INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Gestão de dados em arquiteturas de microsserviços

LUÍS MIGUEL SOUSA FERREIRA
outubro de 2020

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Data Management in Microservice Architectures

Luís Miguel Sousa Ferreira

Dissertation to obtain the Master's degree in Informatics Engineering, Specialization in Software Engineering

Supervisor: Isabel de Fátima Silva Azevedo

Porto, October 2020

Dedicated to my parents, sisters, family and friends.

Abstract

Currently, the information explosion significantly driven by the widespread use of services has become a challenge. The systems can now be redistributed or subdivided into small services, called microservices. However, this process makes the system extra complex, because although on the one hand, it simplifies the installation and management of the software, on the other hand, it increases the management difficulty due to the high number of interactions. In a scalable and large-scale system, it is difficult to assess which components have the greatest influence on the waiting time measured by users, so effective data management is fundamental. However, these components cannot be analyzed separately and the use of source code makes it difficult to correlate and identify the source of bugs or incorrect data. Therefore, it is essential to have a well-designed solution that matches the needs of the system.

The study of an internationally renowned company was carried out to achieve the best solution to data management. An analysis of bibliographic references was also carried out, thus producing a complete approach to the theme.

Through qualitative research and company analysis, this work aims to address the concepts of data management areas in microservices architectures, listing their characteristics, culminating with the presentation of a set of recommendations and solutions for some challenges encountered in this management and with the development of a data consistency monitoring tool. This case study focused on microservice architectures whose complexity of their systems and the volume of data are very high, reaching hundreds of thousands of transactions per second.

Keywords: Software, Microservices, Architecture, Data Management

Resumo

Atualmente, a explosão da informação significativamente impulsionado pelo amplo uso de serviços tornou-se um desafio. Os sistemas agora podem ser redistribuídos ou subdivididos em pequenos serviços, chamados microsserviços. Porém, este processo torna o sistema extremamente complexo, pois embora por um lado simplifique a instalação e gestão do software, por outro lado, aumenta a dificuldade de gestão devido ao elevado número de interações. Num sistema escalável e complexo, é difícil avaliar quais os componentes que têm maior influência no tempo de espera medido pelos utilizadores, portanto, a gestão eficaz dos dados é fundamental. No entanto, esses componentes não podem ser analisados separadamente e o uso do código-fonte torna difícil correlacionar e identificar a fonte de bugs ou dados incorretos. Portanto, é essencial ter uma solução bem projetada que corresponda às necessidades do sistema.

Foi realizado o estudo de uma empresa de renome internacional de modo a encontrar a melhor solução para a gestão de dados. Foi também realizada uma análise de referências bibliográficas, produzindo assim uma abordagem completa do tema.

Através de pesquisa qualitativa e análise da empresa, este trabalho visa abordar os conceitos da área de gestão de dados em arquiteturas de microsserviços, apresentar as suas características, culminando com a apresentação de um conjunto de recomendações e soluções para alguns desafios encontrados nessa gestão e com o desenvolvimento de uma ferramenta de monitorização de consistência de dados. Este estudo de caso concentrou-se em arquiteturas de microsserviços cuja complexidade dos seus sistemas e o volume de dados são muito elevados, podendo atingir centenas de milhares de transações por segundo.

Palavras-chave: *Software*, Microsserviços, Arquitetura, Gestão de dados

Acknowledgments

First of all, I would like to thank my whole family, especially my parents and sisters, who helped me stay motivated, helped me in the most difficult moments, and who provided me with everything I needed to successfully achieve all the goals, not only professional but also personal.

Also thank my girlfriend for the support, understanding, and motivation that she gave me during this year when I wrote the thesis. I apologize for the several weeks that I was unable to be with her during the process of developing the thesis. She has always been by my side, and I know how proud she is of me.

I also have to thank my thesis advisor Isabel Azevedo of ISEP, who was always available to help me with any doubts that have arisen, who have responded very quickly to all the emails I sent (even those almost at the end of the evening), and that scheduled several meetings throughout the process to help me. Her help was indispensable for me to be able to build this document and achieve the proposed objectives.

Finally, I would like to thank all the professionals of the company studied who were always available to help me and provided extremely important information for this document.

My sincere thanks to all of you!

Table of Contents

List of Figures

List of Tables

List of Acronyms

AHP	Analytic Hierarchy Process
API	Application Programming Interface
CAGR	Compound annual growth rate
CAP	Consistency, Availability, Partition Tolerant
CQRS	Command-Query Resource Segregation
DDD	Domain-Driven Design
DSRM	Design Science Research Methodology
EDA	Event-Driven Architecture
GDPR	General Data Protection Regulation
HIPAA	Health Insurance Portability and Accountability Act
HTTP	Hypertext Transfer Protocol
IS	Information systems
MDD	Model-Driven Development
NCD	New Concept Development
REST	Representational State Transfer
SOA	Service-Oriented Architecture
SQL	Structured Query Language
XML	Extensible Markup Language

1 Introduction

In this first chapter, the motivations and relevance of the study carried out focusing on the underlying theme will be highlighted. It aims to present the context and an explanation related to the organization of the respective work.

1.1 Context

The designation “Microservice Architecture” has appeared in recent years as a specific way of developing compartmentalized software for services and with independent implementation. Although there is no exact definition of this kind of architecture, there are certain similar characteristics about the organization, business capacity, automated deployment, intelligence in the terminals, and decentralized control of languages and data. As for software architecture, it emerges as a subdiscipline of software engineering. Architecture presents itself in a general way as a division of a whole in different parts with specific relations between them. This division enhances the ability of groups of people to be able to work collaboratively to solve a complete and more complex problem than any other way that these people could do it individually (Bass, Clements, & Kazman, 2012).

In monolithic applications, all services are developed on a single code base shared between several developers, when those developers want to add or change services they must ensure that all other services continue to work. The complexity increases as more services are added, limiting companies' ability to innovate with new versions and features. Additionally, when new versions of applications are deployed in production, the full set of services is restarted, providing bad experiences for users. This type of architecture also represents a single point of failure, because if the application fails, the entire set of services will fail and compromise the

system as a whole. However, it has other characteristics that become advantageous such as its ease of execution, greater ease of development (mainly due to the uniformity of the technology stack between all layers), the simplicity of the tests and requires a smaller development team to maintain the application.

Concerning systems with microservice architectures, the application is divided into subsectors or services with the possibility of being implemented independently (see Figure 1). Each development team carries out its approach in its microservice and autonomously to the other development teams. This context allows the integration of programmers in the development teams, since the service code base is simplified, ceasing to present itself as a system with only one point of failure, with each microservice deploying independently of the others. As for the disadvantages, the fact that there is a duplication of code common to several modules stands out, being necessary to deploy each service, which may not be a disadvantage. They usually use an HTTP API as a means of communication with each other.

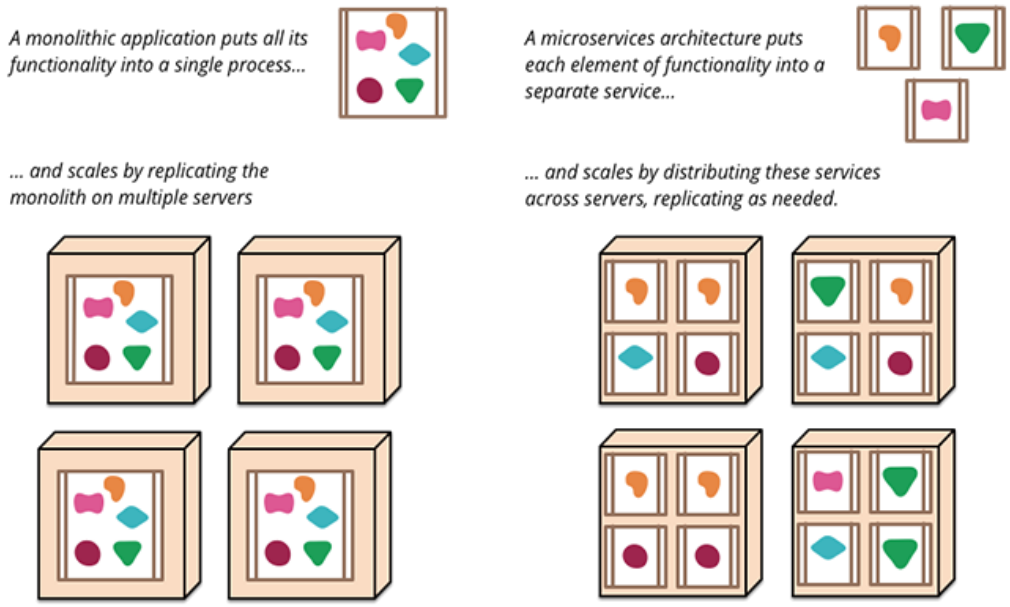

Figure 1 – Distribution in monolithic architectures vs microservices (Fowler, 2015)

Just as microservices decentralize decisions about conceptual models, they also decentralize decisions about data storage. While monolithic applications prefer a single logical database for data persistence, in microservices it is common for each service to manage only its database. They can use different database technologies to best suit the needs - an approach called Polyglot Persistence.

Figure 2 shows the difference between monolithic architectures and microservices about the distribution of services as well as the databases.

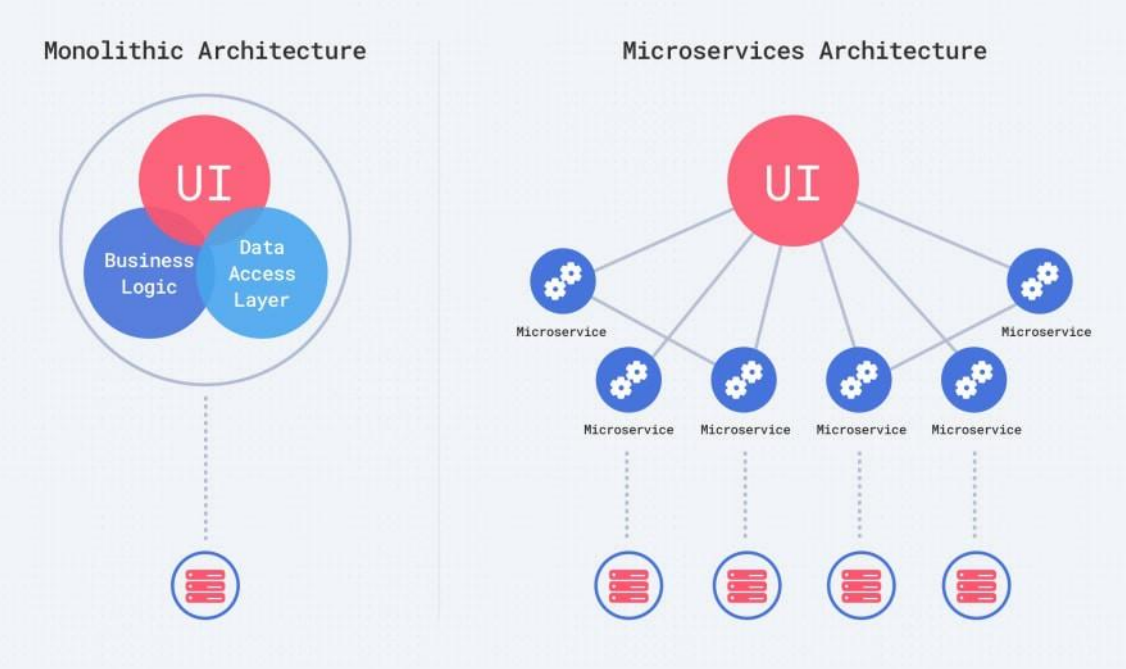

Figure 2 – Monolithic architecture vs Microservice architecture (Barashkov, 2019)

Microservice architecture is the current trend capable of overcoming the low scalability and flexibility of the classic style of monolithic client-server applications, in which modifications to small sections of service may require the construction and deployment of a completely new version.

On the other hand, companies' systems are increasingly complex as they need to respond to complex business requirements and need to be able to manage large volumes of data in the most effective way.

The architecture of systems that operate on a large scale is often highly application-specific - there is no one-size-fits-all, generic, scalable architecture. The problem can be the volume of readings, writes, data to be stored, the complexity of the data, the response time requirements, the access patterns, or even a mixture of all this (Kleppmann, 2017).

In this context, microservices reinforce the engineering of intelligent and innovative systems to support the management of large volumes of data more flexibly and appropriately. These two areas are then expected to find an intersection and move forward together: on the one hand,

an enormous amount of data to be processed; on the other, the need for adequate, flexible, and independent microservices to manage them. Both areas are closely related, also naturally distributed.

Before the era of big data, Structured Query Language (SQL) data stores were the perfect standard for centralized systems. However, big data requires a horizontal expansion for scalability purposes, which contrasts sharply with SQL's inability to scale well, meaning to reduce or stretch resources as needed. Huge volume requires scalability, while variety requires the manipulation of different types of data structures. The speed of data access requires low latency and high throughput. Also, these types have changed in different application systems since the current time. Data sources must provide all the security necessary to recover data reliably for business processes.

Some components of the ACID properties (Atomicity, Consistency, Isolation, Durability) guaranteed by the SQL databases need to be compromised to obtain better availability and scalability in the distributed system.

1.2 Problem

Since each microservice has its data store, there is no single source of truth, and maintaining data consistency may become problematic. Some prototypes have avoided this problem by implementing microservices with a shared database, but such solutions are generally seen as being out of pattern with the share-nothing microservices architecture (Baškarada, Nguyen, & Koronios, 2018).

Implementing complex business processes through microservices may require the addition of a message broker. Such asynchronous messaging may lead to inconsistencies in individual data stores. There was little understanding regarding how data consistency may best be ensured in a microservice architecture, given that the same data may be stored in multiple databases in a range of formats (Baškarada et al., 2018). This can cause obvious problems with data updates. While it was acknowledged that using a centralized data store may alleviate some of these problems, as mentioned, there was also general agreement that this would introduce an unacceptable level of centralization that is counter to the spirit of decentralized microservice architecture (Baškarada et al., 2018).

With the era of digitalization, the production of data per minute has been increasing. This large

amount of data produced per minute makes it difficult to store, manage, use, and analyze.

The problem description will be presented in a more complete and detailed way in chapter 5.

1.3 Objectives

Based on the problem identified for the theme on which this document is concerned, it was possible to identify the general shape of the objectives of this work.

In general, the proposed objectives are the identification of the challenges and solutions for data management in microservices architectures whose data volume is very high (volume defined in detail in section 6.1) and also the implementation of a tool related to data management in this type of architecture.

In chapter 5, the definition of the objectives will be presented in a more refined way, after the development of topics and concerns closely related to data management and microservices.

1.4 Motivation

Large companies worldwide that adopted microservices have gone through many challenges. They need strategies to manage their data in the best and most effective way.

This work focused on the case study of one of these companies that has a microservice architecture with more than 500 services. It is a company founded in Portugal in 2018 and has more than 1.000 workers. Recently in 2018, this company entered the New York Stock Exchange.

A case study will serve as a tool to understand the form and reasons that led to some specific decisions. The analysis and solutions found and presented can be valuable for the scientific community and professionals in the field of informatics engineering. Sharing the achieved trade-offs and solutions can be of great value and bring numerous benefits to all stakeholders and it is a practice increasingly adopted by companies such as Netflix and Amazon.

Regarding the data consistency analysis tool, it is of great importance for the identification of possible data inconsistencies and detect problems more quickly and efficiently, thus allowing users to correct the data. Moreover, it can alert for the need of changes. In this company, whenever this type of data inconsistency exists, incidents/bugs are created. These incidents/bugs are subsequently addressed to the respective teams to be corrected. This tool

served as a proof of concept in this company and should be adapted later to allow its daily use, thus allowing to reduce that number of incidents.

1.5 Research methodology

Design Science Research Methodology (DSRM) is a well-known and accepted framework to implement design science research in Information systems (IS). This methodology was used on the development of this work and includes six steps (Peffers et al. 2007):

1. **Problem identification and motivation:** in this first step the problem is defined, its importance is presented and the value of the solution is justified. This justification allows to accept the results and to know the researcher's understanding of the related problem. The detailed problem identification and motivation is presented in chapter 5 and its value analysis and justification in chapter 2;
2. **Define the objectives:** the objectives can be quantitative or qualitative and require knowledge of the state of the problem and current solutions and their efficacy. Thus, the detailed objectives are present in chapter 5 and the assessment of the state of the problem and the current solutions present in chapter 4;
3. **Design and development:** includes the identification of functionality and architecture, taking into account several quality attributes. Then, is developed the designed artifact.
4. **Demonstration:** demonstrate the efficacy of the artifact to solve the problem. This could involve its use in experimentation, simulation, a case study, proof, or other appropriate activity. In the solution presented in this document, its value analysis can provide some clarifications on how it can solve the identified problem;
5. **Evaluation:** observe and measure how well the artifact supports a solution to the problem. This activity involves comparing the objectives of a solution to actual observed results from the use of the artifact in the demonstration. It requires knowledge of relevant metrics and analysis techniques. At the end of this activity, the researchers can decide whether to iterate back to step 3 to try to improve the effectiveness of the artifact or to continue to communicate and leave further improvement to subsequent projects.
6. **Communication:** communicate the problem and its importance, the artifact, its utility

and novelty, the rigor of its design, and its effectiveness to other researchers and audiences. This phase will be composed by the presentation of the work developed to an evaluation committee.

Another research approach adopted is case study. Case study is a suitable research methodology for software engineering research since it studies contemporary phenomena in their natural context (Runeson & Höst, 2009). The area of software engineering involves the development, operation, and maintenance of software and related artifacts. Software engineering research focuses on investigating how this development, execution, and support is carried out, under different conditions, by software engineers and other stakeholders. That is, software engineering is a multidisciplinary area involving areas where case studies usually are conducted. Many research questions in software engineering are suitable for case study research (Runeson & Höst, 2009). This case study took five steps into account:

1. Case study design: objectives were defined, and the case study was delineated;
2. Preparation for data collection: procedures and protocols for data collection were defined. Within these procedures, to highlight the meetings with the professionals of the company in question and questionnaires;
3. Collecting evidence: execution with data collection on the studied case;
4. Analysis of the collected data and validation of the results obtained within the company;
5. Reporting.

1.6 Document Structure

The document is structured in seven chapters, which are followed by references and appendix sections.

1. **Introduction:** the first chapter presents the context of this document and its structure, as well as the research methodology;
2. **Value Analysis:** it presents the entire business and innovation process, followed by the value proposition that indicates the set of products (tangible) or services (intangible) that create value for a specific segment of customers. The current market value is also described and the Canvas Business Model is presented;

3. **Background:** it presents several themes that are crucial for readers to understand the following chapters;
4. **State of the Art:** presents the current state of the literature regarding data management in microservices and analyzed and compared the current approaches to data monitoring and alarms, as well as data recovery in microservices;
5. **Problem Analysis:** describes the problem, the intended objectives and what is the approach and development process;
6. **Case Study:** this chapter presents the proposed solution, in the context of the company under study, regarding the management of large volumes of data in microservices architectures. This solution includes presenting the most significant challenges encountered as well as the suggested strategies to solve them;
7. **Data Consistency Monitoring Tool:** presents the analysis, design and implementation of the data consistency monitoring tool;
8. **Evaluation:** evaluates the quality of the final work using an industry questionnaire answered by experienced professionals of the field and hypothesis testing;
9. **Conclusions:** describes the main conclusions of this work. It also presents the objectives achieved, the difficulties encountered and also the limitations and future work.

2 Value Analysis

Value analysis is a scientific process that helps in managing decision making. It comprises a group of techniques aimed at systematically identifying unnecessary costs in a product or service and eliminating them efficiently, without jeopardizing attributes such as quality and efficiency.

