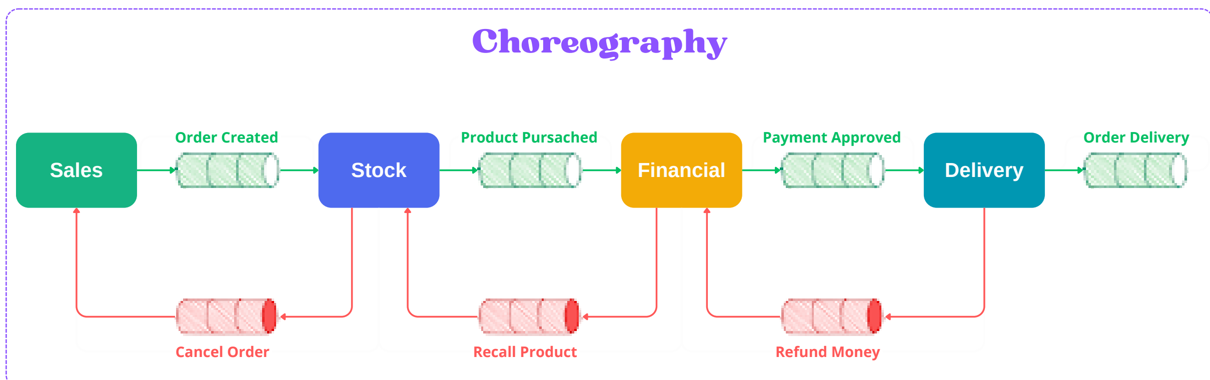


Figure 2.3: Representation of the choreography saga model. Adapted from [2].

Workflow

Figure 2.4 illustrates a generic choreography model workflow. The stages involved are as follows:

1. The Saga begins when service S1 broadcasts an event denoted as S1.

2. The activation of service S2 is triggered by the occurrence of event S1.

3. The S2 service then emits the S2 event after completing its assigned processes.

4. Event S2 starts service S3.

5. After service S3 is finished, it sends an event S3 into the system.

6. After event S3 completes their operations, it re-triggers service S1, completing the Saga.

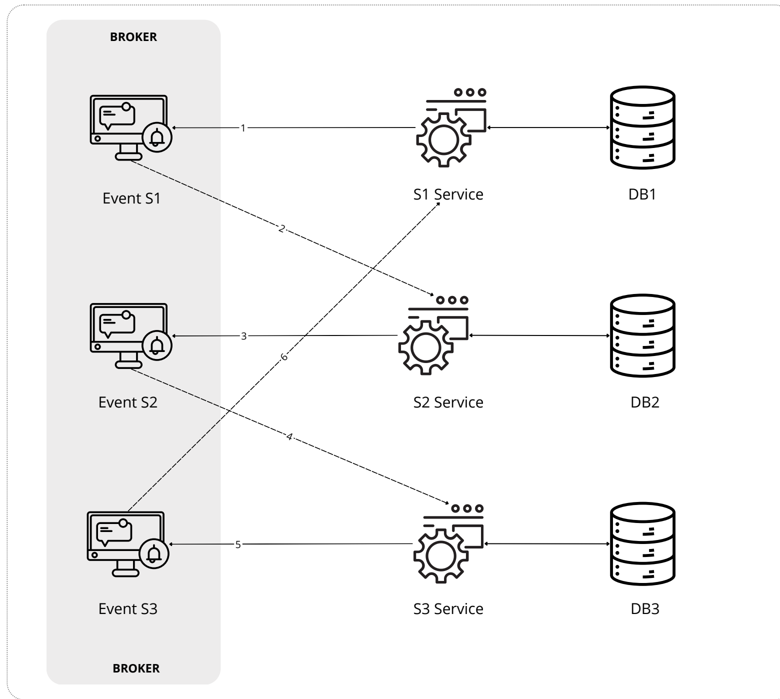


Figure 2.4: Generic choreography workflow. Adapted from [3].

Advantages and Disadvantages

The choreography approach for microservices can be attractive for some applications and system architecture scenarios since it offers several notable benefits [3, 39–42].

Firstly, it emphasises loose coupling, in which every microservice operates independently. Due to their independence, individual microservices can evolve or deploy without impacting the entire system. Active process data is queued in the message broker during deployment to ensure continuous service; the stability and continued operation of the system depends on this decoupling.

The choreography approach also makes it easier for microservices to interact at low frequencies. There is less need for continuous communication, since information sharing occurs only when there is a state change. Due to this characteristic, this approach can be applied in Wide Area Networks (WANs), where frequent communication may not be the most efficient method. The choreography pattern improves system performance and resource usage by reducing communication overhead.

Furthermore, the scalability and durability of the microservice architecture are supported by the decentralised structure of choreography. Unlike orchestration's centralised control, the failure of one microservice does not always impact the others. Due to its resilience, the system can handle changes in workload and continues to work successfully in challenging circumstances.

Despite its benefits, the choreography approach has drawbacks and limitations that should be carefully taken into account when designing the system [3, 40–42]. The lack of centralised process visibility is one significant disadvantage; process monitoring becomes challenging without a central controller. This can lead to complex and challenging struc-

tures, sometimes called *spaghetti architectures*[2], which are expensive to maintain. Since choreography is decentralised, it requires strong management and monitoring techniques to oversee everything while maintaining control of the system.

Additionally, the lack of a single controller makes the design increasingly challenging as choreography evolves. Each microservice must independently react to relevant events, necessitating thorough awareness and synchronisation techniques. Although this independent behaviour enhances flexibility, it also requires careful management of dependencies and inter-service interactions.

Another issue is weak data reuse and atomicity. Microservices often maintain separate copies of their data, which can limit service reuse across multiple applications and lead to inconsistent transactional boundaries. This fragmentation of data and functionality challenges strong atomicity and effective data management across microservices.

Lastly, when a microservice is temporarily unavailable, the choreography approach could result in unpredictable response times. The availability of each participating microservice is essential for the completion of the process, which can cause unexpected delays. For user interfaces that require immediate input, this variability in response time is less than ideal, highlighting the balance between system autonomy and responsiveness.

2.1.5 Orchestration in Microservices

In the orchestration approach, several atomic services are successively triggered by a central controller, also known as an orchestrator, to manage the end-to-end application process flow. By using request-reply interactions, this approach ensures that every service is initiated and completed before the next one begins.

Each atomic microservice is the only one with access to its own data, and it is the orchestrator that allows communication between services. The orchestrator's queries are processed by an API, and events are published to the orchestrator using an event broker by an internal function. All services, along with the orchestrator, are subscribed to the broker to receive notifications of important events.

Using the same e-commerce scenario from Figure 2.2 as a reference, Figure 2.5 illustrates the orchestration model. In this approach, the Saga Pattern is executed consistently due to a central orchestrator that manages the interactions between services and organises the workflow.

[2]The term *spaghetti architecture* refers to a complex, tangled, and difficult-to-maintain software or system architecture. It is a metaphorical term derived from the visual similarity of such architectures to a bowl of spaghetti [43].

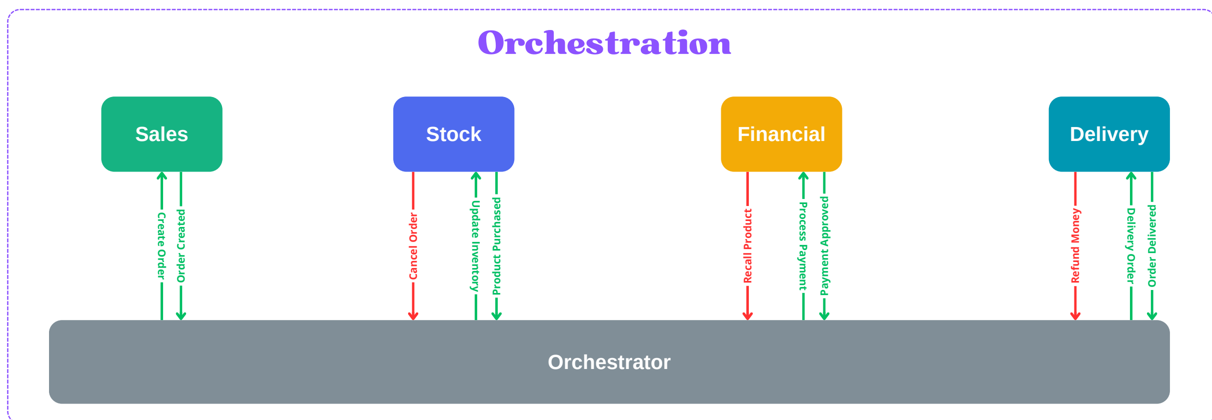


Figure 2.5: Representation of the orchestration saga model. Adapted from [2].

Workflow

Figure 2.6 illustrates a general orchestration model workflow. The stages involved are as follows:

1. The Saga begins by service S1 broadcasts an event to the coordinator.
2. The orchestrator receives the event from service S1.
3. (a) The orchestrator then sends an event through channel S2.
   (b) Channel S2 starts service S2.
   (c) After service S2 is finished, it broadcasts an event back to the orchestrator.
4. Service S2 provides the event to the orchestrator.
5. (a) The orchestrator broadcasts another event over channel S3.
   (b) Channel S3 starts the task of service S3.
   (c) Service S3 reports an event to the orchestrator following its completion of its operations.
6. The orchestrator acknowledges the event from service S3, completing the saga.

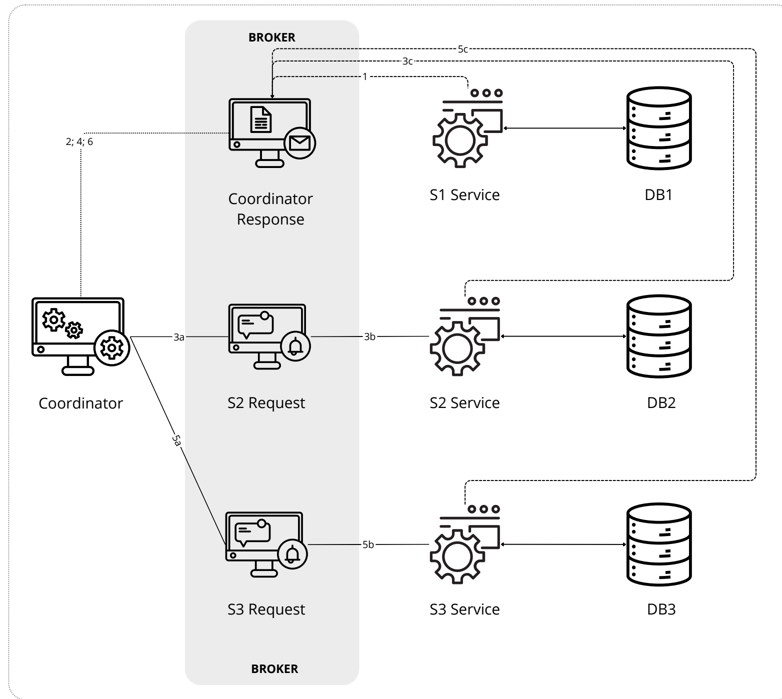


Figure 2.6: Generic orchestration workflow. Adapted from [3].

Advantages and Disadvantages

The orchestration approach has several noteworthy advantages. As highlighted by [40–42] one main benefit is the ability to clearly identify the process is an important benefit; tracking the state during run-time is simple as the processes are accessible from beginning to end. This improves the administration and supervision of the entire workflow, which is especially helpful for complex processes that involve multiple microservices.

Furthermore, since orchestration employs direct invocation-based communication, it simplifies application design. This simplicity is facilitated by the clear visibility of end-to-end processes, making managing and designing complex workflows easier. The typical complexity of the microservice architecture is reduced by this simplified design method.

Moreover, the robust atomicity provided by orchestration ensures high reusability. Microservices offer atomic actions by encapsulating single entities and granting them exclusive access to their own data. Microservices are highly reusable due to their encapsulation, which also enhances their modularity and flexibility. These services can be easily integrated into various applications without interfering with each other.

Another advantage is the predictability of response times. Predictable reaction times are ensured by the invocation-based request-reply interaction, which is a fundamental feature of orchestration, even for incorrect replies. For user interfaces requiring quick feedback, this predictability is essential for delivering a dependable and consistent user experience.

Despite its advantages, the orchestration technique has a few drawbacks [40–42]. A prominent drawback is the tight coupling of microservices. Despite being independently deployable, microservices often require downtime during deployment to prevent interruptions in the flow of the application process. An active load-balanced setup can help mitigate this problem, but it remains a significant risk.

High chattiness, or the frequent data exchanges between microservices at every stage of the application process flow, is another drawback. This high degree of communication can lead to increased latency and possible performance degradation, making the orchestration approach less suitable for microservices distributed across a WAN.

The orchestrator acts as a single point of failure; as the primary control point, its failure could bring the system to a complete halt and reduce its resilience. The orchestrator can be deployed in a high availability configuration to mitigate this issue, but there is still a risk of a single point of failure.

Lastly, orchestration's synchronous structure may result in longer execution times. The overall execution time, which is the sum of the periods spent by each service, can lead to longer end-to-end processing times. This cumulative delay may impact the system's responsiveness and efficiency, particularly in time-sensitive situations.

2.1.6 Comparative Analysis: Choreography vs. Orchestration

The decision between orchestration and choreography in the context of microservices architecture plays an important role in determining how services interact, communicate, and manage their behaviours. Based on specific project goals and system features, both approaches influence architectural decisions while posing their own set of particular challenges.

The purpose of the comparative analysis is to present a systematic evaluation of the two approaches by considering important factors including resilience, control and coordination, adaptability, performance consequences, and applicability for various use cases.

The Table 2.3 summarises the differences between choreography and orchestration approaches across different dimensions critical to MSA.

The trade-offs and considerations to be taken into account when choosing between choreography and orchestration are highlighted by each item in the table. Choreography provides decentralised robustness and flexibility, but when trying to preserve consistency, it needs strong mechanisms for event management and coordination. Conversely, orchestration offers transactional integrity and centralised control, but it may also provide scalability issues and a single point of failure.

Table 2.3: Comparative analysis of choreography and orchestration approaches in microservices architecture. Source: author.

Criteria	**Choreography**	**Orchestration**
Control Mechanism	**Decentralised**: each service reacts independently to events.	**Centralised**: a central orchestrator manages and coordinates services.
Communication	**Event-driven**: services communicate via message brokers or events.	**Request-reply**: services communicate directly, often via API.

Dependency Management	**Loose coupling**: services are loosely coupled and independent.	**Tight coupling**: services depend on the orchestrator.
Service Independence	**Highly independent**: services can evolve and deploy independently.	**Dependent**: services rely on the orchestrator for workflow and communication.
Managing Failures	**Reactive**: services react to failures independently.	**Proactive**: orchestrator manages retries and compensations.
Visibility	**Limited**: monitoring and tracing may be more complex.	**Comprehensive**: orchestrator monitors and manages the entire process.
Scalability	**Flexible**: each service can scale independently.	**Centralised**: scaling may depend on the orchestrator.
Performance	**Efficient**: reduced overhead due to event-driven nature.	**Varied**: can introduce latency depending on architecture.
Complexity	**Potential for spaghetti architectures** without proper design.	**Simplified**: clear structure and oversight by orchestrator.
Atomicity and Consistency	**Eventual consistency**: may require compensating actions.	**Strong consistency**: ensured through orchestration.
Resilience	**Distributed**: failure in one service may not affect others.	**Centralised**: orchestrator is a single point of failure.
Deployment	**Independent**: services can be deployed and evolved separately.	**Coordinated**: all services and orchestrator need synchronism.

2.1.7 Practical Implementations and Case Studies

As covered in previous sections, Microservices Architecture (MSA) is becoming a growing trend among organisations due to its advantages. However, the practical implementation of these concepts requires dedicating careful thought to a number of variables, including fault tolerance, scalability, and service communication. This enables a deeper understanding of how businesses have applied microservices to enhance business agility, address system complexity, and improve performance by examining real-world examples.

This section looks at some of the case studies from recognised companies that have effectively incorporated microservices architecture. These implementations offer important insights into how microservices may address challenging problems across a range of markets by demonstrating the practical use of design patterns, orchestration, choreography, and other architectural alternatives.

2.1.7.1 Netflix

> *Our microservice architecture decouples engineering teams from each other, allowing them to build, test and deploy their services as often as they want. This flexibility enables teams to maximise their delivery speed* [44].

As one of the first organisations to embrace microservices, Netflix faced multiple challenges in trying to scale their monolithic architecture to accommodate millions of clients around the world [45, 46]. By dissolving services such as user management, content distribution, and recommendation algorithms, the shift to microservices allowed them to improve their scalability and resilience to failure.

