INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Reactive Microservices - An Experiment

JOSÉ PEDRO RIBEIRO DA COSTA FERREIRA
Junho de 2022

POLITÉCNICO
DO PORTO

Reactive Microservices

An Experiment

José Pedro Ribeiro da Costa Ferreira

Dissertation for obtaining a Master's Degree in
Informatics Engineering, specialization area
Software Engineering

Advisor: Isabel Azevedo

Porto, June 2022

“Design and programming are human activities, forget that and all is lost”

Bjarne Stroustrup

Resumo

Os microserviços são geralmente adotados quando a escalabilidade e flexibilidade de uma aplicação são essenciais para o seu sucesso. Apesar disto, as dependências entre serviços transmitidos através de protocolos síncronos, resultam numa única falha que pode afetar múltiplos microserviços. A adoção da capacidade de resposta numa arquitetura baseada em microserviços, através da reatividade, pode facilitar e minimizar a proliferação de erros entre serviços e na comunicação entre eles, ao dar prioridade à capacidade de resposta e à resiliência de um serviço.

Esta dissertação fornece uma visão geral do estado da arte dos microserviços reativos, estruturada através de um processo de mapeamento sistemático, onde são analisados os seus atributos de qualidade mais importantes, os seus erros mais comuns, as métricas mais adequadas para a sua avaliação, e as *frameworks* mais relevantes.

Com a informação recolhida, é apresentado o valor deste trabalho, onde a decisão do projeto e a *framework* a utilizar são tomadas, através da técnica de preferência de ordem por semelhança com a solução ideal e o processo de hierarquia analítica, respetivamente. Em seguida, é realizada a análise e o desenho da solução, para o respetivo projeto, onde se destacam as alterações arquiteturais necessárias para o converter num projeto de microserviços reativo.

Em seguida, descreve-se a implementação da solução, começando pela configuração do projeto necessária para agilizar o processo de desenvolvimento, seguida dos principais detalhes de implementação utilizados para assegurar a reatividade e como a *framework* apoia e simplifica a sua implementação, finalizada pela configuração das ferramentas de métricas no projeto para apoiar os testes e a avaliação da solução.

Em seguida, a validação da solução é investigada e executada com base na abordagem *Goals, Questions, Metrics* (GQM), para estruturar a sua análise relativamente à manutenção, escalabilidade, desempenho, testabilidade, disponibilidade, monitorabilidade e segurança, finalizada pela conclusão do trabalho global realizado, onde são listadas as contribuições, ameaças à validade e possíveis trabalhos futuros.

Palavras-chave: Microserviços, Reatividade, Arquitetura de Software, Design Orientado ao Domínio, Framework Lagom

Abstract

Microservices are generally adopted when the scalability and flexibility of an application are essential to its success. Despite this, dependencies between services transmitted through synchronous protocols result in one failure, potentially affecting multiple microservices. The adoption of responsiveness in a microservices-based architecture, through reactivity, can facilitate and minimize the proliferation of errors between services and in the communication between them by prioritizing the responsiveness and resilience of a service.

This dissertation provides an overview of the reactive microservices state of the art, structured through a systematic mapping process, where its most important quality attributes, pitfalls, metrics, and most relevant frameworks are analysed.

With the gathered information, the value of this work is presented, where the project and framework decision are made through the technique of order preference by similarity to the ideal solution and the analytic hierarchy process, respectively. Then, the analysis and design of the solution are idealized for the respective project, where the necessary architectural changes are highlighted to convert it to a reactive microservices project.

Next, the solution implementation is described, starting with the necessary project setup to speed up the development process, followed by the key implementation details employed to ensure reactivity and how the framework streamlines its implementation, finalized by the metrics tools setup in the project to support the testing and evaluation of the solution.

Then, the solution validation is traced and executed based on the Goals, Questions, Metrics (GQM) approach to structure its analysis regarding maintainability, scalability, performance, testability, availability, monitorability, and security, finalized by the conclusion of the overall work done, where the contributions, threats to validity and possible future work are listed.

Keywords: Microservices, Reactive, Software Architecture, Domain Driven Design, Lagom Framework

Acknowledgements

I would like to express my gratitude to both my advisor Isabel Azevedo and professor Susana Nicola for all of their help up to this point, as well as their willingness to answer my questions whenever I needed them.

Thank you to professor Ciro Martins for his detailed revision and remarks in the intermediate evaluation of this dissertation, helping me correct and improve some crucial aspects of the paper.

I would also like to thank my family and friends for all the support during the development of my dissertation.

A special thank you to my colleagues André Madureira, Gustavo Ferreira, João Fonseca, and João Dias for all their aid and for sharing my hardships along this journey.

Table of Contents

Table of Figures

Table of Tables

Table of Code

Acronyms and Symbols

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
ACM	Association for Computing Machinery
AHP	Analytic Hierarchy Process
API	Application Programming Interface
ATD	Architectural Technical Debt
AWS	Amazon Web Services
BASE	Basically Available, Soft State, Eventually Consistent
C4	Context, Containers, Components, and Code
CAP	Consistency, Availability, Partitioning
CI	Consistency Index
CI/CD	Continuous Integration (CI) and Continuous Delivery (CD)
CPU	Central Processing Unit
CQRS	Command and Query Responsibility Segregation
CR	Consistency Ratio
DC/OS	Data Centre Operating System
DNS	Domain Name System
DTO	Data Transfer Object
FAST	Function Analysis System Technique
FCFS	First Come First Serve
FFE	Fuzzy Front End
FTGO	Food To Go
GQM	Goals, Questions, Metrics
HTML	HyperText Markup Language

HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
IoT	Internet of Things
IP	Internet Protocol
ISEP	Instituto Superior de Engenharia do Porto – translated to Oporto Higher Institute of Engineering
JVM	Java Virtual Machine
KPI	Key Performance Indicator
LOC	Lines of Code
NPD	New Product Development
OCI	Oracle Cloud Infrastructure
OHS	Open Host Service
OWASP	Open Web Application Security Project
PL	Published Language
REST	Representational State Transfer
RI	Random Consistency Index
RPC	Remote Procedure Call
RQ	Research Question
RSS	Resident Set Size
SDLC	Software Development Life Cycle
SLA	Service Level Agreement
SQL	Structured Query Language
SSL	Secure Sockets Layer
TD	Technical Debt
TOPSIS	Technique of Order Preference by Similarity to Ideal Solution
UML	Unified Modelling Language

Symbols

λ	Lambda
Δ	Delta

1 Introduction

This chapter provides the context for the work accomplished, followed by the problem to be solved, and the objectives to achieve, before concluding with the research methodology of the paper. Finally, a summary of the document's structure is provided.

1.1 Context

Each year, as the need for distributed and flexible software grows, more and more people are studying and adopting microservices, not only to improve their software, through high maintainability and testability, low coupling, and ease of deployment but also to optimize the resource allocation and flow of information, by organizing its teams around business capabilities and allowing for more diverse teams (Richardson, 2021). Microservices development, on the other hand, is not trivial. Many companies try to use this architecture in projects where it is completely inappropriate. That's why choosing the best architecture for a project and how to implement it is such a crucial stage in assuring its viability.

1.2 Problem Statement

Beyond the characteristics of reactive systems is their capability to remain responsive in the face of failure and under variable workloads (Bonér et al., 2014). Microservices are generally adopted when the scalability and flexibility of an application are essential to its success. Despite this, dependencies between services transmitted through synchronous protocols, result in one failure potentially affecting multiple microservices. The adoption of responsiveness in a microservices-based architecture can facilitate and minimize the proliferation of errors between services and in the communication between them, by prioritizing the responsiveness and resilience of a service. Some open-source frameworks can be used to obtain reactive microservices, such as Lagom, Spring Reactive, Micronaut, and Quarkus. However, it is not fully understood what the effects of responsiveness on some quality attributes are.

1.3 Objectives

In this dissertation, four main assignments were set to explore and evaluate reactive microservices:

1. Explore and document the most frequent and relevant concerns, pitfalls, metrics, and frameworks in reactive microservices;
2. Study and choose a microservices project, to be migrated to reactive microservices;
3. Explore and document the journey of its migration and lessons learned;
4. Evaluate the migrated project with the studied metrics.

1.4 Research Methodology

The study methodology utilized to address the above-mentioned question consisted of the following phases:

1. Bibliographic research on the current state of the art comprised by:
    i. The most relevant concerns for developers when using reactive microservices (see sections 2.4.1);
    ii. What to avoid in the implementation of reactive microservices (see sections 2.4.2);
    iii. Metrics to use on reactive microservices evaluation (see section 2.4.3);
    iv. The most optimal frameworks for developing reactive microservices (see section 2.4.4);
2. Conduct a value analysis of the proposed solution (see Annex A);
3. Identify and perform key decisions that must be made to create and implement the intended solution (see section 3);
4. Analyse the requirements and design the architecture of the proposed solution (see section 4);
5. Implement and document the process of migration of the chosen project (see section 5);
6. Formulate and execute the tests and evaluations to be had, based on the previous metrics gathered (see section 6).

1.5 Document Structure

This paper is divided into eight sections:

1. **Introduction** – This is the current chapter. Here its firstly given a context of the problem, followed by the problem itself, the objectives traced and the research methodology, finalized by the main outcomes and this document structure.
2. **State of the art** – Next is the state of the art, which contains a brief overview of what a traditional microservice and a reactive one, followed by a systematic literature review,

where 4 research questions are formulated and investigated, followed by a summary on what was discovered.

3. **Value analysis** – In this chapter, the value of the proposed solution is determined, as well as the choices for the project and framework to use and its reasoning.
4. **Analysis and Design** – In this chapter, it is documented the initial analysis and design of the proposed solution.
5. **Solution Implementation** – In this chapter, the implemented solution will be showcased, focusing on the necessary project setup for a smooth development environment, the implementation details relative to the traced requirements, the employed tests, the setup for the metrics to be used in tests and evaluation and finally the summary of this migration process and some lessons learned.
6. **Tests and Evaluation** – Next, the evaluation process and tests are formulated and executed, following the previously gathered metrics per quality attribute in the state of the art, finalized with the analysis of its results.
7. **Conclusion** – Finally a conclusion of the overall work is made, starting with the main contributions of this project, followed by the recorded threats to the validity of the project, the future work to be had to minimize some of these threats and further improve its value, concluded by some final remarks on the process to achieve this report as a whole.
8. **Annex –** This supplementary chapter contains part of the value analysis, which was developed for an intermediary delivery, some of the diagrams which size was too big to be easily read in the body of the document and some of the implementation code and diagrams aid with the understanding of the solution.

2 State of the Art

This chapter details the results of research into reactive microservices, their most common practices and guidelines. This includes a systematic literature review documenting four research questions, as a foundation for the work to be done, as well as pattern identification to aid the future development of the value analysis.

2.1 Traditional Microservices vs Reactive Microservices

Monolithic applications can be effective, but as more applications are moved to the cloud, consumers are becoming increasingly frustrated with them. Because change cycles are intertwined, a change to a small component of the program necessitates rebuilding and deploying the entire monolith. It's generally difficult to maintain a solid modular structure over time, making it more difficult to keep changes that should only affect one module within that module. Scaling an application demands scaling the complete application rather than just the bits that require more resources (Fowler & Lewis, 2014).

These frustrations prompted the creation of the microservice architectural style, which is a means of building a single application as a collection of small services, as shown in Figure 1, each of which operates in its process and interacts through lightweight mechanisms, most frequently a hypertext transfer protocol (HTTP) resource application programming interface (API). These services are based on business capabilities and can be deployed by completely automated software. These services may be written in separate programming languages, use different data storage methods, and can even be developed by different teams, having the bare minimum of centralized management (Fowler & Lewis, 2014).

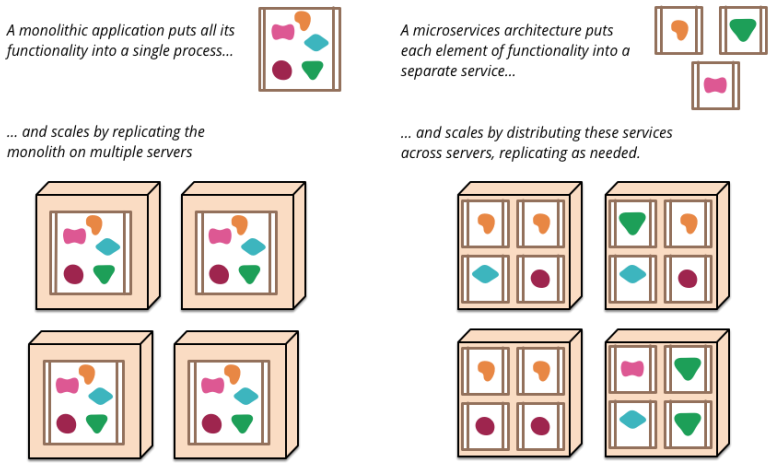

Figure 1. Comparison of a monolithic and microservice architecture (Fowler & Lewis, 2014)

Reactive microservices are a subset of these microservices that increase their flexibility, scalability, and decoupling. This makes them more adaptable to change and easier to build. They are far more tolerant of failure, and when it does happen, they handle it gracefully rather than with tragedy. Reactive Systems are quick to respond and provide users with useful interactive feedback. (Bonér et al., 2014)

Reactive systems are based on 4 principles:

- **Responsiveness:** Responsiveness is the foundation of usability and utility, but it also means that faults can be discovered promptly and successfully dealt with. Rapid and consistent reaction times are the emphasis of responsive systems, which set reliable upper bounds to provide a consistent level of service (Bonér et al., 2014);

- **Resilience:** In the event of a failure, the system remains responsive. Replication, containment, isolation, and delegation are used to achieve resilience. Failures are contained within each component, isolating them from one another and ensuring that sections of the system can fail and recover without affecting the entire system. Each component's recovery is assigned to another external component, and high availability is provided where necessary through replication. A component's client does not handle its failures (Bonér et al., 2014);

- **Elasticity:** The system stays responsive under varying workloads. Changes in the input rate might cause reactive systems to increase or decrease the resources allocated to service these inputs. This entails architectures with no contention points or central bottlenecks, allowing for the sharding or replication of components and the distribution of inputs among them. By providing appropriate live performance measures, reactive systems support both predictive and reactive scaling algorithms. They achieve elasticity on commodity hardware and software platforms at a minimal price (Bonér et al., 2014);

- **Message Driven:** Asynchronous message-passing is used in reactive systems to create a border between components that assures loose coupling, isolation, and location transparency. This barrier also allows failures to be delegated as messages. By structuring and monitoring the message queues in the system and providing back-pressure as appropriate, explicit message-passing offers load management, flexibility, and flow control. As a means of communication, location transparent messaging allows failure management to work with the same syntax and semantics throughout a cluster or within a single host. Non-blocking communication allows recipients to consume resources only when they are actively using them, resulting in lower system overhead (Bonér et al., 2014).

2.2 Building Reactive Microservices

A systematic mapping review was used to gain a better understanding of reactive microservices, their challenges, good and bad practices, and which tools are better suited to build and evaluate them.

As demonstrated in Figure 2, our systematic mapping study's process phases begin with the creation of research questions, followed by a search for relevant articles, screening of papers, keywording of abstracts, and ultimately data extraction and mapping. Each process step has a result, with the systematic map representing the finishing result of the process (Petersen et al., 2008).

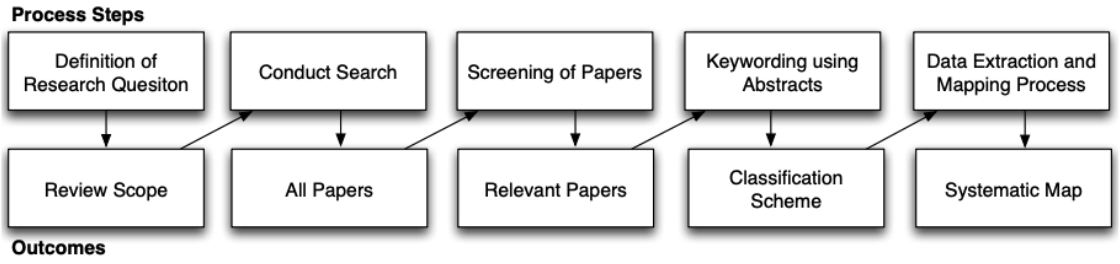

Figure 2. Steps to a systematic mapping review (Petersen et al., 2008)

2.3 Software Engineering Systematic Mapping

A systematic map is a strategy used in software engineering to provide an overview of a research topic and determine the number, type, and quantity of research and findings present within it, generally seeking patterns by mapping the frequency of publishing across time (Petersen et al., 2008).

2.3.1 Definition of research questions

To begin, research questions must be formulated to indicate what the researcher wishes to achieve with his investigation. Table 1 provides all the information about the research question identifier (RQ_x), for future use, as well as the question itself and its reasoning.

Table 1. Research questions

Id	Question	Reasoning
RQ_1	What are the most relevant concerns for developers when using reactive microservices?	To have a better grasp of how to create and maintain reactive microservices.
RQ_2	What should be avoided in the implementation of reactive microservices?	To learn from the errors and difficulties that others have encountered.
RQ_3	What metrics should reactive microservices be evaluated on?	To be able to properly assess and compare the state of reactive microservices.
RQ_4	What are the optimal frameworks for developing reactive microservices?	To increase the quality and performance of reactive microservices development and their overall condition.

2.3.2 Conducted search criteria

To obtain a broader set of study, all sources of information were digital libraries, which are the most utilized resource for software engineering-related issues. Institute of electrical and electronics engineers (IEEE) Explore, Google Scholar, and the association for computing machinery (ACM) Digital Library were the databases used for this search. The following queries were used:

- microservices AND reactive AND (challenges OR practices OR patterns);
- microservices AND reactive AND (pitfalls OR difficulties OR problems);
- microservices AND reactive AND (metrics OR evaluation OR quality);
- microservices AND reactive AND (framework OR tool).

2.3.3 Screening of papers for inclusion and exclusion

It is required to do a screening after completing the search and obtaining a list of findings. As shown in Figure 3, this implies first removing duplicate studies from the set and then using inclusion and exclusion criteria to narrow down the studies that are relevant (Petersen et al., 2008).

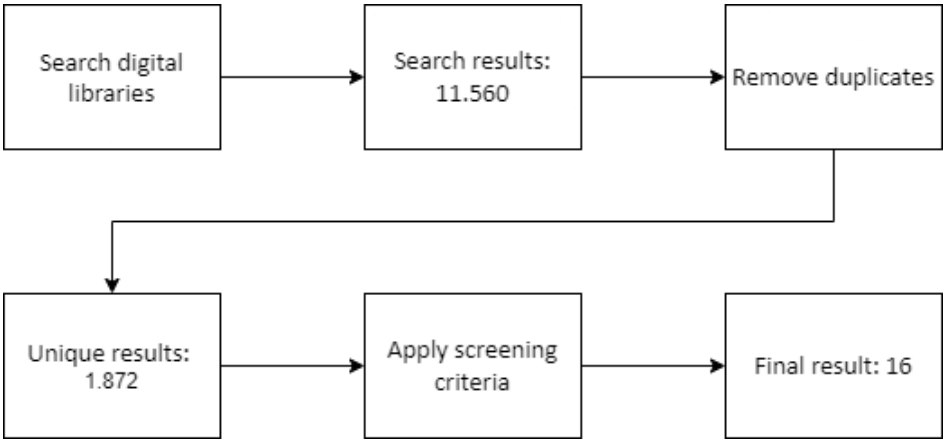

Figure 3. Screening of papers procedure

Table 2 shows the criteria that were considered to obtain the results:

Table 2. Employed screening criteria

Screening Criteria	
Inclusion	Reactive microservice patterns, practices and challenges reported by experienced practitioners
	Practical case studies where reactive microservice methodologies were applied
	The benefits and disadvantages of reactiveness when compared with traditional microservices
	Research on evaluating microservices, and reactiveness
	Investigation of frameworks or tools to develop reactive microservices
Exclusion	Commercial publications
	Papers that are not written in either English or Portuguese
	Papers that are not in the software engineering scope and neither explicitly refer to microservices or reactiveness
	The paper must be accessible to ISEP academic department
	Papers that do not offer evidence for the perspective of the author

After deleting duplicates and applying the criteria to the findings, the studies in Table 3 were obtained:

Table 3. List of papers obtained following the screening procedure

ID	**TITLE**	**REFERENCE**
1	Understanding and addressing quality attributes of microservices architecture: A Systematic literature review	(Li et al., 2021)
2	An extensible data-driven approach for evaluating the quality of microservice architectures	(Cardarelli et al., 2019)
3	The applicability of palladio for assessing the quality of cloud-based microservice architectures	(Klinaku et al., 2019)
4	From a Monolithic Big Data System to a Microservices Event-Driven Architecture	(Laigner et al., 2020)

ID	TITLE	REFERENCE
5	The Saga Pattern in a Reactive Microservices Environment	(Stefanko et al., 2019)
6	A Framework for Evaluating Continuous Microservice Delivery Strategies	(Lehmann & Sandnes, 2017)
7	Architectural technical debt in microservices: a case study in a large company	(de Toledo et al., 2019)
8	Development Frameworks for Microservice-based Applications: Evaluation and Comparison	(DInh-Tuan et al., 2020)
9	Designing Distributed, Scalable and Extensible System Using Reactive Architectures	(Tovarnitchi, 2019)
10	Reactive Microservices Architecture Using a Framework of Fault Tolerance Mechanisms	(Rasheedh & Saradha, 2021)
11	Microservices: A Mapping Study for Internet of Things Solutions	(C. Santana et al., 2018)
12	Reactive microservices for the internet of things: a case study in fog computing	(C. J. L. de Santana et al., 2019)
13	Asynchronous Queue Based Approach for Building Reactive Microservices	(Brilhante et al., 2017)
14	Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments	(Gotin et al., 2018)
15	A Black-box Monitoring Approach to Measure Microservices Runtime Performance	(Brondolin & Santambrogio, 2020)
16	A Comparative Study of Microservices Frameworks in IoT Deployments	(du Plessis et al., 2021)
17	Attributes Assessing the Quality of Microservices Automatically Decomposed from Monolithic Applications	(Cojocaru et al., 2019)

2.3.4 Keywording using abstracts

Keywording is a technique for shortening the time it takes to construct a categorization scheme while also guaranteeing that the system considers previous research. There are two phases to keywording. The reviewers begin by reading the abstracts and looking for keywords and themes that indicate the paper's contribution. While doing so, the reviewer also determines the research's context. When this is completed, the set of keywords from several articles is integrated to produce a high-level understanding of the research's nature and significance. This aids reviewers in defining a collection of categories that is reflective of the whole population. When abstracts are too inadequate to allow for significant keyword selection, reviewers might opt to look at the paper's introduction or conclusion instead. Once a final set of keywords has been selected, they may be grouped and utilized to create the map's categories (Petersen et al., 2008).

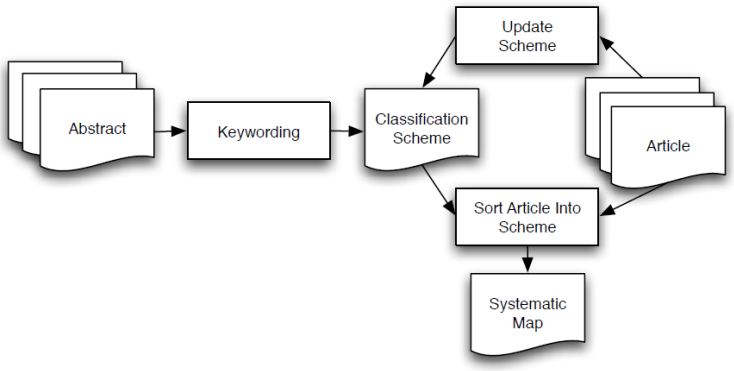

Figure 4. Building the Classification Scheme (Petersen et al., 2008)

Four research topics are defined in the study, as shown in Table 4:

Table 4. Outlined research topics

Research Topics	*Description*
Development of reactive microservices	Explore techniques, guidelines and principles utilized in the development of reactive microservices
Pitfalls developing reactive microservices	Examine problems and anti-patterns to avoid during the implementation of reactive microservices
Metrics to evaluate reactive microservices	Study what metrics, tools or attributes to focus on while developing and maintaining reactive microservices.
Tools to implement reactive microservices	Investigate development frameworks or tools to implement reactive microservices.

2.3.5 Data extraction and mapping of studies

The data from the articles were collected and represented in a collection of concerns for each of the issues listed in Table 4 previously mentioned. The first column of each table represents these issues, with the second column containing the studies that back them up.

Table 5. Development of reactive microservices

Description	*Reference paper ID*
Quality attributes	1, 2, 3, 4, 9, 10, 11, 17
Challenges	2, 4, 9, 10, 11, 12, 14
Service communication	4, 9, 10
Transaction processing	5, 9

Table 6. Pitfalls developing reactive microservices

Description	*Reference paper ID*
Drawbacks	4, 6
Risks and issues	7, 10, 12, 13
Architectural technical debt	7
Cascading failure between microservices	10

Table 7. Metrics to evaluate reactive microservices

Description	*Reference paper ID*
Evaluation of software quality attributes	1, 2, 6, 8, 10, 12, 13, 17
Maintenance tools	2, 3, 6, 8, 10, 13
Evaluation of quantifiable metrics	1, 6, 8, 10, 13, 14, 17
Application assessment with continuous delivery	6, 7
Application Monitoring	14

Table 8. Tools to implement reactive microservices

Description	*Reference paper ID*
Development framework or tool comparison	8, 16
Development framework or tool analysis	8, 10, 16

After meticulously documenting and segmenting each study, the patterns and common themes and subjects between the papers can now be uncovered. As seen in Figure 5, although microservices and reactivity are both well-known topics, the integration of reactive principles in microservices is relatively new, 75% of the findings were produced in the last three years, and all publications were issued in the last five years.

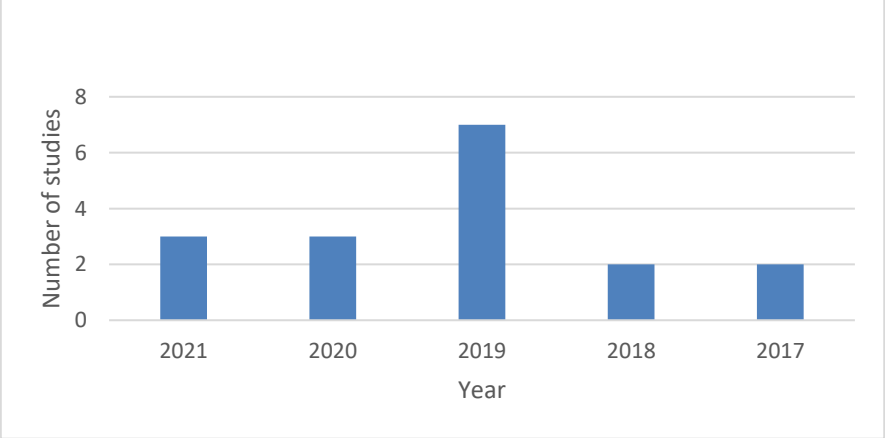

Figure 5. Number of selected studies per year of creation

Another interesting pattern identified is the most popular sectors where reactive microservices are being employed, with the Internet of Things (IoT) having a 50% occurrence rate in the research, followed by cloud applications having a 40% occurrence rate and finally big data having a 10% occurrence rate, as pictured in Figure 6.