It seeks to achieve the maximum possible value through a continuous process of planned action and aims to reduce costs in terms of value.

Although initially the group of techniques, aimed at the systematic identification of unnecessary costs and the exploration of performance improvement channels, was used mainly in the field of engineering from which the name of value engineering arose, this group is now used in several areas such as marketing, purchasing, financing, etc. Given the wide applicability of this technique, it is now called value analysis.

The analysis presented will use the New Concept Development (NCD) model (Koen et al., 2001) to understand what the customers' needs are, as well as the ideas generated and selected to arrive at the final business concept. Finally, the CANVAS model will also be presented (Osterwalder et al., 2010) to demonstrate easily the general concept of the business and is also used the AHP method that allows the use of qualitative and quantitative criteria in the process of evaluating ideas and development, facilitating the understanding and evaluation of the project.

2.1 New Concept Development Model

The New Concept Development (NCD) model is an iterative model that provides a universal language and vision that allows clarifying the vision and culture of an organization/project. This model is defined by three components:

- Influencing factors: there are several influencing factors such as organizational capacities, business strategy, the outside world (distribution channels, customers, competitors, and government), and science and technology. These factors influence the innovation process (Koen et al., 2001).

• Key elements: there are 5 key elements, opportunity identification, opportunity analysis, generation and enrichment of Ideas, selection of ideas, and definition of the concept.

• Engine: responsible for putting into motion/execution the five key elements, under the leadership and culture of the organization (Koen et al., 2001).

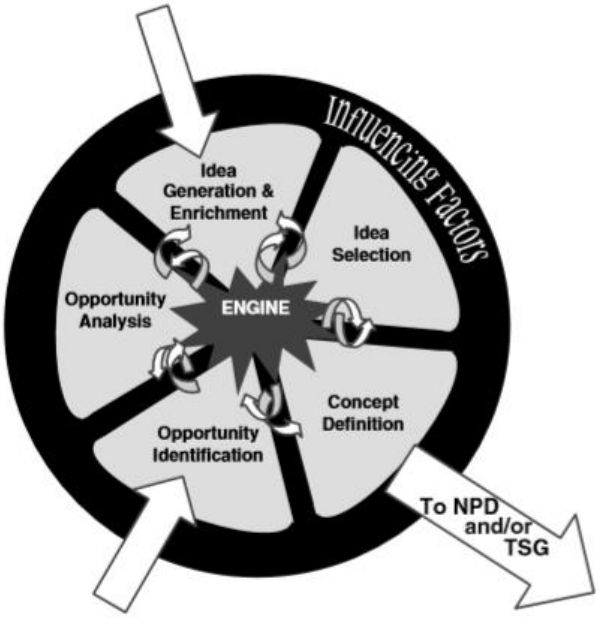

Figure 3 – New Concept Development Model (Koen et al., 2001)

2.2 Opportunity identification

This first key element aims to identify opportunities that can be followed, which can be, for example, an innovative opportunity, a problem, or a customer need.

The identification of opportunity arose through study and investigation, using the analysis of the trend and evolution of technology over time.

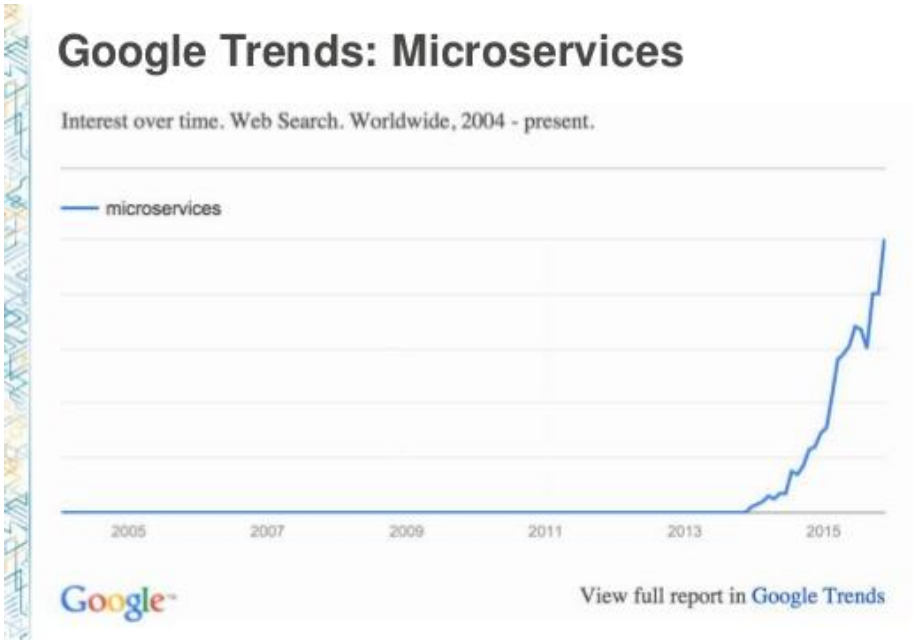

Figure 4 – Google Trends: Microservices (Google Trends, 2020)

The microservice architecture is increasingly adopted by organizations as an integrated mechanism for the development of their solutions. The main trends that affect mainly include the low coupling between services, deployment and domain-oriented design, isolation of failures and resources, among others.

The microservice architecture market is expected to grow at approx. 33 billion dollars by 2023, at a 17% Compound annual growth rate (CAGR) between 2017 and 2023 (Di Francesco, Malavolta, & Lago, 2017).

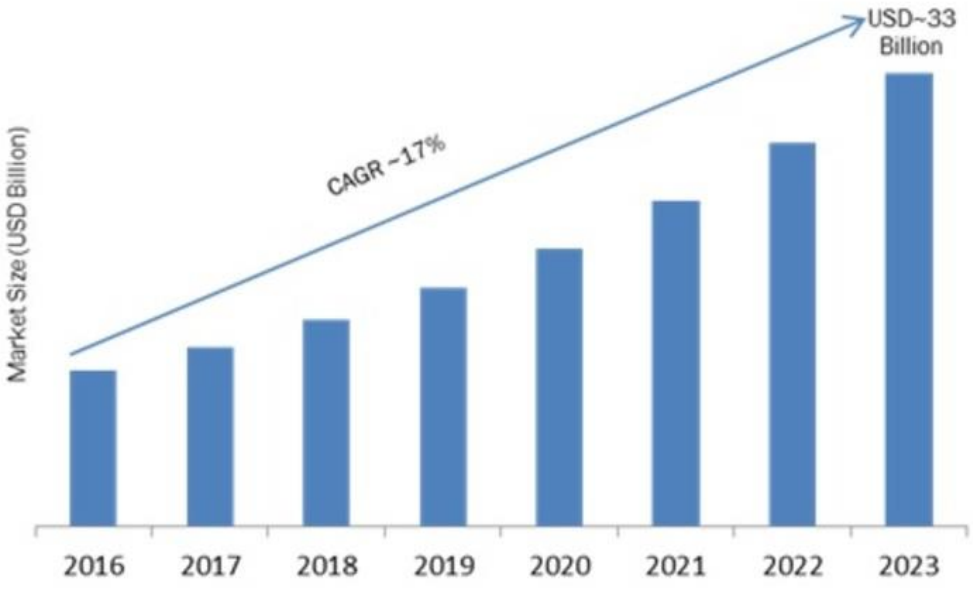

Figure 5 – Growth of microservice architecture (Google Trends, 2020)

2.3 Opportunity Analysis

This key element aims to analyze the opportunity identified above to understand its viability. This analysis involves verifying the opportunity to confirm that it is worthwhile, understanding the unmet needs of customers, and assessing the capacity and competence of the business.

The opportunity identified in the previous section is therefore analyzed so that it is possible to understand it better and understand the possibilities of value that it can offer.

Since this type of architecture began to gain more popularity and began to be implemented in large companies worldwide, interest at a global level has been increasing. With this incremental interest, some articles and webpages on this topic started to appear, but most of them only focus on explaining what this type of architecture is and also on how to migrate from a monolithic architecture to microservices.

Some authors have also written about the topic of data management and there are several attempts to document standards and best practices, however, these standards and

improvements are spread inconsistently and confused by the so-called “grey literature”, through some means such as reports, blogs, systems documentation, among others.

There is, therefore, a lack of a clear guide, with clear guidelines and solutions, for data management in microservice architectures. This type of document can be essential for recent companies that want to choose to develop their systems according to this architecture or for companies looking to migrate their monolithic system to this type.

2.4 Idea Generation and Enrichment

This key is the most embryonic form of a new product or service. It generally consists of a high-level view of the envisaged solution needed to solve the problem identified by the opportunity by generating or modifying some ideas for the identified opportunity.

The following are the ideas that emerged to meet the objective of this work. These ideas have different approaches to the topic of data management in microservices architectures.

- Idea 1: Definition of Bad-Smells associated with poor data management in microservices architectures. The purpose of this idea is to create a guide for defining the Bad-Smells in terms of data and possible solutions;
- Idea 2: Model-Driven Development (MDD) and microservices. The purpose of this idea is to generate the microservice architecture using the previously defined model;
- Idea 3: A case study of Data Management in Microservices Architectures. This idea involves studying a globally recognized company and defining a case study with the best practices and standards for architectures whose data size and complexity are very high, reaching hundreds of thousands of transactions per minute. This case study will help to better manage the data and all the components that need that data;
- Idea 4: Implement a prototype of a data consistency monitoring tool;
- Idea 5: Simulation tool related to data management in microservices. The purpose of this idea would be to apply the content indicated in idea 3 to a simulation tool.

2.5 Ideas Selection

This section of the NCD model is where the most valuable ideas are chosen. This process is usually affected by several factors such as the probability of success at the technical level and the strategic adjustment.

According to Thomas Saaty (1980), the Analytic Hierarchy Process (AHP), translates into an effective tool to work with complex decision making and can help those who are making decisions to define priorities and make the best decision (Moutinho, Hutcheson, & Beynon, 2014).

This method allows the use of qualitative as well as quantitative criteria in the evaluation process and its main idea is to divide the decision problem into hierarchical levels, thus facilitating its understanding and evaluation.

This method was chosen to select the ideas that bring the most value based on the ideas presented in the Idea Generation and Enrichment.

According to the ideas generated, the hierarchical model tree in Figure 6 was built.

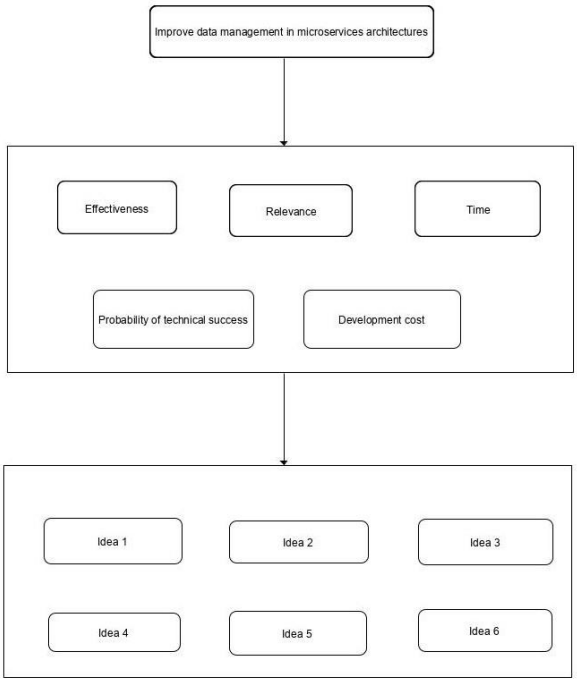

Figure 6 – AHP hierarchical model tree

The first layer of the tree represents the main objective. The criteria selected to assess which is the best idea to arrive at the solution are:

- Effectiveness: if the solution meets the principles of data management in microservices;
- Relevance: the relevance of the idea according to the opportunity identified;
- Time: if the idea meets the time requirements as this document has a limited period;
- Probability of technical success: if the ideas presented will be achieved at a technical level, taking into account the project's constraints;
- Development cost: what is the cost associated with each idea presented.

According to these criteria, it is possible to assess which is the best idea to achieve the main objective.

The possible values for the attribution of importance are presented below, according to the Fundamental scale defined by Saaty (Saaty, 2012).

The Fundamental Scale for Pairwise Comparisons		
Intensity of Importance	**Definition**	**Explanation**
1	Equal importance	Two elements contribute equally to the objective
3	Moderate importance	Experience and judgment moderately favor one element over another
5	Strong importance	Experience and judgment strongly favor one element over another
7	Very strong importance	One element is favored very strongly over another; its dominance is demonstrated in practice
9	Extreme importance	The evidence favoring one element over another is of the highest possible order of affirmation
Intensities of 2, 4, 6, and 8 can be used to express intermediate values. Intensities of 1.1, 1.2, 1.3, etc. can be used for elements that are very close in importance.		

Figure 7 - Fundamental Scale (Saaty, 2012)

Table 1 includes the assignment of weights for each criterion, according to the AHP scale, and describes the standardized matrix with the final weights, for each criterion.

Table 1 – AHP evaluation table

Rating criteria	Effectiveness	Relevance	Time	Probability of technical success	Development cost
Effectiveness	1	1/2	5	3	2
Relevance	2	1	4	4	3
Time	1/5	1/4	1	1/2	1/2
Probability of technical success	1/3	1/4	2	1	3
Development cost	1/2	1/3	2	1/3	1
Total	4,03	2,33	14	8,83	9,5

After defining the weights applied for each criterion, it is intended to determine the relative priority of each criterion. Thus, it is necessary to normalize the comparison matrix. Each value in the matrix is divided by the total of its column.

Table 2 – AHP normalized matrix

Rating criteria	Effectiveness	Relevance	Time	Probability of technical success	Development cost
Effectiveness	0,248	0,215	0,357	0,340	0,210
Relevance	0,496	0,429	0,286	0,453	0,316
Time	0,050	0,107	0,071	0,057	0,053
Probability of technical success	0,083	0,107	0,143	0,113	0,316
Development cost	0,124	0,143	0,143	0,038	0,105
Total	1	1	1	1	1

After calculating the normalized matrix, it is then possible to define the priorities of each criterion for the idea selection process.

Table 3 – Evaluation criteria priorities

Rating criteria	Weights
Relevance	0,396
Effectiveness	0,274
Probability of technical success	0,152
Development cost	0,110
Time	0,068

In conclusion, according to the results of the method applied, the most important criterion is the relevance of the idea. Then the effectiveness and thirdly the probability of technical success. Of all the ideas presented, the ones that are most relevant and most effective for this project are idea 3 (“Case study of Data Management in Microservices Architectures”) and 4 (“Implement a prototype of a data consistency monitoring tool”). Therefore, for this project, and according to the hierarchical assessment, ideas 3 and 4 will be selected to respond to the needs of the business.

2.6 Concept definition

Based on the analysis carried out in the previous chapters, it becomes possible to formulate a concept for the problem of data management in microservices architectures.

This project aims to carry out a study of the company, which uses a microservice architecture, to identify the state of data management and thus create a case study for architectures whose size and complexity of the data is very high, with best practices and standards to perform all this data management. This case study should provide valuable information on the best and most effective ways to structure and manage data in a microservice architecture.

Also, the implementation of a prototype of a data consistency monitoring tool must be carried out.

2.6.1 Value proposition

The value proposition aims to summarize the information that was presented in the previous chapters.

As already presented in the previous sections, as it was possible to analyze, the option for the type of microservices architecture has been gaining more impact year after year, with large world companies already using this type of architecture. It has been gaining more and more impact due to the great advantages it brings to those who use them. Microservices can scale independently, that is, they are deployed completely independently. The updates as they are independent are much simpler and have much less impact since it can, for example, be carried out by only one team and without impacting other sectors. It is possible and common to use polyglot technology, that is, developers are free to choose the most appropriate language and technological stack for the construction of the service, thus also allowing the existence of storage of polyglot data in technological terms, adapting the type of database related to the intended needs. Another advantage is the monitoring of them, being one of the most important aspects in a distributed system, as it allows to take proactive actions in the service that for example is consuming more resources than normal.

Despite being increasingly a trend, this type of architecture is still only a few years old, and therefore there is still a flaw in its documentation. Also, little documentation exists regarding data management and what exists is not very consistent and is only partially addressed.

This work aims to help fill this gap, through the study of the company complemented with an analysis of the literature, allowing the identification of patterns and best practices, starting with the principle for the creation of the case study. This case study will focus on the architecture of the company, which has a large volume of data flow and whose architecture needs to be carefully structured to be able to respond to the great needs and associated requirements.

Also within the scope of data management, the data management prototype will be implemented, containing data consistency monitoring, being presented in detail later in this document.

With this work, all companies that intend to create or migrate their system according to this type of architecture will be able to use this case study and tool to assist this process, and will thus be able to reduce the associated costs, avoid waste, reduce failures at the architectural

level and thus having a more prepared system that guarantees greater maintainability, reliability, reusability, better performance among others.

To present this idea in a more structured way, the Canvas business model was created, which is presented in the next section.

2.6.2 Canvas Business Model

The business model can be represented using the Canvas model (Osterwalder et al., 2010), to make perception easier and more intuitive. This model has nine different topics that allow the strategic planning of a business. Each of these topics is presented in Table 4.

Table 4 – Business Model Canvas

<table>
<tr>
<td rowspan="2">Key partners
Literature search resources like Google Scholar;

A company participating in the study and experimentation.</td>
<td>Key activities
Case study to data management in microservice architecture;

A prototype implementation of data management, containing data consistency monitoring.</td>
<td rowspan="2">Value propositions
Identify standards and best practices, through the case study and literature review;

Will allow companies to reduce flaws at the architectural level and improve various quality attributes;

The data management tool will also help to minimize failures and reduce costs.</td>
<td>Customer relationships
Validation of the case study and tool in a company environment.</td>
<td rowspan="2">Customer segments
All literature platforms;

Companies that are interested in implementing microservices architecture and companies that already use this approach.</td>
</tr>
<tr>
<td>Key Resources
Existing documentation on literature platforms;

Knowledge acquired from this architecture at the business level.</td>
<td>Channels
Literature platforms;

Technology websites.</td>
</tr>
<tr>
<td colspan="3">Cost Structure
Residual costs validating the tool.</td>
<td colspan="2">Revenue Streams
Reduce costs associated with technical charges.</td>
</tr>
</table>

3 Background

This chapter aims to present the fundamental concepts related to microservice architecture and the management of large volumes of data, as they are essential themes for the correct perception of the entire document.

3.1.1 Typical concerns in data-intensive applications

Many applications today are data-intensive, as opposed to compute-intensive. The raw CPU power is rarely a limiting factor for these applications – more significant problems are usually the amount of data, the complexity of the data, and the speed with which they are changing. A data-intensive application often uses standard building blocks that provide the generally required functionality. For example, many applications need to:

- Store data so that they, or another application, can find it again later (databases);
- Remember the result of an expensive operation, to speed up readings (caches);
- Allow users to search for data by keyword or filter it in various ways (search indexes);
- Send a message to another process, to be handled asynchronously (flow processing);
- Periodically crunch a large amount of accumulated data (processing batch).

Despite these standards, the reality is not that simple. There are many database systems with different characteristics because different applications have different requirements. There are several approaches to caching, to create search indexes, and more. When creating an application, it is still necessary to understand which tools and strategies are most appropriate for the task at hand. Sometimes a single-engine may not be enough to satisfy a business requirement, requiring the combination of different tools, which may not be easy to achieve (Kleppmann, 2017).