A thoughtful use of some design patterns, particularly decomposition and cross-cutting concern, can be seen in Netflix's architecture. By employing a business capability decomposition pattern (see Section 2.1.2.1), Netflix segmented core functionalities into isolated microservices, such as the user profile service, content delivery service, and payment processing service. This allowed Netflix to reduce interdependencies between services, enabling faster development cycles and independent scalability across its platform.

To address cross-cutting concerns, Netflix initially employed Hystrix[3] [47, 48] which implemented the circuit breaker pattern (see Section 2.1.2.5) to manage failures. Hystrix allowed Netflix to monitor the health of microservices and ensure that the platform remained operational under stress, minimizing downtime during high-demand periods. However, by 2018, Netflix began transitioning to Resilience4j [49, 50], particularly for new projects. Resilience4j offered a more modular and lightweight approach, giving Netflix greater flexibility by allowing teams to select only the necessary resilience features for specific use cases.

Furthermore, the coexistence of choreography and orchestration approaches ensures the robustness and scalability of the service. Netflix has embraced a choreography for loosely coupled services, using event-driven communication to enable independent interactions between microservices without the need for a central coordinator. This approach allowed Netflix to achieve greater scalability and fault tolerance, as individual service failures did not cascade through the system, ensuring uninterrupted service continuity.

In addition, Netflix acknowledged the vitality of orchestration in some situations, especially when handling sophisticated workflows that required a centralised controller. Netflix Conductor [51, 52], a microservice orchestration engine created by Netflix, is a good example of this. This is applied in multiple-stage billing operations and content recommendation generation to manage interactions between different services.

In short, Netflix's use of both orchestration and choreography created a balance between control and flexibility. For tasks requiring sequential execution, orchestration provided the necessary oversight, while choreography ensured autonomy and fault isolation in event-driven services. This dual approach has enabled Netflix to build a microservice architecture that is highly scalable, resilient, and adaptable, capable of meeting the growing demands of its global user base.

[3] Though sometimes categorized as an observability pattern due to its metrics features, Hystrix primarily addresses cross-cutting concerns through fault tolerance mechanisms.

2.1.7.2 Amazon

> *Many startups, and even projects inside big companies, start out this way. They take a monolithic approach because it is very quick to get moving quickly. But over time, as that project matures, as you add more developers to it, as it grows and the code base becomes larger and the architecture becomes more complex, that monolith will add overhead to your process, and that the software development lifecycle will begin to slow down* [53].

Amazon, one of the largest e-commerce platforms, also transitioned from a monolithic architecture to a microservice architecture to address issues related to scalability, fault tolerance, and agility. This shift enabled the company to decompose complex systems into smaller, independently deployable services, each responsible for distinct tasks such as order processing, inventory tracking, and shipping [54]. By decentralising responsibilities, Amazon was able to scale its systems more efficiently, providing greater flexibility in managing individual components.

A key aspect of this transition was Amazon's adoption of the database-per-service pattern (see Section 2.1.2.3), where each microservice is paired with its own database. By ensuring that services could run without interruption, this architectural decision improved system availability and performance. This design choice was crucial to improve system performance and availability, as services could operate without interdependencies that could otherwise cause downtime or bottlenecks. The autonomy granted by this pattern also allowed Amazon to optimise each database according to the specific needs of the service it supports.

In addition, Amazon introduced the saga pattern (see Section 2.1.3) to manage distributed transactions between services. Rather than relying on traditional, monolithic transactions that spanned multiple components, Amazon decomposed these processes into smaller, manageable steps, each executed by its respective microservice. The Saga pattern allowed for greater fault tolerance, as compensating transactions could be executed to roll back changes in the event of failure, ensuring both consistency and resilience in Amazon's operations.

For complex workflows such as order fulfilment and data analysis, Amazon opted for an orchestration approach [55]. Using a central orchestrator, such as AWS Step Functions [56], the company gained precise control over transaction flows, ensuring that tasks were completed in the correct sequence and that failures could be monitored and mitigated centrally. This approach minimised the risk of system-wide failures in critical workflows, improving overall reliability.

However, Amazon also recognised the need for flexibility in other areas of its architecture. For scenarios where strict coordination was unnecessary, such as event-driven communication, Amazon used a choreography-based approach. Services such as Amazon EventBridge [57] enabled microservices to communicate asynchronously, reacting to events without the need for a central controller. This event-driven architecture provided greater scalability and autonomy, allowing services to evolve independently while maintaining system-wide coherence.

By combining orchestration and choreography, Amazon achieved a balance between control and flexibility. The company typically employs orchestration for highly structured step-by-step processes, while using choreography for more dynamic loosely coupled inter-

actions. This hybrid approach enables Amazon to take advantage of the benefits of both approaches, ensuring that its architecture remains scalable, fault tolerant, and adaptable to the evolving demands of its global customer base.

In summary, Amazon's architectural evolution demonstrates the effective application of microservices principles to meet the challenges of large-scale distributed systems. By adopting a hybrid approach that balances the rigor of orchestration with the agility of event-driven choreography, Amazon ensures the flexibility and resilience needed to support its rapid growth and complex operations.

2.1.7.3 Uber

> *The foundation of our platform power is its microservice-based architecture, a commonly used structural style in which applications consist of interoperating services. Microservice architectures, which can support stable deployments and modularity, are well known to be scalable. With diverse engineering teams at Uber working on interoperating services, it is important to ensure that our stack can both safely roll out new changes and reliably reuse parts of the architecture in a modular way. Taken together, these functionalities allow for high developer velocity and quick release turnaround times, giving us the flexibility to build on independent schedules (...)* [58].

Similar to several other businesses, Uber started off with a monolithic architecture designed for a single service in a single place. Although useful in the beginning, the monolithic architecture proved to be a challenge as it tried to satisfy all the demands of its rapidly expanding business [59]. Uber transitioned to a microservices architecture, splitting its large system into smaller, independently deployable microservices, to overcome these limitations. This shift enabled Uber to grow its services more successfully, offering more operational flexibility and regional customisation.

Their architectural evolution relies on the decomposition of services by subdomain and the decomposition of services by business functions (see Section 2.1.2.1), which groups services according to business funcionalities [60]. Due to the fact that individual services may be changed or enhanced without affecting the system as a whole.

A key feature of Uber's architecture is the use of the event-driven pattern, which enables real-time communication between services [61, 62]. For example, the transport management service broadcasts an event when a new ride is requested, triggering responses from other services such as driver dispatch, pricing, and navigation. Uber can respond to customer requests instantly because of its methodology, which also increases system resilience and speeds up service delivery [62]. Under choreography-based approach, all services operate independently of a central coordinator, listening for and reacting to pertinent events. Since there are less dependencies between services due to this decentralisation, the system is more adaptable, scalable, and resilient to failures.

In areas of Uber's microservices architecture where more organised cooperation is needed, orchestration is essential. Uber uses orchestration in complicated processes to ensure that multiple services are performed in the right order, enhancing reliability and consistency [63]. A notable example is in trip management, where driver allocation, route calculation, and payment processing are coordinated by an orchestrator [64]. This central control ensures that all steps are executed in the right sequence and allows efficient error

management in the event that errors arise.

Payment processing is another crucial activity that requires safe and smooth transaction management, and here orchestration plays a key influence. Payment methods are verified, costs are accurately estimated, and invoices are created appropriately through orchestration [64]. Uber has the ability to give its clients an easy and safe payment experience by centrally organising these processes. In addition, orchestration is necessary to maintain users with the objective of performing tasks such as identity verification, preference management, and the creation and updating of profiles [64]. In this case, orchestration ensures data integrity and consistency between services, ensuring that user data is properly stored.

Uber can reconcile flexibility and the demand for centralised control in complicated operations through the integration of the two approaches in its microservice architecture. This hybrid strategy gives the business the flexibility it needs to grow internationally without compromising the efficiency and dependability that are essential to its platform.

2.2 Related Work

Microservice architecture has emerged as a key component in modern software development, providing scalability, resilience, and agility. Two popular approaches in microservice architecture are choreography and orchestration. This section examines previous research and contributions that employed these techniques, establishing a broad context of the available literature.

Kristianto and Zahra [3] conducted a significant study that used the MySQL database, the Spring Boot Framework, and Kafka as a message broker to develop models deployed on the Google Cloud Platform using Kubernetes. They evaluated the load using JMeter, focussing on metrics such as response time, throughput, and Central Processing Unit (CPU) usage to compare choreography and orchestration performance.

Their tests involved different numbers of services (starting with two, four, six, and eight services) and progressively increasing the number of concurrent users from one to one thousand in multiples of one hundred. Furthermore, the study evaluated performance with and without a load balancer by adjusting the number of service instances.

Regardless of the number of services, choreography consistently outperformed orchestration in scenarios without a load balancer in terms of response times. The constant communication between the coordinator and the service had a more adverse impact on the orchestration. While employing a load balancer and increasing service instances enhanced overall performance in heavy traffic environments, choreography still maintained a superior response time.

In terms of productivity, both in contexts with and without a load balancer, choreography performed better than orchestration. The performance of both models was improved by installing a load balancer, particularly in scenarios where there were many users and substantial traffic loads. The presence of a coordinator in the orchestration increased the probability of bottlenecks, which lowered productivity. Both models benefited from a load balancer, especially in high-traffic scenarios. However, orchestration was less intensive CPU than choreography, although the high level of communication in orchestration increased workload and CPU usage.

Singhal [41] performed an analysis of microservice choreography and orchestration approaches in a healthcare application. Three primary metrics were evaluated: energy consumption, memory usage, and time consumption. Compared to orchestration, Singhal concluded that choreography consumes less time while developing microservices. Furthermore, the study demonstrated that choreography consumes less memory than orchestration, suggesting that the choreography approach makes better use of memory resources. Once again, choreography appeared to be more effective in terms of energy usage than orchestration.

Singhal's experiment concluded that choreography is faster and more effective in terms of memory, energy usage, and performance. However, Singhal noted significant challenges in programming and monitoring numerous events without a central orchestrator. Choreography is beneficial when dealing with fewer microservices or low complexity in event triggers. Conversely, orchestration's centralised control simplifies management and is advantageous for handling complex transactions, despite its slower performance.

Rudrabhatla [38] conducted another analysis of these approaches, especially related to the Saga design pattern for distributed transactions in isolated NoSQL databases. The results demonstrated that orchestration takes significantly more time than choreography. However, Rudrabhatla highlighted the difficulty in developing and tracking choreography when each microservice triggers many events. Coordination among developers or teams becomes challenging without a central orchestrator. Choreography is beneficial for distributed transactions that involve fewer microservices or simple event triggers. In contrast, orchestration offers easier maintenance due to centralised management, making it suitable for complex transactions despite its slower operation.

Through the evaluation of several performance measures, these studies provide comprehensive insights about the choreography and orchestration approaches used in microservice architectures. In summary, choreography performed better in terms of response time, memory usage, and energy consumption, particularly when load balancers were not used and in high-traffic scenarios. However, the lack of a single coordinator made scheduling and monitoring highly challenging, and it became complicated when multiple events and microservices were involved.

On the other hand, orchestration benefited from centralised management, which made it simpler to manage complicated transactions and minimised CPU usage, even if it had higher latency and resource consumption. Despite its lower performance, orchestration appears to be more successful in scenarios requiring centralised control and simpler maintenance. Therefore, the trade-off between microservice management complexity and performance requirements must be taken into account when deciding between choreography and orchestration.

Chapter 3

Design and Specifications

A microservices architecture structures an application as a collection of small autonomous services that communicate with each other using simple protocols, most often an Hypertext Transfer Protocol (HTTP) API. Each service works independently within its own process. This method has multiple benefits including scalability, flexibility, implementation independence, and quick evolution. It fits especially well in complex and dynamic systems.

It is essential to take into consideration equally the functional and non-functional requirements of the system while designing and implementing a microservices architecture, in addition to determining the most appropriate architectural patterns. Saga, which addresses distributed transactions in microservice systems, is one of these patterns.

With a focus on comparing orchestration and choreography using the Saga pattern, we will discuss the specifications of the microservice architecture design and implementation for the meal ordering system in this section.

3.1 Problem Overview

In the swiftly changing digital marketplace, food service companies are increasingly dependent on meal ordering platforms to stay competitive and effectively cater to customers. These platforms enable users to explore menus, place orders, monitor delivery status, and process payments with ease.

Nonetheless, creating and deploying a reliable meal ordering system that guarantees scalability, high availability, and effective management of intricate transactions poses considerable challenges. This is especially crucial as food service companies aim to cope with fluctuating demand, offer varied payment methods, and ensure real-time coordination among different stakeholders. This need arose from the experience of a small freelance food delivery operation, where initial processes relied on manual management.

Customers would directly message the seller to detail their meal preferences and order specifics. The seller had to oversee all logistics, including cost monitoring, handling urgent orders, and estimating ingredient quantities. This manual method resulted in inefficiencies such as laborious coordination, possible miscommunications, and logistical hurdles. A major complication was the diverse payment options clients could select—from immediate

payments to settling bills weekly or monthly. These varying payment timelines, combined with manual tracking of orders and customer preferences, made it difficult to maintain precise records and deliver timely service.

Managing multiple clients, each with distinct requirements, increased the risk of mistakes, leading to unsatisfied customers and higher operational costs. This situation underscored the urgent need for a more efficient, automated solution that could address the shortcomings of the manual system while allowing clients to maintain their usual routines. The proposed solution was to create an integrated meal ordering system that could automate the entire ordering process, from menu exploration to payment.

Such a system would enable clients to easily access the daily menu, place orders, check account balances, and manage preferred payment schedules, all while reducing the need for direct manual effort from the seller. By automating these functions, the proposed meal ordering system aims to lower the risk of errors, boost efficiency, and provide a scalable platform capable of accommodating an expanding clientele.

3.2 Requirements

To ensure that the system satisfies user expectations, it is essential to clearly identify functional and non-functional requirements. Functional requirements outline the characteristics that the system must have, while non-functional requirements address factors like security, scalability, and performance.

3.2.1 Functional Requirements

Functional requirements are all the features, demands, and competencies that the software must satisfy with the goal to complete a process. They answer the key question ”***What does the application do?***”.

During the development of the meal ordering system for this project, specific functional requirements were noted in Table 3.1.

Table 3.1: Functional system requirements. Source: author.

ID	Functionality	Description	Priority	Notes
RF-001	User Registration	The system must allow for the registration of new users via the administrator interface.	High	Includes generating a random 6-digit code.
RF-002	User Activation	The system must activate a user's account when the activation code is provided.	High	A code of activation was transmitted via message service.
RF-003	Order Placement	The system must enable clients to make orders through the interface.	High	Includes providing the menu and receiving order confirmations.

RF-004	Order Confirmation	The system must send an order confirmation to the client and generate a summary of the order.	High	N.A.
RF-005	Invoice Creation	The system must create invoices for orders and associate them with clients.	High	N.A.
RF-006	Invoice Payment	The system must process payments for outstanding invoices and update client debt.	High	Includes receiving payment confirmations via messaging.
RF-007	Menu Delivery	The system must send the daily menu to clients who interact with the application.	Medium	N.A.
RF-008	Debt Summary and Update	The system must provide a summary of outstanding debts and update debt records accordingly.	Medium	N.A.

3.2.2 Non-functional Requirements

Non-functional requirements relate how the application is used in terms of its performance, usability, reliability, security, availability, maintenance, and underlying technology. These are listed in Table 3.2 and deal with how the software's user will benefit from the functionality.

Table 3.2: Non-functional system requirements. Source: author.

ID	Type	Description	Priority
RNF-001	Data Consistency	The system must ensure consistency across microservices, especially during transactions.	High
RNF-002	Usability	The user interfaces must be intuitive and user-friendly for both administrators and clients.	Medium
RNF-003	Documentation	The system must provide comprehensive API documentation using OpenAPI.	Medium
RNF-004	Data Storage	The system must use MongoDB for non-relational data storage.	High
RNF-005	Security	The system must ensure encryption of sensitive data both in transit and at rest.	High

3.3 Architecture

Similarly to all other information systems, the architecture plays an essential role as it acts as the system's core. The architecture of the application provides proof to the stability of a software system.

The proposed architecture is composed of several microservices that collaborate to offer each feature required for a full-featured meal ordering application. The software consists of five elementary modules: the client module, the user module, and all the rest of the microservices, each of which has its own database, as shown in Figure 3.1. Each of these modules is deployed on an individual server.