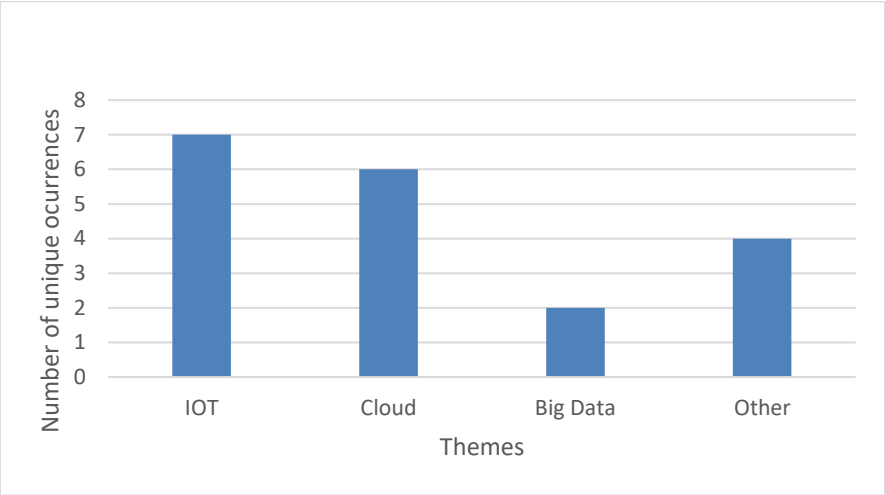

Figure 6. Number of occurrences in selected studies per theme

These correlations are supported by the high synergy between microservices and cloud computing, which can amplify some of the benefits of microservices through tools such as autoscaling and high availability. Regarding IoT, its correlation to reactive microservices can be explained by the similarity of their core principles, which both seek to decentralize the application to provide higher flexibility and scalability.

2.4 Data Analysis

Once the data has been retrieved and mapped into the appropriate categories within each categorization system, it is feasible to assess how the obtained findings answer the study questions stated in section 2.3.1.

2.4.1 RQ_1 - What are the most important concerns that developers should pay special attention to when implementing reactive microservices?

2.4.1.1 Quality Attributes

Starting with the quality attributes, according to the research carried out by Li et al., despite the rapid expansion and adoption of microservices, the influence on quality attributes, particularly which quality criteria are more difficult to implement, remains unexplored and unclear. A thorough literature analysis was conducted, with 72 articles reviewed, to clarify and expand the

knowledge on quality attributes and the techniques that should be used to improve them in a project. (Li et al., 2021).

Scalability and performance were the two most concerning quality aspects in the analysed research, followed by availability, monitorability, and security, with testability being the least significant. Quality attributes such as the trade-off between performance and scalability, or the reliance between monitorability and scalability, were also discovered to be related. Figure 7 illustrates which techniques should be used to refine these quality attributes (Li et al., 2021):

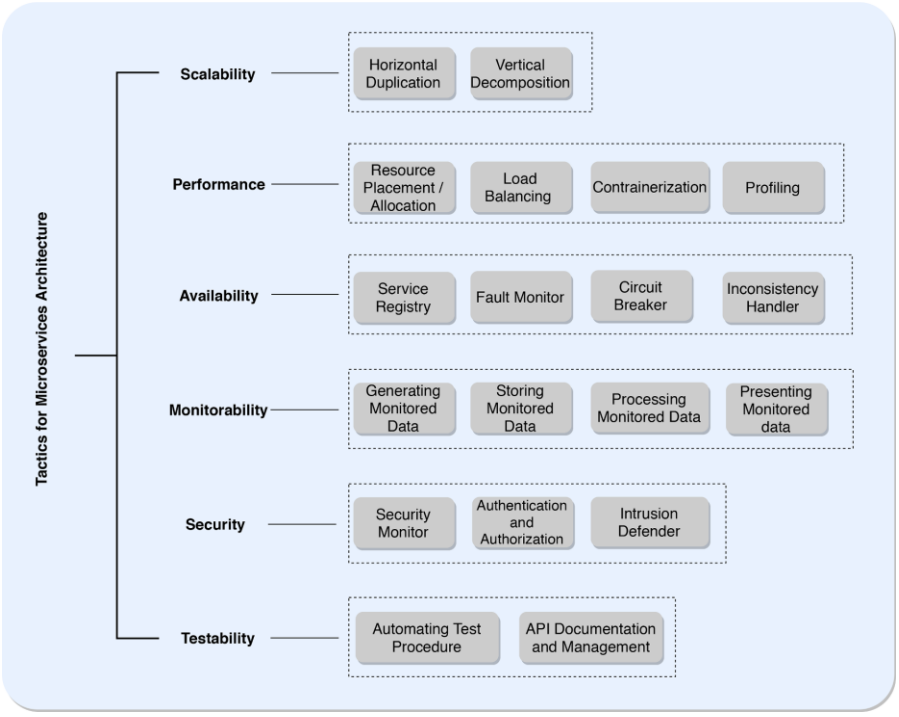

Figure 7. Tactics to use in each quality attribute (Li et al., 2021)

2.4.1.2 Challenges

Regarding the challenges found, Laigner et al. and Gotin et al. dive deeply into a set of challenges found throughout the development life cycle of reactive microservices:

Defining microservices

According to Laigner et al., two seemingly domain concepts that generate distinct domain events lead to duplicate concepts, which resulted in redundant efforts on each requirement modification when new information is obtained during the development life cycle. As a lesson learned, it was recommended to follow Fowler's suggestion, which advocates for the Monolithic-First method, according to which a project “shouldn’t start with microservices, even if you’re sure your application will be big enough to make it worthwhile.” (Fowler, 2015). It's critical to wait for requirements to develop before adequately defining microservices. Emerging research, on the other hand, examines model-driven development of microservice-based systems, which might help ease some of the impediments to microservices development (Laigner et al., 2020).

Data modelling

Another issue faced by Laigner et al. was the fast-paced development process, which made accurate data modelling difficult due to the adoption of new technologies. They had to use APIs to encapsulate schema-less and denormalized data models rather than the normalized data models and data consistency assurances found in monolithic systems because of the distributed architecture. Furthermore, while using domain events to communicate with services increases the domain's expressiveness, the variety of services and technologies made debugging problems difficult for developers. The application's convoluted data flow frequently led to misunderstandings and slowed the process of identifying the source of problems (Laigner et al., 2020).

Embracing failure

Some microservices-oriented frameworks, such as Spring, lack comprehensive support for failure management in processes that span numerous microservices. In the absence of distributed transactions, for example, the developer should hard-code recovery logic for such workflows. Additionally, they argue for a programming paradigm that defines fault-tolerance attributes that can be reasoned about on requests spanning many microservices, given the low granularity nature of microservice instances and the difficulties of thinking about each microservice local state globally (Laigner et al., 2020).

Service Communication

To fully understand the service communication issues encountered by Gotin et al., first, there must be introduced a set of concepts related to queue growth and state (Gotin et al., 2018).

There are three basic states for a queue regarding the difference between the number of messages received and processed, indicated by Δm:

- **Steady**, when the number of messages received is the same as the messages processed ($\Delta m = 0$);
- **Filling**, when the number of messages received is higher than the messages processed ($\Delta m > 0$);
- **Draining**, when the number of messages received is lower than the messages processed ($\Delta m < 0$);

Because its policy is first-come-first-served (FCFS), a filling state causes a delay on the application layer due to a wait time for each message in the queue. The queue is said to be **congested** when the time it takes for a message to transit through it approaches the maximum desired time. When the time it takes for a message to flow through a queue exceeds the maximum length that the message broker system can handle, the queue is said to be **flooded**. (Gotin et al., 2018).

The underlying issue of a congested or flooded queue is based on the provisioning of consuming microservices. Under provisioned microservices lead to a filling queue state since the consumption rate is lower than the production rate. For this reason, it eventually transits to a congested or flooded state inducing a performance degradation on the application layer and may result in reliability issues such as a rejection of messages. Overprovisioned microservices have a low utilization but do not degrade the message queue state since the consumption rate exceeds the production rate, but its costs will increase unnecessarily, being crucial to balance it out. However, due to the typical pay-as-you-go cost model of cloud infrastructure, each provisioned resource increases the operating costs, being necessary to gather information relative to the queue health and to balance its power accordingly (Gotin et al., 2018).

Transaction Processing - The Saga Pattern

A **saga** is a collection of operations that can be linked with one another. Each **operation** in the saga symbolizes a unit of work that the compensatory action has the potential to undo. The saga ensures that either all operations succeed or the corresponding **compensating actions** for all performed operations are conducted to cancel the incomplete processing (Stefanko et al., 2019).

An operation is a section of the saga that reflects a specific work phase. Each saga can be broken down into a series of operations, each of which can be executed as a transaction with complete atomicity, consistency, isolation, and durability (ACID) guarantees. Each operation must have its compensation action. The compensation action's goal is to undo the previous operation's work semantically. It is important to highlight that this may not be the opposite action that restores the system to its previous condition before the operation or the saga began (Stefanko et al., 2019).

The saga pattern, in contrast to the conventional transaction paradigm, softens the ACID constraints to provide availability and scalability while also including built-in failure management. Because the saga commits each action separately, updates to the saga that aren’t yet fully committed are immediately visible to other parallel activities, breaking the isolation property. The consistency, availability, and partitioning (CAP) theorem is an alternative BASE model that prioritizes availability over the consistency granted by ACID (Stefanko et al., 2019). The specified system properties are:

- Basically Available – The system guarantees availability with regards to the CAP theorem;
- Soft State – Due to eventual consistency, the state may change over time even if no immediate modification request is made;
- Eventual Consistency – The system's state is allowed to be inconsistent, but if no additional update requests are received, the state will gradually return to the consistent state (Stefanko et al., 2019).

The concept of sagas can be naturally extended to distributed systems. As an architectural pattern, the saga pattern emphasizes on integrity, reliability, and quality, as well as communication patterns between services. This enables the saga definition in distributed systems to be recast as a series of requests sent to specified participants invocations. These demands may give ACID assurances, but they are not limited and must be ensured by each participant. Similarly, each participant must expose the idempotent compensating action request handler. Distributed saga management, like centralized saga management, necessitates a transaction log and a saga execution component, both of which must be distributed and durable in an optimal environment. Because all components are now distributed, the saga management system must deal with new issues that did not exist in the localized context, the most significant of which being network and participant failures that can occur between remote invocations. However, typical principles from a non-distributed setting continue to be applicable (Stefanko et al., 2019).

2.4.2 RQ_2 - What should be avoided in the implementation of reactive microservices?

To find out what to avoid in the implementation of reactive microservices, several research and case studies were analysed. Two of the most interesting cases were from de Toledo et al. who evaluated the impact of architectural technical debt in reactive microservices and Lehmann & Sandnes, which investigated and compared different microservice delivery strategies.

2.4.2.1 Architectural Technical Debt

Starting with the de Toledo et al. study, and introducing the concept of technical debt (TD):

The term technical debt was initially used to inform non-developers about the threats of delivering "not quite right code". TD is a sub-optimal design or implementation that provides short-term gains but raises the system's long-term costs, compromising its evolvability and maintainability (de Toledo et al., 2019).

The definition of TD includes three key concepts (de Toledo et al., 2019):

- **Debt** – the presence of sub-optimal solutions. A system's debt can be measured as the number of sub-optimal implementations compared to the ideal option;
- **Interest** – due to the existence of a debt, an additional payment must be paid. It can also be seen as the amount that would be saved if the debt didn't exist;
- **Principal** – the cost of developing the system while staying out of debt, or the cost of refactoring it. Accumulating debt may be advantageous if the interest rate is low.

As a result, when to accrue or pay off the debt is a key question in the study. Two further ideas were used in this case study to reason about what affected the choice to correct TD and how to achieve so:

- **Risks** – they have the potential to influence today's decision-making or to be a cause of concern in the future. Fear of things going wrong might influence the likelihood of making that option;
- **Solution** – A strategy for resolving TD or lowering the amount of interest paid(de Toledo et al., 2019).

Architectural Technical Debt (ATD) is a subset of TD that is concerned with the architecture of a system. Some of the issues that ATD created in the case study are:

- Coupling between services;
- In the communication layer, it is necessary to deal with a lot of information regarding services;
- Unnecessary dependencies between development teams;
- Unnecessary implementation of transformations and filters;
- Too many different data types to manage (de Toledo et al., 2019).

This led to the conclusion that, while some TD may not be harmful to the development of reactive microservices, it must be actively measured and analysed to prevent losing some of the key benefits of this architectural approach, such as strong decoupling of services and even software teams.

2.4.2.2 CI/CD

Continuous delivery is a term that combines two concepts: continuous integration and continuous deployment, CI/CD. Continuous integration is the practice of integrating changes into the mainline early, such as the master branch, if the team uses Git versioning tools. The process of releasing changes to end-users as soon as they reach the mainline is known as a continuous deployment (Lehmann & Sandnes, 2017).

When these components are linked, a development workflow emerges in which developers merge their modifications into the production-ready version of the code base regularly, and those changes are promptly delivered to end-users. (Lehmann & Sandnes, 2017)

In the framework developed by Lehmann & Sandnes, various microservices CI/CD approaches were assessed using seven criteria (Lehmann & Sandnes, 2017):

- Testability ease – qualitative variable defined by 3 levels:
    - Trivial, if it requires a single setup per project and properly mirrors the production environment;
    - Time-consuming, if it needs setup per machine but still properly mirrors the production environment;
    - Uncertain, if it involves setup per machine and does not properly mirror the production environment.
- Abstraction expressiveness – qualitative variable defined by 3 levels:

    - Highly expressive, if there are no error-prone manual stages, and just a small amount of learning is necessary;
    - Somewhat expressive, if there are some manual stages;
    - Manual, if there are many error-prone manual stages.
- Environment parity – qualitative variable defined by 3 levels:
    - Equal, if any software bug can be found in any environment;
    - Distinguishable, if there are a few minor differences that can be quickly addressed;
    - Disparate, if all development, testing, and production machines must be manually checked for parity.
- Number of manual steps – integer value of the number of manual steps;
- Minutes to build, verify, and deploy – average overall time to deploy a service, in minutes;
- Availability adequacy – qualitative variable defined by 3 levels:
    - Adequate, if there is automatic scaling of computing nodes in response to increased and decreased server load with zero downtime installations;
    - Excessive, if there are deployments with no downtime and simple manual scaling of computing nodes;
    - Error-prone, if the deployment or resource scaling involves a lot of error-prone manual steps.

The results achieved from the comparison of a manual deployment strategy and a Kubernetes-based deployment, made by Lehmann & Sandnes, are shown in Table 9:

Table 9. Evaluation of a manual vs a container-based deployment (Lehmann & Sandnes, 2017)

	Manual deployment strategy	*Container-based deployment*
Testability ease	Uncertain	Trivial
Abstraction expressiveness	Manual	High
Environment parity	Disparate	Distinguishable
Number of manual steps	80 per day	0
Minutes to build, verify, and deploy	Unknown	Unknown
Availability adequacy	Error-prone	Adequate

Concluding, while creating reactive microservices, the significance of a well-structured and well-thought deployment plan is critical to avoiding long-term complications. Although it can slow down initial development by requiring more technologies and processes to set up and investigate, which aren't the primary goals, it can completely obstruct the ability to deploy and scale an application in the long run. As more microservices are introduced to the deployment,

the knowledge base and the number of mistakes grow exponentially, making this process extremely hard to share with new team members and coordinate between teams.

2.4.3 RQ_3 - What metrics should be used to evaluate reactive microservices?

Following the challenges presented by Gotin et al., this research also gives a range of assessment methods aimed at optimizing the service communication layer and resolving some of the issues reported. It's worth noting that this project was carried out in a cloud environment, which provides features like threshold-based rules auto-scaling, which make the process of scaling up or down a service much simpler.

Because the internal status of the message broker system provided a barrier in this research, they wish to examine the usefulness of depending directly on message queue metrics for scaling choices instead of the standard central processing unit (CPU) measure. Queue-specific metrics like arrival rate (ingress), departure rate (egress), and queue length may be monitored by several message brokers. Because the arrival rate and departure rate are not feasible for scaling options due to the threshold-based rules auto-scaling configurations, they analyse the end-to-end latency between message transmission and receipt in a consuming microservice to estimate the queueing delay. The arrival rate, for example, is unaffected by scaling decisions and so provides no feedback. As a result, they look at the growth of the queue, which contains both measures (Gotin et al., 2018).

The metrics that were evaluated were the **average CPU utilization**, the number of enqueued messages – **length of the message queue** – the difference between arrival and departure rate– **message queue growth** – and the wait time for a message in the queue before it was processed – **message queue delay** (Gotin et al., 2018).

Furthermore, the research made by Cojocaru et al. aids in connecting a quality attribute with measurable metrics. Two categories of analysis techniques were established to evaluate the metrics:

- **Static analysis** is an approach that does not require the examined application to be run. Without incurring the complexity of executing the program, scanning a microservice's code before exposing it to other microservices can help find and mitigate issues and vulnerabilities.
- **Dynamic analysis** is a technique that necessitates the execution of the software to be assessed and can identify flaws in the application’s behaviour and code logic.

Table 10 summarizes the findings by relating the quality attributes previously exposed in 2.4.1 with their respective metrics, which can either be a quantifiable statistic or simply the existence of a given tool or attribute, and its type of analysis (Cojocaru et al., 2019; Gotin et al., 2018; Li et al., 2021):

Table 10. Metrics by quality attributes

Quality Attribute	Metric / Attribute	Description	Analysis Technique
Maintainability	Granularity	Also referred to as the size of each microservice. Although it can be measured through the number of lines of code (LOC), a more useful application of this metric is by determining the relative size between microservices.	Static
	Cohesion	Reflects the extent to which a microservice's operations focus on a single functionality. Measuring cohesion systematically can be difficult due to its semantic essence but a comparison between microservices can be done to determine the components with low cohesion.	Static
	Coupling	The degree of coupling is measured by dividing the number of calls to a microservice by the number of invocations the microservice makes to other microservices.	Static
	Open interfaces	Can also describe the granularity of the service through the number of tasks and their parameters open through exposed interfaces.	Static
	Ease of deployment	A common scenario entails the deployment of completely automated containers, each of which runs a microservice. Execution timelines and instance graphs, as well as use-case and sequence unified modelling language (UML) diagrams, are commonly used to test it.	Dynamic
Scalability	Usage frequency	The percentage of requests made to the evaluated microservice compared to all requests made throughout the whole system.	Dynamic
	Number of synchronous requests	The number of synchronous requests is supported by the exposed interfaces. A significant diversity of requests suggests a lack of scalability.	Dynamic
	Horizontal/ vertical scalability	The ability of a microservice to continue to function appropriately when its size changes, either horizontally or vertically, without incurring performance penalties.	Dynamic
	Isolation	The isolation of the microservice from others, with whom it should only communicate through the disclosed interfaces. Similar to low coupling.	Dynamic
Performance	Response time	The expected time between when a request to a microservice is submitted and when the result is provided. It only considers the execution time, not the network delay time. When monitoring synchronous calls, the longest response time is used, but for asynchronous calls, the average time spent is used.	Dynamic

Quality Attribute	Metric / Attribute	Description	Analysis Technique
	The average size of messages	The average message size in each queue. Can be used comparatively between queues to pinpoint the less performant ones.	Dynamic
	Queue growth	The difference between message arrival and departure rate.	Dynamic
	Average CPU utilization	Average CPU utilization, to be compared between each service.	Dynamic
Testability	API documenta-tion and management	The existence of a streamlined API documentation and management page allows the automation of test procedures.	Dynamic
	Test automation	Automatic microservice testing process to support the continuous integration of microservices.	Dynamic
Availability	Uptime percentage	The percentage of time a microservice is available within a certain time frame. Most modern cloud platform Service Level Agreements (SLA) offer 99.9999% or higher availability, which corresponds to 31.56 seconds of unscheduled downtime per year.	Dynamic
	Successful execution rate	The capacity of a service provider to successfully fulfil requests within a particular time frame is measured by the ratio of successful requests to the total number of requests.	Dynamic
	Fault detection	To identify or predict the occurrence of a defect before the system may take action to recover from faults, applications require continuous monitoring so that their health can be studied to react to failures automatically and responsively with little human intervention, by implementing tools such as fault monitors.	Dynamic
	Health management	A quality trait characterizing a microservice's capacity to cope with failures is also known as resilience to failure. A microservice conforms to this criterion by preserving the internal state and automatically resuming while loading the most recent state before the failure.	Dynamic
Monitorability	Data generation and storage	Each service must be capable of generating universal logs, distributed tracing and applicational metrics, and storing them in either a centralized or decentralized storage system.	Dynamic
	Data presentation	The saved data from the application must be presentable in its singular or aggregable state, and viewable through open-source analytics and monitoring solutions.	Dynamic

Quality Attribute	Metric / Attribute	Description	Analysis Technique
Security	Third-party weaknesses	Regards the security of each dependency, which can directly impact the security of the application.	Dynamic
	Security monitor	Security monitor is a strategy that uses monitors at various levels to observe anomalous behaviour or assaults on microservices.	Dynamic
	Authentication and authorization	Authentication is the process of confirming a user's or a party's identity, and authorization is the means through which a principal is mapped to the activity that an identity is allowed to do. The microservices resilience to attacks is strengthened by applying these policies in the application.	Dynamic

2.4.4 RQ_4 - What are the most relevant frameworks to build reactive microservices?

2.4.4.1 Frameworks

A few exclusion criteria were established before discovering and analysing the most applicable frameworks for building reactive microservices. To be relevant to the study, a framework must be open source, offer explicit support for reactivity, and support a programming language that the researcher has previously studied.

The list, the criteria and comparison of frameworks made in Table 11 are based on studies conducted by du Plessis et al., DInh-Tuan et al. and Rasheedh & Saradha, and updated with the most recent information from each framework's GitHub and documentation pages. The following frameworks were evaluated:

Lagom

Lagom is a Java and Scala framework that uses Akka and Play in its underlying system. For communicating between decoupled microservices, Lagom preferentially utilizes Kafka, while data persistence is handled using Event Sourcing and Command Query Responsibility Segregation (CQRS). Lagom may be run on Kubernetes, data centre/operating system (DC/OS), or any other cloud/on-premises deployment platform that allows for private networking between servers. By default, Lagom is asynchronous, and it includes a service registry and discovery implementation(Lagom, 2016b) (Lagom, 2016a).

Spring Boot

Spring Boot eliminates many of the configuration pains that come with utilizing Spring, allowing for swifter application development. Although Spring Boot was not created specifically for reactive microservices, it enables the rapid and efficient creation of several microservices and

supports reactivity, making it one of the most popular Java frameworks for reactive microservices (Spring, 2014a, 2014b).

Quarkus

Quarkus is a full-stack, Kubernetes-native Java framework that runs on the Java virtual machine (JVM). Quarkus promises to offer a quick start-up time and low resident set size (RSS) memory consumption, which enhances scalability and memory utilization. It is suitable for the creation of a variety of Java applications, including reactive, serverless, microservices, and containers. It is written in Java and was created by Red Hat. Support for Web/representational state transfer (REST) services, databases, communications, and security are just a few of the key features (Quarkus, 2019a, 2019b).

Vert.x

Vert.x is a reactive, event-driven, polyglot software development toolkit that runs on the JVM and was created by Eclipse developers. Support for concurrent and asynchronous communication, database support, event streams, and registries are among its key features. Although Vert.x is developed in Java, it also supports the building of applications in Groovy, JavaScript, Ceylon, and Ruby (Vert.x, 2012b, 2012a).

Micronaut

Micronaut is a full-stack framework for constructing microservices and serverless applications that are built on the JVM. It was created by OCI (Oracle cloud infrastructure) and has faster start-up times and lower memory consumption than other JVM-based microservices frameworks. This is accomplished by pre-compiling the framework, which reduces the amount of computation necessary during runtime. Built-in cloud support, unit testing, service discovery, and reactive programming support, both client and server-side, are some of the characteristics of the Micronaut framework. Micronaut, like Vert.x, is written in Java, but it also supports development in Groovy and Kotlin (Micronaut, 2018a, 2018b).

Moleculer

Moleculer is advertised as a Node.js microservices framework that facilitates the creation of efficient, dependable, and scalable services. Moleculer has several features, including a built-in service registry and dynamic service discovery, as well as modular transporters and serializers. When compared to other Node.js microservices frameworks, Moleculer is written in Javascript and has a very high request time performance (Moleculer, 2017b, 2017a).

2.4.4.2 Comparison Criteria

In terms of comparison criteria, three major groups were identified: maturity, which is used to assess how refined a framework is, ease of implementation, which is used to assess the depth

of documentation, tools, guides, and the overall community, and features, which is the most technical category.

Maturity

To evaluate the maturity of an open-source framework, metrics such as the release cycle and analytics from open-source code hosting services like GitHub were chosen. Several criteria can be used to determine the maturity of software, four of these aspects have been chosen: release date, number of Github commits, the number of Github releases and number of Github contributors (du Plessis et al., 2021) (DInh-Tuan et al., 2020).

Ease of Implementation

Most developers emphasize ease of implementation when choosing a framework because it has a direct impact on productivity. To determine the ease of implementation, five criteria were chosen: The first is the breadth and depth of the documentation offered. The second is the level of information and scope of tutorials or usage guidelines supplied for new framework developers. The third parameter is the base/core programming language, and the fourth parameter is the amount of Stackoverflow tags, which indicates the extent of community support for a certain framework. The existing development environment/build tools are the final consideration (du Plessis et al., 2021) (DInh-Tuan et al., 2020).

Features

The features that are provided by a framework are another important consideration when selecting a framework. The features selected as criteria in this paper have been chosen due to their relevance to reactive microservices (du Plessis et al., 2021) (DInh-Tuan et al., 2020):

- **Essential services** – APIGateway, service discovery and registry, load balancing, and serialization are examples of essential services; some of these will just be identified as supported in this category and will be fully detailed in another.
- **Databases** – include support for multiple databases, either shared or per-service, as well as CQRS and event sourcing.
- **Observability** – Support for log aggregation, performance metrics, distributed tracing, and health checks is included.
- **Cross-cutting concerns** – Service registry, client and server-side discovery, external configuration, and deployment platforms are among the utility services that connect and manage the various microservices, included in this category.
- **Communication** – supported communication protocols are either synchronous, asynchronous, or remote procedure invocations.
- **Fault tolerance** – contains support for circuit breaker, timeout, and retry concepts.
- **Other features** – Other aspects that don't fit into any of the other categories but are still important to emphasise, such as utility tools (Interactive CLI and web dashboard), caching, security and required software, are included.