There are some concerns that matter for most applications, mainly three: reliability, scalability, and maintainability:

- Reliability: this quality attribute says that a system must continue to function correctly (performing the correct function at the desired performance level), even in the face of adversity (hardware or software failures and even human error).

- Scalability: is the term to describe a system's ability to handle the increased load. It is not a one-dimensional label that can be attached to a system: it makes no sense to say "X is scalable" or "Y is not scalable". Instead, discussing scalability means discussing the question: if the system grows in a specific way, what are the options for dealing with growth? How can computing resources be added to handle the additional load?
- Maintainability: most of the software cost is not in its initial development, but in its ongoing maintenance, fixing bugs, maintaining operating systems, investigating flaws, adapting it to new platforms, modifying it to new use cases, and adding new features. Therefore, the software must be designed to minimize pain during maintenance. Three design principles can help:
    - Operability: make it easy for operations teams to keep the system running smoothly;
    - Simplicity: make it easier for new engineers to understand the system, removing the complexity of the system as much as possible;
    - Evolvability: make it easy for engineers to make changes to the system in the future, adapting it to unforeseen use cases as requirements change (extensibility) (Kleppmann, 2017).

3.1.2 Data models

Data models are perhaps one of the most significant parts of software development because they have such a profound effect: not only on how the software is written but also in problem solving strategies.

While historically data was designed as a large tree (the hierarchical model), this was not a good idea to represent relationships from many to many. In this segment, the relational model emerged to solve this problem.

The best-known data model today is probably the SQL, based on the relational model proposed by Edgar Codd in 1970. In SQL data is organized into relations (called tables in SQL), where each relation is an unordered collection of tuples (rows in SQL).

Relational and SQL databases work well for large servers and storage media. However, as different, more extended, and continually evolving data sets became more common in e-

commerce applications, programmers needed something more flexible than SQL. NoSQL is that alternative.

NoSQL databases are created for specific data models and have flexible schemes that allow programmers to build and manage modern applications. NoSQL is also more agile because it is not based on the concept of tables and does not use SQL to manipulate or analyze data.

The more recent non-relational “NoSQL” databases diverged in two main directions:

- Document databases target use cases where data comes in self-contained documents and relationships between one document and another are rare.
- Graph databases go in the opposite direction, targeting use cases where anything is potentially related to everything.

These three models (document, relational and graphic) are widely used today and each is good in its respective domain, with no single solution for all use cases.

4 State of the art

In this chapter, the analysis of what is known or exists associated with managing large volumes of data in microservices is presented. To define the best possible solution to the problem, it is essential to study what is intimately relevant to the topic.

4.1 Literature review design and execution

This section aims to study all the relevant information associated with managing large volumes of data in microservices, obtained through research of published articles and web resources. The digital platforms used to search for scientific articles were:

- **Google Scholar**: is a free and open access research tool created by Google in 2004, but which only in 2006 began to research the Portuguese language. Included in the toolkit available by Google Scholar are full texts of articles in journals, citations, books, theses, dissertations, technical reports, or text summaries. It also facilitates those who are conducting searches by allowing various filters such as the date of publication (Gibson, 2015).

- **IEEE Xplore:** is a powerful resource for research and access to scientific and technical content published by the Institute of Electrical and Electronics Engineers (IEEE) in conjunction with editorial partners. IEEE Xplore provides access to more than four million full-text documents from some of the world's most-cited publications in electrical, computer and electronic engineering. It has more than two million documents in a robust and dynamic HTML format (Xplore, 2019).

- **ACM Digital Library:** is a database with full-text articles and bibliographic literature covering computing and information technology. This renowned repository includes the complete collection of ACM publications, as well as an extended bibliographic database with the main computational works of several academic editors. DL's integrated features, functionality, and content are a critical resource for computing professionals. DL was developed to facilitate the dissemination and sharing of

information, interoperability, user-centered design, and collaboration for computer professionals, researchers, practitioners, and teachers (Toyama, 2010).

As previously mentioned, the microservice architecture emerged recently, more precisely in 2014, so studies of managing large volumes of data in microservice architectures are still evolving and still need further exploration. Some of these articles on the referred platforms were studied and analyzed and the results of this study were compared to understand which points of improvement and which are not yet explored and which are an asset in this area.

The following approaches were identified in research performed in January 2020. This research covered the universe of articles in the area published from 2014 to January 2020 to ensure only cutting-edge knowledge. Some terms such as “data management microservices”, “intensive data management microservices” and “data challenges microservices” were used for the research.

1. **“A Systematic Literature Review on Microservices”** (Shadija et al., 2017)

   This article aims to address a study that was carried out around microservices, presenting emerging patterns and possible gaps and also to address the emerging paradigm of cloud software. This document aims to discover current trends in microservices, the motivation behind microservice research, emerging patterns, and possible research gaps. They intend that with the results obtained they can help researchers and professionals in the field of software engineering who wish to know the new trends in microservices, SOA, and cloud computing.

   To carry out this study, they resorted to a systematic mapping study that aimed to analyze the existing literature but made a change. This modification was that they analyzed the keywords according to the entire article, instead of the keywords according to the abstract. The reason for the change from the original process was to improve the classification criteria by adding new areas.

   With this study they intended to answer the following research questions:

   - RQ1: What kind of research is carried out in microservices?

- RQ2: What are the main practical motivations behind research related to microservices?
- RQ3: What are the emerging standards and tools in microservice solutions?

2. **"Method of Maintaining Data Consistency in Microservice Architecture"** (Fan, Han, Zhang, & Wang, 2018)

In this article, the topic of a classic problem is addressed, such as the consistency of data in the field of distributed systems research. The diversity and complexity of data sources and business needs in the era of big data bring many new challenges to research on this topic and, at the same time, advance research in this field. How to ensure data consistency in microservices is becoming an important topic and this is what is explored in this article. A method is proposed to improve data consistency in the microservice architecture.

3. **"Architectural Technical Debt in Microservices"** (Soares de Toledo, Martini, Przybyszewska, & Sjoberg, 2019)

This study prepared in 2019 aimed to identify issues, solutions, and risks related to the technical debt of architecture in microservices. They carried out an exploratory case study of a real project with around 1000 services in a large international company. Through the qualitative analysis of documents and interviews, they investigated technical debts of this architecture in the communication layer of a system with microservices architecture. The main contributions are a list of questions about architectural technical debt specific to the communication layer in a system with a microservices architecture, as well as the associated negative impact, a solution to resolve the debt, and its cost.

4. **"Architectural Patterns for Microservices: A Systematic Mapping Study"** (Taibi, Lenarduzzi, & Pahl, 2018)

In this case study, microservices architecture is initially presented and its growing popularity is mentioned, however, it is also mentioned that there is still a lack of understanding on how to adopt an architectural style based on microservices. The purpose of the document is to characterize different patterns of this architecture and the principles that guide its

definition. It consists of a systematic mapping study to identify the reported cases of use of microservices and based on these cases, extracts of common standards and principles. The document is divided into two main contributions. First, they identify various patterns of microservice architecture that are widely adopted and reported on in the case studies identified. Second, they present these patterns as a catalog that includes a summary of the advantages, disadvantages, and lessons learned for each case study pattern.

5. **"On the Definition of Microservice Bad Smells"** (Taibi & Lenarduzzi, 2018)

The purpose of this article was to identify and explore symptoms at the code and architecture level (called Bad Smells in this article). These are symptoms of improper design that can impair understanding of the code and decrease maintenance. To identify a set of microservice-specific Bad Smells, the researchers collected evidence of malpractice through interviews with 72 developers with experience in developing microservice-based systems. Then, they classified the bad practices through a catalog of 11 specific Bad Smells in microservices, often considered harmful by professionals. The results obtained can be used by professionals and researchers as a guide to avoiding the same situations in the systems they develop.

6. **"Supporting Architectural Decision Making on Data Management in Microservice Architectures"** (Ntentos et al., 2019)

In this article, a qualitative and in-depth study of best practices and standards in microservice data management architectures is reported. The objective is to complement these sources of knowledge with an impartial, consistent, and more complete view of current industrial practices than those currently available.

To achieve this objective, a qualitative and in-depth study of 35 descriptions of data management practices in microservices was carried out, which was made available by professionals, which contain informal descriptions of practices and standards established in this field. They based the study on the model-based qualitative research method where

they use these professional sources as impartial sources of knowledge and systematically encode them through established methods of coding and constant comparison to develop a rigorous software model with the specification of practices, standards established and their relationships. This article aims to study the following research questions:

- RQ1: What are the standards and practices currently used by professionals to support data management in microservices architectures?
- RQ2: How are current data management standards and practices in related microservices? In particular, what architectural design decisions are relevant to software architecture in terms of data management in microservices?
- RQ3: What are the factors that influence (decisive factors) in the architecture of data management in microservices for professionals today?

7. **"Case Study on Data Communication in Microservice Architecture"** (Smid, Wang, & Cerny, 2019)

In this article, the topic of data synchronization and improving the performance of the data source is covered. One of the main challenges is how to manage data communication from the original monolith to the new microservices and between the different microservices themselves.

Good design for microservice data communication will reduce the overhead of system communications and improve data transmission performance. In this article, two architectures are presented to improve the performance of data communication, demonstrating them in two case studies of production-level systems. The first case study is about data synchronization between the legacy monolithic system database and the microservice databases.

The proposed architecture uses message queues and streaming platforms, such as Kafka and Debezium, to automatically capture and synchronize changes to the database.

8. **“Query Strategies on Polyglot Persistence in Microservices”** (Villaça, Azevedo, & Baio, 2018)

In a microservice architecture, solutions are built through collaboration with external and internal parties, who promote distributed services across networks.

Microservices architecture comprises autonomous services that work together, using different technologies to achieve specific purposes. Characteristics for this technology include organization around the business domain (which defines microservices’ boundaries), evolutionary design, and collaborative work. In this architecture, independently deployable units of software achieve specific purposes through cooperation with other components.

A microservices feature is “Decentralized Data Management”, where services manage their database, eventually using different instances of the same database technology, or even entirely different data store technologies. A scenario where Data Management technology diverges among collaborative services is known as polyglot persistence. Polyglot Persistence implies choosing the best fit persistence model for each task to be performed.

An issue in this scenario is to query data across multiple services, within or beyond the organization. As an example, a single report may correlate information from services that deal with specific data sources such as a graph, XML, document-oriented and relational databases – configuring a polyglot persistence setting. This work characterizes and analyses available solutions to query data in a microservice architecture, based on academia and industry. Strengths and weaknesses of the solutions are provided, according to relevant software quality characteristics based on ISO 25010 model, aiming to guide efforts on future researches.

This work focuses specifically on issues that arise when querying data from services that use different database technologies. It characterizes the state of the art of query strategies for microservices employing polyglot persistence based on academia and industry approaches. Strengths and weaknesses of each approach are provided, along with use cases.

9. **“The pains and gains of microservices: A Systematic grey literature review”** (Soldani, Tamburri, & Van Den Heuvel, 2018)

   Microservices are gaining more and more momentum in enterprise IT. There exist secondary studies analyzing research trends on microservices in the academia, all concluding that academic research on microservices is still in its early stage. At the same time, companies are working day-by-day on the design, development, and operation of microservices, as also witnessed by the huge amount of grey literature on the topic. The authors observed a sort of gap between academic research and industry practices, especially to figure out which are the technical/operational “pains” and “gains” of microservices.

   In this paper, they try to fill the above-mentioned gap by complementing the academic state of the art on microservices with a systematic analysis of the industrial grey literature on the topic. More precisely, they aim at identifying, taxonomically classifying, and systematically comparing the existing grey literature on pains and gains of microservices, from their design to their development. They conducted a systematic review of 51 selected industrial studies, published from 2014 till the end of 2017, then exploited such taxonomies to classify and compare the selected industrial studies, to distill the actual recognition of pains and gains of microservices by the IT industry.

A comparative analysis between the articles described above was necessary to understand the characteristics of these documents and, thus, be able to perceive what flaws exist in them. This analysis made it possible to identify the guidelines to be used in the construction of the case study on managing large volumes of data in microservices.

To make the comparison, some factors were taken into account, which is described below:

Table 5 – Article analysis factors

Main theme	This factor aims to understand the main problem under analysis in the document.
Article type	Identifies the type of article, which can be a literature review, report, case study, questionnaires, etc.
Date	This factor shows the date of publication of the article on one of the described digital platforms.
Purpose of the article	In addition to the main theme, it is important to understand the main objective of the authors with the preparation of the article.
Target Audience	To whom the article is addressed.

With the definition of the factors to be analyzed, Table 6 was used to compare the results.

Table 6 – Comparison of articles

	Main theme	**Article type**	**Date**	**Purpose of the article**	**Target Audience**
"A Systematic Literature Review on Microservices"	Architectural best practices	Systematic literature review	2017	Discover patterns and gaps in the literature	Literature and professionals
"Method of Maintaining Data Consistency in Microservice Architecture"	Data consistency in microservice architectures	Literature revision	2018	Propose a method for improving data consistency	Literature
"Architectural Technical Debt in Microservices"	Technical debt of architecture in microservices	Case study of an international company	2019	Identify issues, solutions, and risks related to technical debt	Literature and professionals

	Main theme	Article type	Date	Purpose of the article	Target Audience
"Architectural Patterns for Microservices: A Systematic Mapping Study"	Standards in a microservice architecture	Systematic literature review	2018	Characterize patterns with a summary of the advantages, disadvantages, and lessons learned	Literature
"On the Definition of Microservice Bad Smells"	Bad Smells in microservice architectures	Interviews with professionals and a literature review	2018	Identify and explore symptoms at the code and architecture level	Literature and professionals
"Supporting Architectural Decision Making on Data Management in Microservice Architectures"	Practices and standards in microservice data management architectures	Qualitative and in-depth study of 35 professional descriptions about data management practices in microservices	2019	Develop software model with the specification of practices, established standards and their relationships	Literature
"Case Study on Data Communication in Microservice Architecture"	Data synchronization and performance in microservices	Case study of production-level systems	2019	Introduce two architectures to improve data communication performance of microservices by demonstrating them in two case studies from their production-level systems	Literature and professionals
"Query Strategies on Polyglot Persistence in Microservices"	Strengths and weaknesses of different query strategies	State of the art of query strategies	2018	State of the art of query strategies for microservices employing polyglot persistence based on academia and industry approaches	Literature

	Main theme	Article type	Date	Purpose of the article	Target Audience
"The pains and gains of microservices: A Systematic grey literature review"	Identify, classify and compare grey literature on pains and gains of microservices	Systematic grey literature review of pains and gains	2018	Complementing the academic state of the art on microservices with a systematic analysis of the industrial grey literature on pains and gains of microservices	Literature and professionals

After the comparative analysis, it is possible to conclude that all of them have as their target audience the literature, and five of them also aim to target professionals in the area. It should also be noted that of the five articles that target both the literature and professionals in the field, only one of them is based on literature review and contact with professionals in the area, while the others are only based on the current literature review. The referred article that aims at these two aspects is "On the Definition of Microservice Bad Smells" because it has the type of article and target audience closer to what is intended. However, the main theme of this is the Bad Smells in microservices architectures, focusing only on identifying and exploring symptoms at the level of code and architecture, it does not have any type of systematic data management guide.

The three articles whose main theme is closest to the intended one are "Method of Maintaining Data Consistency in Microservice Architecture", "Supporting Architectural Decision Making on Data Management in Microservice Architectures" and "Case Study on Data Communication in Microservice Architecture". Although the theme is closer to data management in microservices architectures, the first is only based on increasing data consistency and has no contact with professionals in the area, or with the target audience in the construction of the same. The second is the closest to the intended one, although it is only based on a qualitative and in-depth study of 35 informal professional descriptions, with no study in the current literature. The third is only focused on improving the data communication performance of microservices.

The article “Query Strategies on Polyglot Persistence in Microservices” focuses on an important point of the main theme, which is the topic of queries in microservices, with polyglot databases and with a large volume of data.

Also, it still has several topics that are not covered, and that is of great importance.

4.2 Data management practices

The current technological solutions related to the second objective of the dissertation are presented in. This objective refers to the monitoring of data consistency. Despite the study carried out on what exists in the data management literature, these technological tools/solutions can be of great importance within a microservice ecosystem, allowing to maintain all the data of the sub-systems monitored and sending the necessary alerts.

While all of the metrics presented in Appendix B – Technological framework are extremely important, even more so in a microservice architecture, none of these tools have data consistency analysis features across the various microservices. In other words, it is currently possible to extract several metrics from a service, but it is not possible to compare the data consistency across multiple services.

The topics presented below are closely related to the management of large volumes of data in microservices and are of great importance for the correct understanding of the document.

4.2.1 CQRS

The main approach used by people to interact with an information system is to treat it as a CRUD data store. Thus, it simply consists of a mental model of some data structure where it is possible to create new data, read data, update existing data, and delete data when it is no longer needed (Fowler, 2011).

CQRS is a pattern that separates models for reading and writing data. The main idea of this pattern is to divide the operations of a system into two very separate categories:

- Queries: return a result and do not change the state of the system and are free of side effects.
- Commands: operations that change the state of a system.

Therefore, a conceptual model is separated into models for updates and queries, which are referred to as Commands and Queries, respectively.

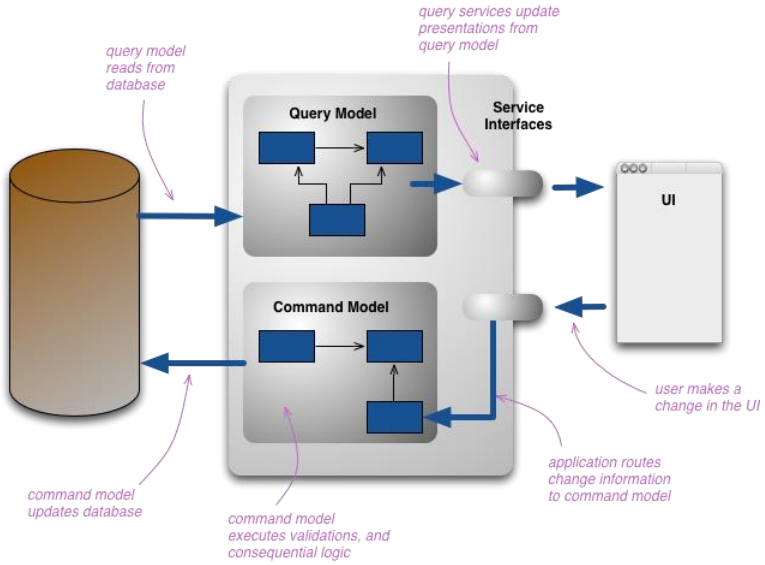

Figure 8 – CQRS pattern (Fowler, 2011)

In many cases, CQRS is used in complex scenarios, where, for example, different physical databases are used for readings (for performing queries) and for writing (updates).

This pattern is particularly important in systems with large volumes of data and that need to perform complex data queries. Medis.NET is an example of a medical information system that aggregates information from more than 2 million patient admission forms, along with medical examinations, laboratory tests and therapeutic treatments. This system began to show slow responses as the amount of information became larger and the data was spread out in separate tables. The performance of the queries was impacted as the amount of data traffic on join operations became much higher. Therefore, in order to reduce the impact on load operations on the main database, a separate component with a new data store was created just for reading data, which allowed to greatly improve its overall performance (Villaça et al., 2018).

A more developed system can also implement Event-Sourcing for the updates database, therefore, only events are stored in the domain model instead of storing only the current state data.

4.2.2 Decentralized Data Management and Polyglot Persistence

A solid data management strategy can mean the difference between the success and failure of a data-driven application. In microservice architectures, several services consume data distributed in several places and that differs from each other. It is therefore essential to prepare microservices to handle large volumes of data in the best way.