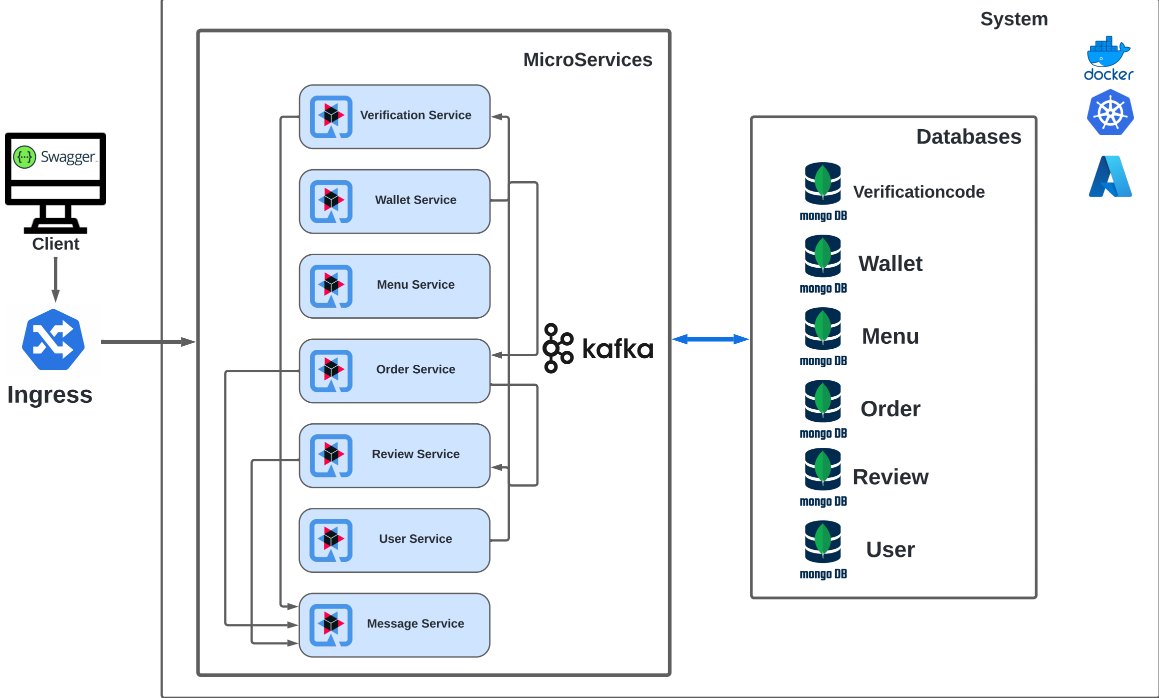


Figure 3.1: Proposed Microservice Archicteture. Source: author.

Table 3.3 lists the specific responsibilities of each module within the proposed architecture. An overview of the main tasks and interactions that are handled by each system component can be found in this table.

Table 3.3: System architectures modules. Source: author.

Module	Description
Users	Management of authentication and authorisation; User profile management; Interaction with the messaging service to send activation codes and important notifications;
Clients	Registration and updating of client information; Maintenance of interaction and order history;
Menu	Sending the daily menu to clients; Processing item selections made by clients;

Orders	Managing the order lifecycle; Interaction with the invoice module to ensure that all orders have associated invoices;
Invoice	Issuance and management of invoices; Processing payments and updating invoice status;
Messaging Service	Facilitating communication between the system and clients;
Review Service	Coordinating client assessments and feedback on orders; Aggregating and analysing review data to improve service quality;

3.3.1 Technology Stack

The specific tools employed for developing the microservices architecture for this project are outlined in this part. Each technology was chosen to satisfy the system requirements, including those related to development, communication, hosting, documentation, orchestration, data storage, and management.

3.3.1.1 Development

The following frameworks have been selected to make the development of microservices and the orchestration layer easier:

3.3.1.1.1 Quarkus

Quarkus is a Kubernetes-native Java framework designed to optimise Java applications for modern cloud environments, including serverless and Kubernetes [65]. Below is an overview of the technology stack and key features of Quarkus.

Quarkus has been optimised for Kubernetes, which simplifies application deployment without requiring an understanding of the platform's complexities. It enables automatic creation of Kubernetes resources, such as container images, without requiring the development of YAML files manually. It aims to integrate easily with OpenJDK HotSpot [66] and GraalVM [67]. This compatibility is essential for high-density memory usage in container platforms since it enables extremely fast startup times and minimal memory use.

Reactive and imperative programming styles are supported by Quarkus. This duality empowers developers to apply reactive non-blocking programming for improved resource usage in cloud-native applications, while simultaneously using recognised imperative code. With features like live reload, zero configuration, and simpler code for common use cases, it aims to enhance developer productivity. Also, it makes use of several popular Java libraries and standards permitting developers to use tools and frameworks within the Quarkus ecosystem.

3.3.1.1.2 Spring

Spring is a comprehensive and widely-used application framework for the Java platform [68]. Since its launch in 2002, Spring has become an indispensable component of Java programming and is available as open source software.

A key element of the Spring ecosystem, Spring Boot [69] optimises applications for fast start times and minimal resource use. It simplifies deployment in Docker [70] and Kubernetes container environments with configurable settings. It allows developers to decide on the most suitable approach for their specific use case by supporting both the conventional imperative and reactive programming styles, and also offers an intuitive full-stack framework through employing a large ecosystem of libraries and following industry standards.

Although Spring Cloud [71] is not as clearly Kubernetes native as Quarkus, it does provide strong support for developing cloud-native applications. It is ideal for microservices designs as it offers tools for distributed configuration, circuit breakers, and service discovery, among other things. This framework contains five key components: (1) Spring Boot, (2) Spring MVC, (3) Spring Data, (4) Spring Security, and (5) Spring Cloud.

1. Spring Boot is a project that makes setting up and developing new Spring applications easier.
2. A powerful Model-View-Controller (MVC) architecture used to develop web applications is offered by the Spring MVC module. It works well with other Spring modules and supports RESTful services;
3. Spring Data simplifies data access and manipulation, offering support for relational databases and non-relational databases like MongoDB (see Section 3.3.1.3). It provides a programming model that is compatible with several data storage.
4. A robust and adaptable system for access control and authentication is known as Spring Security. It provides full security services, such as authorisation, authentication, and prevention against vulnerabilities.
5. The Spring Cloud simplify the development of microservices. Among the crucial components are Spring Cloud Gateway (in order to route and filter requests), Service Discovery (using tools like Netflix Eureka [72]) and, Cloud Providers (compatibility with Amazon Web Services (AWS), Google Cloud Platform, Azure, and more).

3.3.1.1.3 OpenAPI

A standardised mechanism for documenting API endpoints, request/response formats, authentication techniques, and other features is provided by the OpenAPI specification [73], which is used to build and document RESTful APIs. Because of its language-neutral nature, OpenAPI may be used with a variety of backend technologies. This flexibility makes it easier to integrate and develop on different platforms (such as Node.js, the Express Framework, React, and other languages like Python or Java, among others).

The development process for OpenAPI is improved by a number of tools, such as Swagger UI (see Section 3.3.1.4.1), which offers an interactive interface for testing and documenting APIs, and command-line tools that help with type generation, testing and design APIs. As a single source of truth for API contracts, OpenAPI supports type safety and validation

over the whole framework and works directly with specifications without the need for code generation, among other important characteristics. These features enhance flexibility across different technology stacks, improve overall development efficiency, and encourage better communication between frontend and backend teams.

3.3.1.1.4 Camunda

Camunda will be employed to track and manage microservice activities, particularly while applying the Saga pattern. Microservices are orchestrated using workflows and decision automation platform, which serves as a hub for managing sophisticated business processes [74].

The tool was designed to manage and automate business activities in a variety of systems and contexts. It is strong and adaptable. It supports Decision Model and Notation (DMN) for decision management and Business Process Model and Notation (BPMN) for workflow modelling. This enables dynamic decision-making capabilities as well as precise, standardised process specifications.

Camunda offers a range of tools, including the Camunda Modeler for process modelling, the Camunda Engine for Java-written decision automation, Camunda Optimise for monitoring and analytics, and Camunda Tasklist for managing human tasks within workflows. These tools streamline the development and management of process automation, enhancing productivity.

3.3.1.2 Communication

Seamless interaction between each component of a system depends on the presence of effective communication frameworks. The upcoming tools contribute to improve the efficiency and dependability of distributed systems by simplifying data interaction, workflow collaboration, and global service integration.

3.3.1.2.1 Message Broker

To manage asynchronous communication across microservices, Kafka was chosen due to its high productivity, scalability, and capacity for real-time data streams. This message broker is used to encourage effective communication between services.

Kafka is a distributed event streaming platform designed to manage large volumes of data across distributed systems [75]. It achieves great performance by processing billions of messages per day, maintaining high throughput and low latency, and scaling horizontally by adding more brokers to the cluster.

This tool uses multiple brokers to replicate data, offering high availability and fault tolerance. Due to this redundancy, there is never a single point of failure and data is always available, even when servers are down. Kafka stores all messages in a permanent way, acting as a distributed commit log. This makes it possible to record and replay data, improving offline and real-time data processing. Since it supports message queue and publish/subscribe patterns, it can be used in a variety of scenarios, and messages may be processed more effectively when users pull them consciously.

3.3.1.3 Data Storage

Each microservice requires its own database to be scalable and independent. MongoDB [76] is a solid NoSQL database that meets our specifications. It was selected for its adaptability, scalability, and simplicity of interaction with microservices.

Data are stored in flexible documents called Binary JSON (BSON); this gives versatility in data modelling by enabling different data structures inside the same collection. It promotes real-time aggregation for quick searches and high-throughput operations with indexing. Additionally, it provides a strong query language that supports text search, geographical searches, data aggregation, and CRUD[4] operations. It offers an extensive collection of operators for detailed queries.

MongoDB uses self-healing replica sets that automatically fail over to achieve high availability. This ensures continuous operation and redundancy of data in the unlikely event of hardware collapse. Data integrity is guaranteed for complex operations involving several documents or collections via MongoDB's support for multi-document ACID transactions. MongoDB is designed to have a high throughput of writes and reads. Compared to standard relational databases, its document approach enables faster queries, particularly for complex hierarchical data structures.

3.3.1.4 Documentation

We will take advantage of Swagger to ensure that they are accessible and maintained APIs. This tool offers a standardised method for API testing and documentation and is used to describe and communicate with microservices.

3.3.1.4.1 Swagger

Developed around the OpenAPI specification, Swagger is a powerful toolkit designed to make it easier to develop, describe, and apply RESTful APIs [77].

Similarly to OpenAPI, Swagger is compatible with many different technologies. Furthermore, this tool enables development and testing through API testing via the Swagger UI, which allows developers to test API endpoints directly from the documentation interface, and mocking, which allows developers to build mock servers based on the requirements API.

Swagger's integration features include version control for collaboration and change tracking, Continuous Integration and Continuous Delivery (CI/CD) for automated API testing and documentation, and interaction with popular API management systems such as Azure API Management.

[4]CRUD stands for Create, Read, Update and Delete - the four basic functions for interacting with persistent data storage in most software applications. This operations allow for basic persistence and data management.

3.4 Prototype

A particular approach to address distributed transactions in microservice systems is the Saga pattern. It defines a collection of several local operations that involve the same service. The system carries out a number of compensatory operations to reverse the changes made by the earlier local operations in the event that one of the operations fails.

The Saga pattern can be implemented in a microservices architecture in two primary ways: choreography and orchestration. Before diving into more details about these strategies in the following sections, it is important to describe the specific sagas envisioned for this project.

Sagas

Two main sagas have been designed for this meal ordering system to perform several important functions. The first is the procedure for creating a new user. The steps include registering the user, completing the registration, providing the activation information, and enroled the user in the system. If validation fails at any step, compensatory transactions ensure that the user is deleted, maintaining system integrity. Figure 3.2 illustrates the sequence of interactions to create a new user.

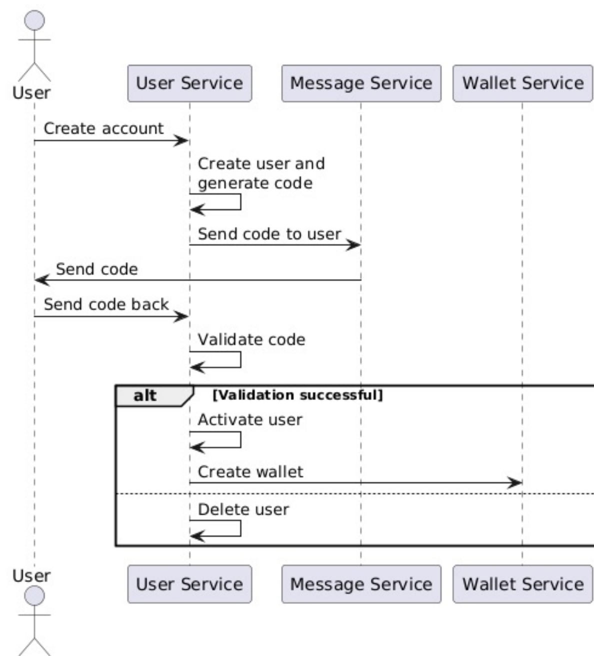


Figure 3.2: Sequence diagram for the saga of creating a new user. Source: author.

The dialogue between a new client and the system is described in the second saga. Everything is covered, starting with the menu selection and order confirmation. If the user has insufficient funds, compensatory transactions ensure that no order is created and the client is notified. After the fifth order, assuming that the order is placed successfully, a review request is sent. Figure 3.3 illustrates the sequence of interactions for the new client's order process.

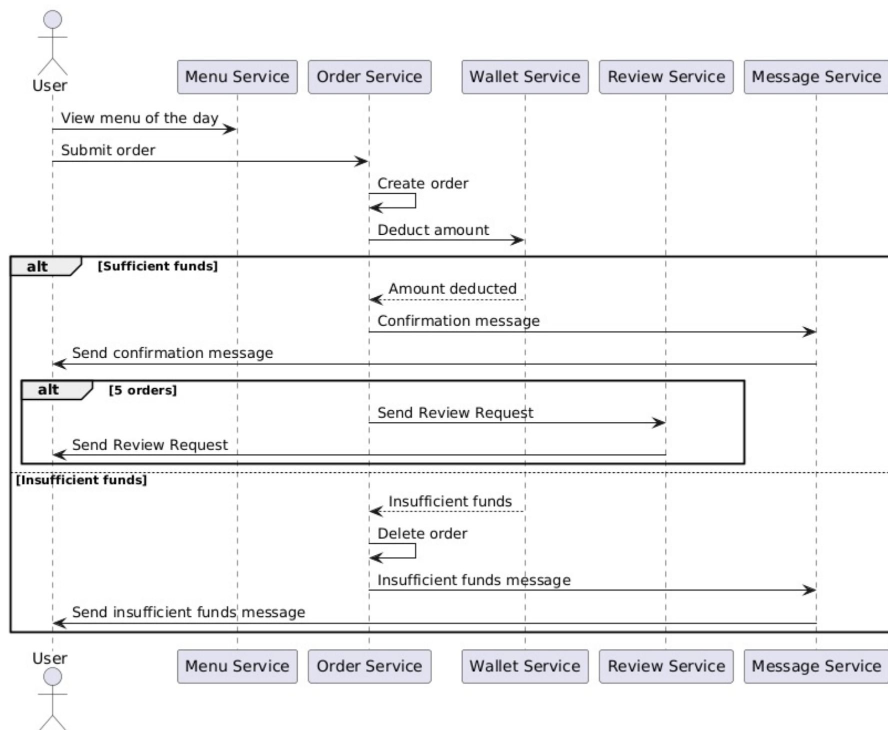


Figure 3.3: Sequence diagram for the saga of new client's order process. Source: author.

Now that the envisioned sagas have been introduced, let's discuss how these sagas can be implemented using the two primary approaches.

Microservices Design

These sagas need a solid and modular microservice architecture to be performed successfully. This section outlines the fundamental design principles and patterns that guide the system as a whole. The structure of microservices is based on standard components that provide scalability, robustness, and maintainability regardless of whether a choreography or orchestration approach is used.

The Boundary-Control-Entity (BCE) pattern is applied to structure all microservices. It is a robust and versatile structural pattern that provides an organised approach for structuring software components. By partitioning the microservices into three distinct layers, each with clearly defined duties, this architectural pattern encourages a clear separation of concerns and makes the system easier to scale and maintain.

The boundary layer behaves as an interface between the microservice and its external actors, be they consumers or other systems. Within the context of microservices, this layer often takes the form of asynchronous message listeners or RESTful API endpoints that encapsulate all communication functionality.

The application logic of the service is centred on the control layer. This layer controls the flow of data across the system, implements comprehensive business rules, and manages the interactions between the boundary and entity components. Lastly, the essential domain objects and data models that are necessary for the service can be found in the entity layer. The entities map the business data to the underlying data storage, ensuring that the microservice maintains a clear boundary around its own data, adhering to the principle of data autonomy. Each service manages its own database and defines the necessary operations in its domain models.

Establishing the system boundaries using an OpenAPI standard based on YAML was the initial stage in designing each microservice by creating interfaces for service routes. This standard ensures that the API structure is uniform for all services.