Table 11. Framework comparison

Criteria \ Framework			Lagom	Spring Boot	Quarkus	Vert.x	Micronaut	Moleculer
Maturity	Release date		March 3rd, 2016	July 10th, 2014	March 20th, 2019	January 29th, 2012	October 23rd, 2018	February 16th, 2017
	Github commits		2955	35916	28129	5250	11276	3909
	Github releases		48	122	159	140	114	68
	Github contributors		142	875	586	218	333	99
Ease of Implementation	Documentation		Moderate	Extensive	Moderate	Extensive	Moderate	Limited
	Tutorials and usage guidelines		Extensive	Extensive	Moderate	Moderate	Limited	Limited
	Base programming language		Java/Scala	Java	Java	Java	Java	JavaScript
	Stackoverflow tags		336	193535	2298	2312	1273	70
	Development environment and build tools	Build tool and dependency management	sbt, Maven	Maven, Gradle, Ant	Maven, Gradle	Maven, Gradle	Maven, Gradle, SDKMan	npm
		Hot reload	Yes	Yes, not default	Yes	Yes	Yes, not default	Yes
		Project template generator	Lagom Tech Hub Project Starter	Spring Initialzr	code.quarkus.io	start.vertx.io	micronaut.io/launch	moleculer-cli
		Interactive CLI	sbt dev console	N/A	N/A	N/A	N/A	REPL console
Features	Essential Services	APIGateway	Supported	Supported	Supported	Supported	Supported	Supported
		Service discovery	Supported	Supported	Supported	Supported	Supported	Supported
		Service Registry	Supported	Supported	Supported	Supported	Supported	Supported

Criteria / Framework			Lagom	Spring Boot	Quarkus	Vert.x	Micronaut	Moleculer
		Load balancing	Akka	Spring Cloud Load Balancer	Stork	Supported	Client-side load balancing	Server-side load balancing
		Serialization	Supported	Supported	Supported	Supported	Supported	Supported
	Databases	Database per service	Supported	Supported	Supported	Supported	Supported	Supported
		Shared database	Discouraged	Supported	Supported	Supported	Supported	Discouraged
		CQRS / event sourcing	Supported	Supported	Supported	Supported	Supported	N/A
	Observability	Log aggregation	Logback-based SLF4J, Log4J2	Logback, Java Util, Log4J2	Java Util, JBoss, SLF4J, Apache	Java Util, Log4J2, SLF4J	Log4J	Pino, Bunyan, Log4js, Datadog
		Performance metrics	Lightbend	Prometheus, Datadog, Netflix Atlas	Micrometer, SmallRye	Micrometer, Dropwizard	Micrometer, Datadog	Prometheus, StatsD, Datadog
		Distributed tracing	OpenTracing	Spring Cloud Sleuth	OpenTracing	Zipkin, OpenTracing	Jaeger, Zipkin	Built-in
		Health check	Through Akka	Spring Actuator	SmallRye	Built-in	Built-in	Built-in
	Cross-cutting concerns	External configuration	N/A	Spring Cloud Netflix, Archaius	N/A	Built-in	Built-in	Moleculer-runner
		Service registry	Built-in	Netflix Eureka, configurable	SmallRye Stork	Built-in	Consul, Eureka	Built-in
		Client-side discovery	ServiceLocator	Ribbon	Built-in	Built-in	Consul, Eureka	N/A
		Server-side service discovery	Supported	N/A	Built-in	Built-in	Consul, Eureka	Built-in

Criteria \ Framework		Lagom	Spring Boot	Quarkus	Vert.x	Micronaut	Moleculer
	Deployment platform	Docker, Kubernetes, Cloud Foundry, Azure, Heroku, Amazon AWS, OpenShift, Boxfuse, Mesosphere DC/OS					
Communication	Synchronous messaging	Supported	Supported	Supported	Supported	Supported	Supported
	Asynchronous messaging	Websocket, Kafka, MQTT, AMQP	Websocket, Kafka, MQTT, AMQP	Websocket, Kafka, MQTT, AMQP	Websocket, Kafka, MQTT, AMQP	Websocket, Kafka, MQTT, AMQP, STOMP	NATS, Kafka, Redis, MQTT, AMQP
	Remote procedure invocation	Akka gRPC library HTTP/JSON	RMI, Spring's HTTP Invoker, Hessian, Burlap	Built-in gRPC	Built-in gRPC	Built-in gRPC, RabbitMQ RPC	Built-in RPC
Fault Tolerance	Circuit breaker	Built-in	Netflix Hystrix	SmallRye	Built-in	Built-in	Built-in
	Retry	N/A	Spring-retry	SmallRye	Built-in	Built-in	Exponential back-off retry
	Timeout	Circuit breaker: call and reset timeouts	For HTTP components	SmallRye	Built-in	Built-in	Set timeout values for requests
Other Features	Caching	Play	Through dependencies	Built-in	Built-in	Built-in	built-in cache, customizable
	Minimum software requirements	Java 8, sbt 1.2	Java 8, Maven 3.2 or Gradle 4	Java 11, Maven 3.2 or Gradle 4	Java 8, Maven 3, Curl/HTTPie	Java 8	NodeJS 10, npm/yarn
	Security	SSL	Basic security, via dependency	SSL, authentication, authorization	SSL	SSL, authentication, authorization	SSL, authentication, authorization

Starting the analysis of Table 11 with the maturity criteria, the most popular and active frameworks were Quarkus and Spring. Lagom, Spring, and Vert.x have the most detailed documentation and guidance in terms of ease of implementation, however, Spring had a distinct lead in terms of Stackoverflow activity. To sum up the features, all frameworks provided the stated core services and were robust in all areas.

2.5 Summary

To summarize the findings, a better understanding of how to build and maintain reactive microservices through RQ_1 was accomplished, starting with the most important quality attributes and how to ensure them, and progressing through all of the challenges identified in other studies, containing valuable knowledge from previous experiences. In RQ_2, some of the most common issues in implementing reactive microservices were identified, as well as how to avoid them, by studying some of the defects and anomalies reported in previous studies. The best metrics and approaches for qualifying and assessing reactive microservices were published in RQ_3, backed up by some of the previously mentioned outcomes, allowing for a more accurate future evaluation and overall knowledge of reactive microservices. Finally, concerning RQ_4, the information gathered and summarized in Table 11 provided a broader understanding of the various frameworks for building reactive microservices. Through this chapter, it is possible now to not only design and implement the application with the right attributes and tools in mind, but also prepare the evaluation of the software with the gathered metrics.

3 Value Analysis

The fundamental goal of value analysis is to determine how a product's or idea's value may be maximized while keeping costs to a minimum without sacrificing quality. To do so, it considers three major components of the product: its utility, the customer's aesthetic and subjective worth, and the price the market is willing to pay for it (Nicola, 2020c).

The value analysis reflected in this chapter is divided into two phases: first, the project selection will be conducted through the TOPSIS method, followed by the framework selection using the analytic hierarchy process (AHP), a structured technique for organizing and analysing complex decisions. The opportunity analysis phase and the logical connections of the project were added into Annex A as they do not directly contribute to this project's work but can help to understand its value.

3.1 Project Selection

The technique of order preference by similarity to ideal solution (TOPSIS) was used to determine which project should be used for the migration to reactive microservices. TOPSIS is a systematic procedure based on three attributes or criteria: qualitative, quantitative, and cost, all of which may be weighted either positively or negatively. TOPSIS chooses the option that is the most identical to the ideal solution and the least similar to the negative ideal alternative (Nicola, 2020b). To apply the TOPSIS method and find which of these projects is most appropriate, 5 criteria were setup, with their respective weight and explanation:

1. Documentation - positive criteria with 0.25 weight, as it is the project's first contact and will provide most of its necessary context.
2. Community relevance, GitHub stars, forks and people watching - positive criteria with 0.15 weight. The existence of a strong community around a project can help highlight some of its value.

3. Project activity, Number of contributors and commits - positive criteria with 0.15 weight. The activity of a project can help in avoiding outdated software and practices.
4. The practice of microservice-related features - positive criteria with 0.25 weight. The existence of common microservice patterns shows the correctness of the project and helps to avoid initial maintenance.
5. The practice of microservice anti-patterns - negative criteria with 0.2 weight. The existence of anti-patterns may require entire components to be rebuilt increasing the workload of the project.

Then, regarding the projects considered, the list curated by Rahman et al., 2019 was the starting point for the selection. This list is based on input from various platforms as well as a list of microservices-based projects described in academic articles and is composed of over 50 projects. Next, a few criteria were set to filter the projects, to ease the migration process and not compromise the analysis to be done. The projects must:

- Be implemented in a programming language known to the researcher and compatible with the aforementioned frameworks in Table 11;
- Have between 3 and 10 microservices;
- Not employ the reactive principles;
- Have documentation regarding the design process;
- Be written in either English or Portuguese.

Resulting in the following list:

1. FTGO - Restaurant Management (Richardson, 2018a);
2. Eberhard Wolff's 11 Systems (Wolff, 2015);
3. Cloud Native Strangler Example (Bastani, 2016);
4. Kenzan Million Song Library (Perkins, 2018);
5. Tap-And-Eat-MicroServices (Ferrater, 2017).

Finally, with all the information necessary gathered, the criteria can be applied to the alternatives, as shown in Table 12:

Table 12. TOPSIS decision matrix

	Documentation	**Community relevance**	**Project activity**	**Practice of microservice related features**	**Practice of microservice anti-patterns**
FTGO - Restaurant Management	8	4	2	8	3

	Documentation	Community relevance	Project activity	Practice of microservice related features	Practice of microservice anti-patterns
Eberhard Wolff's 11 Systems	2	7	7	8	3
Cloud Native Strangler Example	4	9	7	8	6
Kenzan Million Song Library	2	2	9	7	4
Tap-And-Eat-MicroServices	3	2	3	8	4

To discover its closeness to the ideal answer, this matrix must first be normalized and then weighted, as can be seen in Table 13:

Table 13. TOPSIS normalized and weighted decision matrix

	Documentation	Community relevance	Project activity	Practice of microservice related features	Practice of microservice anti-patterns
FTGO - Restaurant Management	0.203	0.048	0.022	0.115	0.065
Eberhard Wolff's 11 Systems	0.051	0.085	0.076	0.115	0.065
Cloud Native Strangler Example	0.102	0.109	0.076	0.115	0.129
Kenzan Million Song Library	0.051	0.024	0.097	0.100	0.086
Tap-And-Eat-MicroServices	0.076	0.024	0.032	0.115	0.086

Finally, with the matrix normalized and weighted, the distance from the ideal and negative ideal solutions can be determined, to find the alternative closest to the ideal solution.

Table 14. TOPSIS closeness to ideal solution.

	Separation from the ideal solution	Separation from the negative ideal solution	Closeness to the ideal solution
FTGO - Restaurant Management	0.097	0.168	0.634
Eberhard Wolff's 11 Systems	0.156	0.104	0.402
Cloud Native Strangler Example	0.122	0.113	0.481
Kenzan Million Song Library	0.176	0.087	0.331
Tap-And-Eat-MicroServices	0.167	0.053	0.241

As demonstrated in Table 14, the "FTGO - Restaurant Management" project is the best candidate for migration since it is the closest to the ideal solution.

3.2 Framework Decision

To find the most appropriate framework to use the Analytic Hierarchy Process was employed. AHP was designed by Professor Thoma L. Saaty in 1980 (Saaty, 1980) and is one of the most widely used approaches in the discrete multicriteria decision-making environment. In the evaluation process, this method allows for the use of both qualitative and quantitative criteria. The fundamental idea behind this strategy is to break down the choice problem into levels, making it easier to comprehend and evaluate. (Nicola, 2018)

3.2.1 Hierarchic Division

Starting with the hierarchic division of the process, its main objective was to find the most appropriate framework to implement reactive microservices. Following Figure 8 and according to the prior research made in Table 11, the solution should take into account three primary criteria: maturity, ease of implementation, and framework features. On the last level, the alternatives considered were Lagom, Spring, Quarkus, Vert.x, Micronaut, and Moleculer, also based on Table 11.

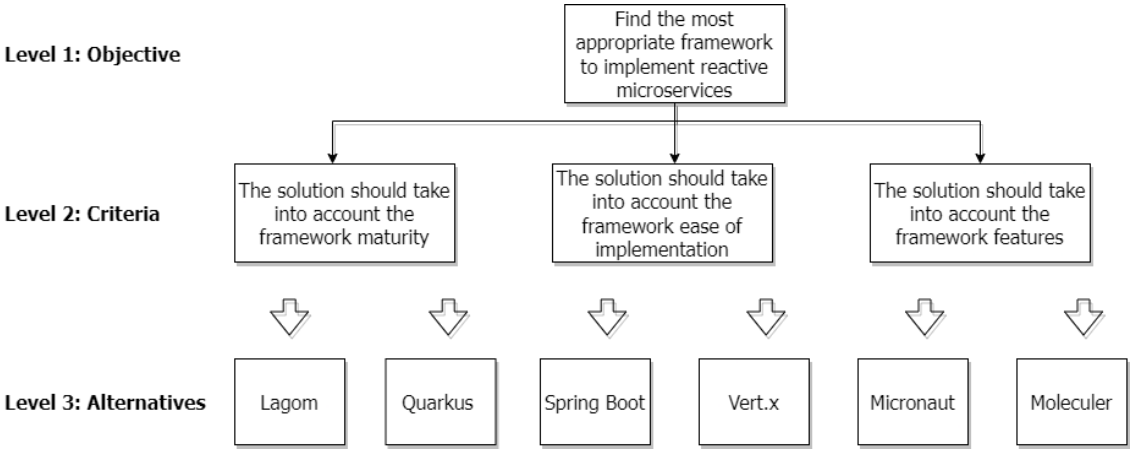

Figure 8. Hierarchic division

3.2.2 Priority Definition

To find out each criterion's priority, a pairwise matrix was utilized, as shown in Equation 1:

Equation 1. Criteria comparison base equation

$$A = [a_{ij}] = \begin{bmatrix} 1 & a_{12} & \cdots & a_{1n} \\ 1/a_{12} & 1 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 1/a_{1n} & 1/a_{2n} & \cdots & 1 \end{bmatrix} \; i = 1,2,\dots,n;\; j = 1,2,\dots,n$$

The criteria are compared between one another, indicating if it is either more, less important or of equal importance, following the levels indicated in Table 15.

Table 15. Importance levels of comparisons (Saaty, 1980)

Level of importance	*Definition*	*Explanation*
1	Equal importance	The two activities contribute equally to the objective
3	Weak importance	Experience and judgement favour slightly one activity over the other
5	Strong importance	Experience and judgement strongly favour one activity over the other
7	Very strong importance	One activity is very strongly favoured over the other
9	Absolute importance	Evidence favours one activity over another with the highest degree of certainty
2,4,6,8	Intermediary value	When looking for a compromise condition between two definitions

As displayed in Table 16, the framework's features were regarded as the most significant, with a strong to very strong importance compared to maturity and a somewhat stronger relevance compared to the simplicity of implementation, since these two factors are more of quality-of-life criteria. The ease of implementation, when contrasted to the maturity criterion was determined to be of weak importance.

Table 16. Criterion comparison matrix

	Maturity	**Ease of Implementation**	**Features**
Maturity	1	1/2	1/6
Ease of Implementation	2	1	1/4
Features	6	4	1

Then, the matrix needs to be normalized to obtain the weights or eigenvalues of each criterion. Table 13 states the results.

Table 17. Weight of the criteria

Criteria	*Relative Priority - Weight*	*Importance*
Maturity	0.106	Low
Ease of Implementation	0.193	Medium
Features	0.701	Highest

The frameworks may now be compared once each criterion has been appropriately ranked. Starting with the maturity criteria in Table 18, Spring and Quarkus were deemed the most mature frameworks, having the greatest number of GitHub commits, releases and contributors, followed by Micronaut, Vert.x and lastly, Lagom and Moleculer.

Table 18. Weight of alternatives by maturity

Maturity	**Lagom**	**Spring**	**Quarkus**	**Vert.x**	**Micronaut**	**Moleculer**	**Weight**
Lagom	1	1/4	1/4	1/2	1/2	1	0.071
Spring	4	1	1	2	2	4	0.284
Quarkus	4	1	1	2	2	4	0.284
Vert.x	2	1/2	1/2	1	1/2	2	0.128
Micronaut	2	1/2	1/2	2	1	2	0.163
Moleculer	1	1/4	1/4	1/2	1/2	1	0.071

Regarding the ease of implementation criteria, although Spring had the most extensive documentation, tutorials and Stackoverflow tags, Lagom was deemed the easiest framework to implement, based on its development environment, build tools and the ability to be developed

on Scala, followed by Spring, Vert.x, Quarkus, Micronaut and lastly Moleculer because of its limited documentation and base programming language.

Table 19. Weight of alternatives by ease of implementation

Ease of Implementation	Lagom	Spring	Quarkus	Vert.x	Micronaut	Moleculer	Weight
Lagom	1	2	3	2	4	5	0.338
Spring	1/2	1	3	2	4	5	0.267
Quarkus	1/3	1/3	1	1/2	2	3	0.112
Vert.x	1/2	1/2	2	1	2	3	0.160
Micronaut	1/4	1/4	1/2	1/2	1	2	0.075
Moleculer	1/5	1/5	1/3	1/3	1/2	1	0.049

Finally, Table 20 is shown the comparison of the feature criteria. Even though the majority of the frameworks were quite robust, the built-in inclusion of features was judged more important as a method to differentiate the framework's emphasis on reactivity. As a result, Lagom was determined to be the framework with the best feature set, followed by Quarkus, Vert.x, Micronaut, Spring, and lastly Moleculer.

Table 20. Weight of alternatives by features

Features	Lagom	Spring	Quarkus	Vert.x	Micronaut	Moleculer	Weight
Lagom	1	3	1	2	2	4	0.284
Spring	1/3	1	1/2	1	1	2	0.124
Quarkus	1	2	1	2	2	4	0.263
Vert.x	1/2	1	1/2	1	1	2	0.132
Micronaut	1/2	1	1/2	1	1	2	0.132
Moleculer	1/4	1/2	1/4	1/2	1/2	1	0.066

3.2.3 Logic Consistency

Before skipping to conclusions, first, the logical consistency of the process executed must be assessed. This is achieved by calculating the consistency ratio (CR) by dividing the consistency index (CI) by the random consistency index (RI), as shown in Equation 2. If the RC is greater than 0.1, the judgments are unreliable because they are too close for the comfort of randomness, resulting in inconsistent values.

Equation 2. Consistency ratio calculation

$$CR = \frac{CI}{RI}$$

The consistency index can be calculated with the value of λmax, which denotes the biggest eigenvalue of matrix A, using the Equation 3:

Equation 3. Consistency index calculation

$$CI = \frac{\lambda_{max} - n}{n - 1}$$

Then, the random consistency index is a fixed value depending on the matrix dimensions, as shown in Table 21.

Table 21. Random consistency index values for n dimensions

Matrix dimension	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
Random consistency index	0	0	0.58	0.90	1.12	1.24	1.32	1.41	1.45	1.49

Finally, the logic consistency can be assured, as all the CR values are below 0.1, as registered in Table 22.

Table 22. Consistency ratios of the comparison matrices produced

Comparison matrix	***Consistency Ratio (CR)***
Criterion	0.010
Alternative's maturity	0.009
Alternative's ease of implementation	0.021
Alternative's features	0.003

3.2.4 Results Analysis

To reach the final decision, each framework's criteria weight must be properly merged. Equation 4 shows the matrix multiplication executed to reach these results.

Equation 4. The final weight of each alternative

$$\begin{bmatrix} L_m & L_e & L_f \\ S_m & S_e & S_f \\ Q_m & Q_e & Q_f \\ V_m & V_e & V_f \\ Mi_m & Mi_e & Mi_f \\ Mo_m & Mo_e & Mo_f \end{bmatrix} \times \begin{bmatrix} m \\ e \\ f \end{bmatrix} <=> \begin{bmatrix} 0.071 & 0.338 & 0.284 \\ 0.284 & 0.267 & 0.124 \\ 0.284 & 0.112 & 0.263 \\ 0.128 & 0.160 & 0.132 \\ 0.163 & 0.075 & 0.132 \\ 0.071 & 0.049 & 0.066 \end{bmatrix} \times \begin{bmatrix} 0.106 \\ 0.193 \\ 0.701 \end{bmatrix} = \begin{bmatrix} 0.272 \\ 0.168 \\ 0.236 \\ 0.137 \\ 0.124 \\ 0.063 \end{bmatrix}$$

It can be concluded that Lagom is the most appropriate framework to implement reactive microservices.

3.3 Summary

Through the conducted value analysis, the framework and project to use were chosen, using the AHP and TOPSIS methods respectively, reducing some potential bias from the decision, and optimizing the judgment's quality. Through the application of these methods and the previously gathered knowledge in the state of the art, in chapter 2, two of the most important choices in this project can safely be called calculated and established options.

4 Analysis and Design

With all the pre-requisites gathered, in this chapter, the previously chosen project "FTGO - Restaurant Management" will be deeply analysed. This is an auxiliary project for the book "Microservice Patterns", written by Chris Richardson, a recognized thought leader in microservices who speaks regularly at international conferences. Chris is the creator of Microservices.io, a pattern language for microservices. He provides microservices consulting and training to organizations around the world that are adopting the microservice architecture (Richardson, 2018b). Firstly, some of its initial requirements will be studied, followed by new reactive-related requirements. Then an examination of its architecture will be conducted, highlighting the key changes to be done through various perspectives and levels of granularity, to achieve a reactive implementation.

4.1 Requirements Engineering

The project documents the management of a restaurant, from the restaurant information to its orders, deliveries and all the logistics in between, and covers the process of moving a typical monolithic application into a microservice architecture. Table 23 and Table 24 showcase the functional and non-functional requirements respectively. Because it is not the focus of this study, no additional function requirements were added, but certain non-functional requirements were introduced to better record and guarantee the practice of reactive patterns.

Table 23. Functional requirements (Richardson, 2018b)

Id	Functional requirement	Description	Maturity
FR_1	Courier management	Manage courier information	Old
FR_2	Restaurant information management	Managing restaurant menus and other information, including location and open hours	Old
FR_3	Consumer management	Managing information about consumers	Old
FR_4	Order management	Enabling consumers to create and manage orders.	Old

Id	Functional requirement	Description	Maturity
FR_5	Restaurant order management	Managing the preparation of orders at a restaurant	Old
FR_6	Courier availability management	Managing the real-time availability of couriers to delivery orders	Old
FR_7	Delivery management	Delivering orders to consumers	Old
FR_8	Consumer accounting	Managing the billing of consumers	Old
FR_9	Restaurant accounting	Managing payments to restaurants	Old
FR_{10}	Courier accounting	Managing payments to couriers	Old
FR_{11}	Kitchen management	Manage kitchen order tickets	Old

Table 24. Non-functional requirements

Id	Non-functional requirement	Description	Maturity
NFR_1	Usage of API Gateway	All services must be addressed through the API Gateway.	Old
NFR_2	Usage of the command query responsibility segregation (CQRS)	All microservices must utilize CQRS.	New
NFR_3	Usage of the SAGA pattern	SAGA transactions must be adopted between all microservices.	Old
NFR_4	Usage of service registry	All services must register themselves on the API Gateway.	Old
NFR_5	Message-based internal communication	All communications between microservices must be asynchronous and message-driven to respect reactive principles.	New
NFR_6	Usage of the circuit breaker	To improve fault tolerance, a circuit breaker must be used.	New
NFR_7	Usage of the fault monitor	To improve fault tolerance, a fault monitor must be used.	New

4.2 Domain Modelling

Regarding the domain model, as shown in Figure 9, the application was constructed following the domain driven design and consists of seven aggregates (Richardson, 2018b):

1. Consumer – manages the common consumer information.
2. Restaurant – manages the restaurant menu and overall information.
3. Order – manages the order information.
4. Kitchen – manages the ongoing tickets of the restaurant.
5. Courier – manages the order delivery and its general status.
6. Accounting – manages the accounts and payments for the restaurants, consumers, and couriers.

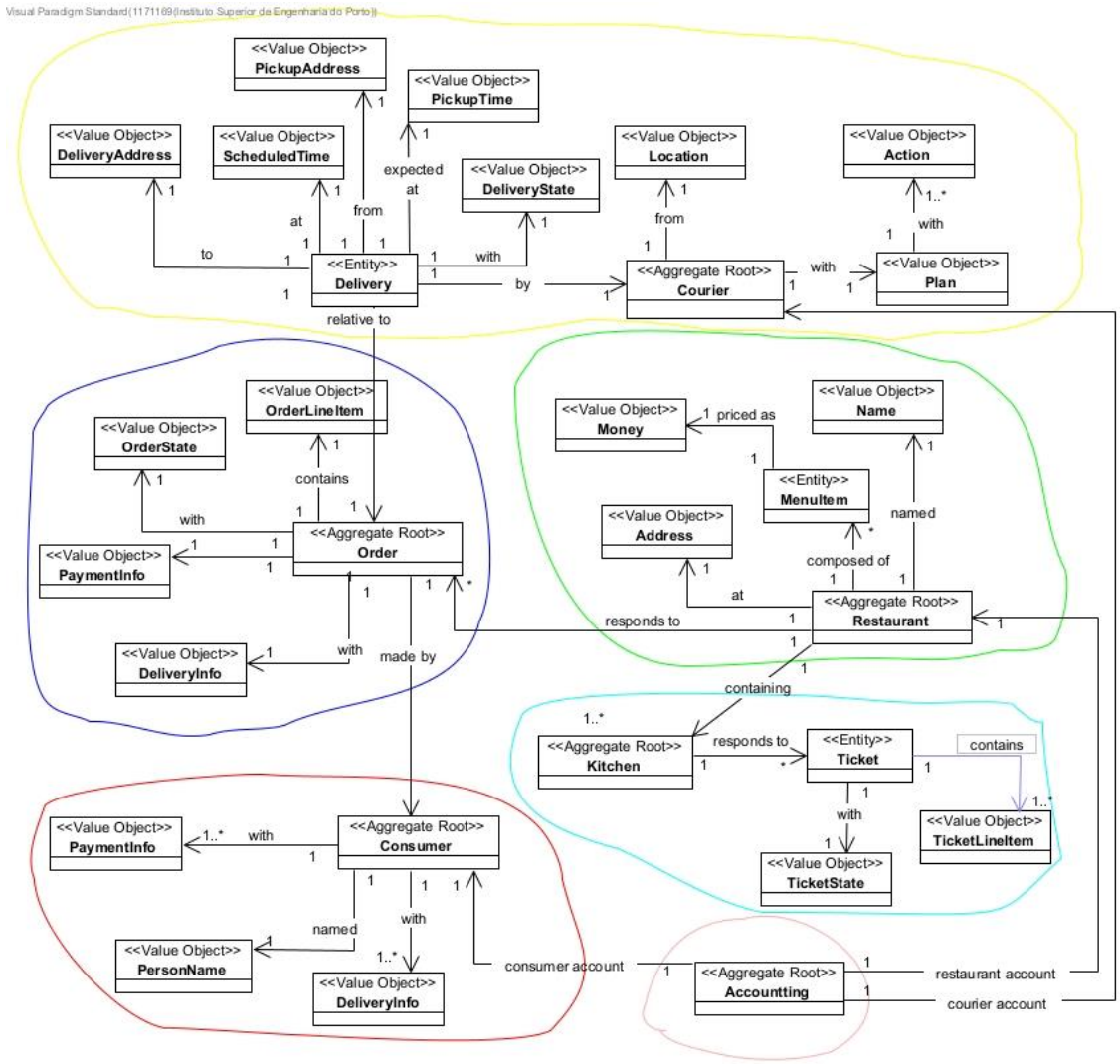

Figure 9. Domain Model

4.2.1 Context Mapper

Next, by employing the context mapper tool, a modular and extensible modelling framework for domain driven design, the current context definition and some of the high-level communication protocols used can be better visualized, ensuring that the reactive principles are correctly utilized in every step of the process. As can be seen in Figure 10, no necessary changes were identified through this view, as these definitions do not constrict the synchronisation of the communication. The protocols specified are:

- Upstream(U)-downstream(D): represents the flow of information.
- Open host service (OHS): describes the function of a bounded context in supplying specific functions that are required by several contexts.
- Published language (PL): the published language describes the shared knowledge two bounded contexts need for their interaction.