The microservice philosophy favors decentralization in all aspects of software design. This focus on decentralization not only guides the organization of business logic, but it also guides how data is maintained (Fowler, 2014).

Microservices favor each service to manage its database, different instances of the same database technology, or different database systems – an approach called Polyglot Persistence.

In traditional monolithic systems, it is quite common to use a monolithic data store, such as an SQL server that contains a single database with many tables. This central database is used as a mechanism for all data persistence, and often parts of the application logic are transferred to the SQL server in the form of queries that use complex joins or even stored procedures. In contrast, the microservice architecture favors decentralized data management, as discussed by Martin Fowler in his articles.

As each microservice has its database, concepts such as redundancy, consistency, and data integrity are concerns that must be taken into account.

When an application is run, it consumes existing data, creates new data, and saves data for later use. If data is not needed immediately, it can be temporarily stored in a cache layer or memory. However, in many cases, it is necessary to store the data for later use. In such cases, the data can be stored in a database or disk independent of the application.

Traditionally, companies have designed their monolithic applications to rely exclusively on relational databases. This meant that all data received the same level of performance, regardless of its unique need. The growth of NoSQL and the big data revolution challenged this point of view with non-relational data storage solutions. It consists of several tools that, in a particular and specific way, solve problems such as handling large volumes of data, executing queries with low latency, and flexible data storage models, such as XML or JSON documents.

Before the era of big data, SQL data stores were the perfect standard for centralized systems. However, big data requires scalability horizontality for scalability purposes, which is in stark contrast to SQL's inability to scale well, that is, reducing or expanding resources as needed. Large volume requires scalability, while variety requires handling different types of data structures. The speed of data access requires low latency and high throughput.

The need to deal with the unbelievable immensity of information in terms of scale, scope, distribution, heterogeneity, and supporting technologies has also made this problem more evident. Some components of the ACID properties (Atomicity, Consistency, Isolation, Durability) guaranteed by SQL databases need to be compromised to obtain better availability and scalability in the distributed system. The relational data model can no longer efficiently handle all of the current needs for analyzing large, often unstructured data sets. NoSQL databases with BASE characteristics (Basically Available, Soft State, and Eventual consistency) have emerged to fill this gap.

NoSQL technologies are not intended to replace relational databases, but only to propose some solutions that in certain scenarios are more appropriate. It is possible to work with NoSQL technologies and relational databases within the same application (Khine & Wang, 2019).

Microservices enable data storage to be handled as a separate service. Also, it allows storing data in various formats, which helps in certain business cases where not all storage options are practical.

This approach has some costs. However, the benefits are worth it. When a relational database is used improperly, it can have an extremely negative impact and significantly impact service capacity.

Polyglot Persistence implies choosing the best persistence model for each task to be performed, considering each one can be designed to best solve different problems (Villaça et al., 2018).

The Netflix company uses polyglot persistence. They have Cassandra, Dynomite, EVCache, Elastic, Titan, ZooKeeper, MySQL, Amazon S3 for some data sets and RDS. Elasticsearch provides excellent search, analysis, and visualization of any data set in any format in almost real-time. EVCache is a distributed in-memory caching solution based on Memcached that was open-sourced by Netflix in 2011. Cassandra is a NoSQL distributed data store that can handle large data sets and can provide high availability, multiregional replication, and high scalability.

Dynomite is a distributed layer of Dynamo, again open-sourced by Netflix, which provides support for different storage mechanisms. Currently, it supports Redis, Memcached, and RocksDB (Tangirala, 2017).

4.2.3 Event Sourcing

Event Sourcing is a way to persist the state of an application by storing an immutable sequence of events. State changes are triggered to update the application's state in response to these events. In Event Sourcing, any triggered event will be stored in an event store. There are no update or removal operations on the data and all events generated will be stored as a record in the database. If there is a failure in the transaction, the failure event will be added as a record in the database. Each registry entry will be an atomic operation.

It offers many advantages, such as:

- Audit: the events are immutable and provide a natural history of what happened in the system.
- Back in time: as the flow of events is persisted, it is very easy to determine the state of the system or an entity at any time, aggregating the events in the period. This provides the ability to answer historical questions about the status of an entity and thus allows the possibility to reverse transactions that were performed incorrectly.
- Event replay: if there are past events that are incorrect, it is also possible to go back in time and send the events in question again. Another advantage is in the bug fixing of the services. This is because all software has bugs, so when a consumer, for example, has a bug and has not consumed the events correctly, the bug can be corrected and the events that were processed incorrectly can be replayed.
- Performance: events are simple, immutable, and independent options that only require an additional operation. Event storage is typically optimized to handle high-performance recordings.
- Scalability: event storage avoids the complications associated with recording complex domain aggregates in relational databases, allowing more flexibility for scalability.

It may seem like a bad idea for each entity to be represented by an event stream as there is no way to reliably query the data. It is even more noticeable, especially in applications with a large volume of data and with intensive use of them, in which much of the commercial value depends

on the analysis of the data. This difficulty would make Event Sourcing inappropriate for most applications and relevant only for very specific and isolated use cases.

However, as discussed above, the CQRS architectural pattern allows to help a lot in this process. As mentioned, this standard describes the concept of having two different models, one for changing information and one for reading them, completely separate from each other. The referred difficulty disappears with the application of Event Sourcing in conjunction with CQRS. The separation of the Write Model and Read Model allows the use of the most appropriate strategy for each model and allows the two to be scaled independently. Event Sourcing is a particularly efficient data recording model, as it works as a log that is being added, where new information is always added, thus allowing a minimum block. As each event is irremovable and immutable, there are no updates or removals which allow for good recording performance. On the other hand, as the reading model is completely independent, it allows the freedom to choose the most appropriate technology to optimize queries, and may even be a technology completely different from the recording model, such as a denormalized and non-relational database built for query-oriented data storage (Rocha, 2018).

4.2.4 Eventual Consistency

Still taking into account CQRS and Event Sourcing, it is possible to understand the separation of the writing and reading model. This means that any changes will be recorded in the Write Model and later synchronized asynchronously to the Read Model, through events. Any changes made will be available for reading in the next milliseconds after recording, meaning that the functionality as a whole is eventually consistent. While the system processes the values it can return obsolete or inconsistent data, a period is known as the inconsistency window. As it is an architecture with a large volume of data, problems can arise such as the accumulation of messages in the message brokers and even the systems have unexpected spikes in use, and thus take a little longer than expected to process.

Consistency management is a critical issue for systems with big volumes of data. Strong consistency models have serious limitations to system scalability and performance due to the necessary synchronization efforts. In contrast, weak and eventual consistency models reduce the performance overhead and enable high levels of availability (Fan et al., 2018).

Although the components are adapted to process messages quickly and have a strong infrastructure behind the service, these types of problems can and will happen. As explained earlier, the CAP theorem illustrates how a system can be available or be consistent in the face of network partitions, but it cannot be both simultaneously. In the highly distributed world and where there are large volumes of data, to have a system available it is necessary to be eventual consistent.

If there are critical features for a business and it is managed with eventual consistency, some problems will arise. Therefore, the ideal is that it is applied according to the needs of the business. There are use cases where availability is the necessary property in the system, but there are also use cases where consistency is also required (Rocha, 2018).

5 Problem analysis

This chapter presents in detail the problem, the objectives, as well as the methodology of the work.

5.1 Problem

Many service-based systems implemented in large companies follow a microservice architecture. Because microservices are used to build distributed systems and promote architectural properties such as independent service development, multilingual technology stacks, including multilingual persistence and loosely coupled dependencies, data management plays a crucial role in this type of architecture. Notably, it is highly advisable to decentralize all concerns in terms of data management. Therefore, this architecture requires, in addition to the inherent challenges existing in distributed systems, sophisticated solutions to guarantee data integrity, data consultation, transaction management, and consistency management among all systems involved. These data must be managed with care to ensure that the performance and security of services are consistent with expectations and policies. When choosing this type of architecture in the company under study, several scenarios and challenges related to data management and large volume of data were considered, such as:

- Does the use case work with data from multiple sources? Does the use case need zero downtime?
- Does the use case depend on highly reliable data or is it focused on improving data quality and traceability?
- Manage data growth;
- Data validation and consistency;
- Choosing the right technologies for data management;
- How to structure the microservice ecosystem for better data management;
- Data communication between microservices;
- How to return data from different microservices?

When dealing with large volumes of data, these challenges and issues have an even higher proportion, and incorrect management of these topics can have a very negative impact on the entire system.

Many standards and practices for data management architectures in microservices have been proposed, but today they are discussed informally mainly in the so-called “grey literature”, consisting of practitioners' blogs, experience reports, system documentation, etc. In most cases, each of these sources well documents only some of the existing practices, but they generally do not provide a clear and validated data management guide in the real world. Instead, the practices reported are largely based on personal experience, often inconsistent or incomplete. This creates considerable risks and uncertainty and risk in the microservice data management architecture, which can be reduced through much more substantial personal experience or by carefully studying a wide range of knowledge sources (Ntentos et al., 2019).

Microservice technology has only recently become a popular topic, which implies the lack of empirical data in older systems that use this technology. This lack also applies to what should be defined (or considered) as good microservice architecture and, consequently, to which sub-optimal solutions may have flaws and debts at the architectural level that can become quite expensive. This architecture is, in fact, based on several structural qualities and characteristics different from traditional systems (Soares de Toledo et al., 2019).

Data replication is a common technique for programming distributed systems and is often essential for achieving performance or reliability goals. However, data replication can compromise data consistency. The fundamental tension between performance (choosing weak consistency) and correction (keeping strong consistency) is a recurring theme when designing competing and distributed systems (Burckhardt, 2015).

Many of these problems and challenges that have been addressed here can be critical factors in managing data consistency. It is essential to minimize these inconsistencies as they can have extremely negative impacts on the systems in question. Being able to detect these inconsistencies as quickly as possible allows them to avoid or reduce their impact.

5.2 Objectives

Software architecture is a branch of software engineering, with increasing importance nowadays. As application development processes evolve, new challenges arise every day. Some of these challenges are to achieve a well-designed architecture with the best quality according to the related quality attributes.

After identifying the theme of the work, the guidelines that are intended to be followed were developed, listing the following objectives:

- Propose a set of recommendations and solutions for challenges encountered in data management in microservice architectures with very high data volume, reaching hundreds of thousands of transactions per minute. This volume, which exists in many large companies around the world, raises some concerns, directly and indirectly, related to their data. Some of the challenges in this type of architecture are the determination of the ideal granularity of the microservices, data consistency, heterogeneity, complexity and strategies of the data queries, data granularity data vs performance, among others. A case study of the company presented in chapter 6 and a literature review were carried out to collect the set of recommendations and solutions for these types of challenges.
- Develop a prototype of a tool to assess the consistency of the data present in a microservice ecosystem. This tool will help organizations and development teams to find possible data inconsistencies in their microservices. For example, compare data that exists in a master service X with data that exists in service Y. Data related to a particular entity may have to be persisted in several different databases, from different microservices, and cross several components. There is the possibility of failures during this journey, either due to processing failures or unavailability of a service, among others, which can lead to data inconsistency. It is essential to have some kind of monitoring of these data inconsistencies. The tool was used and subsequently evaluated at the company under study referred to in section 1.4.

Therefore, research was carried out in three digital libraries: Google Scholar, IEEE Xplore, and ACM Digital Library. This research was focused on data management processes in microservices and issues reported, both in industry and in the literature. Existing solutions for this management were also analyzed.

6 Case Study

This chapter presents a set of recommendations and solutions to the challenges on the topic of data management in microservices architectures whose complexity and volume of data are quite high, reaching hundreds of thousands of transactions per minute.

To this end, a case study of the company described above was carried out, as it is a world reference company that has used a microservice architecture for over 8 years.

6.1 Introduction

This entire case study is related to the architecture and strategies adopted by the company under study for the management of large volumes of data. Below are some metrics related to the current data volume of this company:

- Hundreds of thousands of transactions per minute;
- Databases with 50M + of records.

To start this process precisely and thus be able to correctly identify business capabilities and their boundaries, it is essential to conduct a preliminary study of business needs. These business capabilities represent something that the company adds that manages to add value to its end users. This definition meets one of the most crucial principles in microservices, which says that a service needs to be modeled around business capacity. This principle is also called Bounded Context, which is one of the pillars of Domain-Driven Design (DDD) (Evans, 2015).

The identification of business capabilities and the corresponding services requires a high level of business understanding. For example, these business capabilities for an online shopping application can include:

- Product Catalog Management;
- Inventory Management;
- User Management;
- Product Recommendations Management;
- Pricing Management;

- Others.

After identifying the business resources, it is possible to build the necessary services. Each one can thus belong to a different team, which becomes a specialist in that specific field and a specialist in the most appropriate technologies for those particular services. This generally leads to more stable API limits and more stable teams.

There are some essential features for correctly defining microservices within each boundary. These must be designed to promote high cohesion and low coupling. Some crucial factors to guarantee these principles are:

- Do not share database tables with other microservices as if this happens the databases will be a source of high coupling. One of the fundamental principles when it comes to structuring and developing new services is that they must not cross database boundaries. Each service must rely on its own underlying data storage pool. This allows centralizing access controls, audit log, cache logic, among others;
- A well-designed microservice should have a minimum number of database tables. Although they can manage and record millions or billions of entries, one should focus on actually having one or two database tables;
- When designing a microservice, take into account which services will depend on that new service and what the impact will be on the entire system if that data becomes unavailable. Taking this into account allows to properly design backup and data recovery systems for this service;
- A well-designed service is a unique source of truth for the data, that is, must have its well-defined responsibility and complete information on a given business context.

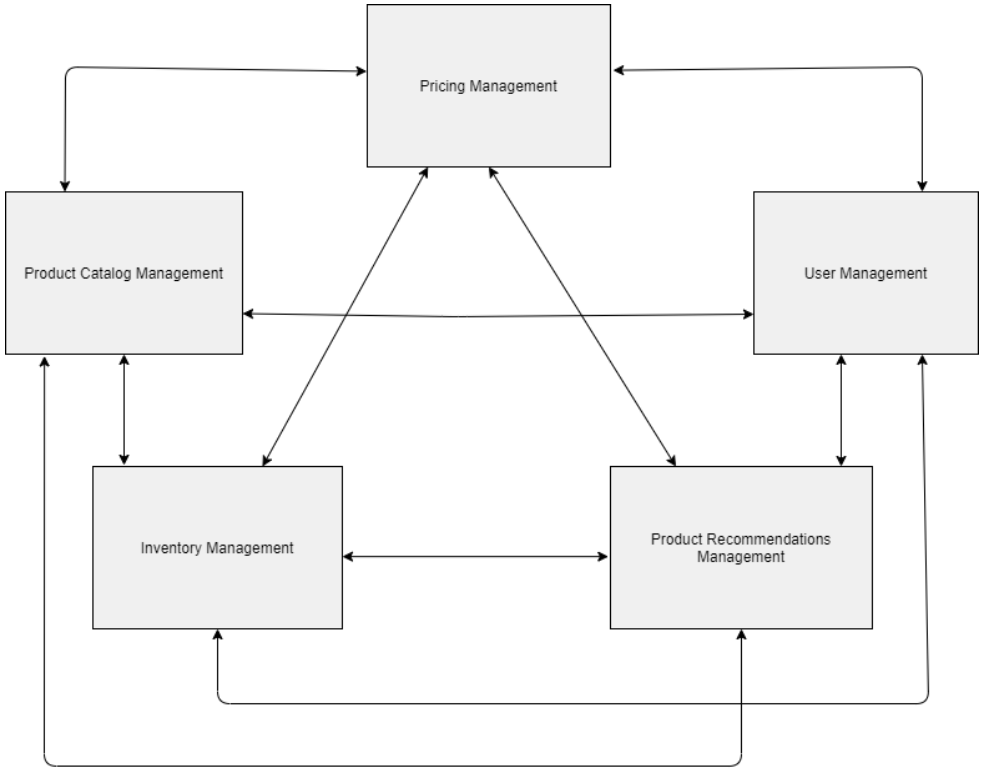

Figure 9 – Boundaries division example

As can be seen in Figure 9, an example of the division previously addressed by business capabilities and the connection between them is presented. Later on, it will be detailed, allowing, not only to observe this division, but also the structuring of the border services (it will be explained in more detail what it is) within each boundary and the communication between them.

6.2 Method

A meeting was held with professionals in the area to specifically structure the aspects that will be addressed in this chapter. Before this meeting, a questionnaire was given to 7 professionals involved to obtain information about their professional background. The complete questionnaire can be found in Appendix A of this document.

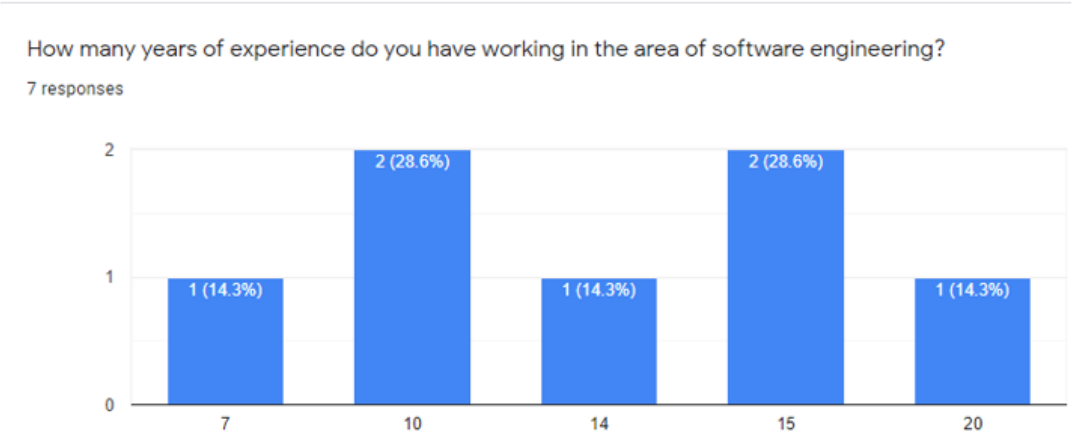

Figure 10 – Questionnaire – Participants professional experience

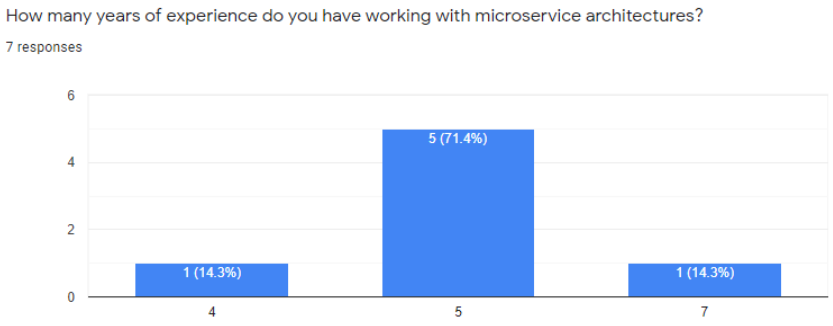

Figure 11 - Questionnaire – Participants professional experience in microservice architectures

As seen in Figure 10, most of the participants already have some years of experience in the area of software engineering, with almost all having 10 or more years of experience. Figure 11 shows that the vast majority have 5 years of experience working with microservices architectures.