Listing 3.1 illustrate an example of a basic API configuration for a service. This standard ensures consistency in communication between services by providing a human-readable description of the API endpoints with the respective requests and response objects.

```
1  openapi: 3.0.3
2  info:
3    title: Menu API
4    version: 1.0.0
5
6  paths:
7    /api/v1/daily-menu:
8      get:
9        summary: Retrieve menu for a specific day
10       parameters:
11         - name: weekDay
12           in: query
13           required: true
14           schema:
15             type: string
16             enum: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]
17       responses:
18         '200':
19           description: Menu for the specified day
20           content:
21             application/json:
22               schema:
23                 $ref: '#/components/schemas/Menu'
24
25 components:
26   schemas:
27     Menu:
28       type: object
29       properties:
30         weekDay:
31           type: string
32           enum: [MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY]
33         dishes:
34           type: array
35           items:
36             $ref: '#/components/schemas/Dish'
37
38     Dish:
39       type: object
40       properties:
```

```
41            name: { type: string }
42            description: { type: string }
43            price: { type: number }
```

Listing 3.1: A sample of an OpenAPI configuration for menu service. Source: author.

To represent the data that will be transmitted between services, Data Transfer Objects (DTOs) are automatically generated once the API interfaces are generated. Listing 3.2 provides an example of a DTO. DTOs are used to ensure that data is structured correctly according to the API schema and sent to the corresponding client that made the request. The system maintains an apparent distinction between the data transport logic and the business logic by preserving the data apart from the domain model.

```
1  @Getter
2  @Setter
3  @AllArgsConstructor
4  @NoArgsConstructor
5  public class DishDTO implements Serializable {
6      private String name;
7      private String description;
8      private BigDecimal price;
9  }
```

Listing 3.2: A sample of a dish DTO in the menu service. Source: author.

The entity layer encapsulates the domain models and represents the objects that interact with the database. By isolating entity data from DTOs, the software preserves the integrity of its business rules and simplified maintenance. This layer of the system was developed applying the Panache framework and MongoDB, offering a direct high-scalability mapping of entities to the database [78]. A code sample of an entity can be found in Listing 3.3.

```
1  @Getter
2  @Setter
3  @MongoEntity(collection="user")
4  public class User extends ReactivePanacheMongoEntity implements Serializable {
5      private String password;
6      private String firstName;
7      private String lastName;
8      private String phone;
9      private RoleDTO role;
10     private StatusDTO status;
11     private String verificationCode;
12 }
```

Listing 3.3: A sample of a user entity using the Panache framework. Source: author.

Following this example, each service's key entities extend the ReactivePanacheMongoEntity class [79], enabling seamless integration with MongoDB using Quarkus Entities can communicate directly with the database by extending from this class, optimising efficiency for highly scalable applications.

Furthermore, as shown in lines 1 and 2, the Lombok library [80] is used to generate the getter and setter methods for entity attributes. Following the JavaBeans standard[5], this method ensures correct data encapsulation and minimises redundant code. The use of annotations enhances software maintainability by avoiding the need to manually write these methods for every attribute, promoting clean and simple code.

The repository layer abstracts away the specifics of data persistence from the control layer and is responsible for direct interactions with the database. This ensures that the repository handles the technicalities of data retrieval and storage, while the logic in the control layer stays focused on business problems.

Listing 3.4 provides an example of the implementation of the repository. The *findByWeekDay* function is used in this example to fetch a Menu object according to the day of the week. To ensure that the system handles missing data cases properly, the function throws a *DataException* with a specific error code if no menu has been identified.

```
1  @ApplicationScoped
2  public class MenuRepository implements PanacheMongoRepository<Menu> {
3
4      Menu findByWeekDay(WeekDay weekday) throws DataException {
5          Menu menu = find("weekDay", weekday).firstResult();
6          if (menu == null) {
7              throw new DataException(ErrorCode.MENU_NOT_FOUND);
8          }
9          return menu;
10     }
11 }
```

Listing 3.4: A sample of a menu repository using the Panache framework. Source: author.

This design provides a modular and maintainable foundation for the microservices in this system. By adhering to the principles of the BCE pattern, the system ensures that responsibilities are clearly defined, which facilitates codebase management and extension.

The control layer, which manages the business logic, is central to how the sagas will be implemented using either choreography or orchestration. In the subsequent sections, we will examine how the sagas are implemented in these two approaches and how the control layer handles service interactions.

3.4.1 Choreography Sagas

The choreography approach employs events to initiate activities and provide compensation so that microservices can communicate with each other without the need for a central coordinator. The autonomous behaviour of each saga in the meal ordering system enables the services to do their tasks in a distributed way by emitting and consuming events. The execution of the two sagas of the system is described in detail in the following sections.

[5] The JavaBeans standard defines a reusable software component model for Java. It outlines a set of guidelines to guarantee correct data encapsulation and enhance component compatibility in Java applications [81].

3.4.1.1 User Creation Saga

The process of creating a new user has few steps, which involve registration and activation to complete the registration. The events provide communication between the microservices in charge of each step.

When someone joins the system for the first time, the registration process begins (see the Listing 3.5). The service creates an user account with a *client* role (line 12) and a *not verified* status (line 13) and then generates a verification code (line 14). Subsequently, a message is sent to the topic *send-message* in Kafka channel (line 3 and 18-23), triggering the next step.

```
1  @ApplicationScoped
2  public class UserService {
3      @Channel("send−message")
4      Emitter<Message> userVerificationCodeEmitter;
5      @Inject
6      UserRepository repository;
7
8      public UserDTO createUser(UserDTO userDTO) {
9          User user = UserConverter.fromDto(userDTO);
10
11         user.setPassword(encoder.encode(userDTO.getPassword()));
12         user.setRole(RoleDTO.CLIENT);
13         user.setStatus(StatusDTO.NOT_VERIFIED);
14         user.setVerificationCode(VerificationCode.generate());
15
16         repository.persist(user);
17         userVerificationCodeEmitter.send(
18                 new Message(
19                         user.getPhone(),
20                         "Verification code: " + user.getVerificationCode(),
21                         "USER")
22                 )
23                 .toCompletableFuture().join();
24
25         return UserConverter.toDto(user);
26     }
27
28     (other methods)
29 }
```

Listing 3.5: A sample of the implementation of the user service in the choreography approach. Source: author.

Once the user receives the verification code, they must submit it back to the system. The *verifyCodeAndUpdateStatus* method is responsible for verifying that the code submitted matches with the one kept in the repository, as seen in the Listing 3.6.

```
1  @Channel("create−wallet")
2      Emitter<String> createWalletEmitter;
3
4  public UserDTO verifyCodeAndUpdateStatus(VerifyCodeAndUpdateStatusRequestDTO
       verifyCodeAndUpdateStatusRequestDTO) throws DataException {
5      String code = verifyCodeAndUpdateStatusRequestDTO.getCode();
6
7      User user = repository.findByPhone(verifyCodeAndUpdateStatusRequestDTO.getPhone());
```

```
8       if (!user.getVerificationCode().equals(code) || isOlderThanFiveMinutes(user.getCreationDate
            ())) {
9           repository.delete(user);
10          throw new DataException(ErrorCode.INVALID_VERIFICATION_CODE);
11      }
12
13      user.setStatus(StatusDTO.ACTIVE);
14      repository.update(user);
15      createWalletEmitter.send(user.getPhone()).toCompletableFuture().join();
16
17      return UserConverter.toDto(user);
18  }
```

Listing 3.6: A sample of the code validation and status update method. Source: author.

If it matches, the service records the changes and sets the user's status to *active*. In this way, the user is ready to perform additional interactions, such as submitting an order. Additionally, a wallet is created for the user by emitting a event to the wallet service through the topic *create-wallet*.

Otherwise, if the code does not match or has expired, the flow is halted, the system deletes the user, and an exception is triggered. This prevents the user from finishing the activation incorrectly. As detailed in Appendix A, the service was developed to provide the user with an adequate HTTP code and an informative message, thus enhancing the user's experience and simplifying diagnosis.

3.4.1.2 Order Creation Saga

The Order Creation Saga oversees every stage of the order lifecycle, from the moment a client requests the menu for a specific day to the order's creation, confirmation, and review. Like the other sagas, the microservices communicate with each other without the need for a central coordinator by independently producing and consuming events. This saga centres on submitting an order and updating the user's wallet.

The saga starts when a user requests the menu for a particular workday. In response, the menu service provides the dishes available that day, as in the following Listing 3.7.

```
1  public MenuDTO getMenuForDay(WeekDay weekDay) throws DataException {
2      if(weekDay == null)
3          throw new DataException(ErrorCode.BAD_REQUEST);
4      return MenuConverter.toDto(menuRepository.findByWeekDay(weekDay));
5  }
```

Listing 3.7: A sample of the menu for the day method. Source: author.

The end user chooses which dishes he wants to order after looking at the menu and places an order. After receiving the order, the service calls another service (Wallet Service) (line 25) to confirm if the user has sufficient funds. Listing 3.8 provides a sample of this premise.

```
1  @ApplicationScoped
2  public class OrderService {
3      @Channel("send−message")
4      Emitter<Message> orderVerificationCodeEmitter;
5      @Inject
```

```
6       OrderRepository orderRepository;
7
8       public OrderDTO createOrder(OrderDTO orderDTO) {
9
10          Order order = OrderConverter.fromDto(orderDTO);
11          order.setDate(new Date());
12          order.setStatus(StatusDTO.NOT_VERIFIED);
13          order.setVerificationCode(VerificationCode.generate());
14          order.setUuid(UUID.randomUUID().toString());
15
16          orderRepository.persist(order);
17
18          if(this.deductFunds(order.getPhone(), getValue(orderDTO).doubleValue())) {(...)};
19
20          return OrderConverter.toDto(order);
21      }
22
23      private boolean deductFunds(String phone, double value) throws DataException {
24          try {
25              WalletDTO walletDTO = walletClient.deductFunds(new WalletDTO(value, phone))
                    ;
26              return walletDTO.getBalance() >= value;
27          } catch (Exception e) {
28              e.printStackTrace();
29              return false;
30          }
31      }
32
33      (other methods)
34  }
```

Listing 3.8: A sample of the implementation of the order service. Source: author.

The user receives a confirmation message if sufficient funds are available (lines 2-8). This behaviour is shown in Listing 3.9. During this stage, the system checks if the user has completed the five orders (lines 10-13). If so, the order service notifies the review service, which forwards a review invitation to the user providing a review link.

However, in the case that the user does not have sufficient funds, the order is cancelled and the service notifies the user, via message service, that their request cannot be processed (lines 18-24).

```
1   if (this.deductFunds(order.getPhone(), getValue(orderDTO).doubleValue())) {
2               orderStatusEmitter.send(
3                           new Message(
4                                   order.getPhone(),
5                                   "Order confirmed with code: " + order.getVerificationCode
                                        (),
6                                   "ORDER")
7                       )
8                       .toCompletableFuture().join();
9
10              if (orderRepository.findAllOrders().stream()
11                      .filter(o -> Objects.equals(o.getPhone(), order.getPhone())).toList().size()
                            % 5 == 0) {
12                  reviewRequestEmitter.send(orderDTO).toCompletableFuture().join();
13              }
14
```

```
15          } else {
16              orderRepository.delete(order);
17
18              orderStatusEmitter.send(
19                          new Message(
20                                  orderDTO.getPhone(),
21                                  "Insufficient funds...",
22                                  "ORDER")
23                      )
24                      .toCompletableFuture().join();
25          }
```

Listing 3.9: A sample of the implementation for checking the user's available funds. Source: author.

3.4.2 Orchestration Sagas

The orchestration approach uses a central orchestrator to manage the saga workflow, coordinating the tasks of several microservices to ensure a smooth operation. With this approach, every stage of the saga can be easily identified and monitored, allowing robust error identification and corrective action as needed.

Below is a detailed description of how the two main sagas in the meal ordering system are being implemented.

3.4.2.1 User Creation Saga

Following the same stages mentioned in the choreography approach, upon the user's first inscription, the *createUser* function is triggered (see Listing 3.10). It works on the same premise as the choreography approach, designating a client role, an not verified status, and generating a code to validate the account.

The only way these two methods (Listing 3.5 and Listing 3.10) differ is in how they execute the flow. Listing 3.10 employs a different technique in which an event is transmitted directly to the orchestrator, which will handle the whole process, instead of Listing 3.5, which sends the event to the messaging service for user interaction.

```
1  @Channel("user-created")
2  Emitter<UserDTO> userCreatedEmitter;
3
4  public UserDTO createUser(UserDTO userRequest) {
5      User user = UserConverter.fromDto(userRequest);
6
7      user.setPassword(encoder.encode(userRequest.getPassword()));
8      user.setRole(RoleDTO.CLIENT);
9      user.setStatus(StatusDTO.NOT_VERIFIED);
10     user.setVerificationCode(VerificationCode.generate());
11
12     repository.persist(user);
13     UserDTO userDTO = UserConverter.toDto(user);
14     userCreatedEmitter.send(userDTO).toCompletableFuture().join();
15
16     return userDTO;
17 }
```

Listing 3.10: A sample of the code of the create user method in orchestration approach. Source: author.

Once the user has been created, the orchestrator listens to the *user-created* topic (line 1) and starts the user activation process. Using the business key from the user's phone and the required variables, the *userCreated* method launches a new process instance initiating the user activation stage, as seen in Listing 3.11.

```
1  @KafkaListener(topics = "user-created", groupId = "orchestrator", properties = {"UserDTO"
       })
2  public void userCreated(UserDTO userDTO) {
3      Map<String, Object> variables = new HashMap<>();
4
5      variables.put("phone", userDTO.getPhone());
6      variables.put("status", userDTO.getStatus());
7      variables.put("code", userDTO.getCode());
8      ProcessEngine processEngine = ProcessEngines.getDefaultProcessEngine();
9      RuntimeService runtimeService = processEngine.getRuntimeService();
10
11     runtimeService.startProcessInstanceByKey("ActivateUser", userDTO.getPhone(), variables);
12 }
```

Listing 3.11: A sample of the code of the create user listener. Source: author.

The Business Process Model and Notation (BPMN) depicted in Figure 3.4 represents the user activation stage. Upon instance initialisation, the message service sends a message to the user with the code that the system previously generated, allowing them to activate their account and continue with additional system activities. From this point on, the flow waits for user participation.

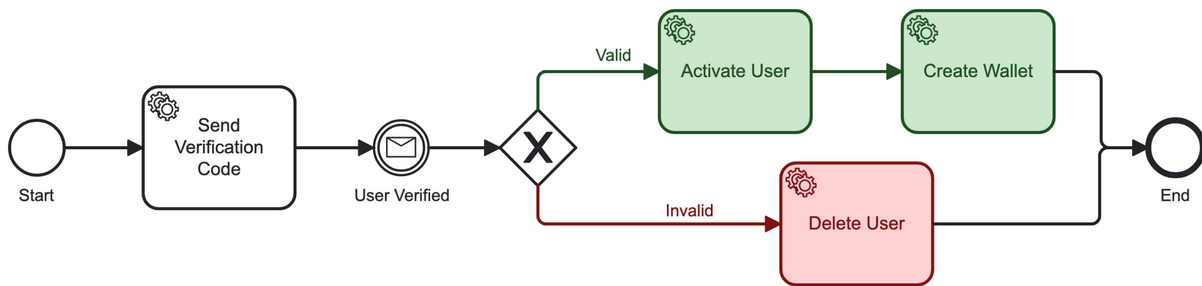


Figure 3.4: BPMN diagram for the user creation saga. Source: author.

When the user submits the code, the orchestrator detects his interaction and launches a method that validates if the code submitted matches the code that has been stored. The user's status is set to *active* if the code is valid and hasn't expired; otherwise, it stays *inactive*. A *user-verified* event is triggered to notify the orchestrator that the task has been completed.

The user's account is activated if his status is active. This is accomplished by making a request to the user API via a Java delegate (see Listing 3.12). As a consequence, a wallet setup request is also submitted to the API.

Unauthorised activation is prevented if the user's status is different from active, as this will stop the flow and delete the user from the system.

```java
1  @Component
2  public class ActivateUser implements JavaDelegate {
3      private final RestTemplate restTemplate;
4
5      @Autowired
6      public ActivateUser(RestTemplate restTemplate) {
7          this.restTemplate = restTemplate;
8      }
9
10     @Override
11     public void execute(DelegateExecution delegateExecution) throws Exception {
12         try {
13             String activateUrl = "http://localhost:8091/api/v1/users/activate/" +
                   delegateExecution.getVariable("phone");
14             restTemplate.getForObject(activateUrl, String.class);
15             System.out.println("User activated: " + delegateExecution.getVariable("phone"));
16         } catch (HttpServerErrorException e) {
17             System.err.println("Error response: " + e.getResponseBodyAsString());
18         } catch (Exception e) {
19             System.err.println("Failed to process user status: " + e.getMessage());
20         }
21     }
22 }
```

Listing 3.12: A sample of the code of the user activation request. Source: author.