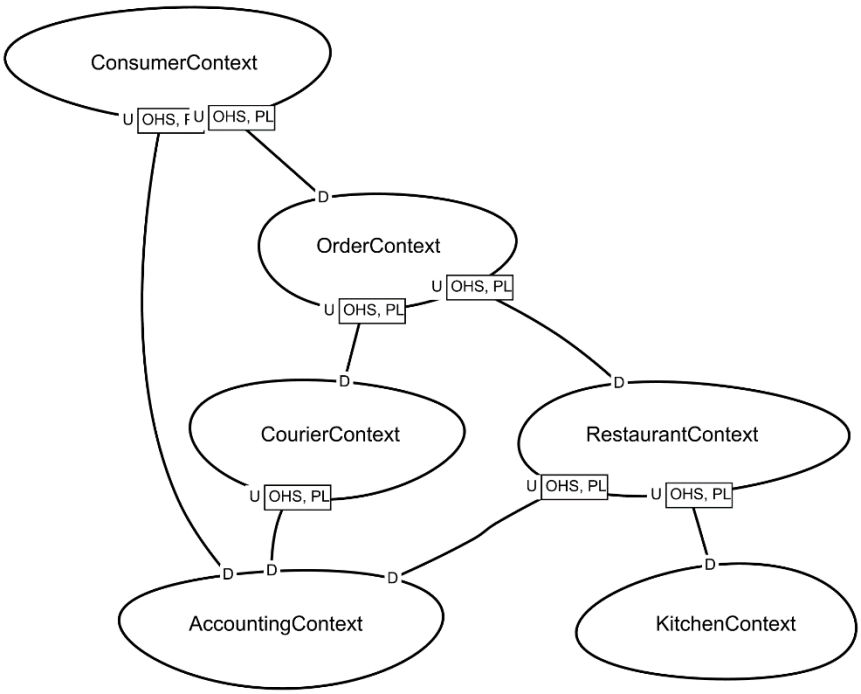

Figure 10. Context mapper model

The consumer context will provide data such as its address and payment info to its orders and also for accounting purposes. The order context will communicate to the restaurant its order items to be validated and then added to its kitchen. After the order context receives the information that its order is ready, it then informs the courier context for it to be delivered. The courier and the restaurant also provide their information to the accounting context, for accounting purposes.

4.3 C4 Model and 4+1 Views

The C4 model is a hierarchical set of diagrams of four levels: context, containers, components, and code, assisting to describe the architecture with different granularities (Brown & Betts, 2018), and the 4+1 views, which consists of describing the architecture in five different views: logical, process, implementation, physical, and use cases. The name 4+1 derives from the use case perspective, which is crucial at the start of the design but superfluous at the end (Staveley, 2011). The application may be examined on numerous scopes and perspectives thanks to the synergy of these tools, eliminating possible ambiguities and enabling the migration to reactive microservices by easily locating required adjustments.

4.3.1 Use Case View

Before starting the documentation of the various C4 levels, the use case view must be first built, as its foundation. Figure 11 illustrates the outlined functional requirements in 4.1:

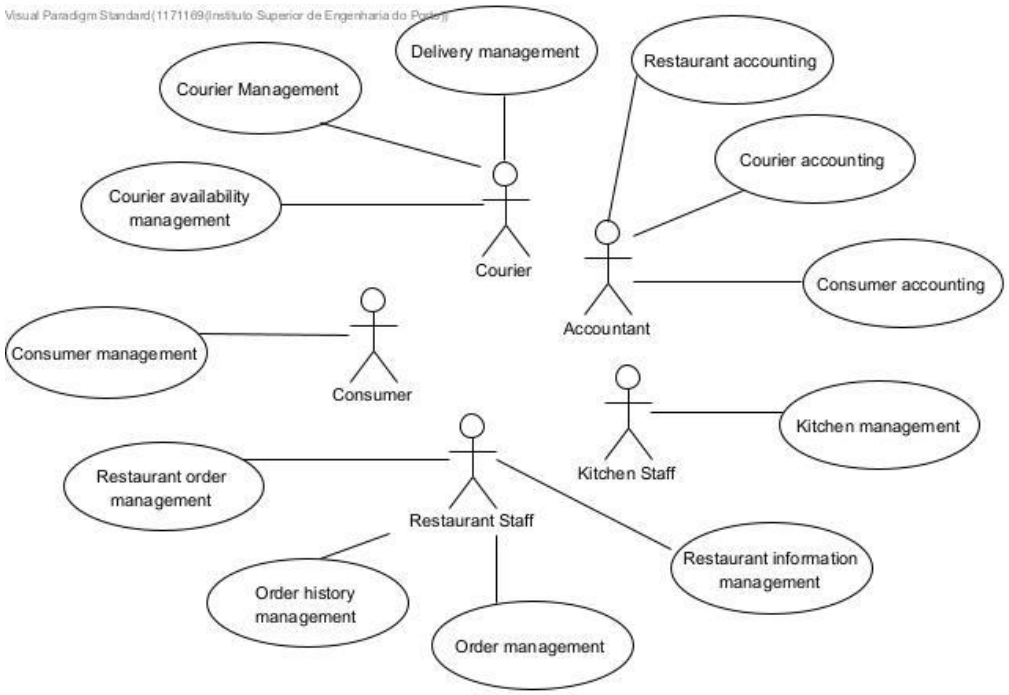

Figure 11. Use case diagram

4.3.2 Context Level

Starting with the most abstract level, the context of the system, 5 categories of actors are identified in the logic view: restaurant staff, kitchen staff, consumers, accountants, and couriers, as seen in Figure 12.

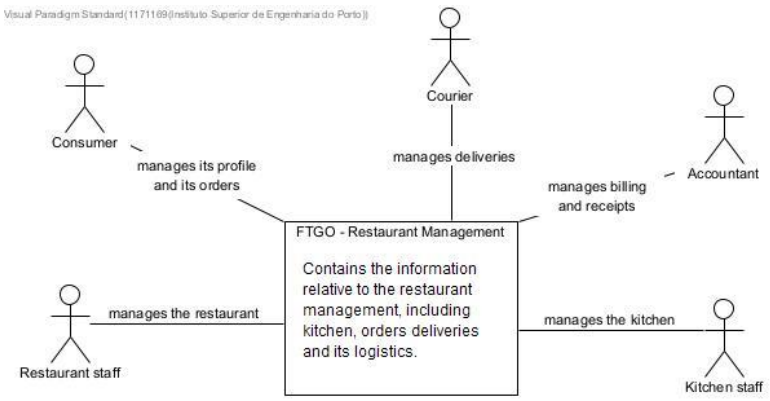

Figure 12. Logic view at the context level diagram

Regarding the interaction with other systems, the current implementation uses third-party payment systems, two databases, MySQL for relational data and DynamoDB as a NoSQL document database for event sourcing and better performance on reading data, and Kafka as the messaging bus system, as can be seen in Figure 13.

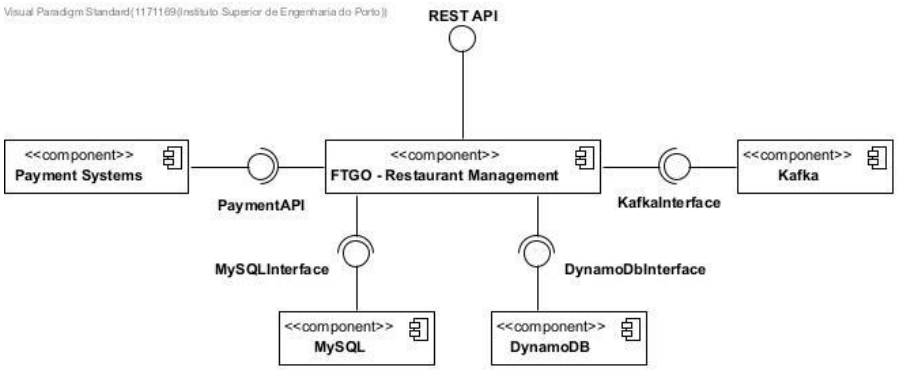

Figure 13. Logic view of the interaction with other systems at the context level

Due to DynamoDB being a closed source dependency, two alternative migrations were idealized:

- In Figure 14, the chosen alternative, DynamoDB was replaced by Cassandra, a NoSQL database reasonably similar to DynamoDB which is natively supported by Lagom and complies with the reactive principles and the BASE transaction principles, while providing linear scalability and high performance (Bekker, 2018).
- Furthermore, the Akka and Play dependencies are added as foundations for tools such as service discovery and CQRS persistence strategies in Lagom (Lightbend, 2018).
- In Figure 15, the only difference in comparison with the first alternative is that the MySQL dependency is removed, and Cassandra is used for both the read and write side of the application, providing a simpler implementation and better development curve, but giving away some of the benefits of CQRS, such as a specialized database for reads, MySQL, and a high performance write database for events, Cassandra.

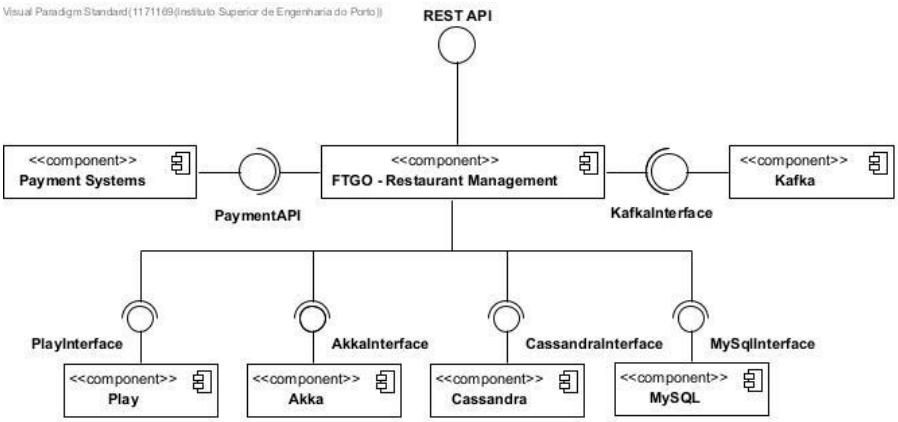

Figure 14. Updated logic view of the interaction with other systems at the context level

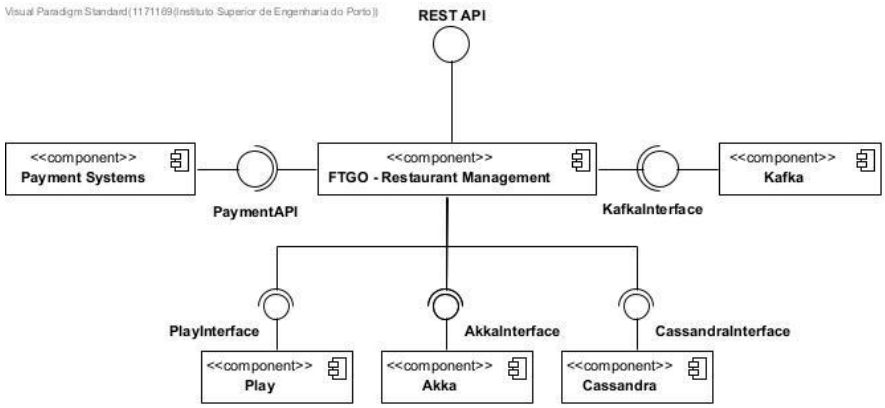

Figure 15. Alternative logic view of the interaction with other systems at the context level

4.3.3 Container Level

Next, the system is expanded to the container level, in Figure 16. Through this granularity, the seven idealized microservices are materialized and coordinated with each external dependency as well as the APIGateway and the service discovery components. Because of the large number of connections, some were left off, such as connecting all business microservices to the Play, Akka, Cassandra, and Kafka interfaces, since this would provide no additional information and make the diagram even more convoluted. It's worth noting that the business microservices don't have any direct interdependencies since they all communicate via asynchronous Kafka messaging. The current container level diagram is not shown as the only differences are the external dependencies, already highlighted at the context level.

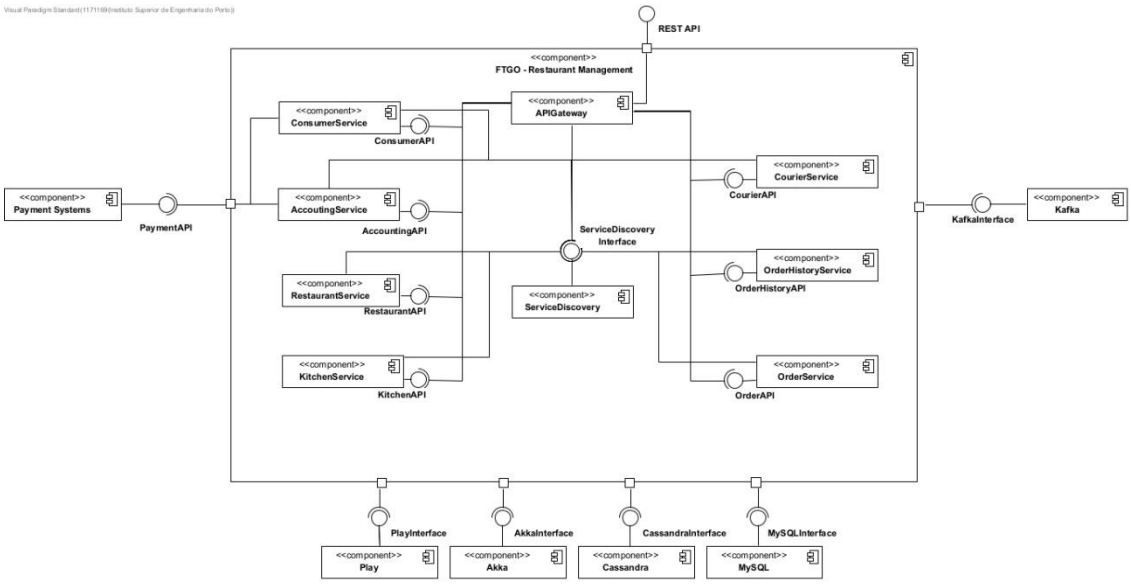

Figure 16. Updated logic view at the container level

4.3.4 Component Level

Moving to the third level, the focus shifts to an individual container to show its integral components. The consumer service was selected for demonstration as it is the most extensive service. The current architecture, represented in Figure 17, is divided into 3 layers:

- **Controller layer**, responsible for handling the read requests of each microservice, through the provided HTTP REST API and forwarding them to the service layer;
- **Messaging layer**, responsible for handling the incoming Kafka messages and forwarding them to the service layer;
- **Service layer**, responsible for the domain object validation, safe keep and execution of its actions.

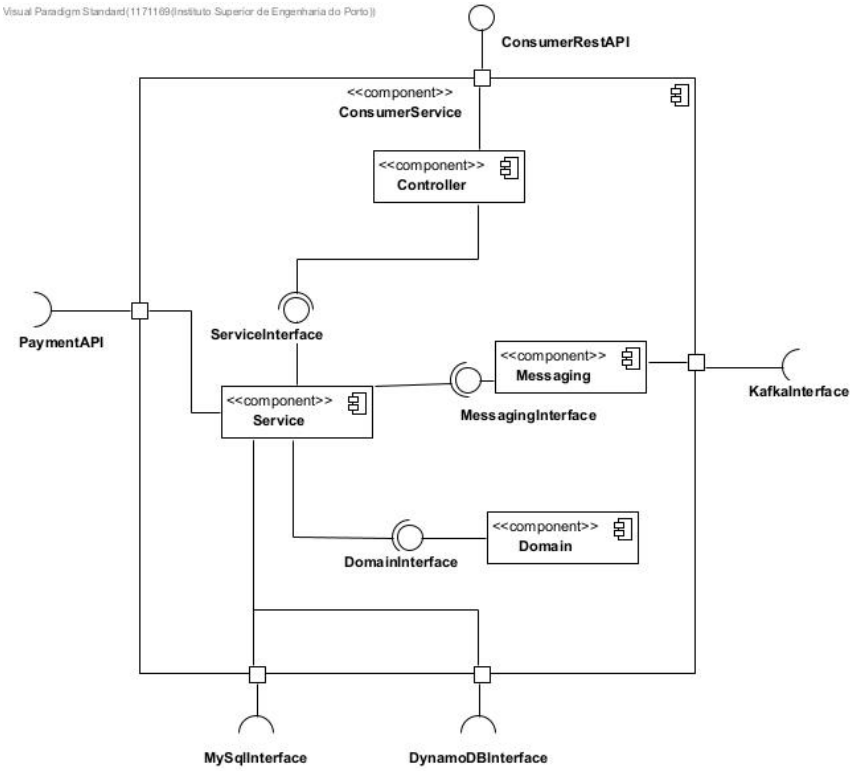

Figure 17. Current logic view of the consumer service component

To improve readability and code flexibility, an auxiliary mapper package and an extra layer were added, and some of the behaviour was shifted between layers and objects:

- **Controller layer**, similar to the old controller layer but is supported by data transfer objects (DTO);
- **Messaging layer**, similar to the old messaging layer, encompasses the CQRS division. Also supported by DTOs;
- **Service layer**, similar to the old service layer, but the object storage responsibility was moved to repository layer.
- **Repository layer**, responsible for managing the information and its integrity.

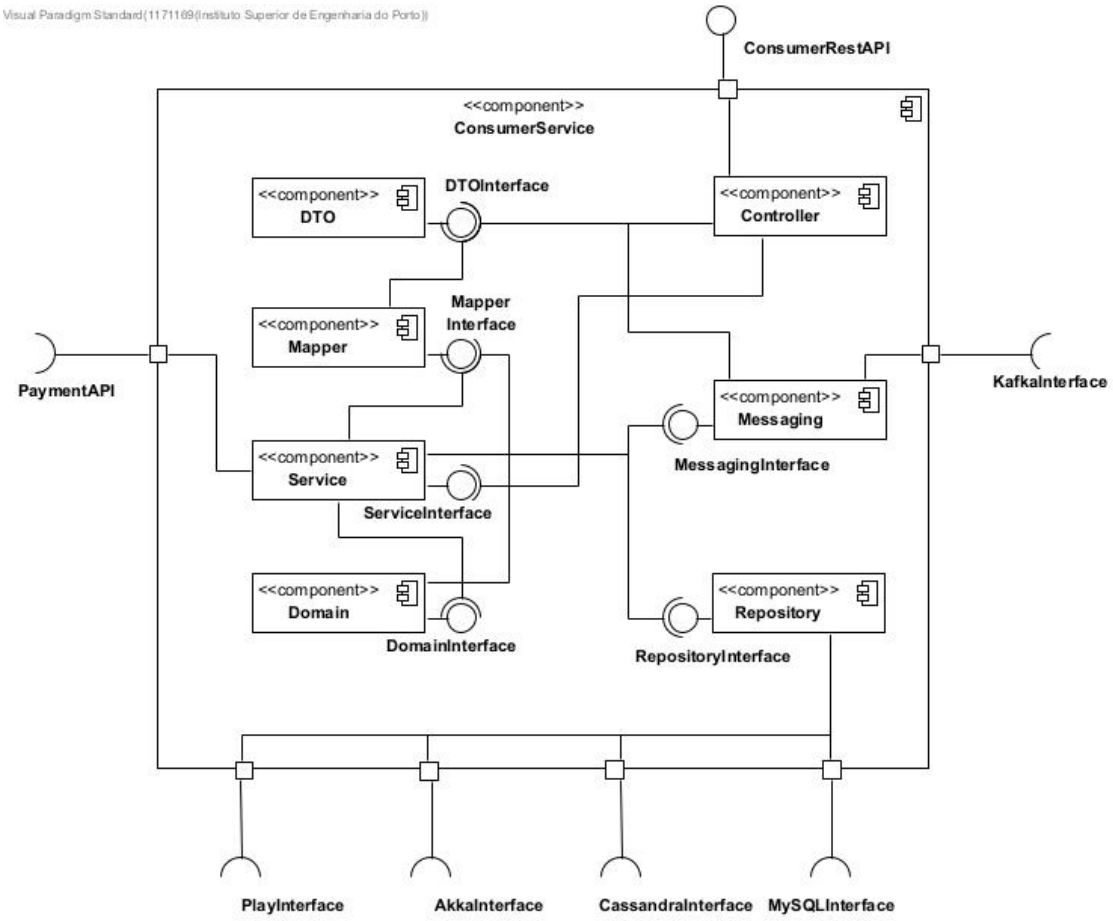

Figure 18. New logic view of the consumer service component

Now, through the implementation view, the system is observed with more focus on the relationships and flow of the various components, as can be seen in Figure 19:

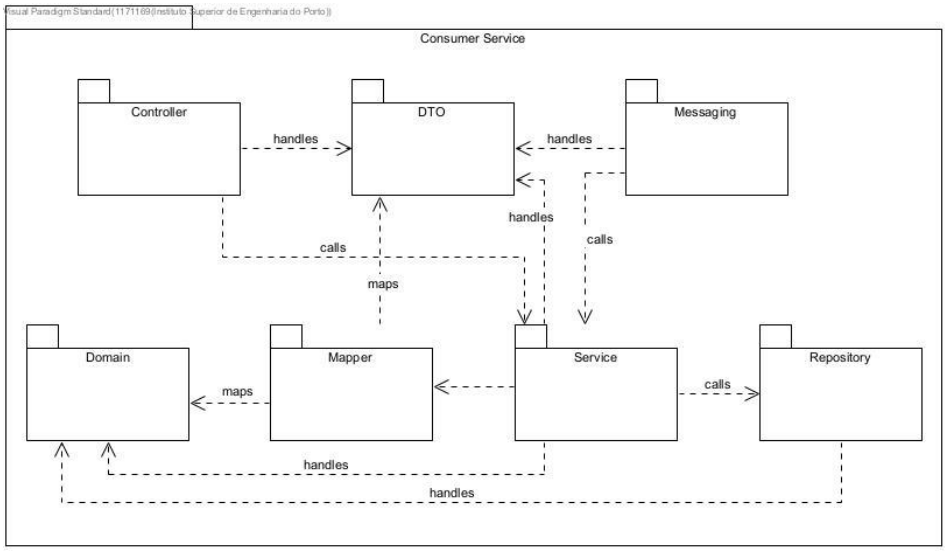

Figure 19. New implementation view of the consumer service component

Ultimately, the process view can be used to demonstrate the concrete flow of a request according to the architecture idealized, while maintaining the level of abstraction indicated. Functional requirement 4, “Order management”, was chosen to explain this, in Figure 20. As the customer sends a request to create a new order, it is validated and saved in the respective service. Then, the success of this creation is then propagated to other services through its Kafka queue, allowing asynchronous communication of this information to related businesses, demonstrating how the domain model's bounded context interdependencies are handled, lowering coupling between services, and ensuring the message-driven principle of reactivity.

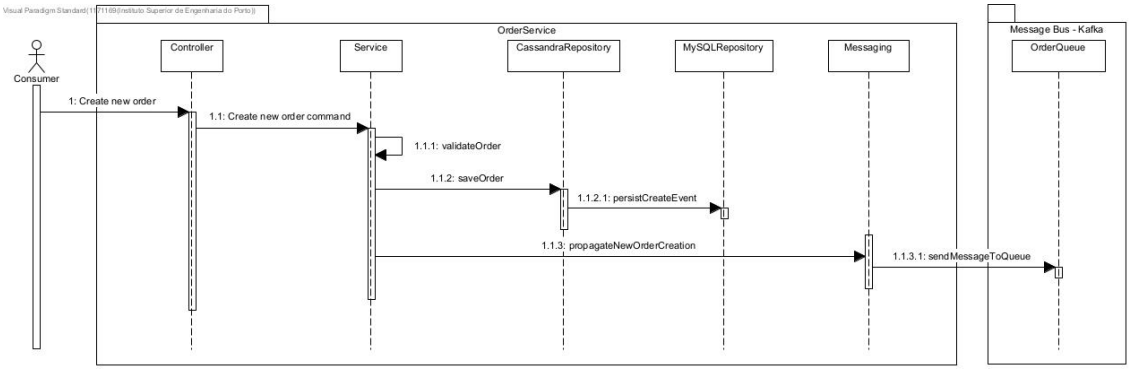

Figure 20. New process view of the order management requirement at the component level

4.3.5 Code Level

Finally, at the most concrete level of the C4 model, the system is extended to the context of the functional requirements. Functional requirement 4, “Order management”, was chosen to expand the logic previously addressed at the component level.

In Figure 21, the process view, explains in more detail, some of the planned code improvements, such as the use of mappers to streamline the transformation of the various data types and the repository layer, allowing for a clear division between the service and repository capabilities. The use of CQRS can also be seen, as the creation request is segregated to the command controller.