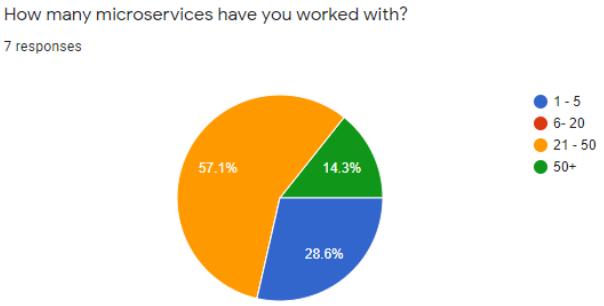

Figure 12 – Questionnaire - Number of microservices that each participant has worked with

Figure 12 presents the number of microservices that each professional has worked with. More than half answered that they have already worked with 21 to 50 microservices.

In this way, solutions are presented for several issues related to the topic of data management in microservices architectures with a large volume of data, including:

- Structuring and segregating the different business areas of the system;
- Structuring of microservices within each business area;
- How to transmit/communicate large volumes of data between microservices;
- What types of databases to use in each microservice and why;
- How to monitor large volumes of data in the system;
- How to correct erroneously or lost data in microservices;
- How to manage data consistency;
- Concurrency control.

6.3 Structuring data in microservice architectures

One of the main concerns in structuring microservice architectures is how to implement it to better handle data and the high volume of transactions. Is there a standard set of principles that can be followed to help build this architecture in the best way?

In general, the answer is no, however, some common themes are addressed informally in the literature and that together with the case study of this company allow presenting a set of relevant recommendations for success in data management and their large volume;

6.3.1 Transmission/communication of large volumes of data between microservices

When thinking about microservices architectures and managing large volumes of data, questions immediately arise about how the communication will take place.

Two concepts have to be taken into account when choosing the best communication strategy:

- Scalability;
- Resilience.

These concepts, further explored in section 3.1.1 for data-intensive applications, are necessary to understand how to carry out this communication, either with direct requests to resources (synchronous communication) or base the communication on Event-Driven Architecture (asynchronous). Event-Driven Architecture (EDA) is an event-oriented architectural design pattern in which communication between components is modeled using Event Streams. This architecture is composed of event producers and consumers. A producer creates events and represents them as messages, not knowing the consumer of the event or the outcome of an event (Microsoft, 2018).

After an event is detected, it is transmitted from the producer to the consumers through the event channels (Event Bus), where an event processing platform processes the event asynchronously.

When it comes to complex communications, which is common in the company under study, transparency is vital. The use of the correct standards to carry out such communications can help to escalate and solve most of the problems that may arise.

The solution suggested and implemented in the company under study involves the use of Event-driven architecture to take full advantage of the asynchrony that this style of architecture promotes. One of the great advantages of this architecture refers to the optimization of time, as there is no resource blocking to attend the requests sent, as traditionally happens in microservices that use the nature of synchronous communication, eliminating the idleness found in traditional implementation with synchronous calls. There are several options available to play the role of Message Brokers such as Apache Kafka, ActiveMQ, RabbitMQ, IBM MQ, among others.

The Message Broker chosen by the company is Apache Kafka, which brings great advantages for data transmission, even more so when the volume of data is quite high as presented in the previous metrics.

Like most similar systems, Apache Kafka allows the tracking of message feeds on message topics, meaning that producers create the data and send it to message topics and consumers read those topics. In Kafka, topics are partitioned and replicated across multiple nodes.

These messages are in the form of byte arrays and so developers can use them to store any object in any format they want, including Avro, JSON, String, Protobuf, among others. These messages can also be accompanied by a key, called a partition key. With the partition key defined,

there is a guarantee that all messages that have the same partition key will be processed by the same partition. One of the biggest advantages of using partition keys is the guarantee of maintaining the ordering of messages within each partition.

The uniqueness of Kafka lies in the fact that it deals with each partition as a log (that is, an organized set of messages) and that each message in a given partition is assigned an offset concerning the current reading position. Kafka is not concerned with tracking which message has already been read by which consumers and only unread messages are interested. It maintains messages for a configurable period, and each consumer is responsible for tracking the location in each log (Mancill, 2018).

Therefore, Kafka can support a massive amount of consumers and retain a large amount of data without incurring a lot of overhead costs and overload.

Among the benefits of Kafka, the most important are:

- Highly scalable: as it is a distributed system it can be scaled quickly and easily without any downtime;
- Highly durable: messages stored on disks and provide replication within the cluster which provides a highly durable messaging system;
- Highly reliable: replicates data and is capable of giving numerous subscribers. Additionally, it can automatically balance consumers in the event of a failure, which means it is more reliable than the similar messaging services that are available;
- High performance: offers high performance for both publishing and subscribing messages, using disk structures capable of offering constant levels of performance, even when dealing with terabytes of stored messages.

6.3.2 Structuring of boundaries and their communication

As described at the beginning of this document, the solution presented for structuring the architecture was outlined by identifying business capabilities and separating them into different boundaries. Now, the border services of the boundaries will be detailed and how the communication and data transmission is done.

To assist in this structuring, the company in question follows a philosophy called "Decentralized Governance".

In this philosophy, microservices are built as totally independent services and dissociated with a variety of technologies and platforms. However, there is a need to define common standards for the design and development of services, so as not to make the system disorganized, which could lead to complications both for its functioning and for the development teams. Since these microservices can be managed by other teams in the future, as in the company under study where there are no "owners" for each microservice, having some structural standards is essential and facilitates teams in many aspects.

In general, this philosophy can be summarized as follows:

- In a microservice architecture, it is not necessary to have centralized governance at the time of design;
- Microservices can make their own decisions about design and implementation;
- The microservice architecture promotes the sharing of common/reusable services;
- At the API Gateway level, important aspects can be centrally managed, such as some aspects of runtime governance, such as SLAs, limitation, monitoring, common security requirements and service discovery.

Therefore, the company makes use of the system's division into boundaries and also the API Gateway standard. The implementation of this standard provides a gateway that acts as a single point of entry for all customers. It can also provide other parallel features, such as authentication and caching.

Figure 13 shows the simplified image of the solution described and implemented by the company.

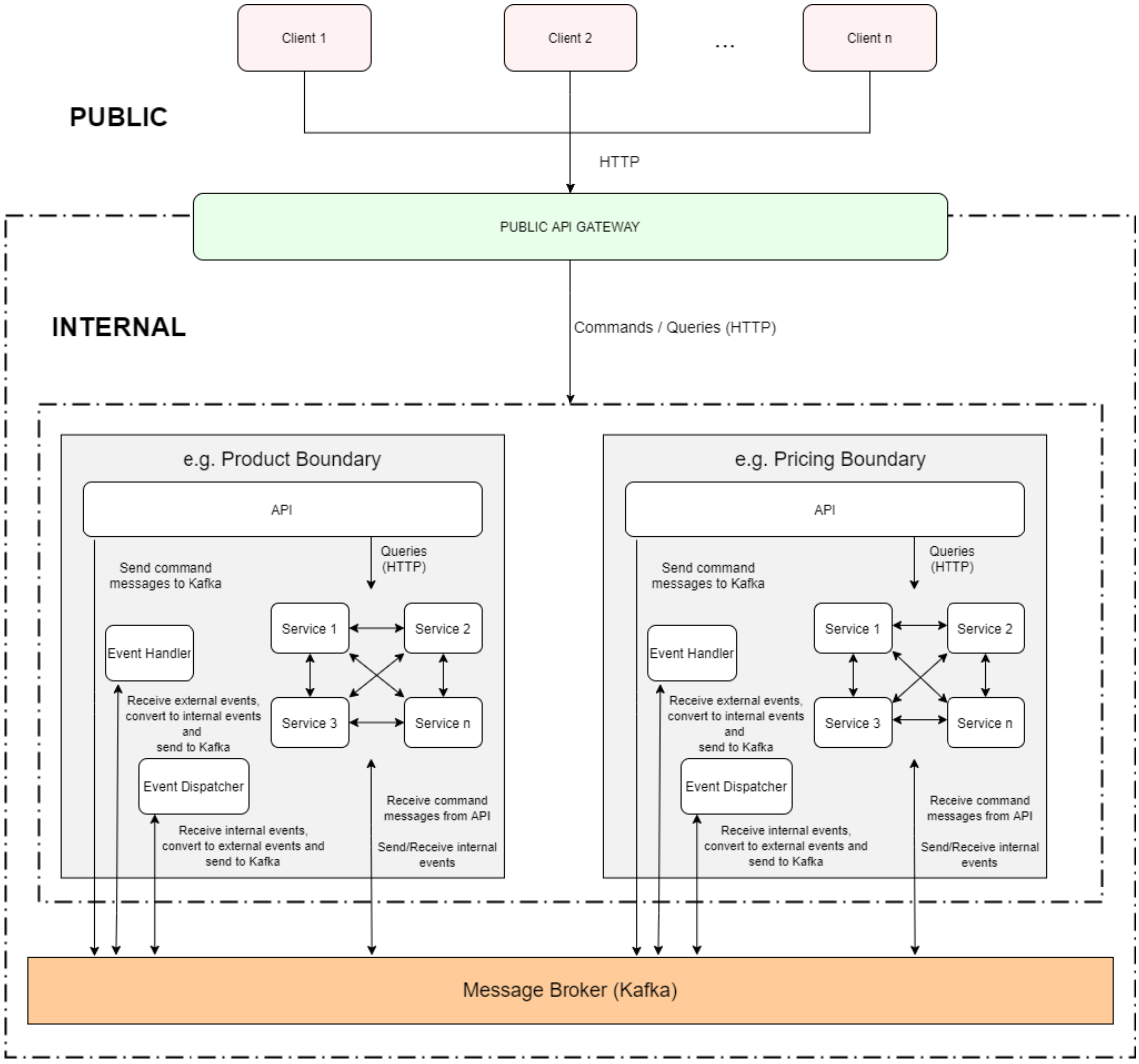

Figure 13 – Detailed boundaries division diagram

Figure 13 shows in general the division in boundaries explained previously and the way clients communicate with them. It is possible to see that the various clients have a single entry point, the API Gateway, which is responsible for forwarding requests as well as dealing with other details such as authentication. This API makes HTTP requests to the API of the different boundaries.

Within each boundary, the following boundary microservices and communication with the external to the boundary are defined:

- Boundary API: responsible for receiving all HTTP requests coming from abroad, both from API Gateway and from microservices of other boundaries. This API then has two distinct paths, thus following the CQRS architectural pattern. These two paths are:

1. Queries: as it is necessary to obtain the answer to queries synchronously, the boundary API communicates with the APIs of the internal microservices responsible for a given resource.
2. Commands: for the remaining operations that are related to create/ update/delete, operations are performed asynchronously. To do this, it creates commands corresponding to the operation and sends those commands to Kafka topics. Subsequently, the service responsible for that context, as it is listening to that Kafka topic, will receive the message and process it.

- Event Handler: responsible for receiving events from outside (from other boundaries), converting to internal events, and forwarding them to the correct Kafka topics to later be received and processed by the respective microservice.
- Event Dispatcher: responsible for listening to internal events (from the boundary), converting them to external events, and sending them to a Kafka topic that will be heard by interested customers.

As shown in the diagram regarding the structuring of boundaries and their communication, is demonstrated the separation between queries and data updates, that is, the separation between Read and Write Models. As explained, in this case, the separation is not only at the level of the conceptual model (different logical layers in the application) but also at the physical level (different microservices for queries -> Read Model, and updates -> Write Model) which thus allows having some benefits extra as discussed above. In other words, when recording an entity in the Write Model, these changes are persisted and events are sent to Kafka, which are then consumed by Read Model, which records in its database optimized for queries.

6.3.3 Decentralized Data Management

In this type of architecture, functions are divided into several microservices, so it is highly recommended not to use the same centralized database as it violates the loosely coupled nature of microservices.

One of the main advantages of this decentralization is to avoid coupling between services. This can happen if the services share the same data schemas. Any changes to the data schema must be coordinated with all services that depend on that database. By isolating data storage, changes have minimal impact and the agility of independent services is maintained.

As a microservice entity can be upgraded as an isolated part, changes offer minimal risk to the rest of the system. In a monolithic architecture, there are more risks involved. The microservice architecture offers greater availability, because if a microservice fails, a part of the application fails, not the entire application.

Another advantage is that each microservice can have the reading/writing patterns, data models, and database types that are best suited to its business logic. For example, a developer can select the RDBMS type data source over the NoSQL data source based on the requirement.

The company under study began to realize that its relational databases started to cause problems as they could no longer support large volumes of data, so they had to find other data processing mechanisms to manage large-scale data.

They began to implement the philosophy of decentralized data management and thus take full advantage of it in conjunction with the microservice architecture and the large volume of data.

In conjunction with this philosophy, the company opted to use NoSQL databases in several parts of the system, as it brings all the advantages presented in section 4.2.2.

The NoSQL movement generated many types of niche databases, such as document stores, columnar databases, and full-text databases. However, NoSQL databases are not perfect, as most of these are technologies that are still evolving. Also, they are specific to certain niches in the different areas of an application.

The polyglot persistence approach is applied in the company under study because it makes use of applications with intensive use of distributed data. Polyglot persistence generally involves the use of different types of databases, as appropriate, in different parts of the target system, such as in this company where each microservice uses the type of database that best suits its needs.

6.3.4 Choosing the right database types

As previously mentioned, there is not only an SQL or NoSQL database capable of fulfilling all the requirements required by a microservice architecture with large volumes of data. However, there is not one that stands out from the others, as each has its advantages and disadvantages. As an alternative, the use of different types of databases within a system has become prevalent.

In this company, there were some difficulties initially to understand which model was best to use to fulfill the requirements of each part of the system. When selecting, performance, reliability, and data modeling requirements must be taken into account.

Performance requirements

With microservices, it is important to design all services to provide the best possible throughput. If a microservice becomes a bottleneck in the data flow, a good deal of the system can be negatively affected.

Reading performance: the metrics commonly used to measure reading performance are the number of operations per second or a combination of how fast queries can be performed and data returned. The speed of data return depends on how well the data is organized and indexed. A microservice for example that provides an e-commerce product catalog may need to run queries that apply various filters, such as product category, price, ratings, etc. The database that will be chosen for this must take into account these specific aspects and requirements of each microservice to be able to give the best responses to the requirements as well as respond quickly to requests, as well as to be able to respond to large volumes of operations by second.

Recording performance: the number of recording operations per second that the service will have to be able to support must be analyzed. Large volumes of data need a database that can perform thousands or millions of write operations per second.

Latency: to offer instant experiences it is necessary to have a low latency database and the implantation of a microservice close to the database allows to minimize network latency.

Resource efficiency: to reflect the microservices design principles and their agility, the database coverage area must be minimal, maintaining the ability to scale as needed (Kumar, 2018).

Data modeling requirements

The advantage of microservices over monolithic architecture is that each service can choose the database that best fits its data model. A microservice architecture can use a data model based on key-value, graphic, hierarchical, JSON, flows, and search engines, among others.

Nature of the data

Not all microservices process or generate data at the same stage in its life cycle. For some microservices, the database may be the source of the truth, but for others, it may be just temporary storage.

The first step in choosing the ideal data storage is to determine the nature of the data for each microservice. The data can be broadly classified into the following categories:

- Ephemeral or short-lived data: a cache server is a good example of ephemeral data storage. It is a temporary data store whose objective is to improve the user experience, providing information in real-time. The microservice is typically tuned for high performance and read operations are intensive.
- Transient data: data such as logs, messages, and signals usually arrive at high volume and speed. Data ingestion services generally process this information before passing it on to the appropriate destination. These data stores must support high-speed writes. The durability requirements for transient data are higher than those for ephemeral data but not as high as transactional data.
- Operational data: information collected from user sessions, such as user profiles, the content of the shopping cart related to an e-commerce solution are examples of operational data. Although the data stored in the database is not a permanent proof of registration, the architecture must do its best to maintain the data for business continuity. For operational data, the requirements for durability, consistency, and availability are high. Generally, operational data is organized into specific data models, such as JSON, graph, relational, key-value, etc.
- Transactional data: Data collected from transactions such as payment processing and order processing must be stored as a permanent record in a database that strongly supports ACID requirements.

The popularity of new applications, such as social networks and the concept of making everything digital, are driving the emergence of NoSQL with several categories of data models. Therefore, determining which type of database is most suitable (SQL or NoSQL data models) for a specific domain will depend only on the requirements of the applications.

Each SQL and NoSQL database has its advantages and weaknesses. NoSQL databases have some similar features, such as the ability to scale horizontally, the ability to increase performance by

connecting more nodes to work as a single unit, and eliminating joins for better performance (Kumar, 2018).

An example of Polyglot Persistence in e-commerce is shown in Figure 14. Availability-oriented activities, such as a shopping cart, are stored in a key-value database. Financial data that requires ACID and is oriented towards strong consistency is stored in relational databases.

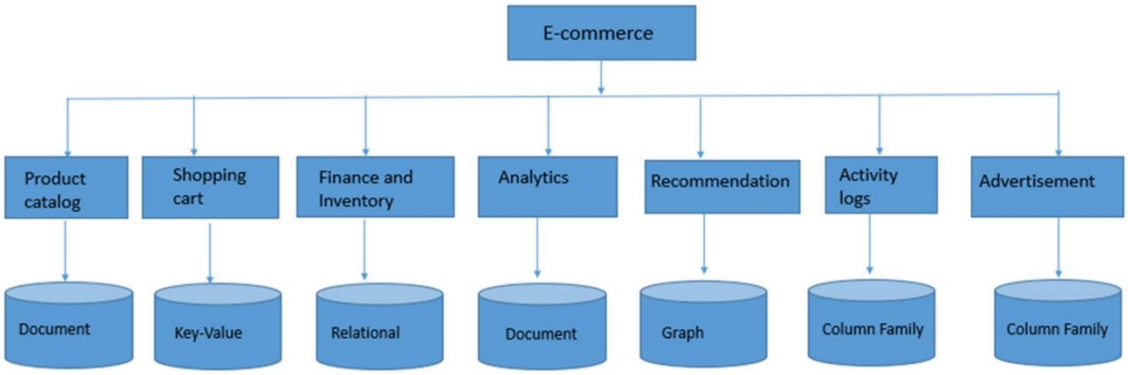

Figure 14 – E-commerce Polyglot Persistence example

Although it cannot be said that database X has to be used for a specific type of microservice, there are some types of database-oriented to help in certain aspects and that best suit the specific needs of each one. With this, and according to the company under study, some aspects were taken into account when choosing the best type of database, which were very valuable and allowed to optimize the applications in question in the best way.

- MySQL, Oracle databases – relational databases continue to be useful for data that changes infrequently and is business-critical. They can be difficult to store and scale and are expensive to maintain.
- Redis - in-memory data structure store, used as a database, cache, and message broker. It is excellent for storing log data files and for caching and is easily scalable.
- MongoDB, CouchDB - this is the most popular type of NoSQL database, the document repositories. Both meet the need for flexible data models that address structured, semi-structured, and unstructured data types.
- Cassandra - columnar database is successful in creating scalable distributed clusters. It is essentially a hybrid database between key-value (tabular data) and a column-oriented database. It is quite popular for systems with large data sets and volumes. Another fundamental characteristic of the architecture is the fact that it is decentralized, that is, different from the master-slave architecture found in other database systems,

in this way, all nodes in the network have the same functions and capacities, there is no single point of failure.

- Elasticsearch - this is a database that allows the addition of complex text queries very quickly. It has features and capabilities that allow it to act as document storage, time-series database, visualization platform, and more. In some cases, it works much faster and more effectively than other databases like MongoDB. It can be used as a database for boundary Read Models.
- InfluxDB, Prometheus - time-series databases are created to receive and render streaming data in real-time to facilitate data analysis. They are used in application monitoring.
- Neo4j - this is a graphical database that allows analyzing how several members of the data set are connected and helps to provide important information about those connections. Typical use is for fraud detection, as it easily identifies suspicious activity in large data samples.
- MariaDB ColumnStore, Apache Hbase - a columnar database focused on fast data readings and writes with efficient data storage.