Each task that comes after updating the user's status implements the *JavaDelegate* interface, meaning that it is in charge of carrying out a particular action inside of a BPMN process. A HTTP request is sent via a *RestTemplate* to the task's API, using variables (such as the user's phone number) that are acquired from the process context.

In the event that the API request is successful, the workflow proceeds, and the console may provide a confirmation message. Specific exceptions are detected and handled in the event of a failure, such as a server crash or connectivity issues. This ensures that the issue is identified and the flow is properly cancelled, preventing continuity with incorrect data.

The sequence diagram presented in Figure 3.5 provides a visual representation of the orchestration of the user creation process by demonstrating the interactions between the services and the orchestrator.

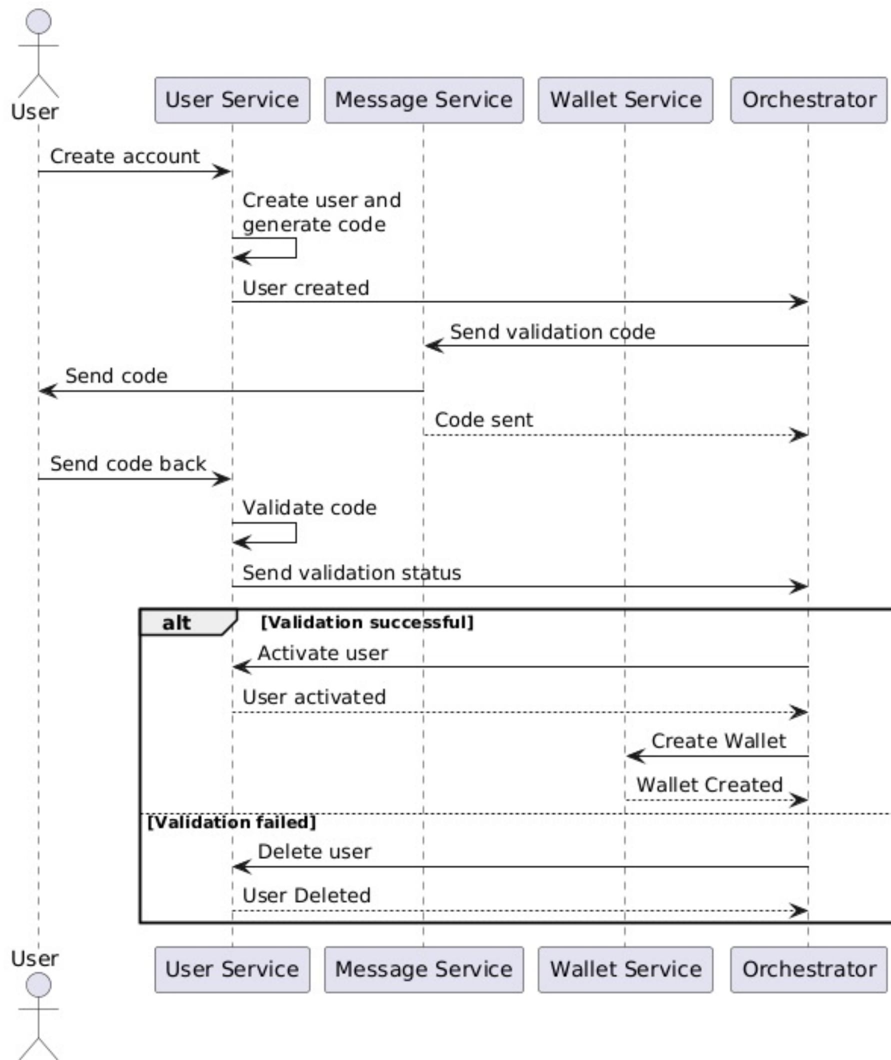


Figure 3.5: Sequence diagram for the orchestrated saga of creating a new user. Source: author.

3.4.2.2 Order Creation Saga

As with the previous approach, this process starts with the end user requesting the menu for a specific day and selecting the meals they desire. After submission, the service creates the order and informs the orchestrator through an event that a new order has been created (see Listing 3.13). From here, the orchestrator then manages the coordination between the wallet service, message service, and review service based on the order's state and the availability of funds.

```
1  public OrderDTO createOrder(OrderDTO orderDTO) {
2
3          Order order = OrderConverter.fromDto(orderDTO);
4          order.setDate(new Date());
5          order.setStatus(StatusDTO.NOT_VERIFIED);
6          order.setVerificationCode(VerificationCode.generate());
7          order.setUuid(UUID.randomUUID().toString());
8
9          orderRepository.persist(order);
10         orderCreatedEmitter.send(OrderConverter.toDto(order)).toCompletableFuture().join();
```

```
11
12          return OrderConverter.toDto(order);
13      }
```

Listing 3.13: A sample of the code of the create order method in orchestration approach. Source: author.

The orchestrator listens for the corresponding event and triggers a series of phases that coordinate the successful completion of the saga. These phases are visually represented in Figure 3.6.

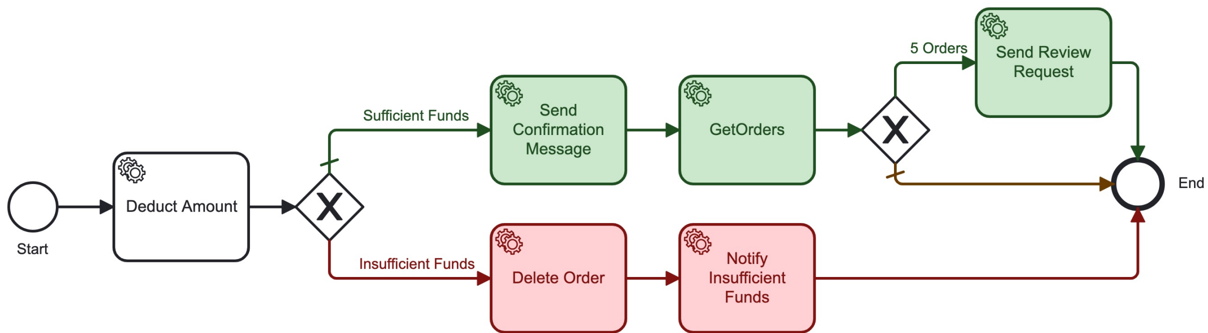


Figure 3.6: BPMN diagram for the order creation saga. Source: author.

The first stage begins with the orchestrator instructing the wallet service to deduct the order amount from the user's wallet. This is accomplished by submitting an HTTP request to the wallet API using a *JavaDelegate* task, using the user's phone number and the order's amount as context variables (see Listing 3.14).

```
1   @Component
2   public class DeductAmount implements JavaDelegate {
3       private final RestTemplate restTemplate;
4
5       @Override
6       public void execute(DelegateExecution delegateExecution) throws Exception {
7           try {
8               String activateUrl = "http://localhost:8094/api/v1/wallets/deduct";
9               WalletDTO walletDTO = new WalletDTO(Float.parseFloat(delegateExecution.
                    getVariable("value").toString()), delegateExecution.getVariable("phone").
                    toString());
10              WalletDTO userWallet = restTemplate.postForEntity(activateUrl, walletDTO,
                    WalletDTO.class).getBody();
11
12              Map<String, Object> variables = delegateExecution.getVariables();
13
14              assert userWallet != null;
15              if (userWallet.getBalance() > walletDTO.getBalance()) {
16                  variables.put("status", "FUNDS");
17              } else {
18                  variables.put("status", "NO_FUNDS");
19              }
20              delegateExecution.setVariables(variables);
21          } catch (Exception e) {
22              Map<String, Object> variables = delegateExecution.getVariables();
23              variables.put("status", "NO_FUNDS");
24              delegateExecution.setVariables(variables);
```

```
25          }
26      }
27  }
```

Listing 3.14: A sample of the code of the wallet deduct request. Source: author.

At this moment, the availability of sufficient funds determines the path that is followed. When the funds are sufficient, the orchestrator proceeds by contacting the message service and requesting it to send the user a confirmation message, using the *JavaDelegate* interface as well.

The orchestrator monitors the number of previous orders the user has placed as part of order tracking (see Listing 3.15). In the scenario that this is the fifth order, the orchestrator initiates an additional stage in which it interacts with the review service to provide the user with a review link and solicits feedback on their experience.

```
1   @Component
2   public class GetOrders implements JavaDelegate {
3       private final RestTemplate restTemplate;
4       @Override
5       public void execute(DelegateExecution delegateExecution) throws Exception {
6           try {
7               String getOrdersUrl = "http://localhost:8093/api/v1/orders/phone/" +
                        delegateExecution.getVariable("phone");
8               OrderDTO[] orders = restTemplate.getForObject(getOrdersUrl, OrderDTO[].class);
9
10              Map<String, Object> variables = delegateExecution.getVariables();
11              assert orders != null;
12              variables.put("numberOfOrders", orders.length);
13
14              delegateExecution.setVariables(variables);
15          } catch (Exception e) {(...)}
16      }
17  }
18
19  @Component
20  public class SendReviewRequest implements JavaDelegate {
21      private final RestTemplate restTemplate;
22
23      @Override
24      public void execute(DelegateExecution delegateExecution) throws Exception {
25          try {
26              String activateUrl = "http://localhost:8095/api/v1/reviews/send−link";
27              SendReviewLinkRequestDTO reviewLinkRequestDTO = new
                        SendReviewLinkRequestDTO(delegateExecution.getVariable("phone").toString
                        (), delegateExecution.getVariable("uuid").toString());
28              restTemplate.postForEntity(activateUrl, reviewLinkRequestDTO, String.class);
29          } catch (Exception e) {(...)}
30      }
31  }
```

Listing 3.15: A sample of the code of the order tracking stage. Source: author.

This process follows a different path if insufficient funds are identified as shown in the Listing 3.16. By contacting the order service and notifying the user that there are insufficient funds, the orchestrator rejects the order. This path guarantees that order attempts

that fail are handled correctly, protecting system consistency and the integrity of the user experience.

```
1  @Component
2  public class DeleteOrder implements JavaDelegate {
3      private final RestTemplate restTemplate;
4
5      @Override
6      public void execute(DelegateExecution delegateExecution) throws Exception {
7          try {
8              String deleteUrl = "http://localhost:8093/api/v1/orders/" + delegateExecution.
                   getVariable("uuid");
9              restTemplate.delete(deleteUrl);
10         } catch (Exception e) {(...)}
11     }
12 }
13
14 @Component
15 public class NotifyInsufficientFunds implements JavaDelegate {
16     private final RestTemplate restTemplate;
17
18     @Override
19     public void execute(DelegateExecution delegateExecution) throws Exception {
20         Message message = new Message();
21         message.setPhoneNumber(delegateExecution.getVariable("phone").toString());
22         message.setDescription("Insufficient funds....");
23         message.setService("ORDER");
24
25         String messageServiceUrl = "http://localhost:8099/api/v1/messages";
26
27         try {
28             restTemplate.postForObject(messageServiceUrl, message, String.class);
29         } catch (Exception e) {(...)}
30     }
31 }
```

Listing 3.16: A sample of the code of the insufficient funds path. Source: author.

Similarly to the choreography approach, in the event of a failure, exceptions are detected and handled to ensure that the cause is identified and the flow is appropriately cancelled, preventing continuity with wrong data.

The sequence diagram presented in Figure 3.7 provides a visual representation of the orchestration of the order creation process by demonstrating the interactions between the services and the orchestrator.

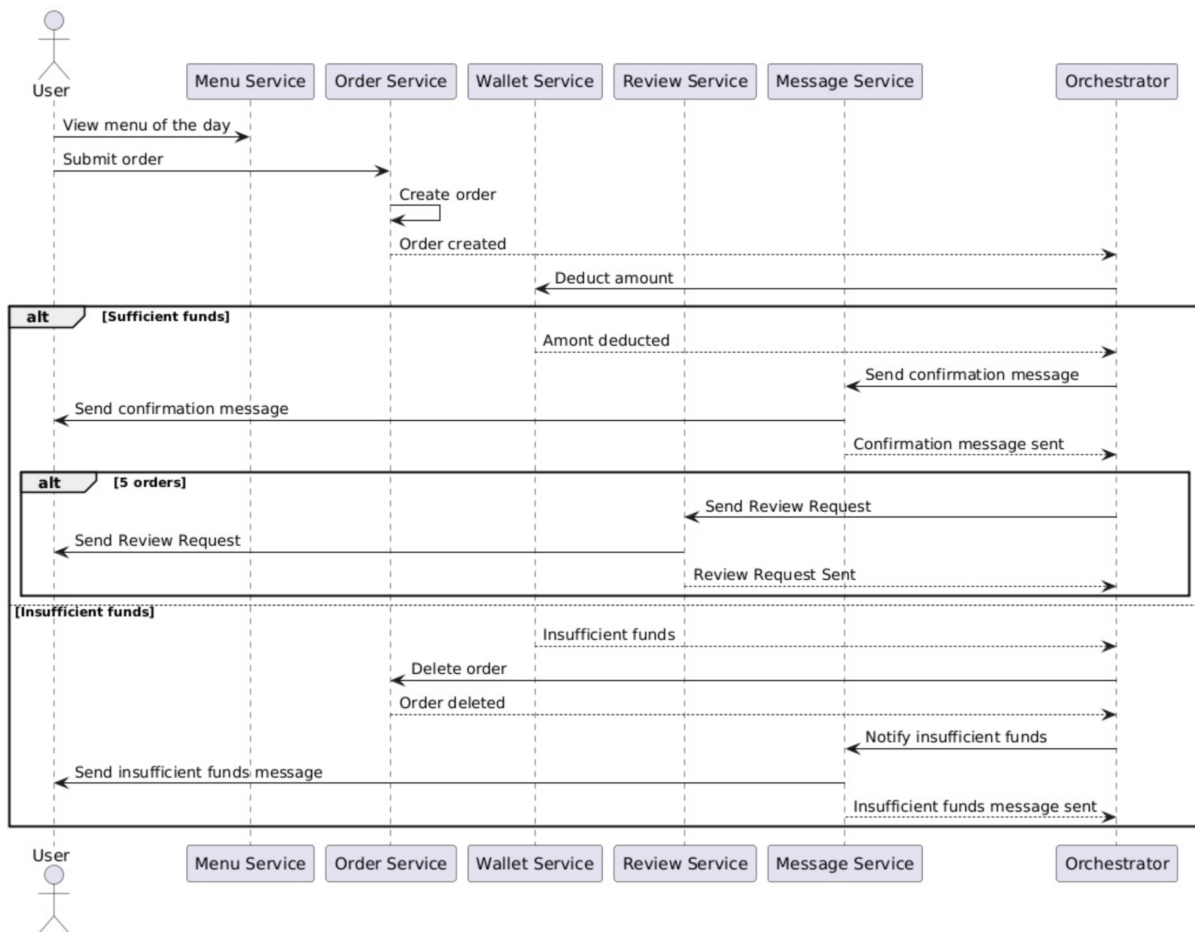


Figure 3.7: Sequence diagram for the orchestrate saga of new client's order process. Source: author.

Chapter 4

Validation and Results

As described in the previous section, both choreography and orchestration approaches provide different ways of executing sagas inside the meal ordering system. Through the use of events, the choreography approach promotes a decentralised communication structure that allows microservices to independently govern their interactions. On the other hand, the orchestration approach centralises control, making it possible for the orchestrator to supervise and manage the workflow of the saga.

This section aims to validate and compare the effectiveness of both approaches based on a set of pre-defined metrics. The testing scenarios and corresponding metrics collected are presented to offer a comprehensive evaluation of each approach's influence on system performance.

4.1 Methodology

In this section, the methodology used to evaluate the effectiveness of choreography and orchestration approaches for the meal ordering system is clarified. To evaluate various aspects of the system's execution, five distinct scenarios were established. The details of each scenario are given in Table 4.1.

Table 4.1: Summary of test scenarios. Source: author.

ID	Scenario	Description	Goal
1	Successful User Validation	This first scenario evaluates the process to register a new user and verify their information. This saga begins with the registration process, in which a verification code is sent to the user and an account with an unverified status is created. Upon submitting the correct code, the user's status is updated to active, the user's wallet is created, and the service states it.	To verify that the system registers new users accurately without errors and updates their status after successful validation.