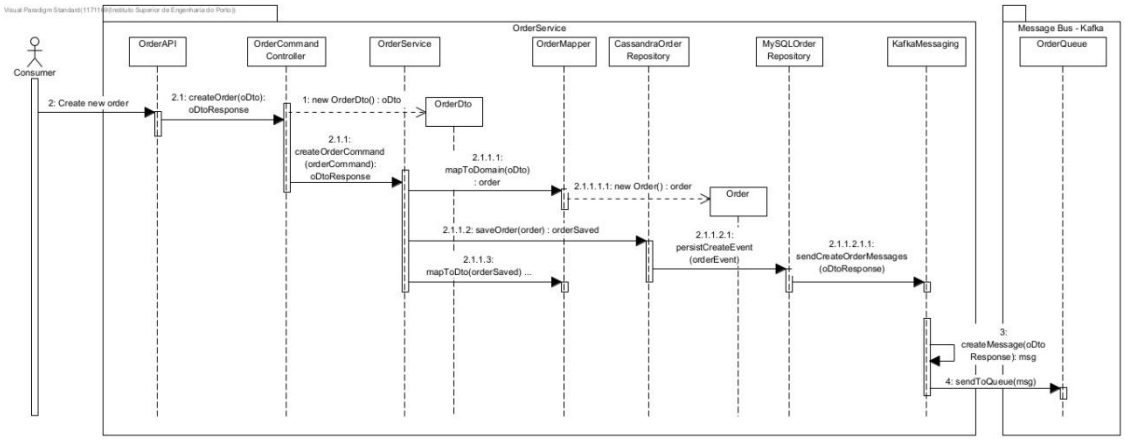

Figure 21. New process view of the order management requirement at the code level

In Figure 22, the component implementation view is materialised into the order management classes and its dependencies can be traced. Through this view, some of the domain model elements are now evidently connected with the rest of the system.

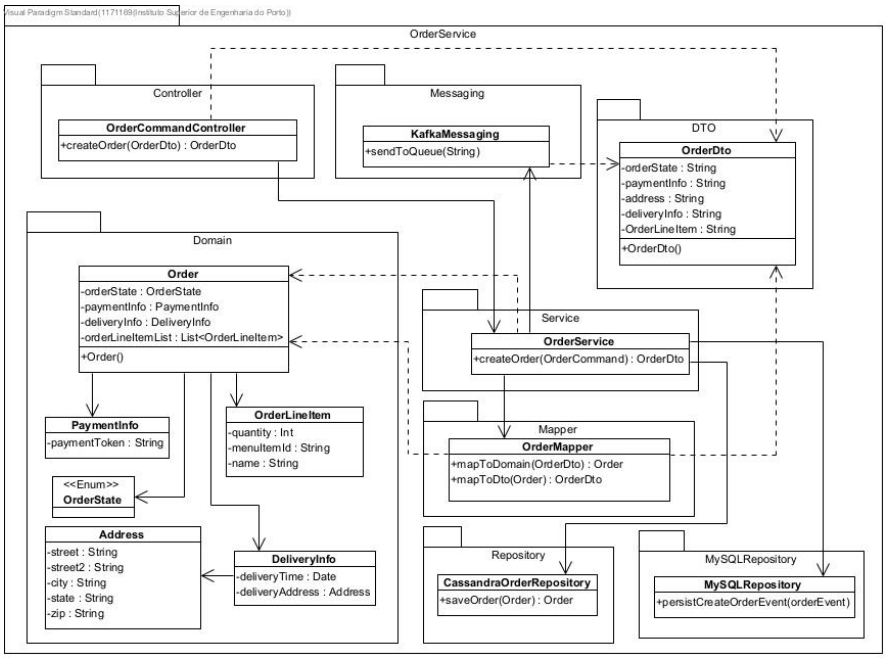

Figure 22. New implementation view of the order management requirement at the code level

4.4 Summary

To summarize, some of the changes to be made are distinctly tracked and assessed through the dissection of the present and new architectures, enabling a smoother migration process by reducing any ambiguities throughout the design. By designing both the new changes and the previous information it is possible to better assess some of the shared code and communications between the microservices.

5 Solution Implementation

The implemented solution will be examined in this chapter, beginning with the software's setup, first by creating an application skeleton using the Lagom project template generator, then by listing the main dependency versions and the reasoning behind them, and finally by containerizing the application. Next, the methods and patterns used during implementation, as well as how the framework helps implementation will be discussed. Following that, the tests created to check the application's quality and correctness will be displayed, finalized by the integration of the metrics tools, which will be used in chapter 6. To aid the reader, the source code can be consulted at *https://bitbucket.org/Jose_Ferreira_1171169/ftgo-reactive*.

5.1 Project Setup

Lagom provides an out-of-the-box solution to the project setup, offering an embedded instance of all the needed external services. Although this is a great way to start the development of the solution, a better environment must be set up to mimic a user-ready production environment and to improve the solution assessment.

5.1.1 Creation of the Project Skeleton

For the creation of the project, Lagom makes use of the sbt template resolver mechanism. Through the sbt “new” command and Gitter8, a templating project originally started by Nathan Hamblen in 2010 which uses a Git repository to host the templates (sbt, 2022), the project skeleton can easily be configured, as can be seen in Annex C - Code 10.

Then, each microservice idealized in 4.2 must be integrated in build.sbt in order to allow sbt to create all the necessary packages and configurations, as shown in Code 1:

```
1  lazy val restaurantApi = (project in file("restaurant-api"))
2    .settings(
3      version := "1.0-SNAPSHOT",
4      libraryDependencies ++= Seq(
```

```
5         lagomScaladslApi
6       )
7     )
8     .dependsOn(ftgoCommon)
9     .aggregate(ftgoCommon)
10
11  lazy val restaurantImpl = (project in file("restaurant-impl"))
12    .enablePlugins(LagomScala, Cinnamon)
13    .settings(
14      version := "1.0-SNAPSHOT",
15      libraryDependencies ++= Seq(
16        lagomScaladslPersistenceCassandra,
17        lagomScaladslPersistenceJdbc,
18        lagomScaladslKafkaBroker,
19        lagomScaladslTestKit,
20        lagomScaladslAkkaDiscovery,
21        macwire,
22        scalaTest
23      ),
24      commonSettings
25    )
26    .settings(cinnamonSettings)
27    .dependsOn(restaurantApi, orderApi, ftgoCommon)
28    .aggregate(ftgoCommon)
```

Code 1. Registration of the restaurant microservice in build.sbt

It is important to highlight that Lagom forces the splitting of a generic API from its implementations, exemplified in Figure 23, attaining a higher level of service decoupling, allowing the existence of several implementations of the same API, and reducing duplicated code (Lightbend, 2022c). This was a necessary adaptation of the architecture that does not affect any of the process views and only slightly impacts the implementation view at the code level displayed in 4.3.5. Nevertheless, a new representation of this diagram was included in Annex C - Figure 47.

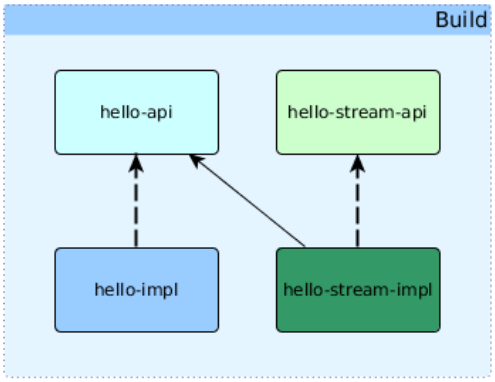

Figure 23. Lagom project structure (Lightbend, 2022c)

5.1.2 Dependency Setup

Regarding the application's dependencies, an attempt was made to use the most recent stable version of each module without impacting other dependencies or the application's overall performance and security. After numerous trials, Table 25 outlines all of the project's major dependencies.

Table 25. Primary dependencies of the application

Dependency	Description	Version
sbt	The dependency manager of the project. Version 1.4.9 was used to minimize java compatibility issues.	1.4.9
Java	JVM version to be used by the Scala compiled code.	11.0.15
Scala	The programming language used for the project. The version used was the latest compatible version with Lagom.	2.13.8
Lagom	The framework of this project is based. This is the current latest stable version of the framework.	1.6.7
Scalafmt	Scala code formatter is used to uniformize all the written code and to lessen the number of bugs and code smells in the application.	2.4.2
Cinnamon	Lightbend Telemetry sbt plugin, to ease the compilation of metrics and statistics.	2.16.0
Sonar-Scala	SonarQube connector for Scala applications.	2.3.0
MySQL connector	MySQL connector for Scala applications.	8.0.29
Macwire	Dependency Injection tool, auxiliary to the Lagom framework.	2.5.7
ScalaTest	Scala testing framework.	3.2.12
ScalaCheck	Scala property-based testing library.	1.16.0
Akka discovery	Service discovery tool, auxiliary to the Lagom framework.	1.6.7

5.1.3 Containerization of the Solution

Completing the initial setup of the application, Docker was used to mimic a production-level environment through containerization, with the help of the sbt-native-packager plugin, which allows the generation of a docker image of every existing service (Lightbend, 2022e).

The deployment diagram in Figure 24 summarizes all the different services and their dependencies added in the docker-compose (Ferreira, 2022b). This diagram is divided into three logical layers: a dependency layer at the bottom, the domain service layer, which will use all of these dependencies, and the application management services, such as service discovery and APIGateway services, which will rely on the domain service layer and supply all external communication.

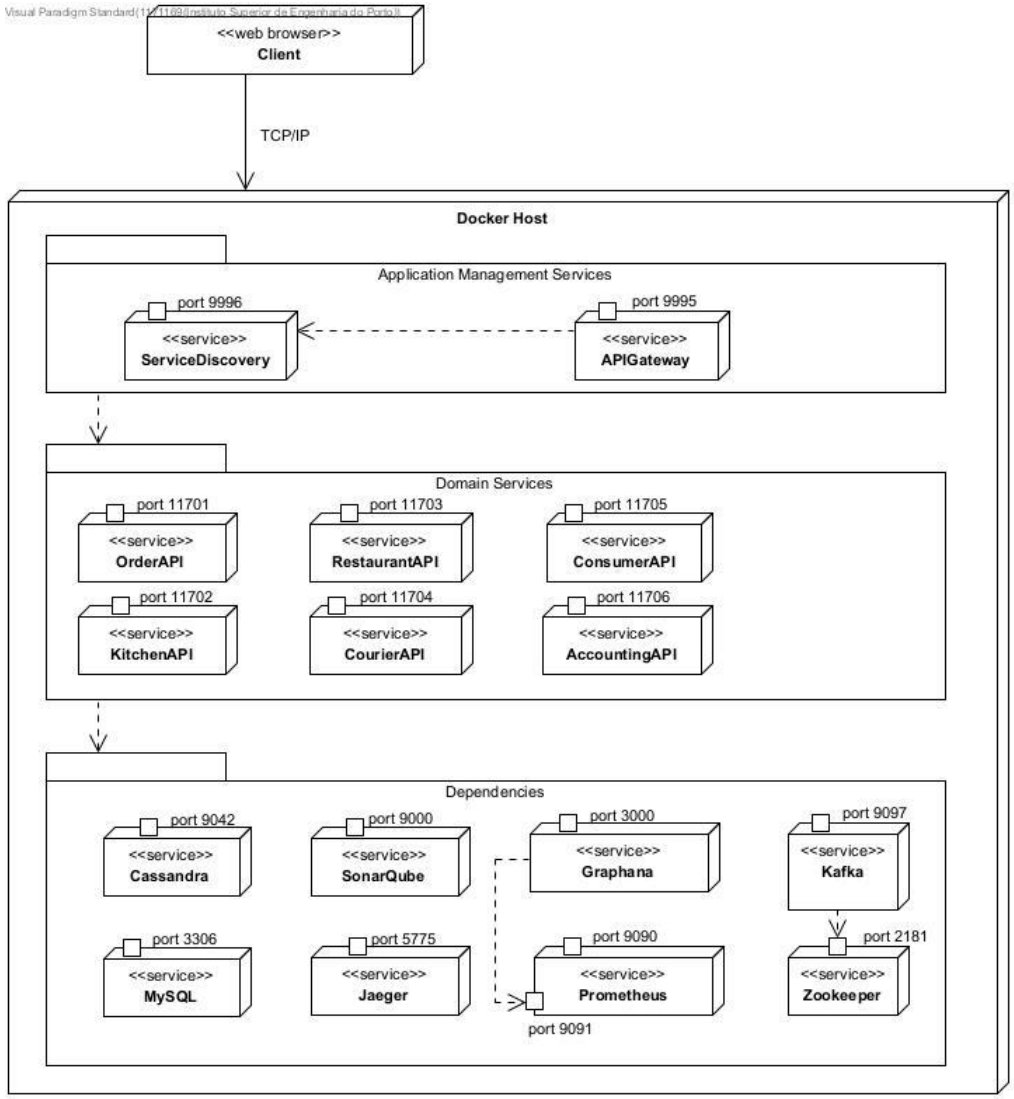

Figure 24. Docker deployment diagram

5.2 Implementation Details

Moving to the implementation details of the solution, the execution of the key non-functional requirements previously listed in Table 24 will be explained and connected with the framework as well as how it eases their implementation.

First, the service discovery and registry component of the application will be showcased, followed by how the APIGateway connects and streamlines the communication between microservices as well as the end-user. Next, the implementation of CQRS and event sourcing will be discussed, highlighting how the usage of these patterns enables the asynchronous message-based communication between services. Finally, the implementation of the SAGA pattern will be reviewed.

5.2.1 Service Discovery and Registry

Due to the nature of microservices, the number of instances of a service and its locations constantly change and adapt to the current characteristics of the system. As seen in Figure 25, the service discovery and registry mechanism allow for reliable access between services, by serving both as a registry where components can register and de-register themselves based on their statuses (Cusimano, 2022).

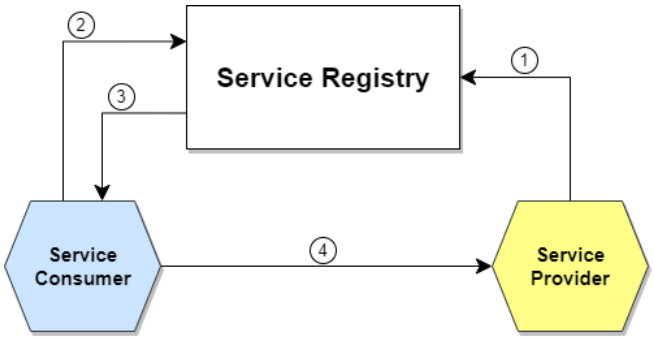

Figure 25. Service discovery and registry (Cusimano, 2022)

Lagom provides by default a static implementation of a mock service discovery module that does not allow the registration of new services or even changes to existing services. Although this solution does not provide great scalability and requires a full reboot of the application if any of the registered services move to a different internet protocol (IP) address, possibly damaging the fault tolerance of the application, it allows for a quick start-up of the development process. It is important to highlight this feature as it permits the developers to delay as much as possible some of the more technical requirements, which usually change with the growth of the project, and focus on the core of the application.

To achieve a more dynamic and robust implementation of the service discovery and registry, Lagom provides built-in integration with Akka Discovery that allows the aggregation of multiple discovery methods, such as through configuration file and domain name space (DNS) discovery (Lightbend, 2022a). The Akka management kit also adds flexibility to the program, enabling developers to pick which modules to include, as seen in Figure 26 (Schlothauer, 2019).

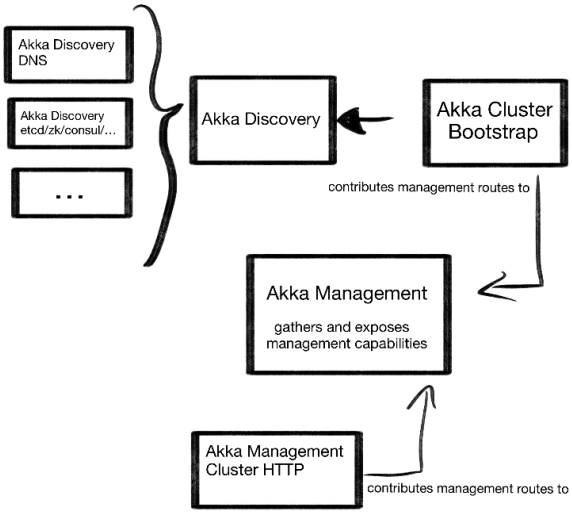

Figure 26. Lagom Akka Discovery modules (Schlothauer, 2019)

5.2.2 API Gateway

Aided by the service discovery and registry component, the API Gateway completes the front-facing layer to external clients by hiding how the backend services are partitioned in the architecture, not only by forwarding requests but by performing the orchestration or aggregation of these services. These characteristics allow for cleaner client-side code, by removing the need to invoke and know multiple services, reduce the number of requests and roundtrips needed, grant increased security by reducing the exposed elements to the outside world and increased scalability, by balancing the load between the registered instances of each service (Cusimano, 2021).

Although Lagom does not provide any production implementation for an API Gateway, generic services, such as Consul and HAProxy, can easily be integrated into the ecosystem. For this work, a Consul implementation was chosen due to its simplistic approach and built-in load balancing, as shown in Figure 27.

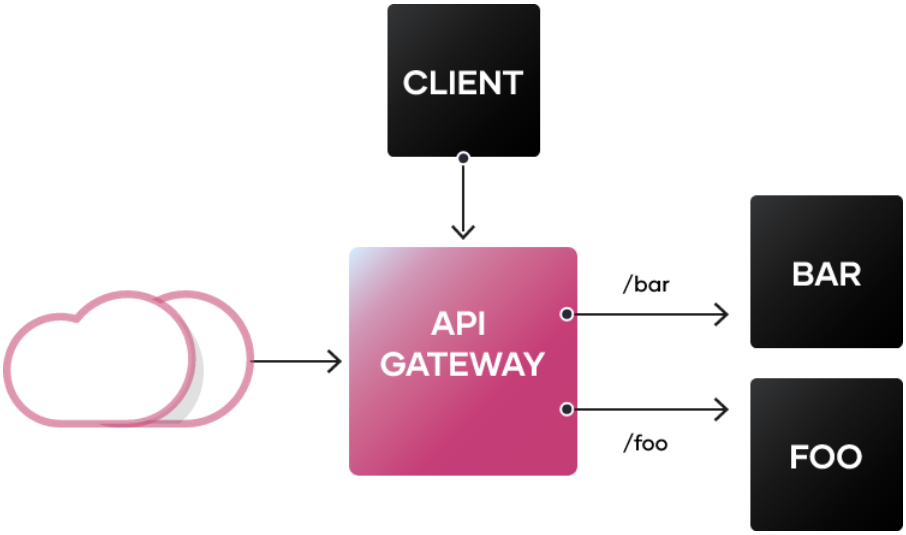

Figure 27. API Gateway flow (Consul, 2022)

5.2.3 CQRS and Event Sourcing

When designing your microservices, remember that each service should own its data and have direct access to the database. Other services should then use the API Gateway to interact with the data. There must be no sharing of databases across different services since that would result in a too-tight coupling between the services. (Lightbend, 2022d)

Lagom's persistence module encourages the usage of event sourcing and CQRS to create this decoupled design. The approach of recording all changes as domain events, which are immutable truths about what has happened, is known as event sourcing. For an Aggregate Root, such as a restaurant with a specific restaurant id, Event Sourcing is employed. Within the aggregate, the write-side is completely consistent. This makes things like preserving invariants and verifying incoming instructions simple to think about. When using this architecture, keep in mind that while the aggregate may respond to requests for a given identifier, it cannot be used to serve queries that span many aggregates. As a result, you'll need to develop a new view of the data that's suited to the service's requests. (Lightbend, 2022d)

Stored events are read and optionally acted on by event stream processors, other services, or clients. Persistent read-side processors and message broker topic subscribers are supported by Lagom. (Lightbend, 2022d)

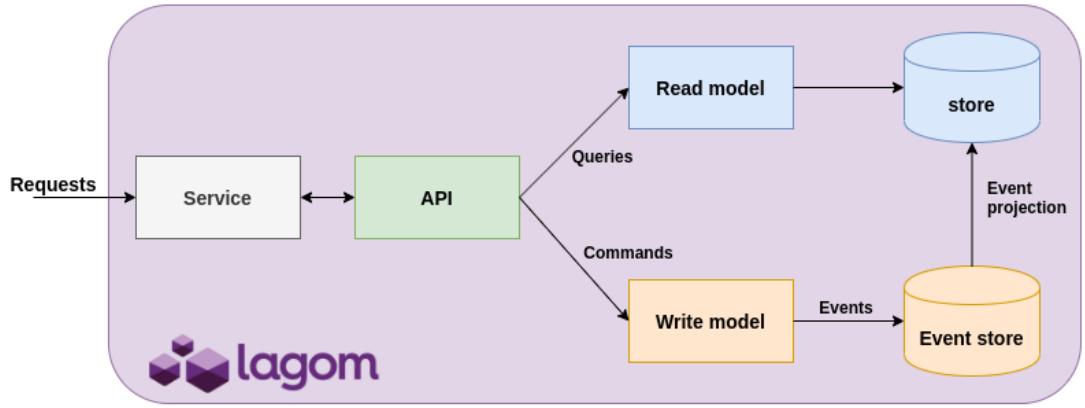

Figure 28. Lagom CQRS flow (Calus, 2020)

As previously mentioned in chapter 4.3.2, Cassandra was used for the write side of the application, to save the incoming commands into events. Using the restaurant aggregate as an example, the companion object *RestaurantState* (Ferreira, 2022c), was used to save the current status of a specific restaurant instance and all its processed commands. Figure 29 contains the result of multiple operations as Cassandra rows, containing the event, its timestamp and associated entity, allowing the replayability of the events.

	persistence_id	timestamp	event (Text)	ser_manifest	tags
1	RestaurantAggregat…	9361bec0-dc25…	{"message":{"id":…	dei.isep.ftgo.impl.messaging.event.RestaurantCreatedEvent	{'dei.isep.ftgo.impl.messaging.event.RestaurantEvent2'}
2	RestaurantAggregat…	fa337e90-ddc4…	{"message":{"id":…	dei.isep.ftgo.impl.messaging.event.RestaurantEditedEvent	{'dei.isep.ftgo.impl.messaging.event.RestaurantEvent2'}
3	RestaurantAggregat…	0409a700-ddc5…	{"restaurantId":"…	dei.isep.ftgo.impl.messaging.event.RestaurantMenuRevisedEvent	{'dei.isep.ftgo.impl.messaging.event.RestaurantEvent2'}
4	RestaurantAggregat…	04b52940-ddc5…	{"restaurantId":"…	dei.isep.ftgo.impl.messaging.event.RestaurantMenuRevisedEvent	{'dei.isep.ftgo.impl.messaging.event.RestaurantEvent2'}
5	RestaurantAggregat…	050f7e40-ddc5…	{"restaurantId":"…	dei.isep.ftgo.impl.messaging.event.RestaurantMenuRevisedEvent	{'dei.isep.ftgo.impl.messaging.event.RestaurantEvent2'}
6	RestaurantAggregat…	0623d470-ddc5…	{"restaurantId":"…	dei.isep.ftgo.impl.messaging.event.RestaurantMenuRevisedEvent	{'dei.isep.ftgo.impl.messaging.event.RestaurantEvent2'}
7	RestaurantAggregat…	e3942c72-ddc4…	{"message":{"id":…	dei.isep.ftgo.impl.messaging.event.RestaurantCreatedEvent	{'dei.isep.ftgo.impl.messaging.event.RestaurantEvent1'}

Figure 29. Example of event list stored in Cassandra

After each event is handled by the write side, it will then be sent to the *RestaurantEventProcessor* (Ferreira, 2022d), to be carried by the read side and stored in MySQL. To do this, Slick was utilized, a functional relational mapping library for Scala that accelerates the development time of table construction and entity saving and allows database queries to be written in Scala rather than SQL, taking use of Scala's static checking, compile-time safety, and compositionality. Contrary to the Cassandra column-oriented database, through SQL the overall data and its relationships can be written with a lot more structure and depth, granting the ability to better organize and filter our data, by sacrificing some of the write performance.

5.2.4 SAGA Pattern

As mentioned previously in 2.4.1, although the usage of SAGA provides more scalability and built-in failure management, it requires the tailored development of each operation and its respective compensating action. Due to this factor, Lagom does not provide an out-of-the-box implementation for the SAGA transaction, but with the usage of the underlying Akka cluster

system, through the persistent entity registry, Kafka topics and HTTP requests a verbose implementation of this pattern is accomplished.

```
1   orderController.orderTopic.subscribe
2    .withGroupId(RestaurantController.RESTAURANT_GROUP).atLeastOnce {
3       Flow[OrderMessage]
4         .map { case OrderCreatedMessage(orderUUID, orderDto) =>
5           persistentEntity
6             .entityRefFor(RestaurantState.typeKey, orderDto.restaurantId)
7             .ask[Confirmation](replyTo =>
8               validateOrder(orderDto, replyTo))(Timeout(10.seconds))
9             .map {
10              case Accepted => Done
11              case Rejected =>
12                persistentEntity
13                  .entityRefFor(OrderState.typeKey, orderUUID.toString)
14                  .ask[Confirmation](replyTo => DeleteOrderCommand(orderUUID,
    replyTo))(Timeout(10.seconds))
15                  .map(confirmation => mapConfirmation(confirmation, "Could
    not remove invalid Order"))
16            }
17          Done
18        }
19    }
```

Code 2. RestaurantService excerpt of order validation with SAGA (Ferreira, 2022e)

Code 2 showcases an implementation of the order item validation in *RestaurantService* with the SAGA pattern. First, in lines 1-2 the order topic is subscribed with the restaurant group id, to be aware of the creation of new orders but avoid duplicate consumption of the same method, in this case, two restaurant instances consuming the same order creation. It is also important to highlight that through the usage of *atLeastOnce* the message consumption will be retried in case of any failure (Ferreira, 2022e).

Then, in lines 4-8, the order validation process will be conducted, followed by an analysis of the achieved response. If the persistent entity returns the *Accepted* confirmation, the process will simply end as this validation was successful. If the response is *Rejected*, the respective rollback action is sent back to the order service, asynchronously, as shown in lines 9-16.

5.2.5 Circuit Breaker

In distributed systems, a circuit breaker is employed to maintain stability and prevent cascade failures. To avoid the failure of a single service from taking down other services, they should be used in conjunction with appropriate timeouts at the interfaces between services (Lightbend, 2022b).

Figure 30 illustrates the various states the circuit breaker can be in and the flow between them. A circuit breaker is in the closed state during normal operation, with exceptions or calls exceeding the defined timeout incrementing the failure counter and successes resetting it to

zero. The circuit breaker is triggered into an open condition when this counter hits the number of maximum failures set.

All calls fail quickly in the open state, except for the first one that indicates the circuit breaker is open. The circuit breaker reaches a half-open condition after a defined reset timeout. In this state, the first call attempted is allowed through without failing fast, but all other calls fail with the same exception as in the open state. If the call allowed through succeeds, the circuit breaker is reset back to the closed state, but otherwise, it is sent back to the open state for the configured timeout (Lightbend, 2022b).