6.3.5 Event Granularity vs Performance

When designing an event-oriented system, such as the system in this case study, it is necessary to model in a better way the events that will be used to transmit changes that are made in that system. These events will be used to transmit changes between all microservices in the system. Therefore, these events must fulfill essentially two criteria:

- Be expressive
- Be useful

In a more simplified way, these events must clearly describe what they do, their purpose, and contain information that is easy for consumers to interpret and process.

When discussing about a system with a data volume of hundreds of thousands of transactions taking place in seconds, being able to correctly define the correct granularity of each event becomes essential. With very fine granularity, events do not have enough information to be useful. If they have a very coarse granularity, they will have a high-performance impact due to

the serialization and deserialization of messages, in the disk space and will stress the message brokers.

Is an event necessary for each change in the value of an entity? Or is it better to use larger events that convey entire entities?

According to the theory, commands and events must be created to reflect the user's intention, thus remaining faithful to Domain-Drive Design (DDD). However, more pragmatically, major negative impacts on the consumers of the events can be avoided by understanding what their needs are. For example, imagine that a system sends "ClientFirstNameChanged" and "ClientLastNameChanged" events with the following schemas:

```
ClientFirstnameChanged
      userId: Guid
      firstName: string


ClientLastnameChanged
      userId: Guid
      lastName: string
```

Figure 15 – Event schemas example

These events clearly show users' intention to change the "firstName" or "Lastname", but consumers are not interested in knowing about these changes separately, they are only interested in knowing the full name to which it was changed. 2 options would solve these problems: either consumer keeps the status of customer names internally in their service or needs to ask the owner service about the missing information they need. Both have dire consequences, even more so with a large volume of data. This is because in the first solution there would be a much greater occupation of disk space and would have the extra effort of creating an internal state of the entities and keeping them always synchronized, which would also increase the replication of the data. As for the second option, as the reading model (Read Model) is eventually consistent, there is the possibility of returning information that is not the most recent as it may not yet have processed the events in question (Rocha, 2018).

Events cannot be too small or too large, they need to have the right granularity for each type of need. Having the instinct to get it right requires extensive knowledge of the system, the business, and consumer applications.

6.3.6 Ordering and concurrency of events

As presented throughout this case study, events and asynchronous communication are used. Therefore, it is essential to have mechanisms that guarantee the ordering of events and mechanisms of competition management in the processing of them by microservices, since the volume of data is quite high and that would bring several of these problems.

The solution suggested and implemented in the company under study:

This is where the use of Apache Kafka can bring great benefits. As addressed when discussing the topic of communication/data transmission, Kafka makes use of topics that are divided into partitions to make horizontal scaling possible. Messages will be distributed across these partitions so that multiple messages can be processed in parallel by attaching consumers on each of these partitions.

For example, a microservice can listen to a topic related to the “User” entity, called “User-Topic”. When creating this Kafka topic, if the partition count is set to 5, it means that the microservice infrastructure can process 5 requests in parallel.

The key provided by Kafka can also be used, called PartitionKey. PartitionKey is an important Header which can be sent along with a message to a Kafka topic. Messages sent with the same PartitionKey will always end up in the same partition of a topic.

For the example above, if there are for example 3 events related to the entity "User" with the same PartitionKey (it could be the UserId), all the above messages will end up in the same partition of "User-Topic". And they will exactly follow the same order as pushed to the topic. This way the events can always be processed in the same order they were emitted.

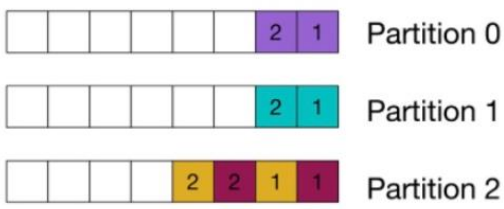

Figure 16 - Ordered messages with PartitionKey in Kafka

With 10 instances of the microservice that manages "Users" to listen to the topic "User-Topic", with these configurations is 100% guaranteed that only one thread of these 10 instances will consume from that specific partition that has that PartitionKey.

No two user-microservice instances will process the same user at the same time. This is important because if the PartitionKey is not used there may be two or more instances of the same microservice listening on the same PartitionKey as the topic and therefore is not guaranteed which event will be processed first. This can lead to incorrect event processing and it is a major problem with RabbitMQ and hence it is not advisable to use it for event-driven architecture.

6.4 Summary

The topics covered in section 4.2 together with what was explored in this chapter allowed to build a set of solutions to the problem of managing large volumes of data in microservices and helped the company under study to structure its system taking into account the need to fulfill various quality attributes such as scalability, maintainability, availability, among others.

The company under study has a system with thousands of microservices that aggregates a huge volume of data. For this company, it was essential to have its business model distributed in a decentralized manner through the creation of boundaries and to have a well-defined communication between them. It was also essential to have communication between boundaries and within each boundary made asynchronously, which allows the system to be much more scalable and with better performance. This asynchronous communication was implemented using a message broker, in this case, Apache Kafka. This message broker was chosen because it offers several advantages such as high scalability, durability, and reliability. This message broker also has mechanisms to deal with the ordering and concurrency of events.

The decentralized management of data applied in this company allowed to avoid the coupling between microservices, since each one has its own database and manages its own data. This allowed increasing the resilience of the system because a failure in a service or database does not affect the whole system. This decentralized management also allows each microservice to choose the type of database that best suits its requirements.

7 Data Consistency Monitoring Tool

The case study carried out and the analysis of the literature on the management of large volumes of data in microservices architectures allowed us to present some of the biggest challenges encountered on the subject and a set of solutions and recommendations.

This chapter aims to present the solution to one of the problems and challenges encountered: Monitoring data consistency in microservices.

7.1 Analysis

This section will present the analysis carried out on the theme, the design alternatives, the requirements, and the implementation of the final solution.

7.1.1 Context

Today all companies rely on data to make accurate business decisions and strategies. Relevant, precise, and actionable data contribute considerably to the growth of an organization. If not managed strictly, this data can become useless and even harmful to the company (Rouse, 2019).

Data management is the process of consuming, storing, organizing, and maintaining data created and collected by an organization. The data management process includes a combination of different functions that collectively aim to ensure that the data in the systems are accurate, available, and accessible (Rouse, 2019).

However, inadequate data management can overwhelm organizations with incompatible data silos, inconsistent data sets, and data quality problems that considerably limit companies because the data is extremely valuable (Rouse, 2019).

It is essential, in microservices architectures, to have active monitoring of the data as they are distributed and potentially replicated in several databases.

As previously discussed, with the application of the CQRS pattern and the consequent separation of Read and Write Models, there is data replication between these two models. This replication arises not only from the application of this standard but also from the needs of each

service, that is, a microservice may have to store an internal state of information that is managed by another microservice (discussed in section 6.3.5). This replication can lead to potential data inconsistencies, so its rapid detection and correction can be a fundamental factor.

As presented in Appendix B – Technological framework, data monitoring tools in microservices were analyzed, and quite complete solutions for monitoring services were found. However, a flaw was found on this topic, as there is a gap in a tool that allows analyzing the consistency of data between microservices.

Maintaining active monitoring of data consistency is fundamental in a company, and can help to perceive potential inconsistencies even before these become problems for the company and thus reduce the number of incidents/bugs created.

7.1.2 Domain Model

Next, the domain model of the second objective of this work is presented. This objective was described in section 5.2, and consists of the implementation of a prototype of a data consistency tool in microservices architectures.

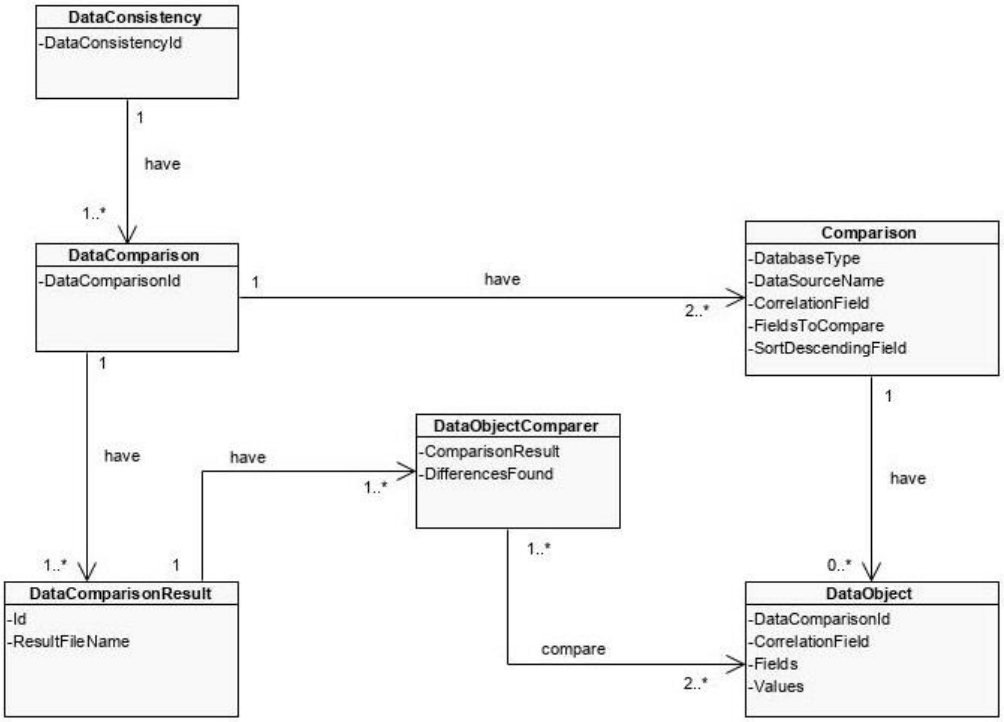

Figure 17 – Domain Model

For a better understanding of Figure 17, the description of each of the entities present in the diagram is presented.

- DataConsistency: the more general entity of the application. This entity has a unique identifier and a list of DataComparisons;
- DataComparison: entity responsible for each comparison to be made. Each comparison is made between two or more Comparison entities;
- Comparison: entity that owns the data for each comparison. These data are:
    - DatabaseType: type of database to which the comparison relates (example: SQL, MongoDB);
    - DataSourceName: name of the data source. Can be for example an SQL table, a MongoDB collection, or an ElasticSearch index;
    - CorrelationField: identifier by which the data will be filtered for comparison. For example, if there is a "*Users*" table in SQL and a "*Persons*" collection in MongoDb, a comparison of the two can be made using the CorrelationField "*UserId*" and "*PersonId*", respectively;
    - Fields: fields that should be included in the data comparison;
    - SortDescendingField: field by which the data will be sorted in descending order. As mentioned in section 7.1.1, each comparison will analyze only a sample of data from the complete set, so it is necessary for each analysis performed to return the most recent data. An example of a value for this identifier would be a field such as "*UpdatedDate*";
- DataObject: entity into which all data returned by the different databases are converted. This object is generic and has the necessary information to make the comparisons;
- DataObjectComparer: represents the result of comparing two or more DataObjects. This entity has information on the result of the comparison (boolean indicating whether they are equal or not) and the list of differences found;
- DataComparisonResult: represents the final result of the comparison. This result is inserted into a file showing all the differences found as well as the percentage of correction of the analyzed data.

7.1.3 Requirements

This section presents the requirements identified for this solution and is divided into non-functional and functional requirements.

7.1.3.1 Non-functional requirements

The non-functional requirements of the application, which have been described throughout the document, are presented in this section and follow the FURPS+ model. As the focus of this work is microservices architectures with large data volumes, the developed tool had to take into account its standards and guidelines.

Non-functional requirements are:

- The system must be highly effective;
- The system must be highly efficient;
- Present a reduced effort to configure the desired comparisons;
- Present a reduced effort to configure comparisons for different scenarios;
- Present a reduced effort to configure data from the databases to be used (url, credentials);
- The system should allow configuring multiple comparisons simultaneously;
- Provide the result of the data consistency analyses in files with information on the inconsistencies found and the percentage of correction of the analyzed comparisons;
- The system should ensure a low response time for each analysis of data consistency, and each analysis should be carried out in a few seconds;

7.1.3.2 Functional requirements

The tool developed will serve to help developers to detect data inconsistencies, so all use cases have the same actor, the developer himself. The tool has only four use cases, and a large part of its value is in the generic algorithm that allows analyzing data from various databases and comparing them.

Figure 18 presents the use-case diagram.

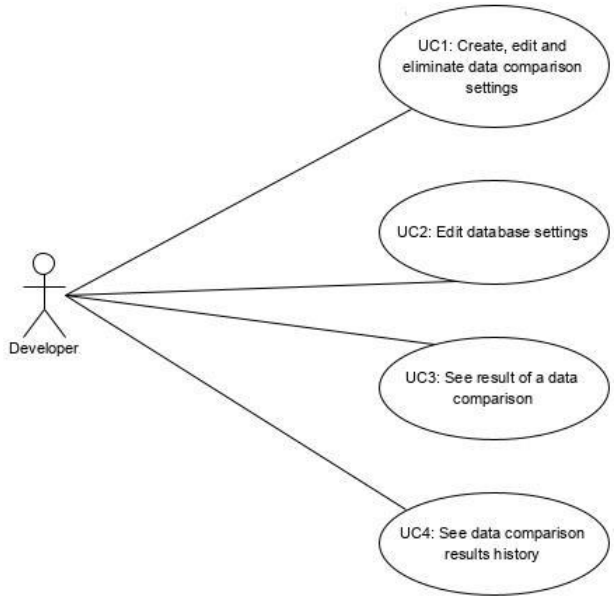

Figure 18 – Use-case diagram

As shown in the non-functional requirements of section 7.1.3.1, the user should have a low effort to configure the desired comparisons. In this way, the user only fills in the information necessary to perform the desired data consistency analysis, according to UC1. This UC allows it to indicate, through a JSON file, the data of the comparisons that it intends to make. In this file, it is possible to configure several comparisons simultaneously, and for each one, the user must indicate:

- DatabaseType;
- DataSourceName;
- CorrelationField;
- Fields;
- SortDescendingField;

Details about each of these fields were presented in detail in section 7.1.2.

In addition to the configuration of the desired comparisons, the user must insert through a JSON file the configuration of the databases involved in the comparisons (UC2). For example, if the user wants to make comparisons of a Mongo Collection, he must indicate in this file the data to access the database (URL and credentials).

Finally, and according to UC3 and UC4, the user can consult the result of the current comparisons as well as the history of all comparisons made previously. This result is presented

in the form of a file showing all the differences found and the percentage of correction of the analyzed data.

7.2 Design and implementation

This section presents the details of the solution and provides details of the implementation for the most relevant points of the same. The technologies, the logical view, and the process view are presented. These views are part of the 4+1 model by the author Philippe Kruchten and allow to define parts of the architecture that will be followed.

7.2.1 Logical view

After defining the domain model and use cases, the high-level design of the prototype was defined, which is illustrated in Figure 19.

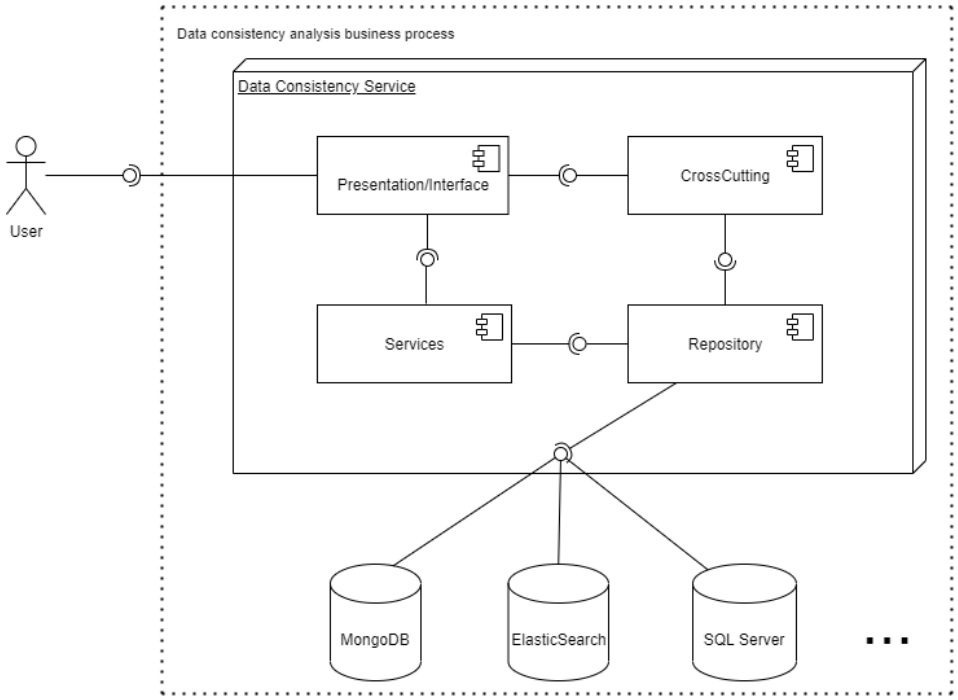

Figure 19 – Logical view

The figure shows the different components that are part of the implemented prototype. These components are:

- Presentation/Interface: responsible for communicating with the client, in this case, the developer who will configure the data consistency analysis. This component is responsible for ensuring that the input and output data are in the correct format;
- CrossCutting: responsible for managing information/services that are transversal to various components of the application;
- Services: responsible for dealing with business logic and rules. It is the component responsible for comparing data. It is also the component that communicates with the Repository layer;
- Repository: responsible for communicating with the different databases involved in the comparisons (MongoDB, ElasticSearch, SQL Server, among others);
- MongoDB, ElasticSearch, SQL Server, and others: components that have the data that will be consulted to perform data analysis.

7.2.2 Process view

This section describes the decomposition of the system into groups of processes and presents the main modes of communication between them.

Figure 20 presents in a high-level format the main process of the prototype, related to the collection of data and comparison of the same to analyze its consistency.

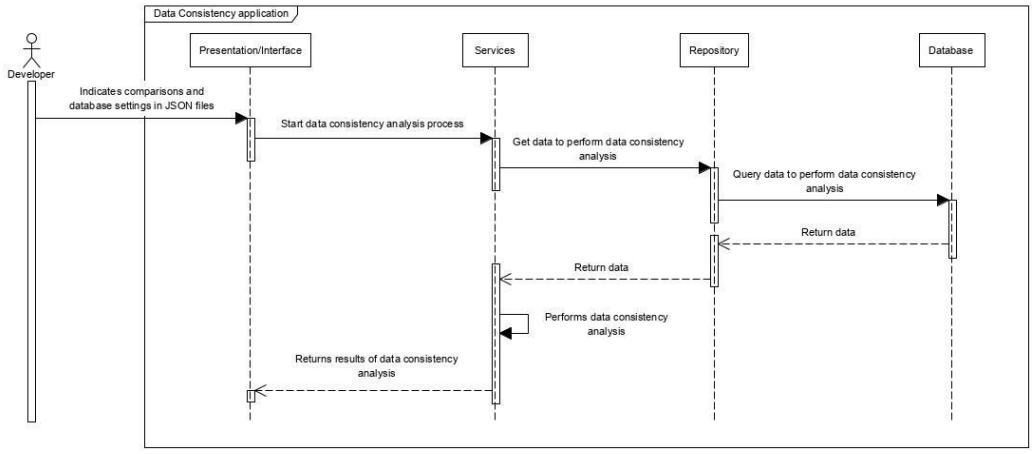

Figure 20 - High-level sequence diagram of the data consistency analysis process

From the figure, we can verify the main steps, at a high level, of the data consistency analysis process, from the user's action to the moment when he gets the answer, in this case, the result of the comparisons. This process has the following high-level steps:

1. The developer starts the data consistency analysis process by configuring two JSON files, the first with information on the desired comparisons, the other file with the information of the database settings;
2. The application receives the information and begins the data consistency analysis process;
3. The service layer receives the request to start the analysis process and asks the Repository layer for data for each comparison;
4. The Repository layer requests the database in question in that comparison to obtain the data for analysis;
5. After obtaining the data, the Repository layer returns the data to the service layer;
6. The service layer receives the data for each comparison and performs the comparison algorithm in order to analyze its consistency. The result of the comparison is then sent to the Presentation/Interface layer;
7. The Presentation/Interface layer receives the results and presents them to the developer.