2	Failed User Validation	In this scenario, an attempt happens to create a user with incorrect data. In order to access the service, the user must provide a verification code. If the code is invalid or has expired, the user will be deleted and an exception will be thrown.	To examine how the system handles failed user validation, ensuring sure that the user is not registered and that the relevant error messages are sent back.
3	Successful Order Creation	Our third scenario evaluates the system's capacity to properly handle a meal order when the user's wallet has sufficient money. The saga begins when the user requests the menu for a specific day. After selecting their meals, the system checks if the user has enough funds. If successful, a confirmation message is sent, and the order is recorded.	To ensure that the system correctly records new orders and deducts the appropriate amount from the user's wallet without errors.
4	Failed Order Creation	In this scenario, a user attempts to create a meal order without sufficient funds in their wallet. The system checks the wallet balance and, if insufficient, cancels the order and notifies the user.	To examine how the system handled the failure, ensuring that appropriate feedback was provided, the wallet status remained unchanged, and the order is deleted.
5	Sending a Review Request	In the last scenario, a user who has completed five successful orders receives a request to submit a review. After the order is confirmed, the system checks if the user has placed five orders. If so, the review service triggers it to send an invitation to the user to provide feedback.	To evaluate the ability of the system to monitor user orders and trigger review requests once the threshold of five orders is reached.

In addition to previously established test scenarios, a survey was conducted among fourteen software development professionals to evaluate their viewpoints and experiences with orchestration and choreography in microservices. Participants have distinct responsibilities (Figure 4.2) and different levels of familiarity (Figure 4.1) with the architecture of microservices.

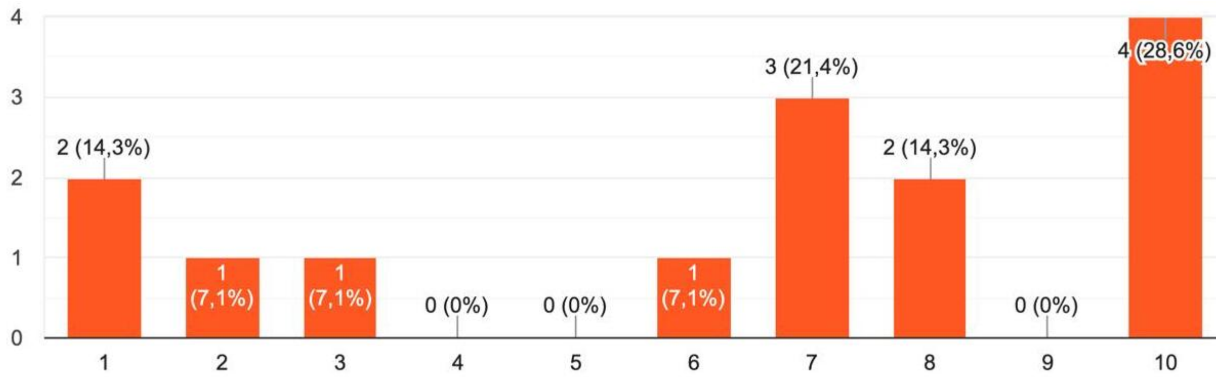


Figure 4.1: Answers from the survey on the familiarity with the microservices architecture of participants. Source: author.

Backend developers comprise the majority of participants, followed by frontend developers, DevOps engineers, and scrum masters and mobile solution architects in lower percentages (see Figure 4.2). In terms of experience, 57.1% of the participants have been employed in software development for more than five years, 28.6% for three to five years and 14.3% for one to two years.

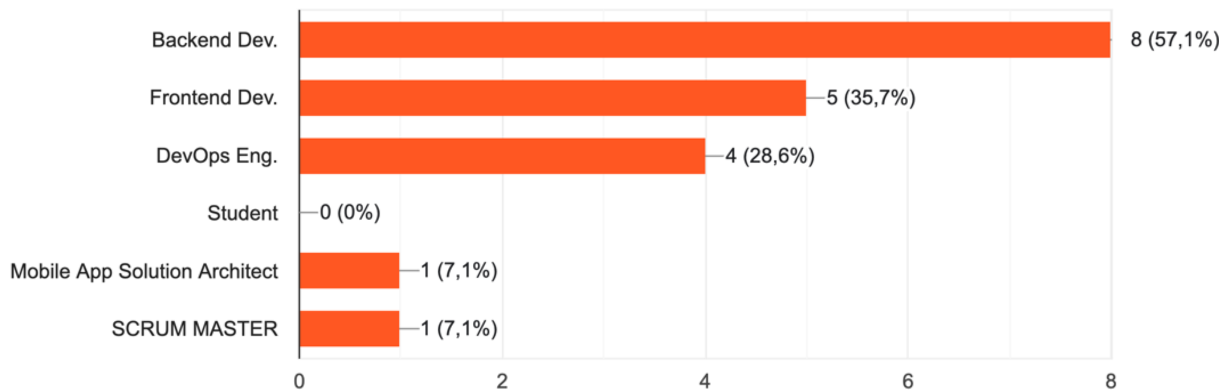


Figure 4.2: Answers from the survey on the roles of participants. Source: author.

4.2 Evaluation of Approaches

The evaluation of the two implemented sagas is evaluated in this section. It is supported by the results obtained from the five test scenarios, which highlight the distinct behaviours and characteristics of the respective implementations. In addition, the evaluation will include the findings of a survey conducted as a component of this study, offering actual support for the analysis.

Through the analysis of the scenarios and responses to the survey, this evaluation explores how each approach affects the system performance and user experience. Flexibility, service independence, management and monitoring, scalability, resilience and failure management, latency, and implementation complexity are the main areas of focus. Each element will be thoroughly examined, supported by the knowledge acquired during the testing process.

4.2.1 Flexibility, Scalability and Service Independence

Flexibility is a prerequisite in highly dynamic systems where services are often added or modified. Due to its minimal connectivity between services, choreography proved to be the most versatile approach. With minimal impact on the rest of the system, each microservice can grow on itself. Despite this, the system can expand naturally and adapt to constant changes without requiring costly reconfiguration.

In contrast, orchestration concentrates responsibility on a single component, which may lead to dependencies that impede the development of distinct services. Although orchestration offers more organised control, in circumstances when frequent changes are required, this centralisation limits the system's flexibility.

When comparing scenario one and scenario three under the choreography approach, we observed that the number of services and interactions involved significantly increased the complexity. The interactions in scenario one are rather simple and are restricted to wallet creation and user verification. Scenario three, on the other hand, demanded more steps and demonstrated how choreography became challenging as services grow, requiring the management of individual interactions and compensations by each service. In comparison, since the orchestrator monitored the whole process, the orchestrated system maintained a more linear and regulated flow even as the number of stages increased.

Scalability was a significant advantage of choreography. Every service can be scaled independently, allowing for minimum dependence in response to shifts in workload. As a result, the system can handle many services with greater efficiency and grow horizontally without suffering bottlenecks.

However, since the orchestrator acts as the main element, orchestration could face scaling challenges. The orchestrator might become a source of conflict as the system grows, affecting performance, and requiring more resources to manage the expanding workload.

Scenario three, which contains a higher number of service interactions, clearly illustrated the differences in scalability between the different sagas. Although services expanded separately in the choreographed saga, it became increasingly clear that event propagation is significantly more complicated than previously thought. On the other hand, when the orchestrator's workload increased, orchestration found it difficult to scale effectively, highlighting the importance of careful planning in bigger systems to prevent overloading the central orchestrator.

According to the answers of the survey (see Figure 4.3), the low coupling between choreography services proved to be easier for 30.8% of the participants to scale and maintain in production environments. However, 15.4% participants preferred orchestration due to its structured management, which promotes scenarios in which service independence is not as critical. The majority were not familiar with any approach in a production environment, and only 7.7% considered all approaches to be equally levelheaded.

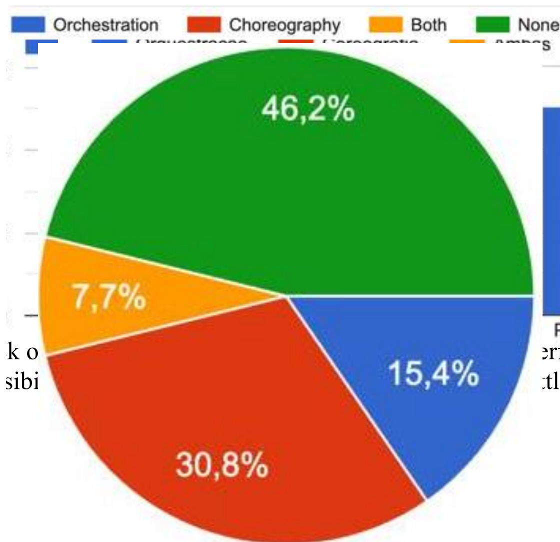


Figure 4.3: Answers from the survey about the scalability in a production environment. Source: author.

4.2.2 Monitoring and Management

Monitoring choreography is challenging as no one component can provide a full overview of the workflow. Since each service is responsible for maintaining its own state, debugging and tracking down interactions become harder, particularly when the system becomes more complex. Monitoring and logging technologies have proven to be necessary to monitor activity across distributed services and debug errors.

On the other hand, orchestration is excellent in delivering transparent management. The orchestrator is responsible for managing and supervising the entire workflow, which enables tracking of the system's state, the detection of failures, and the execution of recovery procedures. In challenging situations where it is imperative to have access to the current state of the system, this centralised monitoring is quite helpful.

In a choreographed system, adding a new service could be easy, but it also means that monitoring will become challenging because each new service demands to have its state handled individually. However, integrating a new service into the orchestration requires setting up the orchestrator to handle interactions; but, workflow management becomes simpler with centralised monitoring.

As shown in Figure 4.4, four participants in the survey experienced the lack of central visibility (a) in choreography as a disadvantage, while orchestration was viewed as having more favourable effects in that field. Additionally, one participant reported that debugging was equally difficult in both ways, and two participants said that choreography and orchestration presented unique difficulties in this regard (b). And, although no one brought up the topic in choreography, five participants acknowledged the impact of performance bottlenecks on orchestration (c).

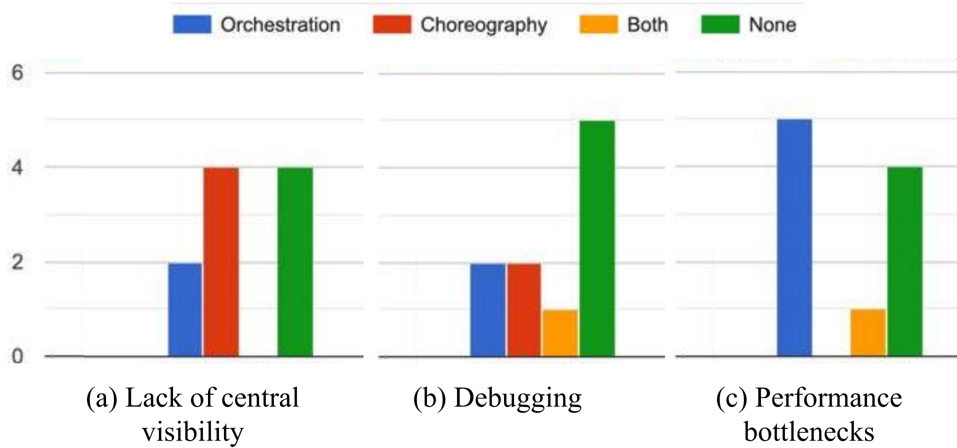


Figure 4.4: Answers from the survey about the monitoring and management of microservices. Source: author.

4.2.3 Resilience and Failure Management

Both approaches offer ways to promote resilience, but they handle failure management differently. In choreography, a distributed approach to resilience is achieved by letting each service handle its own failures. In decentralised systems, where distinct services require autonomy to manage their own recovery, this offers robustness.

By centralising failure management, orchestration empowers the orchestrator to ensure process integrity while coordinating compensatory actions. This centralised control simplifies recovery in intricate workflows, but poses a single point of failure that could threaten the entire process if not properly controlled.

The scenarios two and four most successfully illustrated failure management and resilience. Because each service handled exceptions individually in the coordinated saga of both cases, monitoring and recovering from mistakes became harder due to the dispersed nature of error handling. Conversely, the primary benefit of the orchestrated approach was demonstrated in these scenarios sagas: the orchestrator had better control over the recovery treatments by being able to centrally supervise the reverse process. Thus, in instances with more complicated failure conditions, the orchestration approach's centralised compensating logic outperformed other approaches.

Orchestration performs exceptionally well in centralised monitoring and fault recovery when it comes to resilience and fault management. However, as some interviewees pointed out, this may come at the cost of a single point of failure. The use of external services by 42.9% of the survey participants was observed. This might challenge the choreography approach, particularly if events need to be tracked and acquired in other systems.

The survey's findings, illustrated in Figure 4.5, revealed that, in contrast to orchestration's central management benefits, choreography had more difficulty monitoring event flows and managing race circumstances, with three participants mentioning such concerns and one bringing up racial conditions.

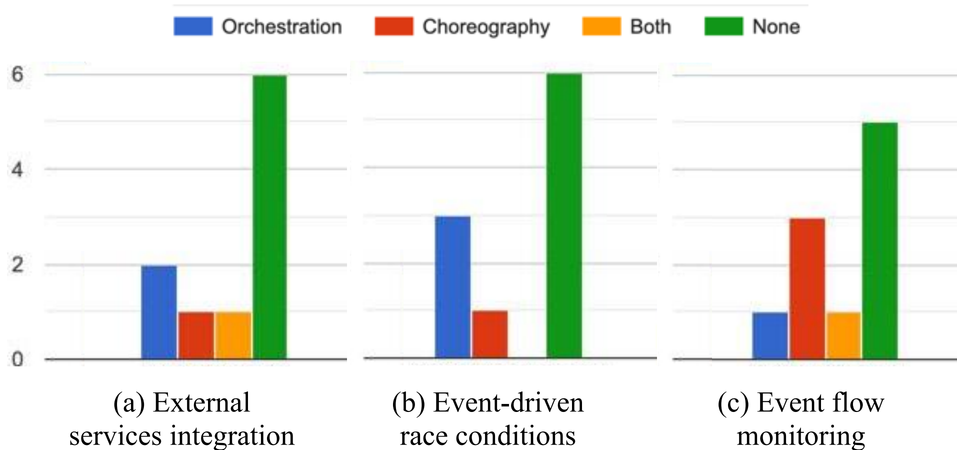


Figure 4.5: Answers from the survey about the resilience and failure management of microservices. Source: author.

4.2.4 Latency

In systems that use asynchronous communication, latency is a crucial performance component. As choreography is decentralised, services can interact directly with each other without passing through a centralised orchestrator, which usually results in lower latency. However, there may be additional costs that might result in higher latency when the number of services increases and the complexity of event-based interactions expands.

On the other hand, due to centralised management in orchestration, it often results in higher latency. More steps and processing time are included when the orchestrator must manage and handle every interaction, particularly in systems with an abundance of services.

The latency results between scenario one and scenario three demonstrate how, as the number of services increases, the type of interactions in choreography can result in additional costs. The smallest number of steps in scenario one coreographed contributed to the lower latency. Conversely, scenario three's increased event propagation and service-to-service communication resulted in higher overall latency.

The orchestrated scenarios consistently demonstrated greater latency. This results because the orchestrator has to supervise every interaction and introduces a control point at every turn in the sagas. This centralisation has less of an effect in simpler situations, such as in scenario one. Nevertheless, in more intricate scenarios such as scenaro three, the number of stages that rely on central coordination results in a boost in the overall execution time as each service had to wait for the orchestrator's commands before moving forward.

4.2.5 Implementation Complexity

The choreography and orchestration approaches to developing a saga differ significantly in complexity. Given that each microservice in choreography is in charge of processing events, compensatory logic, and error management, complexity is inevitably spread throughout the system. Due to its decentralised structure, each service must process events and maintain its own state independently, increasing the difficulty of design and development,

especially in situations where there are a lot of interactions.

In the third scenario, for instance, the Order Service, Wallet Service, and other services have to collaborate through events instead of direct orchestration. As a result, developers must control these event flows and address any failure cases on their own. Additional levels of complexity are introduced by the necessity to implement compensations in each service, as any mistake or discrepancy at one service must cause compensatory actions to be triggered throughout the whole chain of services. A thorough understanding of distributed systems and asynchronous communication is necessary to maintain such an architecture as the system scales.

Furthermore, the usage of event-driven communication frameworks, such as Kafka, places additional technological requirements. Developers should have knowledge of this technology and ensure flexible, consistent, and efficient event pipelines, especially in situations where compensations are necessary to maintain reliability.

The orchestration approach, on the other hand, isolates the saga logic into the orchestrator. The complexity at the service level is greatly reduced as a result. Every service does not have to handle events or compensation logic itself; it only has to perform tasks according to instructions from the orchestrator. Because all of the services are made simpler by this centralisation, developers are able to concentrate on the essential business logic rather than the complexities of saga management.

In the same scenario, for example, the orchestrator supervises the entire workflow ensuring that the activities are completed in the correct order and managing any errors or refunds from a single location. Nonetheless, there are disadvantages to this centralisation. Orchestration makes the development of services easier, but it also makes the orchestrator more complex. The orchestrator is a vital part of the system that has to be carefully designed to handle all potential scenarios, failures, and compensations. Every error or failure in the orchestrator introduces a single point of failure that must be properly controlled and can impact the workflow as a whole.