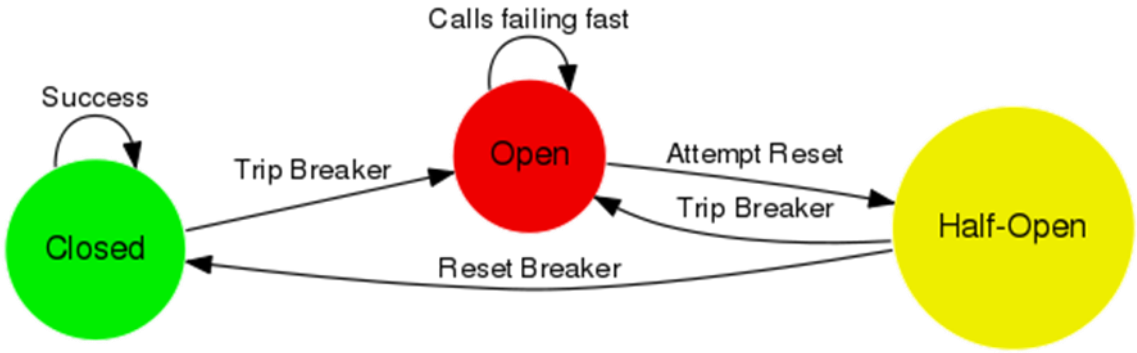

Figure 30. Circuit breaker flow between states (Lightbend, 2022b)

Circuit breakers are used by default on all service calls with Lagom service clients. On the client-side, circuit breakers are utilized and configured, but the service provider defines the granularity and configuration identifiers. By default, all calls to another service are handled by a single circuit breaker instance. To utilize a distinct circuit breaker instance for each method, set a unique circuit breaker identifier for each method. Using the same identifier on many methods may also be used to group similar methods (Lightbend, 2022b). Code 3 reveals the configuration used throughout all the microservices of the project.

```
1  lagom.circuit-breaker {
2
3    default {
4
5      enabled = on
6
7      max-failures = 10
8
9      call-timeout = 10s
10
11     reset-timeout = 15s
12
13     exception-whitelist = []
14   }
15 }
```

Code 3. *Circuit Breaker default configuration*

5.3 Testing

Three methods of testing were established to guarantee the application's correct implementation. The use of Scala for unit testing will be demonstrated first, followed by property-based testing and how it complements the former. Finally, integration testing will be presented as a method of validating the use cases flow and service integration.

5.3.1 Unit Testing

Individual units or components of the software are tested in unit testing, which is a subtype of software testing. The goal is to ensure that each unit of software code works as intended. This category is the bedrock of all software testing, as displayed in Figure 31, achieving a thorough examination of each unit of the software, allowing the detection of bugs and problems early in the software development life cycle (SDLC) (Hamilton, 2022).

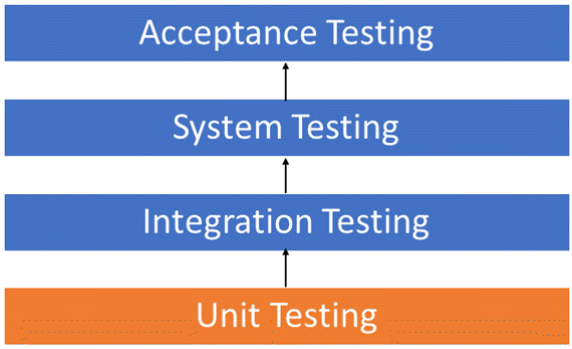

Figure 31. Categories of software testing (Hamilton, 2022)

Scala test library offers simple and verbose test clauses to ensure that all the written code is unambiguous and accessible to new developers. In Code 4 an excerpt of the *AddressMapperTest* class is displayed where it is possible to observe these characteristics:

```
16 class AddressMapperTest extends AnyFlatSpec with should.Matchers {
17
18   def AddressMapper = new AddressMapper
19
20   "Address mapper" should "map an address domain to dto" in {
21     val address = Address(
22       id = UUID.fromString("96b9cbf5-509b-442b-aef7-0c98f3904fdd"),
23       street1 = "Avenida Visconde de Barreiros",
24       street2 = "34",
25       city = "Maia",
26       state = "Porto",
27       zip = "4470-151"
28     )
29     val result = AddressMapper.domainToDto(address)
30
31     result.id.orNull should be("96b9cbf5-509b-442b-aef7-0c98f3904fdd")
32     result.street1 should be("Avenida Visconde de Barreiros")
33     result.street2 should be("34")
34     result.city should be("Maia")
```

```
35      result.state should be("Porto")
36      result.zip should be("4470-151")
37    }
38  }
```

Code 4. AddressMapperTest class excerpt

5.3.2 Property-Based Testing

Property-based testing can be seen as a subset of the unit testing category. While the aforementioned example-based test in Code 4 can be useful to ensure specific edge cases are covered and secure, it is human to fail to anticipate edge cases that may cause errors or unwanted behaviours in the application. Property-based testing solves this issue by detaching the test cases from concrete examples and replacing them with a set of higher-level properties that describe the intended behaviour (Malheiro, 2021).

To implement these properties in Scala, the ScalaCheck library was used, which provides the possibility to create generator type objects, Gen. Through natively supported generators such as ***Gen.uuid*** which generates a random UUID and ***Gen.alphaNumStr*** which generates a random alphanumeric string, it is possible to build an Address entity generator, as shown in lines 5-12 of Code 5.

```
1   class AddressMapperPropertyTest extends Properties("AddressMapper") {
2
3     def addressMapper = new AddressMapper
4
5     val genAddress: Gen[Address] = for {
6       id      <- Gen.uuid
7       street1 <- Gen.alphaNumStr
8       street2 <- Gen.alphaNumStr
9       city    <- Gen.alphaNumStr
10      state   <- Gen.alphaNumStr
11      zip     <- Gen.alphaNumStr
12    } yield Address(id, street1, street2, city, state, zip)
13
14    property("any valid domain address can be mapped to a dto") = {
15      forAll(genAddress)(address => {
16        val result = addressMapper.domainToDto(address)
17
18        result.id.orNull.equals(address.id.toString) &&
19        result.street1.equals(address.street1) &&
20        result.street2.equals(address.street2) &&
21        result.city.equals(address.city) &&
22        result.state.equals(address.state) &&
23        result.zip.equals(address.zip)
24      })
25    }
26  (…)
27  }
```

Code 5. AddressMapperPropertyTest class excerpt

With this generator, it is now possible to map a property that will evaluate if any valid domain address can be mapped to a dto, as shown in lines 14-24 of Code 5. By default, each property is tested with 100 randomly generated cases, ensuring the correct functioning of the examined behaviour.

5.3.3 Integration Testing

Next, integration tests were employed to cover and regulate not only each microservice, by examining its flows with a black-box approach, but also its integration with other microservices. To automate this process, Postman's collections and environments were used.

Figure 32 showcases the collection setup and how each of them analyses their respective services without compromising the existing application data, by removing every created object during the execution of the collection.

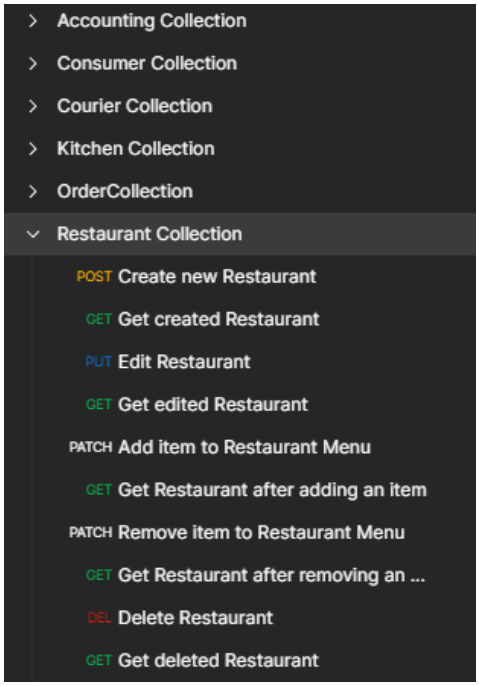

Figure 32. Restaurant management flow tests

Figure 33 depicts the environment that was utilized, which enabled the persistence of several variables across tests and simple access to all data handled, giving more in-depth testing and analysis of the application.

Figure 33. Fragment of the FTGO environment in Postman

Finally, Figure 34 shows the automatic execution of the restaurant library, which enables the possibility of a combination of these test suites with CI/CD practices.

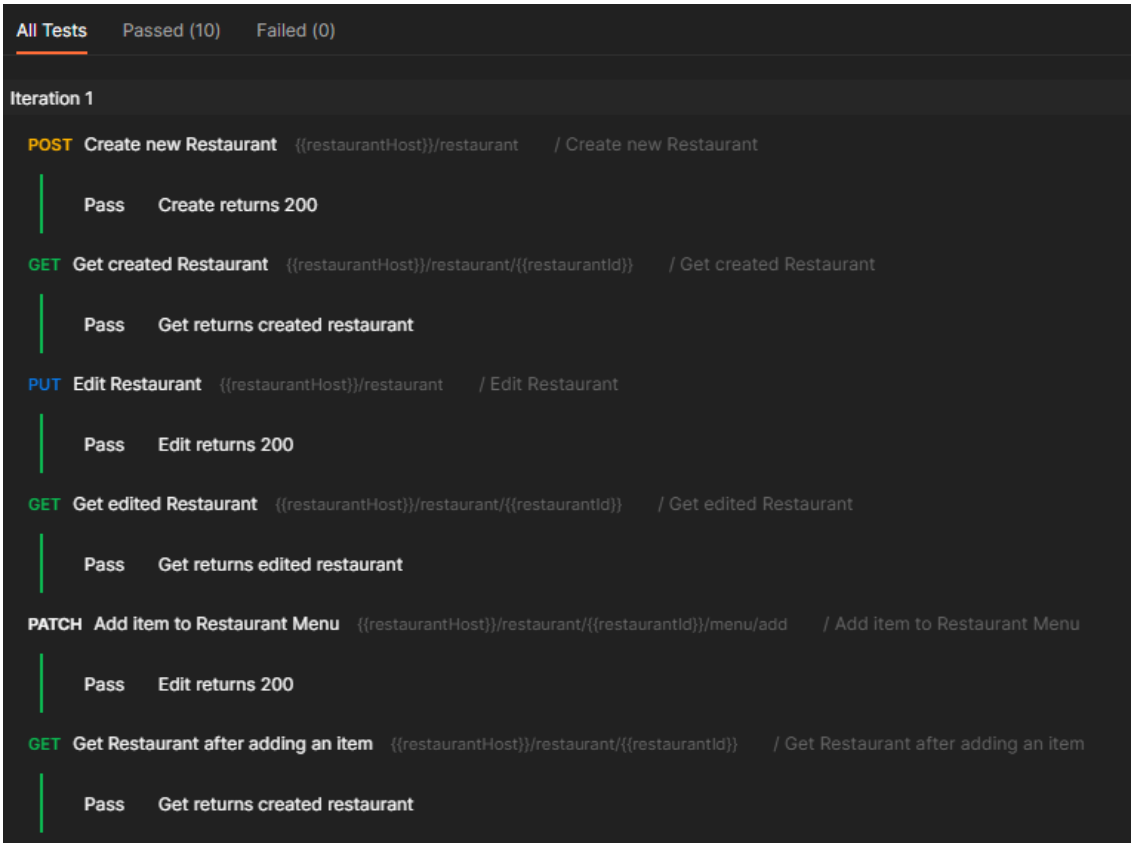

Figure 34. Example run snippet of the restaurant tests

5.3.4 Acceptance Testing

Acceptance Testing is a type of software testing in which a system is evaluated for its acceptability. The primary goal of this test is to determine whether the system complies with the business requirements and whether it is suitable for delivery (Patel, 2019). Due to the

inability to assess and test some of the requisites planned, such as the circuit breaking, tracing and the SAGA pattern, two acceptance tests were drawn.

In the first test, the order creation use case was used to support the testing of both the correct tracing of the application's communications and the healthy state of its circuit breakers. Table 26 exposes the steps and criteria used in its evaluation.

Table 26. Acceptance Test 1 – Successful creation of a new order

Acceptance Test 1 – Successful creation of a new order	
Procedure	Through the order service API, create a new order with an existing restaurant ID, the *created* order state, payment and delivery information, and a valid menu item from the chosen restaurant.
Criteria	1. Jaeger correctly shows the data flowing between order and restaurant service;
	2. Circuit breaker stays closed throughout the whole operation;
	3. Grafana correctly shows the metrics relative to the entire process.

The second test involves a failure scenario where a service becomes unreachable to test the circuit breaker in an open and half-open state. To do this, an asynchronous request between the order service to an unavailable restaurant service was made. Table 27 shows the assessment procedure and criteria.

Table 27. Acceptance Test 2 – Unavailable restaurant service

Acceptance Test 2 – Unavailable restaurant service	
Procedure	After deactivating the restaurant service, create a new order through the order service API with an existing restaurant ID, the *created* order state, payment and delivery information, and a valid menu item from the chosen restaurant. After 60 seconds, activate the restaurant service and send the same request.
Criteria	1. After failing the request 10 times, the circuit breaker switches to an open state, as shown in Grafana;
	2. Jaeger shows that the order request did not arrive at the restaurant service;
	3. After 15 seconds, the circuit breaker switches to a half-open state, as shown in Grafana;
	4. After waiting 60 seconds and sending a new request, the circuit breaker switches back to a closed state, as shown in Grafana;
	5. Jaeger shows that the order request did arrive at the restaurant service;

5.4 Metrics Setup

Concluding the solution implementation, the tool's setup will be presented, emphasizing the metrics to be collected from each tool and how Lagom, Scala and sbt aid in this process.

5.4.1 SonarQube

SonarQube is a code review tool that automatically detects bugs, vulnerabilities, and code smells in your code (SonarSource, 2006). Its integration with the project is done through the sbt Sonar-Scala plugin, which automatically registers both the global project and its subprojects for multiple views on the overall code quality, as can be seen in Figure 35. Each given metric will be further analysed in the Testing and Evaluation chapter.

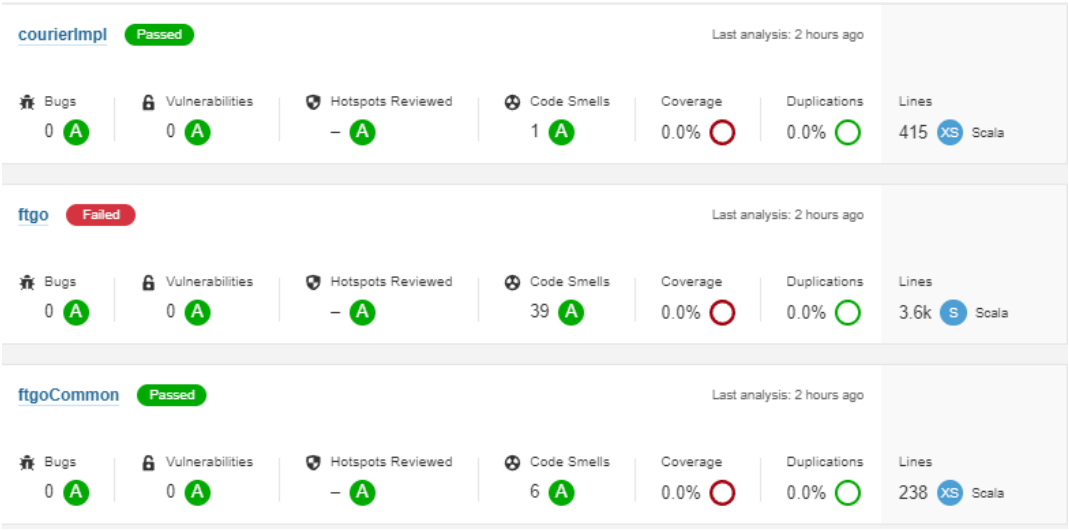

Figure 35. SonarQube project's view

5.4.2 Lightbend Telemetry (Cinnamon)

Lightbend Telemetry, also known as Cinnamon, offers information on Lightbend-based applications. As seen in Figure 36, it accomplishes this by instrumenting frameworks and toolkits such as Akka, Scala, Play, and Lagom-based applications. A Java agent does the instrumentation when the program first starts up. Lightbend Telemetry gathers data about the application in real-time depending on a specified configuration. (Lightbend, 2022f)

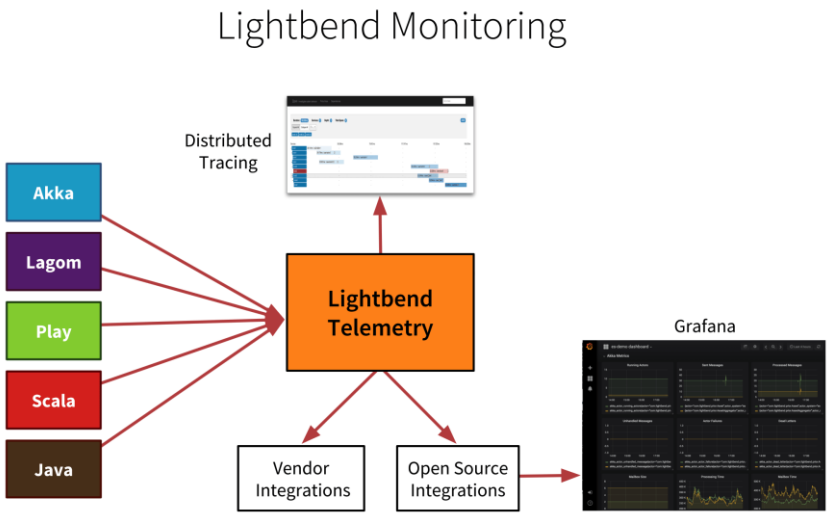

Figure 36. Lightbend Telemetry information flow (Lightbend, 2022f)

Three open-source integrations were chosen to retrieve, process and showcase the data of each microservice:

- **OpenTracing**, built-in Lagom through an extension. Used for distributed tracing;
- **Prometheus**, added to Lagom through a backend plugin. Used for event monitoring, querying, and alerting;
- **Grafana**, Web application for multi-platform open-source analytics and interactive visualization. It delivers web-based charts, graphs, and alarms, supported by Prometheus metrics.

5.4.2.1 OpenTracing

OpenTracing is a vendor-independent API that allows developers to seamlessly integrate tracing into their applications. Many tracing software vendors are supporting OpenTracing as a standardized approach to instrument distributed tracing. OpenTracing aims to provide a shared understanding of what a trace is and how to use it in our applications. This enables the simulation of processes such as application interaction, internal operations, or asynchronous jobs, depicted as spans, in a directed acyclic graph (OpenTracing, 2022).

The following state is encapsulated by each Span:

- The name of the operation;
- A start timestamp;
- A finish timestamp;
- A collection of zero or more key-value span tags with strings as keys and strings, booleans, or numeric types as values;
- A collection of zero or more span logs, each of which is a key-value map with a timestamp. The values can be of any type, but the keys must be strings;

- A SpanContext, which holds any OpenTracing-implementation-dependent state (such as trace and span ids) required to refer to a separate span across a process boundary, and baggage Items, which are key-value pairs that traverse process boundaries;
- References to zero or more causally related Spans (via the SpanContext of those related spans) (OpenTracing, 2022).

The OpenTracing data model is illustrated in Figure 37 for the flow of a new order being created in the order service, propagated to the restaurant, and finally to the kitchen service. It is feasible to trace the whole order process across all systems and show it in a uniform diagram using the OpenTracing spans (OpenTracing, 2022).

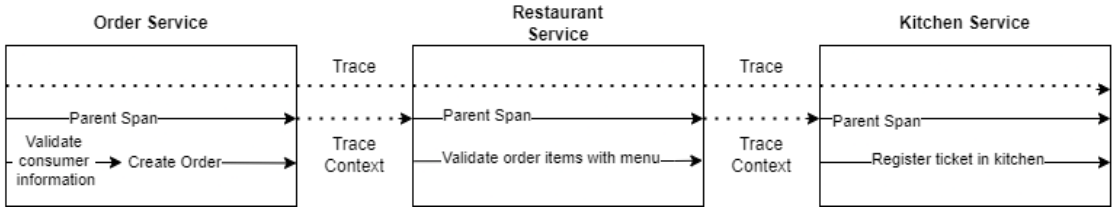

Figure 37. Order creation flow - OpenTracing diagram

For configuring tracing, or integrating tracing with Lagom, Code 6 lists all the necessary setups. In lines 5-10 the Kafka tracing is enabled through the Akka library Alpakka, for the underlying Akka operations lines 14-20 record the necessary configurations and finally lines 23-44 present the configurations to trace Lagom HTTP calls.

```
1   cinnamon {
2
3     application = "restaurant"
4     (…)
5     opentracing {
6             alpakka.kafka {
7               consumer-spans = on
8               consumer-continuations = on
9               trace-consumers = on
10              trace-producers = on
11            }
12    }
13
14    akka.actors {
15      default-by-class {
16        includes = "/user/*"
17        report-by = class
18        excludes = ["akka.http.*", "akka.stream.*"]
19      }
20      traceable = on
21    }
22
23    lagom.http {
24      servers {
25        "*:*" {
26          paths {
27            "*" {
```

```
28              metrics = on
29              traceable = on
30            }
31          }
32        }
33      }
34      clients {
35        "*:*" {
36          paths {
37            "*" {
38              metrics = on
39              traceable = on
40            }
41          }
42        }
43      }
44    }
```

Code 6. OpenTracing configuration excerpt from Restaurant Service

5.4.2.2 Jaeger

Uber Technologies has developed Jaeger, an OpenTracing compliant distributed tracing system. It analyses distributed context propagation, transaction monitoring, root cause analysis, service dependency analysis, and performance optimization to monitor and troubleshoot microservice-based distributed systems (Jaeger, 2022). Figure 38 depicts the Jaeger information flow starting from the application's jaeger agent sending an HTTP request into the Jaeger server, to be stored in its database.

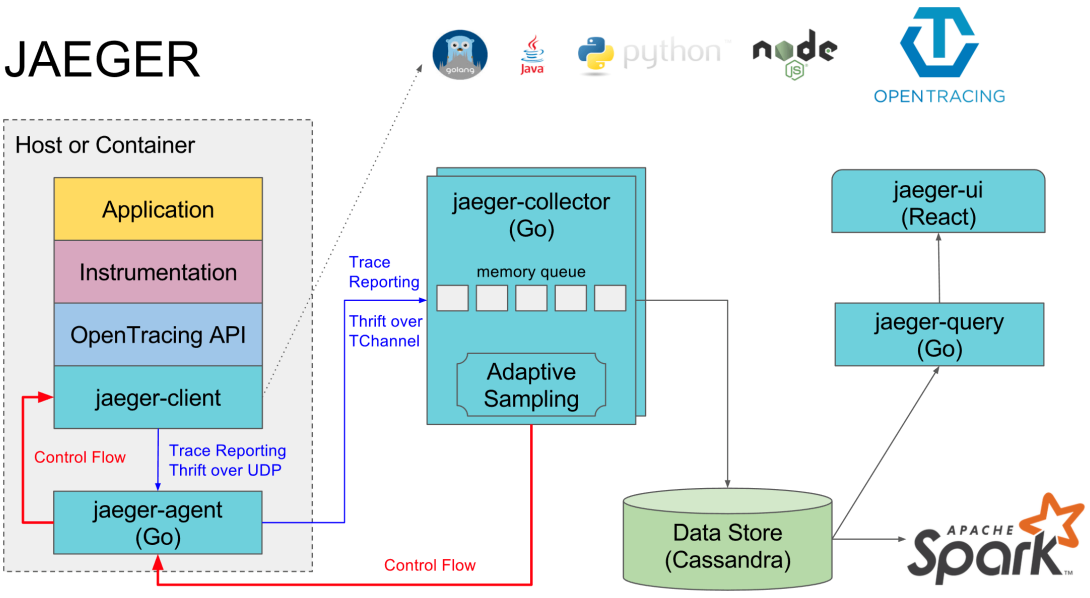

Figure 38. Jaeger information flow from the application (Gökalp, 2019).

Because Jaeger just functions as a gateway for OpenTracing, transforming and processing its data into monitoring and troubleshooting information, the majority of setups are done on the OpenTracing side. Code 7 represents the Jaeger host's required declaration.

```
1   cinnamon {
2
3     application = "restaurant"
4     (…)
5      opentracing {
6             (…)
7         jaeger {
8             host = restaurant
9         }
10     }
```

Code 7. Jaeger configuration excerpt from Restaurant Service

5.4.2.3 Prometheus

Prometheus is characterized by its multi-dimensional data format, which includes time series data identified by metric name and key/value pairs, as well as PromQL, a sophisticated query language for leveraging this dimensionality. Its simple integration through an HTTP pull method and the ability to register targets via service discovery fit in seamlessly with the Lagom architecture, as displayed in Figure 39 (Prometheus, 2022).

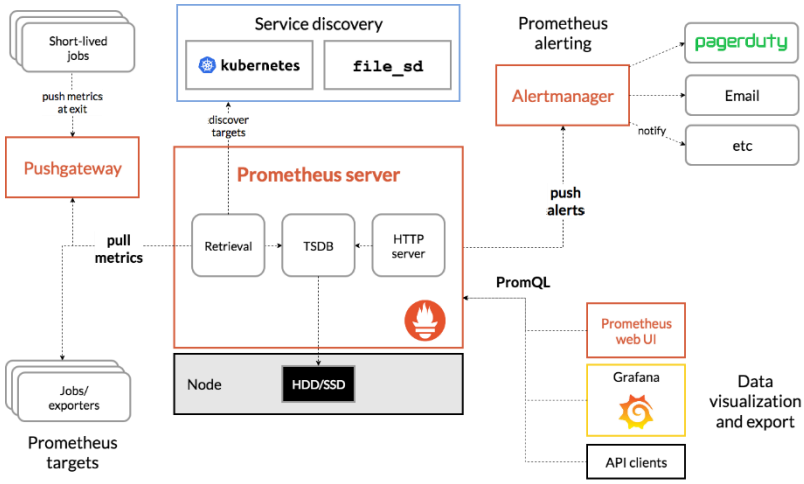

Figure 39. Prometheus flow of information (Prometheus, 2022)

Code 8 describes the necessary configuration to enable the HTTP server and start to receive the application metrics. In lines 7-13, the Prometheus server is registered with the restaurant keyword as its host, to be mapped by Docker, and in lines 14-36 the intent to record the API's metrics is declared.

```
1   cinnamon {
2
3     application = "restaurant"
4
5     chmetrics.reporters += "slf4j-reporter"
6
7     prometheus {
8       exporters += http-server
9
10      http-server {
11        host = restaurant
```

```
12      }
13    }
14    lagom.http {
15      servers {
16        "*:*" {
17          paths {
18            "*" {
19              metrics = on
20              traceable = on
21            }
22          }
23        }
24      }
25      clients {
26        "*:*" {
27          paths {
28            "*" {
29              metrics = on
30              traceable = on
31            }
32          }
33        }
34      }
35    }
36  }
```

Code 8. Prometheus configuration excerpt from Restaurant Service

5.4.2.4 Grafana

Grafana is an open-source web-based analytics and interactive visualization software. It allows data to be ingested from several sources, searched, and displayed on customisable charts for ease of interpretation, as displayed in Figure 40 (Tendonge, 2021).