7.2.3 Technologies

In accordance with the state of the art and the requirements/restrictions of the prototype, the technologies presented in the next sections were selected.

7.2.3.1 Development environment

The prototype was developed using the IDE Visual Studio Enterprise 2019 and used the C # programming language together with the .NET Core 3.1 framework.

For this prototype, three different database types were used, thus allowing the analysis of data from any of these databases: MongoDB, Microsoft SQL Server, and ElasticSearch.

7.2.3.2 Framework .NET Core 3.1

The developed project made use of .NET Core 3.1 technology, as it is aimed at testing, high performance, native dependency injection, open-source and community focus, which has contributed to the increase in its popularity.

The use of .NET allows the use of external packages for the prototype through NuGet. NuGet is the library manager for the .NET platform and allows the possibility to produce and consume packages. Some external packages were used in the prototype, and the most important ones are covered in this chapter.

7.2.3.3 Databases

As shown in section 7.2.3.1, three databases are used for data consistency analysis: MongoDB, SQL Server, and ElasticSearch. It is thus possible to make comparisons between any of these databases.

As the prototype was developed using Microsoft tools, external packages obtained through NuGet were used to connect the application to each of the databases. These packages are:

MongoDB C#/.NET Driver

This driver is available through NuGet under the name MongoDB.Driver is easy to install and use, and brings all the necessary resources to add, change, remove, update and search for information (with filters, sorting, among others) of documents within a particular collection. This driver allows searching data from MongoDB collections by encoding classes in C# that correspond to existing collections in the database. However, for the project in question, this approach is not enough because one of the requirements is that the user can indicate any collection that he wants to compare, so it is impossible to create the mapping in C # classes of the collections. To meet this requirement, the generic object of this driver, BsonDocument, was used, which allows the properties and respective values of the collection to be loaded generically.

Dapper

Dapper is a micro ORM for .NET. It is a simple driver, which helps in mapping objects from SQL queries. It is a high-performance library for data access that was created by the StackOverflow team and is open-source (Kanjilal, 2019).

This library also allows performing data queries indicating the specific class to map the results to or generically obtain the results. As the objective is to return data generically, the dynamic object was used, with each dynamic object corresponding to a row of the SQL table and containing information on the fields and respective values of it.

NEST

NEST is the official high-level .NET client for Elasticsearch. It is a client that has the advantage of automatically mapping all request and response objects. Provides a Domain Specific Language (DSL) query that maps directly to the Elasticsearch DSL query. NEST uses and exposes the low-level client Elasticsearch.Net internally (Mauri, 2020).

In addition to allowing queries to specify the C# class to which the results will be mapped, this library also allows data to be returned dynamically using the dynamic object. This object has information about the fields and respective values of each ElasticSearch index, thus allowing the analysis of data consistency of any existing index.

7.2.4 Data consistency configuration

The prototype provides two input files for the user with the configurations they want. The first file serves to indicate the data for comparisons and the second to indicate the configuration of the databases. These two JSON files are explained in more detail below:

7.2.4.1 Comparison file

This file was called "*ComparisonConfiguration*.json" and Figure 21 shows an example of its configuration.

```
{
    "DataComparisons": [
        {
            "Id": "57346120-9840-45df-b836-5295a1c507cb",
            "Comparisons": [
                {
                    "DatabaseType": "SQL",
                    "DataSourceName": "Product",
                    "CorrelationField": "ProductId",
                    "FieldsToCompare": "ProductName, ProductCategoryId",
                    "SortDescendingField": "UpdatedDate"
                },
                {
                    "DatabaseType": "ElasticSearch",
                    "DataSourceName": "Product",
                    "CorrelationField": "Id",
                    "FieldsToCompare": "ProductName, productCatId",
                    "SortDescendingField": "insertedDate"
                }
            ]
        },
        {
            "Id": "cf105d57-62c4-4e63-bb9a-7411ee2c343d",
            "Comparisons": [
                {
                    "DatabaseType": "ElasticSearch",
                    "DataSourceName": "ProductCategory",
                    "CorrelationField": "id",
                    "FieldsToCompare": "CategoryName, CategoryTreeId, CategoryNumber",
                    "SortDescendingField": "UpdatedAt"
                },
                {
                    "DatabaseType": "MongoDb",
                    "DataSourceName": "ProdCategory",
                    "CorrelationField": "CategoryId",
                    "FieldsToCompare": "Name, TreeId, CategoryNumber",
                    "SortDescendingField": "UpdatedDate"
                }
            ]
        }
    ]
}
```


Figure 21 - Example data comparison file

The root element of this file, "*DataComparisons*", is a list that allows the insertion of several groups of comparisons simultaneously. In the example in the figure, there are two different comparison groups, each identified by the "*Id*" field.

Within each group of comparisons, there is a list called "*Comparisons*" that indicates the database and respective fields to be compared. In the case of the first group of comparisons, consistency analysis is carried out between an SQL table with the name "*Product*" and an ElasticSearch index also with the name "*Product*". The type of database is indicated by the "*DatabaseType*" field and the name of the data source by the "*DataSourceName*" field.

The remaining fields that are indispensable for comparison are also listed: "*CorrelationField*", "*FieldsToCompare*" and "*SortDescendingField*". These fields were presented in detail in section 7.1.2.

7.2.4.2 Settings file

In this file called "*appsettings*.json" the user must indicate the access settings to the database. The Figure 22 shows an example of that file, in this case with the configuration of access to the MongoDB, SQL Server and ElasticSearch databases. The file also has a Hangfire configuration which will be explained in the next section.

```
{
    "ApplicationSettings": {
        "Logging": {
            "LogLevel": {
                "Default": "Information",
                "Microsoft": "Warning",
                "Microsoft.Hosting.Lifetime": "Information"
            }
        },
        "SQLServer": {
            "ConnectionString": "Server=(LocalDB)\\MSSQLLocalDB; Database=Data_Consistency;Integrated Security=SSPI;Trusted_Connection=true;Application Name=DataConsistency;"
        },
        "Mongo": {
            "ConnectionString": "mongodb://localhost:27017/SVC_DATA_CONSISTENCY"
        },
        "ElasticSearch": {
            "Server": "http://localhost:9200",
            "Username": "",
            "Password": "",
            "AllowAllCertificate": true,
            "RequestTimeoutInMinutes": 3
        },
        "HangfireSettings": {
            "CronExpression": "*/2 * * * *"
        }
    }
}
```

Figure 22 – Settings file example

7.2.5 Data consistency algorithm

The data consistency analysis algorithm aims to be able to analyze a sample of data from the databases configured in the input file in a few seconds of execution. The data sample analyses 10,000 objects for each database. Figure 23 shows in more detail the flow of the data consistency analysis algorithm.

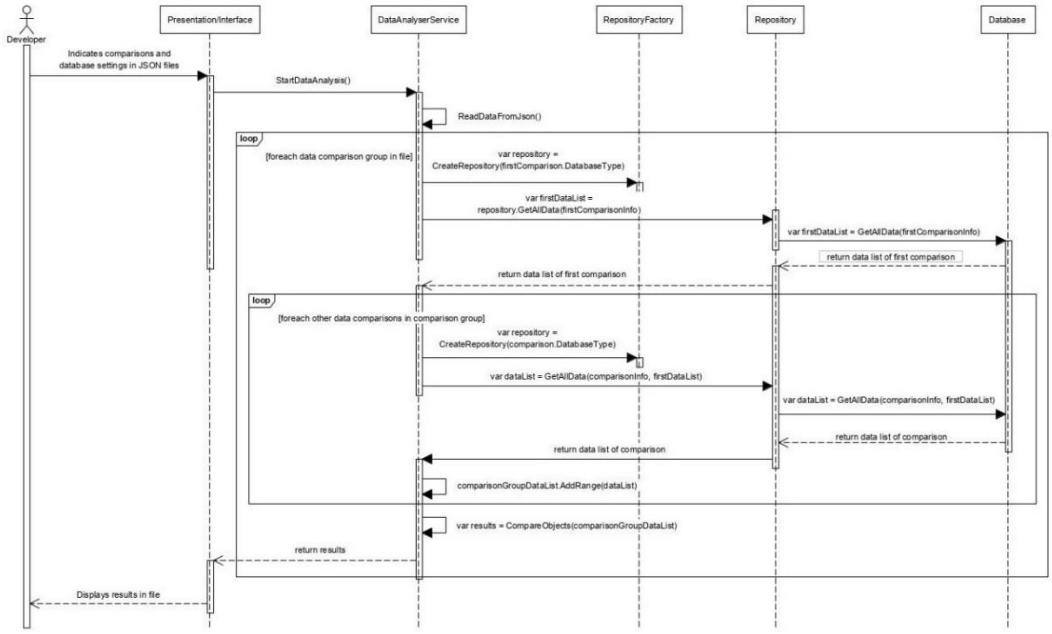

Figure 23 – Data consistency analysis sequence diagram

To complement the diagram, the steps that are performed by the algorithm are presented:

1. After the developer has set up the comparison and database files, the system starts the comparison process by reading the input file;

2. The system analyzes the first group of comparisons, and within the group, it analyses the first comparison configuration and fetches the sample of data from the indicated database;
3. After obtaining the data sample from the first comparison, the system will search for the corresponding data in the other comparison settings of the same group, using the field "*CorrelationField*" indicated in the configuration;
4. After obtaining the data from each of the databases, the system joins them into a common data list. With this list the system performs the comparison of the objects;
5. The system returns the results to the developer in the form of a file. This file has information of all differences found and the percentage of correction of the analysed data;
6. If there are more comparison groups in the configuration file, the system resumes from step 2.

In addition to the packages presented in section 7.2.3.3, were also used two more external packages (obtained through NuGet) in the algorithm:

• Hangfire: package that allows the execution of recurring background jobs. In the case of this algorithm, a recurring background job was created, which by default runs every 5 minutes (configurable through the "*appsettings.json*" file) and which reads the comparisons file and performs the consistency analysis;

• ObjectsComparer: package that allows class comparison, recursively verifying member by member. It also allows the creation of customized comparison rules for certain properties, fields, or types. This package was used to compare the objects under analysis.

8 Evaluation

This chapter aims to evaluate the implemented solutions. It defines the metrics to be used, specifies the indicators and sources of information, the research hypotheses to be tested and also describes the approach used to evaluate the solution.

8.1 Indicators and sources of information

This chapter presents the indicators and sources of information used to evaluate the solution. The indicators define which are the parameters/hypotheses that should be analysed. In turn, the information sources define who or what will provide the information needed for the assessment process.

8.1.1 Indicators

The relevant indicators for evaluating the solution are:

- Relevance: this indicator will allow defining whether the recommendations and solutions to challenges of data management and the data consistency monitoring tool are relevant to their target audience.
- Errors:
    - The number of errors: number of data inconsistencies that the tool can detect;
    - Relevance/importance of errors: detected faults, whether they are relevant or important.
- Time: the time required for the tool to perform data consistency analysis. This time may be very important in the diagnosis of possible existing technical failures.

8.1.2 Sources of information

So that each indicator presented in section 8.1.1 can be measured, it is necessary to have sources of information that provide the necessary data. The sources of information are divided by the two objectives. For the first objective, related to the proposal of recommendations and

solutions to the challenges encountered in the management of large volumes of data in microservices, the sources of information are:

- Professionals with great experience in the field of microservices. The opinion of these professionals is of great importance as they have extensive experience in working with microservices and in the management of large volumes of data.

Regarding the second objective, related to the creation of the prototype for analyzing data consistency in microservices, the sources of information are:

- Test scenario environment: testing environment with scenarios created exclusively to check the indicators;
- Environment with real scenarios: environment with production data from the case study company.

8.2 Research hypothesis

Each indicator defined in the previous chapter results in a hypothesis to be evaluated. These hypotheses, aim to validate the performance and quality of the solution.

This approach will be carried out following the hypothesis test model called Paired T-Test (Mcgready, 2006). In this model, types of hypotheses are defined:

• H0: null hypothesis, considered as the hypothesis that is intended to be rejected;

• H1: alternative hypothesis, considered as the hypothesis that determines the solution as valid.

A tool that allows data to be monitored is a way to identify potential problems and / or alert to aspects that may represent problems in the future.

The hypotheses for each of the indicated indicators will be presented below.

- Relevance:
    - H0: the data consistency monitoring tool and recommendations/solutions to challenges of data management are not relevant for professionals and do not bring advantages to their architecture;

- H1: the data consistency monitoring tool and recommendations/solutions to challenges of data management are relevant to professionals and also bring advantages to their architecture.

- Errors:
  - H0: the number of data inconsistencies detected by the tool is on average equal to or less than the number of data inconsistencies detected by the professionals.
  - H1: the number of data inconsistencies detected by the tool is on average higher than the number of data inconsistencies detected by professionals.

- Time:
  - H0: the time taken by the tool to perform data consistency analysis is equal to or greater than the current time it takes professionals to analyze data consistency for the same sample of data.
  - H1: the time taken by the tool to perform data consistency analysis is less than the current time it takes professionals to analyze data consistency for the same sample of data.

8.3 Evaluation methodology and results

This section presents the evaluation methodologies and their results. The tests and simulations performed as well as the input data necessary for the evaluation of the different hypotheses defined are also presented.

8.3.1 Evaluation of recommendations and solutions for data management

The set of recommendations and solutions for managing large volumes of data in microservices architectures will be presented to professionals with extensive experience in the microservices area.

The objective of the assessment will initially be to collect information about the profile of the participants to be able to describe in more detail their experience at a professional level, thus obtaining information such as years of experience in the area and their current role in the organization where works.

Each participant will be given the set of recommendations and solutions, and for each problem and respective recommendation/solution, the participant will evaluate according to the Likert scale (Likert, 1932) indicated in Table 7.

Table 7 – Likert scale

Strongly disagree	Disagree	Neutral	Agree	Strongly Agree
1	2	3	4	5

As indicated in the table, the evaluation is carried out on a scale of 1 to 5, with 1 indicating that the participant strongly disagrees with the solution presented and 5 indicates that he strongly agrees.

8.3.2 Evaluation of the data consistency monitoring tool

To evaluate the tool and according to the hypotheses, tests were performed on the tool in two different scenarios. The first scenario was in a testing environment, and the second scenario was the productive environment of the company under study.

8.3.2.1 Test environment

Before evaluating the tool in this scenario, it was necessary to generate a dataset in this environment. 10.000 objects were created in each of the three target analysis databases:

- SQL: a table named "*Products*" was created and was populated with 10.000 test objects. This table has the information present in Table 8.

Table 8 - SQL table "*Products*" of the test environment

Field	Type
Id	uniqueidentifier
Name	nvarchar(100)
Description	nvarchar(500)
PartNumber	int
CategoryId	uniqueidentifier
UpdatedDate	datetime

- MongoDB: a "*ProductCatalog*" collection was created and populated with 10.000 objects. This collection has the information present in Table 9.

Table 9 – MongoDB collection "*ProductCatalog*" of the test environment

Field	Type
Id	UUID
ProductName	string
ProductDescription	string
ProductPartNumber	Int32
ProductCategId	UUID
UpdateDate	Date

- ElasticSearch: an index called "*ProductBase*" was created and populated with 10.000 objects. This index has the information present in Table 10.

Table 10 – ElaticSearch index "*ProductBase*" of the test environment

Field	Type
ProductId	keyword
Name	text
Description	text
PartNumb	integer
ProductCategId	keyword
UpdatedDate	date

The 10.000 objects that were created in each database are all interconnected. Thus, it is possible to simulate what is required in microservices architectures where the same data can be replicated in several databases, as explained in detail in sections 4.2.2 and 6.3.5. These 10.000 objects were initially created with the same information in the three databases.

After populating the information in the databases, the comparison file was configured as shown in Figure 24.

```
{
    "DataComparisons": [
        {
            "Id": "57346120-9840-45df-b836-5295a1c507cb",
            "Comparisons": [
                {
                    "DatabaseType": "SQL",
                    "DataSourceName": "Products",
                    "CorrelationField": "Id",
                    "FieldsToCompare": "Name, Description, PartNumber, CategoryId",
                    "SortDescendingField": "UpdatedDate"
                },
                {
                    "DatabaseType": "MongoDB",
                    "DataSourceName": "ProductCatalog",
                    "CorrelationField": "Id",
                    "FieldsToCompare": "ProductName, ProductDescription, ProductPartNumber, ProductCategId",
                    "SortDescendingField": "UpdateDate"
                }
            ]
        },
        {
            "Id": "cf105d57-62c4-4e63-bb9a-7411ee2c343d",
            "Comparisons": [
                {
                    "DatabaseType": "MongoDB",
                    "DataSourceName": "ProductCatalog",
                    "CorrelationField": "Id",
                    "FieldsToCompare": "ProductName, ProductDescription, ProductPartNumber, ProductCategId",
                    "SortDescendingField": "UpdateDate"
                },
                {
                    "DatabaseType": "ElasticSearch",
                    "DataSourceName": "ProductBase",
                    "CorrelationField": "ProductId",
                    "FieldsToCompare": "Name, Description, PartNumb, ProductCategId",
                    "SortDescendingField": "UpdatedDate"
                }
            ]
        }
    ]
}
```


Figure 24 - Comparison file used in test environment

The file indicates that two groups of comparisons will be carried out. The first compares the SQL "*Products*" table with the MongoDB "*ProductCatalog*" collection, and the fields to be compared are those described in "*FieldsToCompare*". The second group, on the other hand, compares the "*ProductCatalog*" collection with the "*ProductBase*" index of ElasticSearch.

Then three tests were performed with different scenarios. Inconsistencies were created purposefully in the databases in:

- Scenario 1: 50 objects with inconsistencies;
- Scenario 2: 500 objects with inconsistencies;
- Scenario 3: 5.000 objects with inconsistencies.

For each scenario, the tool was executed and the following metrics were extracted:

- Number of inconsistencies detected;
- Data consistency analysis execution time.

Results

- Scenario 1:

Table 11 - Test scenario 1 results

Number of detected inconsistencies	50
Execution time	15 seconds

- Scenario 2:

Table 12 - Test scenario 2 results

Number of detected inconsistencies	500
Execution time	22 seconds

- Scenario 3:

Table 13 - Test scenario 3 results

Number of detected inconsistencies	5.000
Execution time	43 seconds

According to the results obtained in the three scenarios, it is possible to conclude that the number of inconsistencies detected by the tool had a 100% accuracy rate since the number of inconsistencies found is equal to the number of expected inconsistencies.

Regarding the execution time, it increased by several seconds, especially in scenario 3, in which the number of inconsistencies detected was much higher than in the others and which represented 50% of the total sample analyzed. In the best scenario, the execution took 15 seconds, and in the worst 43 seconds.