The survey's findings provide additional insight on the state of technology and the knowledge required to implement each strategy (see Figure 4.6 and Figure 4.7). For example, 78.6% of participants were familiar with Kafka, the most widely used message broker, whereas 35.7% were familiar with RabbitMQ. Nonetheless, a sizable portion (14.3%) had no prior experience with event-driven systems.

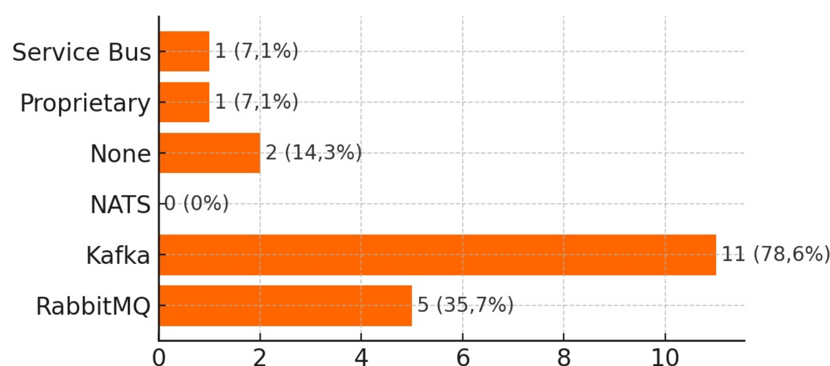


Figure 4.6: Answers from the survey on the familiarity with choreography approach tools. Source: author.

In terms of orchestration tools, AWS Step Functions and Camunda were the most familiar both indicated by 21.4% of participants, while 57.1% had no familiarity with orchestration tools.

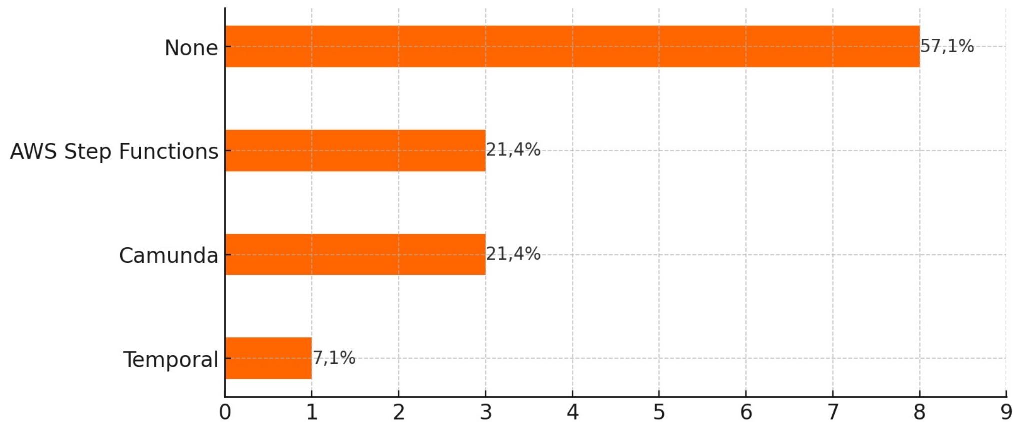


Figure 4.7: Answers from the survey on the familiarity with orchestration approach tools. Source: author.

These findings highlight the fact that although decentralising choreography distributes complexity across the services, it also necessitates expertise in specific technologies, such Kafka or RabbitMQ, in order to handle the system's event-driven nature. Even if orchestration is easier at the service level, in order to prevent becoming a single point of failure or bottleneck in the system, the orchestrator must be carefully designed and have a thorough grasp of orchestration platforms. Achieving a balance between technological proficiency and complexity management is crucial for the effective use of both strategies.

4.3 Validation Overview

Microservice architecture, which offers scalability, robustness, and agility, has become an essential part of modern software development. In microservice architecture, orchestration and choreography are two popular approaches. Depending on the specifications of the system, the relationships between services, and the performance goals, each approach offers unique advantages and challenges. This section reviews the results of our study as well as relevant literature, including research by [3, 38, 41]. These studies provide helpful background information to evaluate orchestration and choreography in terms of performance, scalability, and maintainability.

As stated by our investigation, choreography's main benefits are flexibility and service independence. Microservices with choreography are loosely connected and communicate with each other using event-driven techniques, making scalability and shifts easier without significant impact on the entire system. This finding is supported by [3] which demonstrates that in scenarios without a load balancer, choreography consistently performed better than orchestration, especially in terms of response times. This independence is especially beneficial for systems that need to update or modify services regularly. However when systems get larger and more complex, it gets harder to handle a lot of distinct independent interactions.

However, orchestration adds the possibility of performance bottlenecks while centralising control, making it easier to coordinate service interactions. As a single point of coordination for all microservices, the orchestrator improves operational dependability and management, especially in complex systems. This was shown in situations involving several transactions, where the organised method of orchestration offered more control over error management and compensations. [38] pointed out that because centralisation creates a single point of contact for monitoring and controlling process execution, it promotes easier maintenance in distributed transactions. However, [3] noted that this pivotal position might have an adverse impact on performance, particularly in cases with high traffic, as it tends to increase latency and resource consumption due to constant interaction with the orchestrator.

Choreography often performs better in terms of scalability since microservices may scale independently in the absence of a central controller. This is particularly useful for systems that manage several users at once [3]. Because of its adaptability, choreography is perfect for systems with different demands as it enables services to grow horizontally without compromising the structure as a whole. But as [41] pointed out, when more services are added, it becomes more challenging to oversee and manage interactions in the absence of a central coordinator.

On the other hand, since the orchestrator coordinates all service interactions, scalability problems arise with orchestration. System performance may be slowed down by the orchestrator if there are more services and interactions than there is capacity for. Since the orchestrator provides visibility into the whole process, orchestration still has advantages in situations where control and workflow monitoring are essential. Additionally, [3] discovered that although orchestration performs worse than choreography in terms of response times, in some situations it is more resource-efficient since it uses less computational resources.

Another area where these two techniques contrast is in failure management. By letting each service handle its own failures and compensations, choreography improves resilience by allocating responsibility for addressing failures amongst services. However, since each microservice must separately maintain its state and error recovery, this decentralised approach makes it more difficult to recover from failures that impact several services [3, 41]. Conversely, orchestration controls failure management, giving the orchestrator more control over how failures are recovered and compensations are handled across services. High-complexity scenarios addressed in our study demonstrate how this centralised method streamlines error handling in complex workflows and facilitates recovery from system-wide failures.

Choreographed systems frequently have lower latency since services may communicate directly with each other without a central orchestrator. Compared to orchestration, choreography consistently produced lower latency in situations with simpler systems with fewer services. However, maintaining low latency becomes more difficult as the number of services and events rises due to the complexity of managing distributed event flows. On the other hand, since an orchestrator is involved in each transaction, orchestration results in higher latency. Communication is slowed down by this additional step, especially in complex workflows with several phases. Likewise to our findings, [3] observed that when the number of services increased, the performance difference between choreography and orchestration grew, with choreography continuing to outperform orchestration in response

times even in situations with more traffic and more advanced services.

Each of the two approaches has a different level of implementation complexity. With choreography, complexity is divided across services, and each microservice is responsible for overseeing its own compensations, events, and states. Both [38, 41] noted that although choreography improves performance and makes better use of resources, its distributed architecture makes it more challenging to set up and maintain. Service-level implementation is made easier by orchestration, which centralises workflow management and error handling inside the orchestrator. But this centralisation transfers complexity to the orchestrator, who has to deal with all potential processes, failures, and adaptations. Therefore, a strong and well-designed orchestrator is essential to avoid bottlenecks.

In conclusion, the decision between orchestration and choreography is mostly based on the specific requirements of the system and how efficiently performance, scalability, and management complexity are balanced. When scalability, low latency, and service independence are important concerns, choreography performs very well. Due to its event-driven, loosely coupled design, which enables effective scalability of services, it is perfect for dynamic, high-traffic environments. Yet as systems get bigger, this flexibility comes with greater complexity management of relationships.

However, orchestration works well in systems that require workflow management, error handling, and centralised control. Orchestration streamlines the management of complex transactions and provides simpler maintenance via a central coordinator, although introducing latency and extra resource consumption.

These dilemmas are amply demonstrated by the studies we reviewed and our own conclusions. The following graphs provide visual representations of the comparison between choreography and orchestration across key performance metrics for both sagas.

In Figure 4.8, we compare the performance of choreography (blue area) and orchestration (orange area) in the user creation saga, which illustrates a simpler flow of service interaction. The blue region dominates the parameters of scalability, flexibility, and autonomy, making it evident how strong the choreography is.

However, as the larger orange area indicates in these characteristics, orchestration offers more control over process monitoring and error management. This suggests that orchestration works best in controlled scenarios, while choreography works better on smaller systems or those that prioritise independence and low latency.

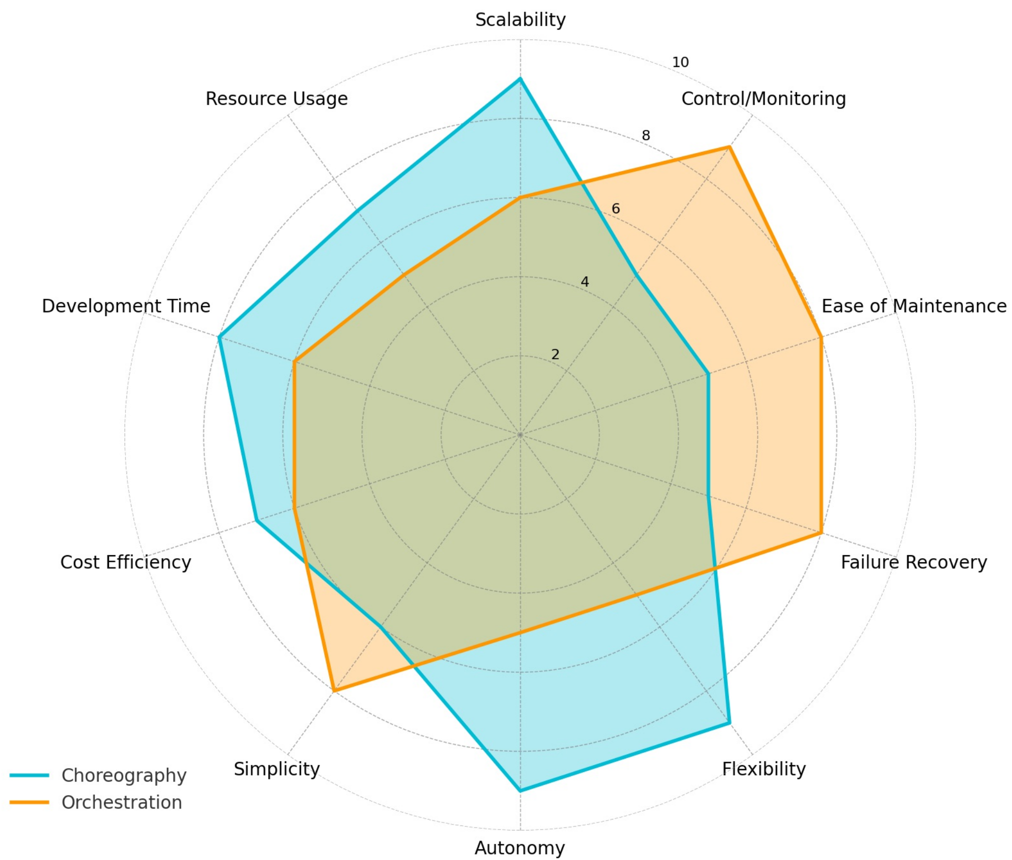


Figure 4.8: A network graph to aid in decision-making for simpler saga when comparing choreography and orchestration. Source: author.

The comparison analysis for the order creation saga, a more sophisticated interaction flow with several dependencies, is shown in Figure 4.9. The benefits of orchestration (orange region) are more evident here, especially when it comes to centralisation's key role in error handling, workflow control, and simplicity.

Choreography (blue area) continues to have an advantage in terms of scalability, flexibility, and autonomy. Despite these advantages, it should be noted that it may provide better performance in dynamic high-traffic scenarios but at the expense of more complicated service interaction management.

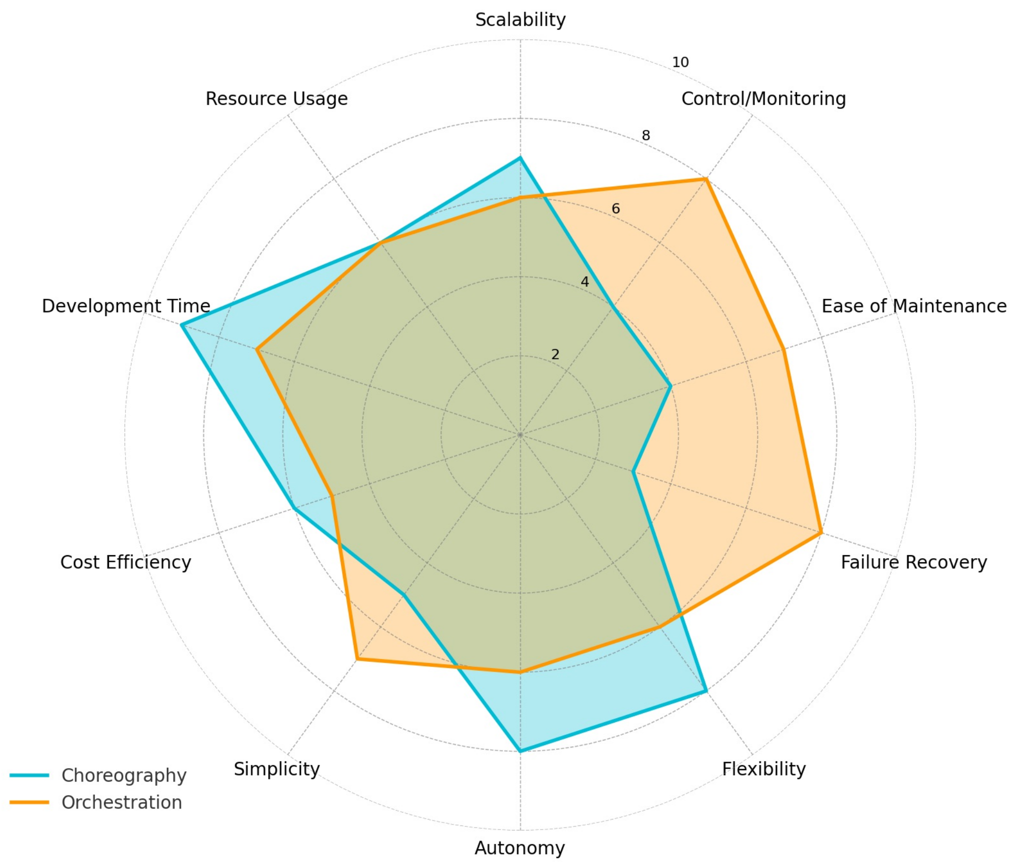


Figure 4.9: A network graph to aid in decision-making for complex saga when comparing choreography and orchestration. Source: author.

These graphs aid in illustrating the advantages and disadvantages of both strategies for different systems complexity levels. The choreography in user creation saga is excellent because to its loose coupling and independence, but the orchestration offers stronger control over handling failures. In order creation saga, choreography still has benefits in highly scalable, distributed systems, but orchestration's centralisation becomes increasingly important as transaction complexity rises.

Chapter 5

Conclusion

As software development has grown, microservice architecture has become a fundamental architecture that makes systems more robust, scalable, and flexible. The present research explored the benefits and drawbacks of orchestration and choreography, the two primary approaches for executing sagas in microservices.

The results indicate that choreography offers more flexibility and scalability due to its decentralised structure. In simpler scenarios, microservices can function independently and interact directly, resulting in a typically lower latency. However, monitoring interactions in more sophisticated systems can be challenging and requires a thorough understanding of distributed systems and asynchronous communication.

On the other hand, orchestration offers centralised control over fault management and workflow, making it easier to do maintenance and recovery in challenging situations. However, since this strategy requires continuous communication with the orchestrator, it can result in higher latency, particularly in systems with a high volume of services and interactions.

In light of these findings, the decision between choreography and orchestration should be made in consideration of the specific demands of each system. Choreography could be the best strategy for systems that prioritise scalability, service independence, and low latency. However, orchestration is clearly a better option for applications that demand strong process management and error control.

Furthermore, knowledge of the underlying technologies, such as orchestration tools and/or event-driven communication frameworks, is required for the implementation of both approaches. Research findings and related work have demonstrated that the knowledge required to deal with these technologies influences how successful microservices architecture are.