Figure 40. Grafana Kubernetes capacity dashboard (Grafana, 2022)

It also is simple and quick to set up alerts to be notified of unusual behaviour and other desired events. Grafana aids in the creation of graphics from the massive volumes of performance metric data collected by Prometheus beforehand. This will allow you to draw conclusions and take action to keep your application stack healthy (Tendonge, 2021).

5.5 Summary

Concluding the implementation of the designed solution, through Lagom it was possible to implement all the planned features that support reactive microservices with little to no extra effort, due to their high level of framework abstraction. Although the framework heavily contributes to simplifying the process to implement reactiveness in microservices, throughout the implementation of the process it is clear that the framework's maturity and popularity are nowhere near some of the most used such as Spring and Quarkus, and when some implementation details fall outside of the scope of the framework’s documentation, it is extremely hard to find examples or evidence that support it, which slows down the development process.

6 Testing and Evaluation

In this chapter, the tests and evaluation methods to be applied in the future implementation will be documented. As stated in section 2.4.1 the key quality criteria identified were maintainability, scalability, performance, testability, availability, monitorability and security. To evaluate these criteria, goals, questions, and metrics (GQM) approach was conducted, following the metrics listed in 2.4.3.

6.1 Goals, Questions, Metrics

The GQM methodology is a tested technique for implementing goal-oriented metrics across a software project. It begins by identifying the goals to achieve with GQM, then clarifying the questions to answer with the data to collect, as shown in Table 28. A comprehensive picture of the environment can be created and clearly describe how the evaluation process will be done by connecting business objectives and goals to data-driven indicators (LeadingAgile, 2017).

Table 28. Goals, questions, metrics

Quality Attribute	**Goals**	**Questions**
Maintainability	The solution should have high availability.	Can the application achieve the maintainability metrics traced in 2.4.3?
Scalability	The solution should be easily scalable.	Can the application achieve the scalability metrics traced in 2.4.3?
Performance	The solution should perform under heavier workloads.	Can the application achieve the performance metrics traced in 2.4.3?
Testability	The solution testability should not be affected by implementing reactive microservices.	Can the application achieve the testability metrics traced in 2.4.3?

Quality Attribute	Goals	Questions
Availability	The application offers high availability and fault tolerance.	Can the application achieve the availability metrics traced in 2.4.3?
Monitorability	The solution provides high monitorability.	Can the application achieve the monitorability metrics traced in 2.4.3?
Security	The solution offers security regarding its API and data management.	Can the application achieve the security metrics traced in 2.4.3?

6.1.1 Maintainability

Regarding **maintainability**, SonarQube was used to automatically measure the static attributes such as lines of code (LOC), the number and time of code smells and the total of technical debt existing in each project, as previously mentioned in 5.4.1.

Starting with the **granularity** of each microservice, Table 29 lists each microservices and their composing subprojects. The results present a median of 757.33 LOC, a minimum of 521 and a maximum of 937.

Table 29. LOC per microservice

Microservice	LOC in API Project	LOC in Implementation Project	Total LOC
Accounting	103	478	581
Consumer	96	425	521
Courier	120	685	805
Kitchen	126	718	844
Order	177	679	856
Restaurant	146	791	937
Total	**768**	**3776**	**4544**

Although the LOC between the accounting and consumer microservices and the rest seem to be slightly lower than the other microservices, this follows their rationale, as they can be seen as supporting microservices to the primary flow of the application, the creation and delivery of orders. For this fact, the granularity of the microservices can be considered fairly uniform.

Following up with the **cohesion** and **coupling** of the application, these two attributes are usually seen together due to their high synergy. On average, when an application has high cohesion, that also means they have low coupling, since each microservice focuses on a single functionality, and vice-versa, when an application has low cohesion, that typically means they have high coupling due to their microservices executing multiple tasks and functionalities. Due to the ambiguity of evaluating the cohesion, these attributes will be assessed together, starting

with the number of calls to a microservice compared to the number of invocations the microservice makes to other microservices, exposed in Table 30.

Table 30. Degree of coupling of each microservice

Microservice	Number of Calls to a Microservice	Number of Invocations to Other Microservices
Accounting	0	3
Consumer	2	0
Courier	1	1
Kitchen	1	0
Order	1	2
Restaurant	2	1
Total	**7**	**7**

The number of calls to a microservice and of invocations to other microservices is equal to the planned communication protocols in 4.2.1 and the division of the calls per invocation equals 1, meaning that the software presents low coupling heightened by the fact that all communications made between microservices are asynchronous. This also hints at the fact that each microservice presents high cohesion, which can be supported by its process of division through bounded contexts in domain driven design. By establishing the dividing lines of each microservice based on the domain of the application and DDD principles, it is assured that each microservice represents a single aggregate root of the domain which can translate to its focus on a single functionality.

Moving on to **open interfaces**, by analysing the number of operations and their parameters available through exposed interfaces it is possible to evaluate the readability and granularity of a microservice's API, as shown in Table 31.

Table 31. Open interface evaluation of each microservice

Microservice	Number of Operations	Types of parameters
Accounting	7	String, AccountDTO
Consumer	5	String, ConsumerDTO
Courier	6	String, CourierDTO
Kitchen	8	String, KitchenDTO, TicketDTO
Order	5	String, OrderDTO, OrderLineItemDTO
Restaurant	7	String, RestaurantDTO, MenuItemDTO

Through Lagom's separation of the generic API code and its implementations, we can assure the uniformity of requests and their responses, which allied to the REST principles, assure streamlined readability and uniform granularity of the open interfaces.

Finally, regarding the **ease of deployment**, through the automatic generation of docker images and the usage of a docker-compose file, explained in 5.1.3, we can achieve a smooth, single-action generation of containers and deployment.

6.1.2 Scalability

Next, involving **scalability** and starting the analysis with the **usage frequency**, Jaeger was the chosen tool to track distributed tracing, as previously explained in 5.4.2.2, and evaluate the percentage of requests made to the evaluated microservice compared to all requests made throughout the whole system. A single request was made to each available operation in the exposed APIs, to discover subsequent requests made, which were recorded in Table 32.

Table 32. Usage frequency of each microservice

Microservice	Number of Operations	Number of Invocations to Other Microservices
Accounting	7	0
Consumer	5	0
Courier	6	0
Kitchen	8	2
Order	5	8
Restaurant	7	0
Total	**38**	**10**

The results show that the ratio between the requests made to the system and the total requests made to the whole system is only 8.60% which, supported by the fact that these subsequent requests are all made asynchronously, guarantees the scalability of the application.

Next, regarding the **number of synchronous requests**, Table 33 showcases the number of operations and the corresponding synchronous requests:

Table 33. Number of synchronous requests per microservice

Microservice	Total Number of Requests	Number of Synchronous Requests
Accounting	7	4
Consumer	5	2
Courier	6	3
Kitchen	8	4
Order	5	2
Restaurant	7	2
Total	**38**	**17**

The results show that the ratio between the number of synchronous requests and the total number of requests is 44.74%, which although may seem a high number, only read requests are

made synchronously, and through the usage of a separate database for read and write operations, showcased in 4.3.2, the scalability of the write side flow of the application is completely detached from its read side, removing any possible compromise from one another.

Moving on to **horizontal and vertical scalability**, it is necessary to guarantee that a microservice continues to function normally, without performance penalties, when its size or number of instances changes. The usage of event sourcing assures that, by allowing the replayability of all the events, achieving a uniform state between all instances, regardless of their status. By using the underlying Akka cluster in Lagom applications, it is also possible to easily manage all the instances of the microservices.

Finalizing with the **isolation** of the microservices, through the Lagom's division of microservices in API and Implementation projects, only the disclosed interfaces in the API project can be utilized by other microservices, and the implementation code of each microservice contains dependencies only to the API projects of microservices, as shown in the *build.sbt* file (Ferreira, 2022a).

6.1.3 Performance

Regarding **performance**, although Grafana could be used to review real-time applicational metrics such as response time, the average size of messages, queue growth and average CPU utilization, as previously explained in 5.4.2.4, it was not possible to acquire a stable test environment where the performance of the application could not be affected by third party systems, common in a personal computer.

6.1.4 Testability

In terms of **testability**, starting with the **API documentation and management**, Lagom supports the automatic generation of OpenAPI documentation, which is a language-independent interface to RESTful APIs that lets both people and machines explore and comprehend the service's capabilities without access to source code, documentation, or network traffic analysis (Apache, 2022). Through the combination of the OpenAPI plugin with the swagger annotation plugin, it is possible to also generate a hypertext markup language (HTML) page to aid in the exploration of API, as seen in Annex D - Code 11.

Moving to **test automation**, as previously shown in chapter 5.3, Scala provides resources to create and automate both unit and property-based testing, and Postman allows the generation of generic collections, which execution can also be automated. Although currently the acceptance tests designed cannot be automated, due to their highly technical scope, these tests exist to display and prove some of the technical aspects employed in this project, and their automation is not required to guarantee the test automation of the software itself.

6.1.5 Availability

Involving **availability**, to ensure **fault detection** multiple systems are in effect. First, the circuit breaker, mentioned in 5.2.5, will automatically detect a faulty instance or microservice and block its usage by switching it to an open state, protecting the system from potential data corruption or even attacks. Then, the logs and metrics gathered by Prometheus can be transformed into alarms in Grafana and, in case there is an unwanted situation, such as a critical bug or unexpected downtime of a service, an automatic notification can be sent to a developer, reducing the time to fix this unwanted situation. Through these two tools, fault detection and rectification can be ensured in each microservice.

Moving on to the **health management**, through the usage of event sourcing it is possible to ensure the microservice's capacity to cope with failure, by allowing the replayability of all the existing events, achieving a uniform state between all instances, regardless of their current situation. Exemplifying, if an instance of a microservice loses its connectivity to the network for an unknown reason but it is capable to preserve its internal state, it will simply switch to an open state in the circuit breaker until it can regain its connectivity and proceed to consume the existing events, from its initial state before the problem.

Finalizing with the **uptime percentage** and **successful execution rate**, these metrics require similar conditions to the performance metrics, such as a production-level deployment, to remove possible external factors from their evaluation, and a continuous assessment for several months and therefore cannot be reviewed with the conditions of this project.

6.1.6 Monitorability

To ensure the **monitorability** of the application, first, the **data generation** is automatically managed by Lagom's underlying system that produces several generic logs and metrics through the cinnamon plugin, explained in 5.4.2. Then, they're sent to the existing OpenTracing and Prometheus instances which will manage the **data storage**, as explained in 5.4.2.1 and 5.4.2.3 respectively. Finally, regarding **data presentation**, the Jaeger and Grafana instances will consume and process this data to provide the user with several different views and monitors, as shown in 5.4.2.2 and 5.4.2.4 respectively.

6.1.7 Security

Finally, starting the analysis of **security** with the **third-party vulnerabilities**, the sbt dependency check plugin was used to gather the whole dependency tree of the project to be analysed for known, published vulnerabilities. The plugin achieves this by using the Open Web Application Security Project (OWASP) dependency check library which already offers several integrations with other build and continuous integration systems. Figure 41 displays the results of the conducted analysis.

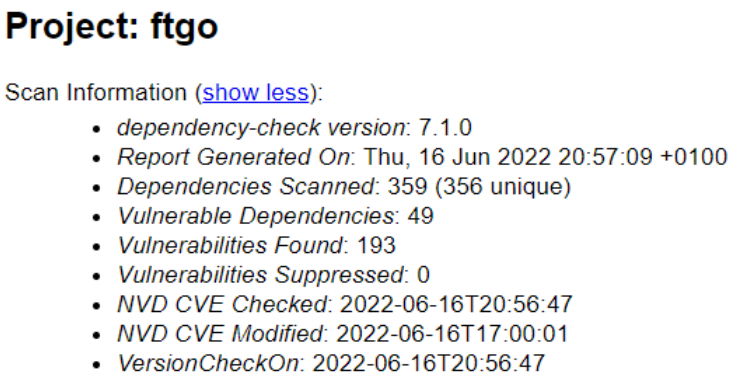

Figure 41. Analysis of dependent libraries

Although an effort was made to reduce as much as possible the number of vulnerabilities of the project, through upgrading and even downgrading some dependencies, the value of weaknesses amounted was too big for a project to be openly used without being exploited. For this reason, this metric was not considered to be guaranteed. It is important to highlight that these vulnerabilities are not related to reactivity itself but the chosen dependencies and framework.

Moving on to the **security monitor**, Grafana is flexible enough to use the existing logs of the application to generate detailed graphics and monitors at various levels to observe and even alert of anomalous behaviour and assaults at each microservice, certainly guaranteeing this metric.

Regarding **authentication and authorization**, the Lagom pac4j library provides an easily configurable authentication and authorization system. Code 9 exemplifies an authorization check on the edit consumer endpoint which only allows the consumer to update its information.

```
override def editConsumer: ServiceCall[ConsumerDto, Done] = {
  authorize(
    requireAnyRole[CommonProfile]("consumer"),
    (profile: CommonProfile) =>
      ServerServiceCall { consumer: ConsumerDto =>
        consumerService.createConsumer(consumer)
      }
  )
}
```

Code 9. Edit consumer authorization check

6.2 Summary

Summing up the evaluation made in Table 34, a total of 24 metrics were idealized and considered, with 17 metrics being deemed attained, represented by the checkmark (✓), 6 failing

to gather the essential circumstances to be assessed, symbolized by the question mark (❓), and just 1 failing to be assured, indicated by the cross (✕).

Table 34. Evaluation results

Quality Attribute	Metric	Result
Maintainability	Granularity	✓
	Cohesion	✓
	Coupling	✓
	Open interfaces	✓
	Ease of deployment	✓
Scalability	Usage frequency	✓
	Number of synchronous requests	✓
	Horizontal/vertical scalability	✓
	Isolation	✓
Performance	Response time	❓
	Average size of messages	❓
	Queue growth	❓
	Average CPU utilization	❓
Testability	API documentation and management	✓
	Test automation	✓
Availability	Uptime percentage	❓
	Successful execution rate	❓
	Fault detection	✓
	Health management	✓
Monitorability	Data generation and storage	✓
	Data presentation	✓
Security	Third-party weaknesses	✕
	Security monitor	✓
	Authentication and authorization	✓

It is possible to conclude the designed and implemented solution can completely accomplish the traced goals relative to the maintainability, scalability, testability, and monitorability and strongly answer some of the most common challenges and adversities found in microservices with flexibility and efficiency, but it still necessary to execute further testing to ensure its performance and availability and resolve the weaknesses found through its dependencies to guarantee the security of the application.

7 Conclusion

In this final chapter, the contributions of this project will be showcased and related to the initially traced objectives, followed by the detected threats to the validity of this work and its causes. Finally, the future work to be had is listed to minimize some of the analysed threats and further improve the value of this project.

7.1 Contributions

Some key objectives to be achieved with this dissertation were defined in chapter 1.3. The following contributions describe its outcomes and how they were achieved:

1. **Research on reactive microservices development experiences, challenges, and mistakes:** through the systematic literature review made in the state of the art, it was possible to highlight some of the most significant and prevalent obstacles and errors encountered while implementing and maintaining reactive microservices (see sections 2.4.1 and 2.4.2);
2. **Research on the metrics to evaluate reactive microservices:** through the same literature review, it was possible to determine the most essential and widely used criteria to assess reactive microservices (see section 2.4.3);
3. **Research on the most relevant frameworks to implement reactive microservices:** as the final question made in the literature review, it was achievable to compile a list of the most relevant frameworks used to construct reactive microservices (see section 2.4.4);
4. **Design and implementation of a project migration to reactive microservices:** after conducting the research mentioned in the previous items, a framework was chosen, as well as an existing microservice application, to be migrated. Through this process, it was possible to document the key changes to be made to a microservice application in both its design, code, and supported functionalities as well as the experience of the author and its learned lessons (see sections 4 and 5);

5. **Evaluation of the implemented solution:** finally, using the selected metrics applied to the developed solution, the project was evaluated regarding multiple quality attributes and its validity, possible improvements, and limitations were determined (see section 6).

7.2 Threats to Validity

Some difficulties have been detected in the manner this dissertation was constructed, raising concerns about the solution's applicability to other projects and situations. They are as follows:

- Due to the limited academic resources available, it was not possible to develop an isolated testing environment, and the consistency and validity of some tests cannot be assured, so they may not be used to certify the solution.
- Because this dissertation focuses on a single application and framework, the results cannot be generalized. To do this, further case studies with other systems, requirements and frameworks would be required, to emphasize its strong and weak elements in each situation.

7.3 Future Work

The study presented in this document is intended to provide answers to the goals modelled in section 1.3. Additional development would be required to make this project usable and practical in the real world, as well as to carry out the remaining planned testing. Some of the topics that might demand more examination are as follows:

1. **Remove dependency vulnerabilities.** To deploy the application publicly, it is first needed to ensure its safety by performing a meticulous analysis of the detected vulnerabilities to assess their validity and possible fix;
2. **Implement a stable testing environment.** After the application is successfully secured, a stable testing environment must be setup, preferably through a third-party cloud computing platform, such as Amazon Web Services (AWS), to ensure its stability and security;
3. **Execute the remaining tests.** Then, the application can be tested against the planned performance and availability metrics, and even expanded to public API testing to remove some possible user bias;
4. **Refactor and document the code.** Regarding the code of the application, due to its highly experimental nature, some of the existing functions and classes are currently not being utilized. All the application code can be analysed and refactored to improve both its readability and cohesion;

5. **Further empirical validation.** While this project attempted to justify the use of reactiveness in microservices, it is only one of many different sorts of implementations that this technique intends to tackle. Case studies with other systems, ideally software in different languages, frameworks, architectures, and potentially even communication protocols, would be needed to enhance confidence in the validation. It would also be noteworthy to undertake some research in production environments.

References

Apache. (2022). *OpenAPI Specification - Version 3.0.3 | Swagger*. Swagger.Io. https://swagger.io/specification/

Bastani, K. (2016). *Microservices: Cloud Native Legacy Strangler Example [Source code]*. Github.Com. https://github.com/kbastani/cloud-native-microservice-strangler-example

Bekker, A. (2018). *DynamoDB vs. Cassandra: from "no idea" to "it's a no-brainer" - KDnuggets*. ScienceSoft. https://www.kdnuggets.com/2018/08/dynamodb-vs-cassandra.html

Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2014). The Reactive Manifesto. *Reactivemanifesto.Org*, *2*(16 September 2014). https://www.reactivemanifesto.org/

Brilhante, J., Costa, R., & Maritan, T. (2017). Asynchronous Queue Based Approach for Building Reactive Microservices. *Proceedings of the 23rd Brazillian Symposium on Multimedia and the Web*, 373–380. https://doi.org/10.1145/3126858

Brondolin, R., & Santambrogio, M. D. (2020). A Black-box Monitoring Approach to Measure Microservices Runtime Performance. *ACM Transactions on Architecture and Code Optimization (TACO)*, *17*(4). https://doi.org/10.1145/3418899

Brown, S., & Betts, T. (2018). *The C4 Model for Software Architecture*. InfoQ. https://www.infoq.com/articles/C4-architecture-model/

Calus, B. (2020). *Event Sourcing and CQRS with Lagom! | by Ben Calus | Reaktika | Medium*. Medium. https://medium.com/reaktika/event-sourcing-and-cqrs-with-lagom-b7190d92c78e

Cardarelli, M., di Salle, A., Iovino, L., Malavolta, I., di Francesco, P., & Lago, P. (2019). An Extensible Data-Driven Approach for Evaluating the Quality of Microservice Architectures. *Proceedings of the ACM Symposium on Applied Computing*, *Part F1477*, 1225–1234. https://doi.org/10.1145/3297280.3297400

Cojocaru, M. D., Oprescu, A., & Uta, A. (2019). Attributes assessing the quality of microservices automatically decomposed from monolithic applications. *Proceedings - 2019 18th International Symposium on Parallel and Distributed Computing, ISPDC 2019*, 84–93. https://doi.org/10.1109/ISPDC.2019.00021

Consul. (2022). *Control access with Consul API Gateway*. Consul.Io. https://www.consul.io/use-cases/api-gateway

Cusimano, S. (2021). *API Gateway vs. Reverse Proxy | Baeldung on Computer Science*. Baeldung. https://www.baeldung.com/cs/api-gateway-vs-reverse-proxy

Cusimano, S. (2022). *Service Discovery in Microservices | Baeldung on Computer Science*. Baeldung. https://www.baeldung.com/cs/service-discovery-microservices

Dannana, S. (2020). *Function Analysis and System Technique - FAST diagram - ExtruDesign*. Extrudesign. https://extrudesign.com/function-analysis-and-system-technique-fast-diagram/

de Santana, C. J. L., de Mello Alencar, B., & Serafim Prazeres, C. v. (2019). Reactive microservices for the internet of things: A case study in Fog Computing. *Proceedings of the ACM Symposium on Applied Computing*, *Part F147772*, 1243–1251. https://doi.org/10.1145/3297280.3297402

de Toledo, S. S., Martini, A., Przybyszewska, A., & Sjoberg, D. I. K. (2019). Architectural technical debt in microservices: A case study in a large company. *Proceedings - 2019 IEEE/ACM International Conference on Technical Debt, TechDebt 2019*, 78–87. https://doi.org/10.1109/TECHDEBT.2019.00026

DInh-Tuan, H., Mora-Martinez, M., Beierle, F., & Garzon, S. R. (2020). Development Frameworks for Microservice-based Applications: Evaluation and Comparison. *PervasiveHealth: Pervasive Computing Technologies for Healthcare*, 12–20. https://doi.org/10.1145/3393822.3432339

du Plessis, S., Mendes, B., & Correia, N. (2021). *A Comparative Study of Microservices Frameworks in IoT Deployments*. 86–91. https://doi.org/10.1109/YEF-ECE52297.2021.9505049

Ferrater, J. (2017). *Tap-And-Eat-MicroServices [Source code]*. Github.Com. https://github.com/jferrater/Tap-And-Eat-MicroServices

Ferreira, J. (2022a). *Jose_Ferreira_1171169 / ftgo-reactive / build.sbt — Bitbucket*. Bitbucket.Org. https://bitbucket.org/Jose_Ferreira_1171169/ftgo-reactive/src/master/build.sbt

Ferreira, J. (2022b). *Jose_Ferreira_1171169 / ftgo-reactive / docker-compose.yml — Bitbucket*. Bitbucket.Org. https://bitbucket.org/Jose_Ferreira_1171169/ftgo-reactive/src/master/docker-compose.yml

Ferreira, J. (2022c). *Jose_Ferreira_1171169 / ftgo-reactive / restaurant-impl / src / main / scala / dei / isep / ftgo / impl / messaging / event / RestaurantState.scala — Bitbucket*. Bitbucket.Org. https://bitbucket.org/Jose_Ferreira_1171169/ftgo-reactive/src/master/restaurant-impl/src/main/scala/dei/isep/ftgo/impl/messaging/event/RestaurantState.scala

Ferreira, J. (2022d). *Jose_Ferreira_1171169 / ftgo-reactive / restaurant-impl / src / main / scala / dei / isep / ftgo / impl / repository / slick / RestaurantEventProcessor.scala — Bitbucket*. Bitbucket.Org. https://bitbucket.org/Jose_Ferreira_1171169/ftgo-

reactive/src/master/restaurant-impl/src/main/scala/dei/isep/ftgo/impl/repository/slick/RestaurantEventProcessor.scala

Ferreira, J. (2022e). *Jose_Ferreira_1171169 / ftgo-reactive / restaurant-impl / src / main / scala / dei / isep / ftgo / impl / service / RestaurantService.scala — Bitbucket*. Bitbucket.Org. https://bitbucket.org/Jose_Ferreira_1171169/ftgo-reactive/src/master/restaurant-impl/src/main/scala/dei/isep/ftgo/impl/service/RestaurantService.scala

Fowler, M. (2015). *MonolithFirst*. https://martinfowler.com/bliki/MonolithFirst.html

Fowler, M., & Lewis, J. (2014, March 25). *Microservices*. https://martinfowler.com/articles/microservices.html

Gökalp, G. (2019). *Distributed Tracing with OpenTracing API of .NET Core Applications on Kubernetes*. Gokhan-Gokalp.Com. https://www.gokhan-gokalp.com/en/distributed-tracing-with-opentracing-api-of-net-core-applications-on-kubernetes/

Gotin, M., Lösch, F., Heinrich, R., & Reussner, R. (2018). Investigating Performance Metrics for Scaling Microservices in CloudIoT-Environments. *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering*. https://doi.org/10.1145/3184407

Grafana. (2022). *Grafana® Features | Grafana Labs*. Grafana.Com. https://grafana.com/grafana/

Hamilton, T. (2022). *Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE*. Guru99. https://www.guru99.com/unit-testing-guide.html

Jaeger. (2022). *Jaeger documentation*. Jaegertracing.Io. https://www.jaegertracing.io/docs/1.11/

Klinaku, F., Bilgery, D., & Becker, S. (2019). The applicability of palladio for assessing the quality of cloud-based microservice architectures. *PervasiveHealth: Pervasive Computing Technologies for Healthcare*, *2*, 34–37. https://doi.org/10.1145/3344948.3344961

Koen, P., Ajamian, G., Boyce, S., & Clamen, A. (2014). *Fuzzy front end: Effective methods, tools, and techniques*. Academia.edu. https://www.academia.edu/4878240/1_Fuzzy_Front_End_Effective_Methods_Tools_and_Techniques

Lagom. (2016a). *Lagom - Microservices Framework*. https://www.lagomframework.com/

Lagom. (2016b). *Lagom [Source code]*. Github.Com. https://github.com/lagom/lagom

Laigner, R., Kalinowski, M., Diniz, P., Barros, L., Cassino, C., Lemos, M., Arruda, D., Lifschitz, S., & Zhou, Y. (2020). From a Monolithic Big Data System to a Microservices Event-Driven Architecture. *Proceedings - 46th Euromicro Conference on Software Engineering and*

Advanced Applications, SEAA 2020, 213–220. https://doi.org/10.1109/SEAA51224.2020.00045

LeadingAgile. (2017). *GQM Approach: Agile Metrics - LeadingAgile*. https://www.leadingagile.com/2017/05/agile-metrics-gqm-approach/

Lehmann, M., & Sandnes, F. E. (2017). A framework for evaluating continuous microservice delivery strategies. *ACM International Conference Proceeding Series*, *17*. https://doi.org/10.1145/3018896.3018961

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. A. (2021). Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. In *Information and Software Technology* (Vol. 131, p. 23). Elsevier. https://doi.org/10.1016/j.infsof.2020.106449

Lightbend. (2018). *Lightbend Podcast: When To Use Play, Lagom, Or Akka HTTP For Your Project | Lightbend*. https://www.lightbend.com/blog/lightbend-podcast-when-to-use-play-lagom-or-akka-http

Lightbend. (2022a). *Lagom - Akka Discovery Integration*. Lagomframework.Com. https://www.lagomframework.com/documentation/1.6.x/scala/AkkaDiscoveryIntegration.html

Lightbend. (2022b). *Lagom - Consuming services*. Lagomframework.Com. https://www.lagomframework.com/documentation/1.6.x/java/ServiceClients.html

Lightbend. (2022c). *Lagom - Defining a Lagom build*. Lightbend. https://www.lagomframework.com/documentation/1.6.x/java/LagomBuild.html

Lightbend. (2022d). *Lagom - Managing data persistence*. Lagomframework.Com. https://www.lagomframework.com/documentation/current/scala/ES_CQRS.html

Lightbend. (2022e). *Lagom - Production Overview*. Lagomframework.Com. https://www.lagomframework.com/documentation/1.6.x/scala/ProductionOverview.html

Lightbend. (2022f). *Lightbend Telemetry*. Developer.Lightbend. https://developer.lightbend.com/docs/telemetry/current//home.html

Malheiro, N. (2021). Property Based Testing. In *Departamento de Engenharia Informática Instituto Superior de Engenharia do Porto*. https://moodle.isep.ipp.pt/pluginfile.php/96639/mod_resource/content/7/TAP_T_FP_W6_1_PBT.pdf

Micronaut. (2018a). *Micronaut Framework*. https://micronaut.io/

Micronaut. (2018b). *Micronaut [Source code]*. Github.Com. https://github.com/micronaut-projects/micronaut-core

Moleculer. (2017a). *Moleculer - Progressive microservices framework for Node.js*. https://moleculer.services/

Moleculer. (2017b). *Moleculer [Source code]*. Github.Com. https://github.com/moleculerjs/moleculer

Nicola, S. (2018, November). *Método de Análise Hierárquica*. Instituto Superior de Engenharia Do Porto. https://moodle.isep.ipp.pt/pluginfile.php/187507/mod_resource/content/1/Análise_Valor_Aula_4_21NOV_2018_1hora_AHP.pdf

Nicola, S. (2020a). *Análise de Valor (Value Analysis) FAST and QFD Techniques*.

Nicola, S. (2020b). *TOPSIS*. Instituto Superior de Engenharia Do Porto.

Nicola, S. (2020c, November). *Análise de valor*. Instituto Superior de Engenharia Do Porto.

OpenTracing. (2022). *OpenTracing specification*. Opentracing.Io. https://opentracing.io/specification/

Patel, P. (2019). *Acceptance Testing | Software Testing - GeeksforGeeks*. Geeksforgeeks.Org. https://www.geeksforgeeks.org/acceptance-testing-software-testing/

Perkins, R. (2018). *Kenzan Million Song Library [Source code]*. Github.Com. https://github.com/TheDigitalNinja/million-song-library

Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008). Systematic Mapping Studies in Software Engineering. *12th International Conference on Evaluation and Assessment in Software Engineering, EASE 2008*. https://doi.org/10.14236/EWIC/EASE2008.8

Prometheus. (2022). *Overview | Prometheus*. Prometheus.Io. https://prometheus.io/docs/introduction/overview/

Quarkus. (2019a). *Quarkus - Getting Started With Reactive*. https://quarkus.io/guides/getting-started-reactive

Quarkus. (2019b). *Quarkus [Source code]*. Github.Com. https://github.com/quarkusio/quarkus

Rahman, M. I., Panichella, S., & Taibi, D. (2019). A curated List of project that migrated to microservices. In *Joint Proceedings of the Inforte Summer School on Software Maintenance and Evolution* (Vol-2520). CEUR-WS. https://github.com/davidetaibi/Microservices_Project_List

Rasheedh, J. A., & Saradha, S. (2021). Reactive Microservices Architecture Using a Framework of Fault Tolerance Mechanisms. *Proceedings of the 2nd International Conference on Electronics and Sustainable Communication Systems, ICESC 2021*, 146–150. https://doi.org/10.1109/ICESC51422.2021.9532893

Richardson, C. (2018a). *FTGO example application [Source code]*. Github.Com. https://github.com/microservices-patterns/ftgo-application

Richardson, C. (2018b). *Microservices patterns*. Manning Publications. https://learning.oreilly.com/library/view/microservices-patterns/9781617294549/

Richardson, C. (2021). *What are microservices?* Red Hat, Inc. https://microservices.io/

Saaty, T. L. (1980). *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. McGraw-Hill International Book Company. https://books.google.pt/books?id=Xxi7AAAAIAAJ

Santana, C., Alencar, B., & Prazeres, C. (2018). Microservices: A mapping study for internet of things solutions. *NCA 2018 - 2018 IEEE 17th International Symposium on Network Computing and Applications*. https://doi.org/10.1109/NCA.2018.8548331

sbt. (2022). *sbt Reference Manual — sbt new and Templates*. Scala-Sbt.Org. https://www.scala-sbt.org/1.x/docs/sbt-new-and-Templates.html

Schlothauer, S. (2019). *Build reactive microservices in latest Lagom release v1.5.0*. Jaxenter. https://jaxenter.com/lagom-update-1-5-158203.html

SonarSource. (2006). *SonarQube Documentation*. https://docs.sonarqube.org/latest/

Spring. (2014a). *Spring | Reactive*. https://spring.io/reactive

Spring. (2014b). *Spring Boot [Source code]*. Github.Com. https://github.com/spring-projects/spring-boot

Staveley, A. (2011). *The "4+1" View Model of Software Architecture - DZone Java*. DZone. https://dzone.com/articles/"41"-view-model-software

Stefanko, M., Chaloupka, O., & Rossi, B. (2019). The saga pattern in a reactive microservices environment. *ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies*, 483–490. https://doi.org/10.5220/0007918704830490

Tendonge, R. (2021). *What is Grafana? | MetricFire Blog*. MetricFire. https://www.metricfire.com/blog/what-is-grafana/

Tovarnitchi, V. M. (2019). Designing distributed, scalable and extensible system using reactive architectures. *Proceedings - 2019 22nd International Conference on Control Systems and Computer Science, CSCS 2019*, 484–488. https://doi.org/10.1109/CSCS.2019.00088

Vert.x. (2012a). *Eclipse Vert.x*. Vertx.Io. https://vertx.io/

Vert.x. (2012b). *Vert.x [Source code]*. Github.Com. https://github.com/eclipse-vertx/vert.x

Wolff, E. (2015). *Microservice Sample [Source code]*. Github.Com. https://github.com/ewolff/microservice

Annex A Value Analysis

Business Process & Innovation

The innovation process may be separated into three parts: the fuzzy front end (FFE), the new product development (NPD) phase, and commercialization, as shown in Figure 42. The first section, the FFE, is often considered one of the most promising areas for improving the whole innovation process. The value, volume, and likelihood of success of high-profit concepts entering product development and commercialization are increasingly being focused on the front-end activities that precede this formal and organized process (Koen et al., 2014).

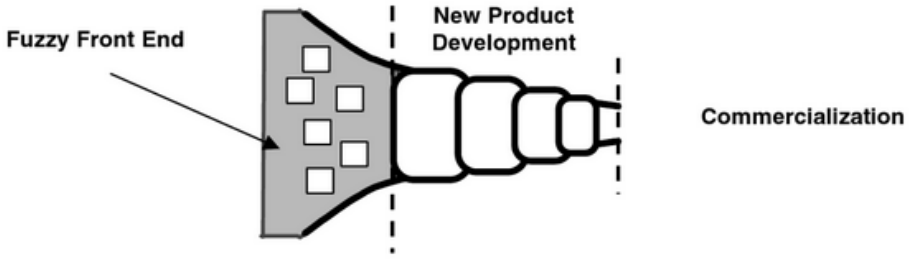

Figure 42. Innovation process phases (Koen et al., 2014)

However, it is seen as an experimental aspect of the creative process, one that is fraught with uncertainty and unexpected, or fuzziness. To eliminate this ambiguity, Koen et al. identified the crucial phases and qualities that played a vital part in the Frontend of the innovation process. As a result, the New Concept Development Model (NCD) was created, a comprehensive framework for managing the front end of innovation that identifies the most efficient approaches (Koen et al., 2014).

The NCD model shown in Figure 43 consists of three key layers (Koen et al., 2014):

- The organization's leadership, culture, and business strategy drive the five essential factors that are controllable by the company, which is known as the engine or central section.
- The FFE's five controlled activity aspects - opportunity discovery, opportunity analysis, idea generation and enrichment, idea selection, and concept definition - are determined by the inner radius region.
- Organizational capabilities, the outside environment - distribution channels, law, government policy, consumers, rivals, and the political and economic climate - and the enabling sciences that may be engaged are all contributing variables. These variables have an impact on the whole innovation process, from conception through commercialization. The business has little influence over these contributing elements.

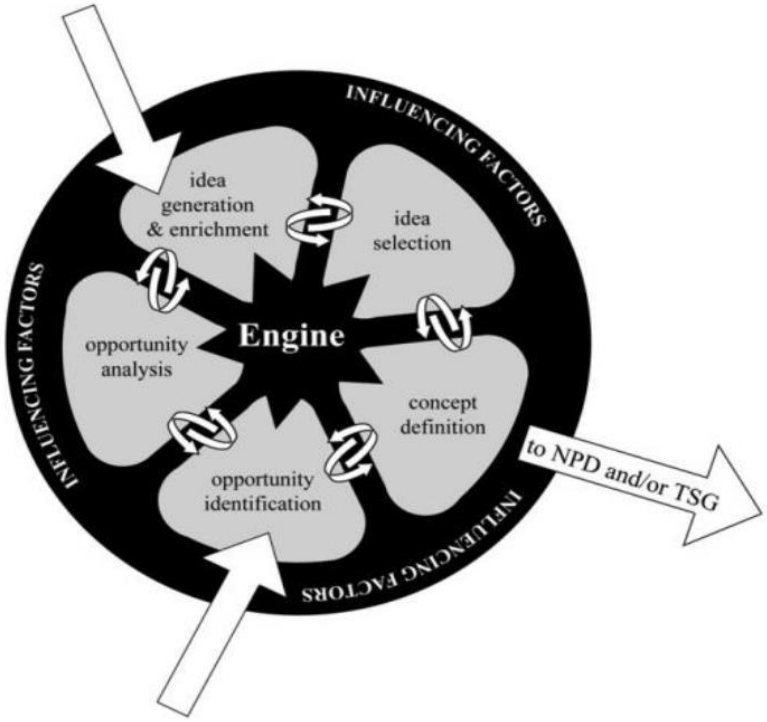

Figure 43. New concept development model (Koen et al., 2014)

This concept is shown as a circle to represent the flow of ideas across all five aspects, as well as the arrows pointing inside the opportunity. The phases of identification, idea creation and enrichment are the most common beginning points for initiatives and ideas (Koen et al., 2014). Using the NCD model, it can be deduced that this project was at the opportunity identification stage.

The opportunity to deliver this study came from the surge of popularity in areas such as the Internet of things, cloud computing and big data, where companies adopting them were looking for software characteristics such as fault tolerance, resilience, responsiveness, scalability, performance, modularity and elasticity, which are heavily linked to both microservices and reactivity, creating this correlation, as previously seen in Figure 6.

Following the model's flow, in the opportunity analysis stage, past implementations of reactive microservices were studied in section 2.4, providing a grasp of their merits, downsides, common practices and pitfalls. This output led to the idea generation and enrichment, where several frameworks were gathered for subsequent analysis and study in the Idea Selection phase, in section 3.2, where the analytic hierarchy process was used to determine the most appropriate framework to use, increasing both the productivity of the development process and the overall quality of the value given.

Finally, after obtaining a solid foundation in how to implement reactive microservices and choosing a framework to use, in the concept definition phase the development of a proof of concept will be created to demonstrate the advantages of constructing reactive microservices over other microservice archetypes and to compare their characteristics.

Function Analysis System Technique

FAST, or function analysis system technique, is a graphical interpretation of the logical connections between the functions of a project, product, process, or service based on the questions “how” and “why”, assisting in objectively thinking about the problem and establishing the scope of the project. The FAST diagram may be used to determine if and how a proposed solution meets the project's requirements, as well as to identify any unneeded, redundant, or missing processes (Nicola, 2020a), as shown in Figure 44.

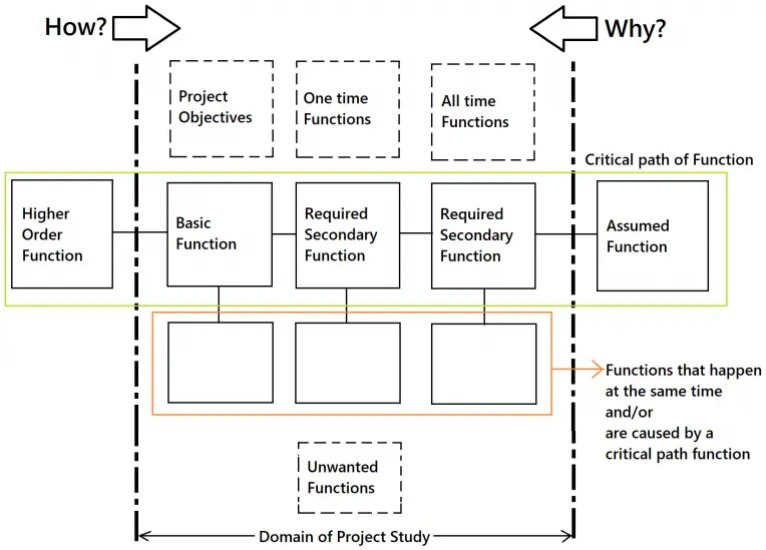

Figure 44. FAST frame diagram (Dannana, 2020)

Three key questions are addressed in a FAST diagram (Nicola, 2020a):

- How do you achieve this function?
- Why do you perform this task?
- What additional tasks must you complete while performing this function?

With these questions in mind and the frame diagram from Figure 44, the execution of this technique was realized in Figure 45.

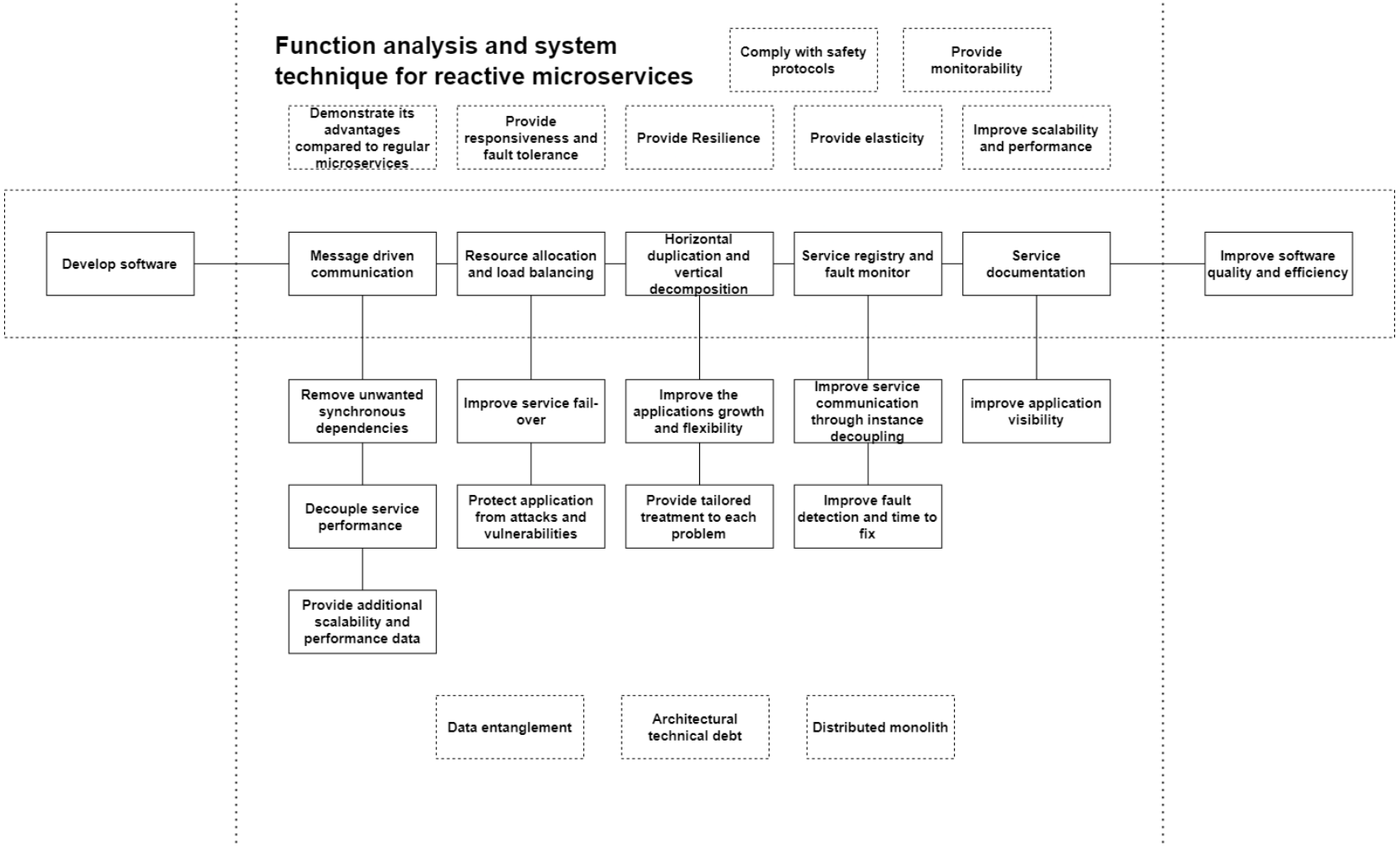

Figure 45. FAST application in reactive microservices

Annex B Domain Model

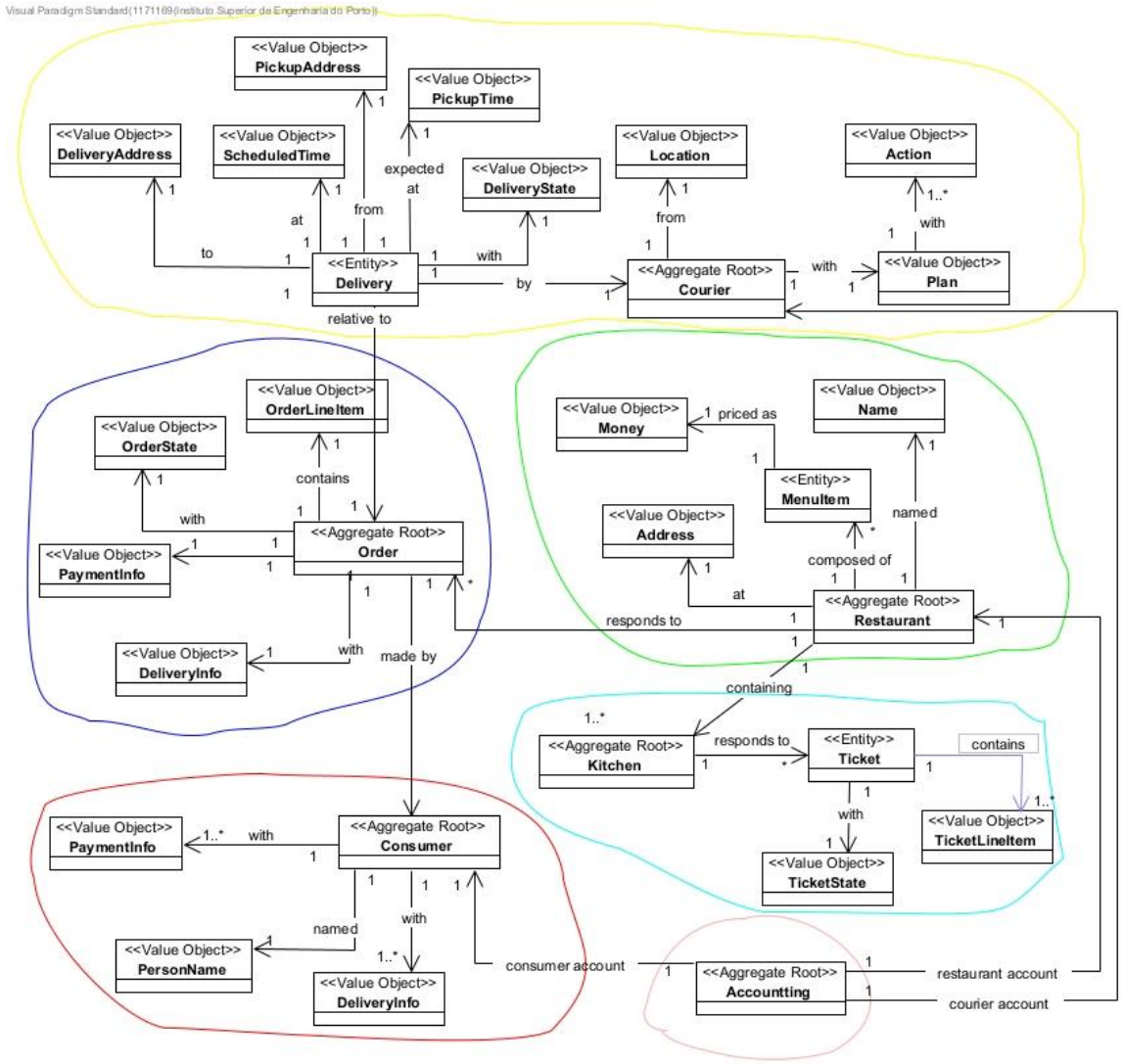

Figure 46. Domain Model

Annex C Implementation Details

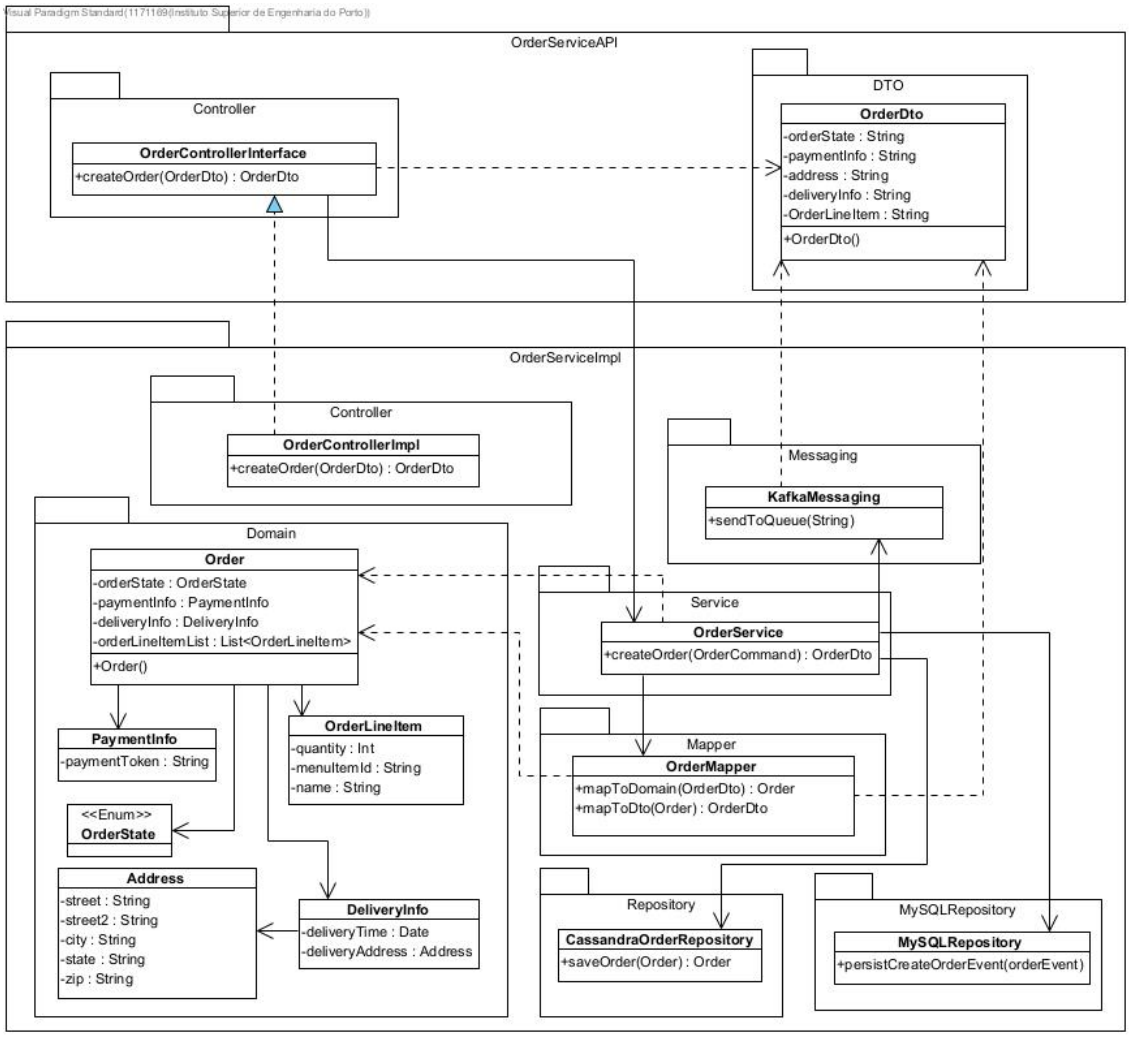

Figure 47. Class diagram code level separation of the API and Implementation projects

```
1  PS C:\Users\JosePedroFerreira\IdeaProjects> sbt new lagom/lagom-scala.g8
2  name [Hello World]: ftgo
3  organization [com.example]: dei.isep
4  version [1.0-SNAPSHOT]: 1.0-SNAPSHOT
5  package [dei.isep.ftgo]: dei.isep.ftgo
```

Code 10. Creation of the project skeleton through sbt

Annex D Testing and Evaluation Details

```
{
  "openapi": "3.0.1",
  "info": {
    "title": "Order API",
    "version": "1.0.0"
  },
  "paths": {
      "post": {
        "summary": "Create new order",
        "operationId": "createOrder",
        "requestBody": {
          "content": {
            "application/json": {
              "schema": {
                "$ref": "#/components/schemas/OrderDto"
              }
            }
          }
        },
        "responses": {
          "200": { "description": "Order creation command sent." },
          "400": { "description": "Malformed order." }
        }
      }
    },
    (…)
    "/order/{orderId}": {
      "get": {
        "summary": "get order by id",
        "operationId": "getOrderById",
        "responses": {
          "200": {
            "description": "",
            "content": {
              "application/json": {
                "schema": {
                  "$ref": "#/components/schemas/OrderDto"
                }
              }
            }
          },
          "400": { "description": "Malformed ID." },
          "404": { "description": "Order not found." }
        }
      }
} (…)
```

Code 11. Excerpt of the OpenAPI endpoint response, available in the Order microservice