It is imperative to do more research on the adaptability of these architectural approaches for different operational environments. Both orchestration and choreography have special advantages, and by comprehending how to use them in bigger scenarios, novel possibilities for the development of robust, scalable, and efficient microservices might become feasible. This investigation provides interesting paths for future research and opens the door to practical challenges beyond the scope of this study.

5.1 Future Work

Although this study has provided insightful information on the decision between orchestration and choreography, there are still a number of areas for future investigation, such as:

1. **Application in more everyday scenarios:** The following studies might concentrate on the practical implementation of these methodologies in other sectors, such as financial services, healthcare, or e-commerce, where demands and constraints can differ considerably.

2. **Combination of approaches:** Exploring hybrid solutions that integrate both choreography and orchestration is another promising field of research. Previous case studies, like the ones covered in Section 2.1.7, have demonstrated that a hybrid strategy may provide the greatest of both worlds by enabling more flexibility and control.

3. **Analysis of new technologies:** As new microservices frameworks and tools appear, it will be crucial to continuously evaluate orchestration and choreography technologies. Future studies could investigate the potential impact of new technologies, including cloud computing and real-time events, on these approaches.

4. **Large-scale performance analysis:** Additional research should concentrate on this topic, since testing and simulations in real-life environments can offer a deeper understanding of how both strategies behave under different load scenarios.

Chapter 6

Bibliography

[1] F. Auer, V. Lenarduzzi, M. Felderer, and D. Taibi, "From monolithic systems to microservices: An assessment framework," *Information and Software Technology*, vol. 137, p. 106600, 9 2021. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950584921000793

[2] I. N. Borges, "Saga pattern for microservices architecture," 3 2023. [Online]. Available: https://blog.avenuecode.com/saga-pattern-for-microservices-architecture

[3] H. Kristianto and A. Zahra, "Performance analysis of choreography and orchestration in microservices architecture," *Journal of Theoretical and Applied Information Technology*, vol. 99, pp. 4220–4230, 9 2021.

[4] A. Sharma, M. Kumar, and S. Agarwal, "A complete survey on software architectural styles and patterns," *Procedia Computer Science*, vol. 70, pp. 16–28, 2015. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S187705091503183X

[5] M. Fowler, "Microservice trade-offs," 7 2015. [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html

[6] S. Newman, "Building microservices," 2015. [Online]. Available: http://safaribooksonline.com

[7] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, 2003.

[8] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: yesterday, today, and tomorrow," 6 2016. [Online]. Available: http://arxiv.org/abs/1606.04036

[9] C. Richardson, "Pattern: Decompose by business capability." [Online]. Available: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

[10] Z. Sunagatov, "Microservice architecture patterns: Decomposition patterns," 3 2023. [Online]. Available: https://hackernoon.com/microservice-architecture-patterns-part-1-decomposition-patterns

[11] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges," *IEEE Access*, vol. 11, pp. 88 339–88 358, 2023.

[12] A. Davis, "An in-depth guide to microservices design patterns," 12 2023. [Online]. Available: https://www.openlegacy.com/blog/microservices-architecture-patterns/

[13] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: A systematic mapping study," in *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2018, pp. 221–232. [Online]. Available: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0006798302210232

[14] A. Akbulut and H. G. Perros, "Performance analysis of microservice design patterns," *IEEE Internet Computing*, vol. 23, pp. 19–27, 11 2019.

[15] M. Abbott, "Microservice aggregator pattern," 3 2020. [Online]. Available: https://akfpartners.com/growth-blog/microservice-aggregator-pattern

[16] G. D. Maayan, "Top 10 microservices design patterns and their pros and cons," 7 2024. [Online]. Available: https://www.computer.org/publications/tech-news/trends/microservices-design-patterns

[17] R. Harmes and D. Diaz, *Pro JavaScript design patterns*. Apress, 2008.

[18] Z. Yueping, L. Yuefan, and X. Kesheng, "The compound pattern on the chain of responsibility and observer," in *2009 International Forum on Computer Science-Technology and Applications*, vol. 3. IEEE, 2009, pp. 420–422. [Online]. Available: http://ieeexplore.ieee.org/document/5384908/

[19] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso, *The Database-is-the-Service Pattern for Microservice Architectures*. Springer Verlag, 2016, vol. 9832 LNCS, pp. 223–233. [Online]. Available: http://link.springer.com/10.1007/978-3-319-43949-5_18

[20] M. K and M. P, "Evaluation of data storage patterns in microservices archicture," in *2020 IEEE 15th International Conference of System of Systems Engineering (SoSE)*. IEEE, 6 2020, pp. 373–380. [Online]. Available: https://ieeexplore.ieee.org/document/9130516/

[21] A. W. S. Inc., "Database-per-service pattern." [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/database-per-service.html

[22] C. Richardson, "Pattern: Shared database." [Online]. Available: https://microservices.io/patterns/data/shared-database.html

[23] A. W. S. Inc., "Shared-database-per-service pattern." [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/shared-database.html

[24] H. Dhaduk, "6 observability design patterns for microservices every cto should know," 1 2023.

[25] C. Richardson, "Pattern: Distributed tracing." [Online]. Available: https://microservices.io/patterns/observability/distributed-tracing.html

[26] IBM, "What is distributed tracing?" [Online]. Available: https://www.ibm.com/topics/distributed-tracing

[27] C. Richardson, "Pattern: Application metrics."

[28] ——, "Pattern: Circuit breaker."

[29] M. Abbott, "The circuit breaker pattern - dos and don'ts," 7 2019.

[30] M. Fowler, "Blue green deployment," 3 2010. [Online]. Available: https://martinfowler.com/bliki/BlueGreenDeployment.html

[31] A. W. Services, "Blue/green deployments on aws," 9 2021. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/blue-green-deployments/welcome.html

[32] G. D. Maayan, "Blue-green deployment for software applications: Pros and cons," 11 2022. [Online]. Available: https://www.computer.org/publications/tech-news/trends/blue-green-deployment-for-software-applications

[33] H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SIGMOD international conference on Management of data - SIGMOD '87*. ACM Press, 1987, pp. 249–259. [Online]. Available: http://portal.acm.org/citation.cfm?doid=38713.38742

[34] C. McCaffrey, K. Kingsbury, and N. Narula, "Distributed sagas," 2015. [Online]. Available: http://gotocon.com/dl/goto-chicago-2015/slides/CaitieMcCaffrey_ApplyingTheSagaPattern.pdf.

[35] C. Richardson, "Pattern: Saga." [Online]. Available: https://microservices.io/patterns/data/saga.html

[36] S. Aydin and C. B. Cebi, "Comparison of choreography vs orchestration based saga patterns in microservices," in *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*. IEEE, 7 2022, pp. 1–6. [Online]. Available: https://ieeexplore.ieee.org/document/9872665/

[37] A. W. Services, "Saga pattern." [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/saga-pattern.html

[38] C. K. Rudrabhatla, "Comparison of event choreography and orchestration techniques in microservice architecture," 2018. [Online]. Available: www.ijacsa.thesai.org

[39] A. Nadeem and M. Z. Malik, "A case for microservices orchestration using workflow engines," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. ACM, 5 2022, pp. 6–10. [Online]. Available: https://dl.acm.org/doi/10.1145/3510455.3512777

[40] A. Megargel, C. M. Poskitt, and V. Shankararaman, "Microservices orchestration vs. choreography: A decision framework," in *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, 10 2021, pp. 134–141. [Online]. Available: https://ieeexplore.ieee.org/document/9626189/

[41] N. Singhal, U. Sakthivel, and P. Raj, "Selection mechanism of micro-services orchestration vs. choreography," *International journal of Web Semantic Technology*, vol. 10, pp. 01–13, 1 2019. [Online]. Available: http://aircconline.com/ijwest/V10N1/10119ijwest01.pdf

[42] T. Heinrichs, “Orchestration vs choreography,” 2 2023. [Online]. Available: https://camunda.com/blog/2023/02/orchestration-vs-choreography/#what-is-choreography-arlk

[43] A. Jaffke, “The spaghetti problem,” 6 2018. [Online]. Available: https://www.linkedin.com/pulse/spaghetti-problem-andreas-andy-jaffke

[44] M. McGarr and D. Marsh, “Towards true continuous integration: distributed repositories and dependencies,” 4 2017. [Online]. Available: https://netflixtechblog.com/towards-true-continuous-integration-distributed-repositories-and-dependencies-2a2e3108c051

[45] A. Lima, “A histÓria da netflix e dos microsserviÇos.” [Online]. Available: https://acervolima.com/a-historia-da-netflix-e-dos-microsservicos/

[46] Y. Izrailevsky, S. Vlaovic, and R. Meshenberg, “Completing the netflix cloud migration,” 2 2016. [Online]. Available: https://about.netflix.com/en/news/completing-the-netflix-cloud-migration

[47] A. N. O. Production, “Hystrix: Latency and fault tolerance for distributed systems,” 11 2012. [Online]. Available: https://github.com/Netflix/Hystrix

[48] B. Christensen, “Introducing hystrix for resilience engineering,” 11 2012. [Online]. Available: https://netflixtechblog.com/introducing-hystrix-for-resilience-engineering-13531c1ab362

[49] R. Winkler, B. Storozhuk, M. Romeh, and D. Maas, “Fault tolerance library designed for functional programming,” 6 2015.

[50] ——, “Resilience4j documentation,” 6 2015. [Online]. Available: https://resilience4j.readme.io/docs/getting-started

[51] Netflix, “Conductor,” 12 2016. [Online]. Available: https://github.com/Netflix/conductor

[52] V. Baraiya and V. Singh, “Netflix conductor: A microservices orchestrator,” 12 2016. [Online]. Available: https://netflixtechblog.com/netflix-conductor-a-microservices-orchestrator-2e8d4771bf40

[53] S. M. Fulton, “What led amazon to its own microservices architecture,” 10 2015. [Online]. Available: https://thenewstack.io/led-amazon-microservices-architecture/

[54] S. Kumar and A. Puthiyavettle, “Architecting a highly available serverless, microservices-based ecommerce site,” 7 2021. [Online]. Available: https://aws.amazon.com/pt/blogs/architecture/architecting-a-highly-available-serverless-microservices-based-ecommerce-site/

[55] I. A. W. Services, “Workflow orchestration.” [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/best-practices-building-data-lake-for-games/workflow-orchestration.html

[56] ——, “Aws step functions.” [Online]. Available: https://aws.amazon.com/step-functions/

[57] ——, “Amazon eventbridge documentation.” [Online]. Available: https://docs.aws.amazon.com/eventbridge/?icmpid=docs_homepage_appintegration

[58] A. Gud, “Why we leverage multi-tenancy in uber’s microservice architecture,” 3 2020. [Online]. Available: https://www.uber.com/en-PT/blog/multitenancy-microservice-architecture/

[59] E. Haddad, “Service-oriented architecture: Scaling the uber engineering codebase as we grow,” 9 2015. [Online]. Available: https://www.uber.com/en-PT/blog/service-oriented-architecture/

[60] A. Gluck, “Introducing domain-oriented microservice architecture,” 7 2023. [Online]. Available: https://www.uber.com/en-PT/blog/microservice-architecture/

[61] L. Lozinski, “The uber engineering tech stack, part i: The foundation,” 7 2016. [Online]. Available: https://www.uber.com/en-PT/blog/tech-stack-part-one-foundation/

[62] D. Sinha, “Understanding uber’s service-oriented architecture design,” 1 2022. [Online]. Available: https://www.techaheadcorp.com/blog/uber-architecture-design/

[63] Lucy, “Conducting better business with uber’s open source orchestration tool, cadence,” 11 2019. [Online]. Available: https://www.uber.com/en-PT/blog/open-source-orchestration-tool-cadence-overview/?uclick_id=2f1e2a42-7c2e-43f9-9d9d-277421cd9d43

[64] K. Buckner, “Improving the user experience with uber’s customer obsession ticket routing workflow and orchestration engine,” 3 2019. [Online]. Available: https://www.uber.com/en-PT/blog/customer-obsession-ticket-routing-workflow-and-orchestration-engine/?uclick_id=2f1e2a42-7c2e-43f9-9d9d-277421cd9d43

[65] Quarkus, “Quarkus.” [Online]. Available: https://quarkus.io/

[66] O. Corporation, “The hotspot group.” [Online]. Available: https://openjdk.org/groups/hotspot/

[67] Oracle, “Graalvm.” [Online]. Available: https://www.graalvm.org/

[68] Broadcom, “Spring.” [Online]. Available: https://spring.io/

[69] ——, “Spring boot.” [Online]. Available: https://spring.io/projects/spring-boot

[70] D. Inc., “Docker.” [Online]. Available: https://www.docker.com/

[71] Broadcom, “Spring cloud.” [Online]. Available: https://spring.io/projects/spring-cloud

[72] Spring, “Spring cloud netflix.” [Online]. Available: https://cloud.spring.io/spring-cloud-netflix/reference/html/

[73] T. L. Foundation, “Openapi.” [Online]. Available: https://www.openapis.org/

[74] Camunda, “Camunda.” [Online]. Available: https://camunda.com/

[75] A. S. Foundation, “Apache kafka.” [Online]. Available: https://kafka.apache.org/

[76] MongoDB, “Mongodb.” [Online]. Available: https://www.mongodb.com/

[77] S. Software, “Swagger.” [Online]. Available: https://swagger.io/

[78] Quarkus, “Simplified mongodb with panache.” [Online]. Available: https://quarkus.io/guides/mongodb-panache

[79] J. by Red Hat, “Class reactivepanachemongoentity.” [Online]. Available: https://javadoc.io/static/io.quarkus/quarkus-mongodb-panache/1.11.2.Final/io/quarkus/mongodb/panache/reactive/ReactivePanacheMongoEntityBase.html

[80] T. P. Lombok, “Project lombok.” [Online]. Available: https://projectlombok.org/

[81] Oracle, “Javabeans specification.” [Online]. Available: https://www.oracle.com/java/technologies/javase/javabeans-spec.html

Appendix A

HTTP Codes and Handling Exceptions

This appendix covers the error handling and HTTP codes used in this project.

HTTP Code	ID	Description	Message
Internal Server Error	500	An unexpected error occurred on the server.	Sorry, there was a technical issue.
Bad Request	400	The request made by the entity is invalid.	Bad Request
Not Found	404	The requested entity could not be found.	Entity not found.
Forbidden	403	The entity is already registered in the system.	Entity already exists.
Forbidden	403	The submitted verification code is invalid.	Invalid verification code.

Table A.1: HTTP response codes and their corresponding error messages returned by services. Source: author.

The services guarantee the transmission of appropriate HTTP status codes and informative messages, resulting in a better user experience and simplifying issue identification. This is achieved through the use of the DataException class (Listing A.1) and the ErrorCode enumeration (Listing A.2).

```
1   public class DataException extends RuntimeException {
2       private final ErrorCode errorCode;
3       private final String errorMessage;
4       private final List<String> errors;
5
6
7       public DataException(ErrorCode errorCode) {
8           this(errorCode, errorCode.getErrorMessage());
9       }
10
11      public DataException(ErrorCode errorCode, String errorMessage) {
12          this(errorCode, errorMessage, null);
13      }
14
15      public DataException(ErrorCode errorCode, String errorMessage, List<String>
            errorMessages) {
16          super();
```

```
17         this.errorCode = errorCode;
18         this.errorMessage = errorMessage;
19         this.errors = errorMessages == null ? new ArrayList<>() : new ArrayList<>(
               errorMessages);
20     }
21 }
```

Listing A.1: Java implementation of the DataException class handling error codes and messages within the service. Source: author.

```
1  public enum ErrorCode {
2      UNEXPECTED_ERROR(Response.Status.INTERNAL_SERVER_ERROR.getStatusCode(),
           500, "Sorry, there was a technical issue."),
3      BAD_REQUEST(Response.Status.BAD_REQUEST.getStatusCode(), 400, "Bad Request"),
4      ENTITY_NOT_FOUND(Response.Status.NOT_FOUND.getStatusCode(), 404, "Entity not
           found."),
5      ENTITY_ALREADY_EXISTS(Response.Status.FORBIDDEN.getStatusCode(), 403, "Entity
            already exists."),
6      INVALID_VERIFICATION_CODE(Response.Status.FORBIDDEN.getStatusCode(), 403, "
           Invalid verification code, entity deleted");
7
8      private final int statusCode;
9      private final long codeId;
10     private final String errorMessage;
11
12     /**
13      * Constructs an ErrorCode with the specified parameters.
14      *
15      * @param statusCode the HTTP status code associated with the error
16      * @param codeId the identifier for the error code
17      * @param errorMessage the error message
18      */
19     ErrorCode(int statusCode, long codeId, String errorMessage) {
20         this.statusCode = statusCode;
21         this.codeId = codeId;
22         this.errorMessage = errorMessage;
23     }
24 }
```

Listing A.2: Java implementation of the ErrorCode enum mapping HTTP codes and messages to specific error conditions. Source: author.