8.3.2.2 Productive environment

To complement the evaluation in the test environment and to be able to carry out a more reliable evaluation, tests were carried out in the productive environment of the company under study.

It is important to mention that the indicator related to the number of inconsistencies is much more difficult to quantify in this type of environment, as this result may be 0 inconsistencies. This value may not indicate that the tool failed because if the real environment is 100% consistent in terms of its data, the expected result is 0. Since this would be a limitation for the evaluation, tests were performed in a simulated environment (presented in the previous section) and complemented with the tests in the productive environment presented below.

Two data analyzes were performed in this environment:

- First analysis: data from an SQL table was compared with an ElasticSearch index. The results are shown in Table 14.

Table 14 - Results of the first analysis in real environment

Number of detected inconsistencies	0
Execution time	45 seconds

- Second analysis: data from an ElasticSearch index that is in use in the real environment were compared with another ElasticSearch index in the same environment but that is not in use and is incomplete (with missing data). The results are shown in Table 15.

Table 15 – Results of second analysis in real environment

Number of detected inconsistencies	3.037
Execution time	51 seconds

As explained, in the first analysis the number of inconsistencies detected was 0, which means that the data is 100% correct. Regarding the second analysis, 3.037 objects with inconsistencies

were found, resulting in a percentage of 30.37% of inconsistent data in the analyzed sample of 10.000 objects.

One of the improvement points of the tool is the optimization of the execution times to even lower values, but it is still within the desired values with the non-functional requirements. If this analysis were performed manually by professionals, it would take several hours to complete since they would have to analyze the entire sample of data manually, that is, analyze field by field and its consistency.

With these results, it is proved that the indicators used for the evaluation related to the number of errors and time of execution were fulfilled, allowing to refute H0, and it is valid to say that the obtained results are valuable and useful for its target audience.

9 Conclusions

This is the final chapter of this document and presents the conclusions, analyzes and compares the objectives defined initially, and compares with the work done. Difficulties encountered, limitations, and possible future work are also presented.

9.1 Achieved objectives

In this document, a value analysis was carried out in the area of microservices that allowed the identification of a specific problem. After defining the problem, the intended objectives for this work were defined, presented in section 5.2. In this section, the achievement of these objectives is assessed and why. Table 16 shows the different objectives and whether they have been achieved or not.

Table 16 – Objectives achievement

Number	Objective	Completeness
1	Propose a set of recommendations and solutions for challenges encountered in data management in microservice architectures with very high data volume, reaching hundreds of thousands of transactions per minute.	Achieved
2	Develop a prototype of a tool to assess the consistency of the data present in a microservice ecosystem.	Achieved

An extensive investigation was executed for each of the identified objectives. This investigation culminated in the realization of the state-of-the-art, where several articles related to the first objective and tools for the second objective were presented and analyzed.

This work contributed to the field of microservices with the presentation of a set of problems and challenges of managing large volumes of data in microservices and the proposed solution for them. These problems and solutions were explored through the case study of the company described in section 1.4. A meeting was also held with several professionals from that company to obtain their input on various topics addressed in this objective.

Regarding the second objective, several strategies and tools were analyzed, and the prototype explained in detail in chapter 7 developed, which allows the monitoring of data consistency in large volumes of data in microservices.

It is possible to affirm that these two objectives contributed to the area of microservices and the management of large volumes of data.

9.2 Difficulties along the way

Throughout the development process, some difficulties arose that directly or indirectly influenced the final result of this work. These difficulties were:

- Data confidentiality: due to issues related to confidentiality, it was not possible to present all the information collected in the case study and in the evaluation of the prototype in the real environment;
- Two complementary objectives: the fact that there were two objectives that complemented each other, doubled the time spent on its development, since each of the objectives had its analysis of the state-of-the-art, development and evaluation. So it was difficult to do this time management;
- Number of participants in the meeting: it was difficult to obtain a reasonable number of professionals present at the meeting presented in section 6.2 on the case study. These difficulties were in finding a day and time when everyone could be present, so it was only possible to have 7 professionals present;
- Time to develop the prototype: as two objectives were defined for this work, it was necessary to manage the time to successfully fulfill both. However, the second objective related to the prototype had a short time for development and therefore has some limitations that are explained.

9.3 Limitations and future work

Although this work has achieved the proposed objectives, there are some possible improvements and limitations that have been identified. These limitations were encountered during the development and had an impact on the final result of the work. Therefore, there is a possibility of improvement in the future.

Regarding the case study, a meeting was held with the professionals, however, it was only possible to have the contribution of seven professionals from two business clusters, because it was hard to schedule a date when more professionals from other business clusters were available. Their contribution would have been quite relevant since the company is divided into several business clusters and has thousands of professionals in the area, so these clusters would possibly have other problems and solutions for the topic in question.

Regarding the second objective, the prototype was implemented and can respond to what was intended, that is, it can detect data inconsistencies in microservices architectures. However, some limitations need to be worked out in the future so that this prototype can become an official tool used by companies. These limitations and improvements are:

- Although the prototype has a recurring job that monitors data at predefined time intervals, each analysis only analyzes 10,000 objects. This value was defined so as not to affect the performance of the databases of the respective microservices. However, it would be essential to be able to perform the analysis of a higher volume of data;
- In the tests performed, the average execution time for the analysis of 10,000 objects was 35 seconds. If a much larger volume of data were analyzed, this execution time would be even higher. There will then be a need to optimize this time in the future. One possible optimization could be the use of Threads that would be executed in parallel. Each thread would search for a batch of data and compare it in parallel with the other Threads;
- Impossibility of comparing objects with more than one level of depth. As a solution, recursion can be implemented in the future when comparing this type of objects;
- Creation of an API that allows the management of comparisons and the consultation of the history of analyzes performed. Thus, it would not be necessary to have the two input JSON files explained in section 7.2.4. With the creation of this API, it would also be possible to consult the results of the analyzes, allowing the removal of the output files currently generated for that purpose.

References

[1]. Barashkov, A. (2019). Microservices vs. Monolith Architecture. Retrieved January 5, 2020, from https://medium.com/pixelpoint/microservices-vs-monolith-architecture-c7e43455994f

[2]. Baškarada, S., Nguyen, V., & Koronios, A. (2018). Architecting Microservices: Practical Opportunities and Challenges. *Journal of Computer Information Systems*, *00*(00), 1–9. https://doi.org/10.1080/08874417.2018.1520056

[3]. Burckhardt, S. (2015). *Consistency in distributed systems*. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (Vol. 8987). https://doi.org/10.1007/978-3-319-28406-4_4

[4]. Di Francesco, P., Malavolta, I., & Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. *Proceedings - 2017 IEEE International Conference on Software Architecture, ICSA 2017*, (April), 21–30. https://doi.org/10.1109/ICSA.2017.24

[5]. Dynatrace. (2019). What is Dynatrace? Retrieved January 15, 2020, from https://www.dynatrace.com/support/help/get-started/what-is-dynatrace/

[6]. Evans, E. (2015). Domain --- Driven Design Reference.

[7]. Fan, W., Han, Z., Zhang, Y., & Wang, R. (2018). Method of Maintaining Data Consistency in Microservice Architecture. In *2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS)* (pp. 47–50). IEEE. https://doi.org/10.1109/BDS/HPSC/IDS18.2018.00023

[8]. Fowler, J. L. M. (2011). CQRS. Retrieved from https://martinfowler.com/bliki/CQRS.html

[9]. Fowler, J. L. M. (2014). Microservices. Retrieved June 8, 2020, from https://martinfowler.com/articles/microservices.html#DecentralizedDataManagement

[10]. Fowler, J. L. M. (2015). Microsserviços em poucas palavras. Retrieved from https://www.thoughtworks.com/pt/insights/blog/microservices-nutshell

[11]. Gibson, N. (2015). Getting Better Results with Google Scholar. Retrieved January 5, 2020, from https://uark.libguides.com/googlescholar

[12]. Grinshteyn, Y. (2019). An introduction to monitoring with Prometheus. Retrieved from https://opensource.com/article/19/11/introduction-monitoring-prometheus

[13]. Haselbock, S., & Weinreich, R. (2017). Decision guidance models for microservice monitoring. *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, 54–61. https://doi.org/10.1109/ICSAW.2017.31

[14]. Kanjilal, J. (2019). How to use the Dapper ORM in C#. Retrieved October 10, 2020, from https://www.infoworld.com/article/3025784/how-to-use-the-dapper-orm-in-c.html

[15]. Khine, P. P., & Wang, Z. (2019). A review of polyglot persistence in the big data world. *Information (Switzerland)*, *10*(4). https://doi.org/10.3390/info10040141

[16]. Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*. *O'Reilly Media, Inc.* Retrieved from https://www.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/%0Ahttp://shop.oreilly.com/product/0636920032175.do

[17]. Koen, P. A., Ajamian, G. M., Boyce, S., Clamen, A., Fisher, E., Fountoulakis, S., Johnson, A., Puri, P., & Seibert, R. (2001). Fuzzy Front End : and Techniques. *Industrial Research*.

[18]. Kumar, R. (2018). Selecting the Right Database for Your Microservices. Retrieved June 15, 2020, from https://thenewstack.io/selecting-the-right-database-for-your-microservices/

[19]. Likert, R. (1932). "A Technique for the Measurement of Attitudes" *Encyclopedia of Research Design*. https://doi.org/10.4135/9781412961288.n454

[20]. Mancill, T. (2018). 20 Best Practices for Working With Apache Kafka at Scale. Retrieved June 5, 2020, from https://dzone.com/articles/20-best-practices-for-working-with-apache-kafka-at

[21]. Mauri, H. (2020). Utilizando o Elasticsearch com NEST no .NET Core 3.1. Retrieved October 5, 2020, from https://medium.com/@hgmauri/utilizando-o-elasticsearch-com-nest-no-net-core-3-1-cd83d559dc5c

[22]. Mcgready, J. (2006). The Paired t-test and Hypothesis Testing John McGready.

[23]. Microsoft. (2018). Event-driven architecture style. Retrieved June 2, 2020, from https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/event-driven

[24]. New Relic. (2010). New Relic documentation. Retrieved from https://docs.newrelic.com/

[25]. Ntentos, E., Zdun, U., Plakidas, K., Schall, D., Li, F., & Meixner, S. (2019). Supporting Architectural Decision Making on Data Management in Microservice Architectures, 20–36. https://doi.org/10.1007/978-3-030-29983-5_2

[26]. Quinn, B. (2020). Datadog Review: Popular Infrastructure Monitoring Service. Retrieved July 12, 2020, from https://www.softwarepundit.com/datadog-review

[27]. Rocha, H. (2018). What they don't tell you about event sourcing. Retrieved June 15, 2020, from https://medium.com/@hugo.oliveira.rocha/what-they-dont-tell-you-about-event-sourcing-6afc23c69e9a

[28]. Rouse, M. (2019). What is data management and why is it important? Retrieved August 18, 2020, from https://searchdatamanagement.techtarget.com/definition/data-management

[29]. Runeson, P., & Höst, M. (2009). Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, *14*(2), 131–164. https://doi.org/10.1007/s10664-008-9102-8

[30]. Saaty, T. L. (2012). How to make a decision. *International Series in Operations Research*

and Management Science, *175*, 1–21. https://doi.org/10.1007/978-1-4614-3597-6_1

[31]. Shadija, D., Rezai, M., Hill, R., Li, S., Zimmermann, O., Ntentos, E., Zdun, U., Plakidas, K., Schall, D., Li, F., & Meixner, S. (2017). Understanding quality attributes in microservice architecture. *Computer Science - Research and Development*, *2018-Janua*(September), 20–36. https://doi.org/10.1007/s00450-016-0337-0

[32]. Smid, A., Wang, R., & Cerny, T. (2019). Case Study on data communication in microservice architecture. *Proceedings of the 2019 Research in Adaptive and Convergent Systems, RACS 2019*, 261–267. https://doi.org/10.1145/3338840.3355659

[33]. Soares de Toledo, S., Martini, A., Przybyszewska, A., & Sjoberg, D. I. K. (2019). Architectural Technical Debt in Microservices: A Case Study in a Large Company, 78–87. https://doi.org/10.1109/techdebt.2019.00026

[34]. Soldani, J., Tamburri, D. A., & Van Den Heuvel, W. J. (2018). The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software*, *146*, 215–232. https://doi.org/10.1016/j.jss.2018.09.082

[35]. Taibi, D., & Lenarduzzi, V. (2018). On the Definition of Microservice Bad Smells. *IEEE Software*, *35*(3), 56–62. https://doi.org/10.1109/MS.2018.2141031

[36]. Taibi, D., Lenarduzzi, V., & Pahl, C. (2018). Architectural patterns for microservices: A systematic mapping study. *CLOSER 2018 - Proceedings of the 8th International Conference on Cloud Computing and Services Science*, *2018-Janua*(Closer 2018), 221–232. https://doi.org/10.5220/0006798302210232

[37]. Tangirala, R. (2017). Polyglot Persistence Powering Microservices, 1–15.

[38]. Toyama, K. (2010). Introduction to the ACM digital library proceedings of ICTD2010. *ACM International Conference Proceeding Series*, (November).

[39]. Villaça, L. H. N., Azevedo, L. G., & Baio, F. (2018). Query strategies on polyglot persistence in microservices. *Proceedings of the ACM Symposium on Applied Computing*, 1725–1732. https://doi.org/10.1145/3167132.3167316

[40]. Xplore, I. (2019). What is IEEE Xplore? Retrieved from https://innovate.ieee.org/what-is-ieee-xplore/

Appendix A

Questionnaire - Data Management in Microservices Architectures

A systematic guide to data management in microservices architectures are being created in the scope of a master's dissertation for the Porto School of Engineering (ISEP).

This guide will be oriented towards microservices architectures that need to be prepared to manage a large volume of data, whose size and complexity of the data is very high, reaching hundreds of thousands of transactions per minute.
This complexity, which exists in many large companies worldwide, raises some concerns, directly and indirectly, related to their data.

For this reason, there is the need to identify and evaluate the common problems and challenges in managing large volumes of data in microservice architectures, how these problems are detected and what are the strategies adopted to deal with these problems.

Questionnaire expected time: 5 minutes

Thank you for your time !
* Required

1. How many years of experience do you have working in the area of software engineering? *

2. What is your role? *

   Mark only one oval.

   - ◯ Software Architect
   - ◯ Software Engineer
   - ◯ Infrastructure Engineer
   - ◯ Engineering Lead
   - ◯ Project Manager
   - ◯ Other: _____

3. What is your level of education? *

Mark only one oval.

- ◯ High School
- ◯ Master's Degree
- ◯ Bachelor´s Degree
- ◯ Doctorate Degree
- ◯ Other: _____

4. How many years of experience do you have working with microservice architectures? *

_____

5. How many years of experience do you have working with large amounts of data in microservice architectures? *

_____

6. How many microservices have you worked with? *

Mark only one oval.

- ◯ 1 - 5
- ◯ 6- 20
- ◯ 21 - 50
- ◯ 50+

7. Which databases have you already worked with? Select all that apply *

Check all that apply.

- [ ] Microsoft SQL
- [ ] MongoDB
- [ ] Neo4J
- [ ] Cassandra
- [ ] MariaDB
- [ ] ElasticSearch
- [ ] Cockroachdb
- [ ] PostgreSQL

Other: [ ] _____

8. What is the overall volume of data records in the databases you work with? *

Mark only one oval.

- ◯ 1 - 100K
- ◯ 100K - 500K
- ◯ 500K - 1M
- ◯ 1M - 25M
- ◯ 25M - 50M
- ◯ 50M+

9. What is the overall number of transactions per minute in each microservice you work with? *

Mark only one oval.

- ◯ 1 - 1K
- ◯ 1K - 5K
- ◯ 5K - 50K
- ◯ 50K - 250K

Appendix B – Technological framework

The current technological solutions related to the second objective of the dissertation are presented below. This objective refers to the monitoring of data consistency. Despite the study carried out on what exists in the data management literature, these technological tools/solutions can be of great importance within a microservice ecosystem, allowing to maintain all the data of the sub-systems monitored and sending the necessary alerts.

Monitoring remains a critical part of the management of any IT system, and the challenges associated with monitoring microservices are especially unique. Traditional monolithic systems are deployed as a single executable or library and have different points of failure and dependencies than those implemented with a microservice architecture. This type of architecture has different and extra intensive monitoring requirements. The business logic applied to a process is distributed among many separate services. Tracking an application requires data correlation from all of these different services (Haselbock & Weinreich, 2017).

As the data do not come from just one source, but many sources and as the data does not have a uniform format, a solution is needed that takes these aspects into account.

Data consistency is a classic problem in the field of distributed systems research. The diversity and complexity of data sources and business needs present many new challenges for data management.

Data integrity can be compromised in several ways, making data integrity practices an essential component of effective corporate security protocols. Data integrity can be compromised through:

- Human error, whether malicious or unintentional;
- Transfer errors, including unintended alterations or data compromise during transfer from one device to another;
- Bugs, viruses/malware, hacking, and other cyber threats;
- Compromised hardware, such as a device or disk crash;

- Physical compromise to devices.

Thus, it is essential to have a correct monitoring of data consistency so that potential consistency problems are detected in a timely manner.

Below are the approaches that are more related to the theme of this work and that meet the proposed objectives. To collect these approaches research of data monitoring tools and data consistency in microservices was carried out.

1. Monasca

Monasca provides monitoring as a highly scalable service solution, performance, and fault-tolerant. Monasca provides an extensible platform for advanced monitoring that can be used by its customers to obtain operational inputs about their infrastructure and applications (SDXcentral, 2019).

Monasca uses REST APIs for high-speed metrics, log processing, and queries. It integrates a streaming alarm mechanism, a notification mechanism, and an aggregation mechanism. This solution follows a microservice architecture, with several services divided into several repositories. Each module is designed to provide a singular service in the general monitoring solution and can be implemented or omitted according to the needs of operators/customers.

2. Logstach

Logstash is a tool for collecting, processing, and forwarding events and logs messages. The tool uses configurable input plug-ins to collect data. After the input plug-ins collect data, that information can be processed using numerous filters that modify and record the event data. Finally, Logstash allows events to be sent to outbound plug-ins that can forward events to a diversity of external programs, including Elasticsearch, local files, among others. (Elastic, 2019).

3. App Metrics

App Metrics is an open-source, cross-platform .NET library used to record metrics for an application. The App Metrics can run on .NET Core or the entire .NET framework. The app provides several types of metrics for measuring things such as the rate of requests, counting

the number of user logins over time, the time needed to run a database query, the amount of free memory and so onwards (AppMetrics, 2019).

4. **Dynatrace**

The Dynatrace solution allows in-depth application monitoring, with code-level visibility. Purepath technology makes it possible to track all orders within the infrastructure, from the user to the Central Systems / Databases. Monitor real user data, application performance, infrastructure, and cloud environments. Dynatrace also automatically detects all application dependencies and tracks transactions across all layers.
It supports the various current development environments, namely technologies involving microservices, containers, and also the technologies produced by the central cloud solution providers (Dynatrace, 2019).

5. **Nagios**

Nagios is a powerful monitoring system designed with scalability and flexibility that allows organizations to identify and solve problems in large and small computer networks before they affect critical business processes (Biswas, 2019).

The tool allows us to detect and repair problems in advance, mitigating the unavailability of systems that may affect end-users and customers.

Some features that Nagios offers to network administrators are:

- Plan infrastructure upgrades before outdated systems fail;
- Respond quickly to problems as soon as they are detected;
- Automatically correct problems when detected;
- Make sure that the organization's SLAs are met;
- Monitor the entire infrastructure and business processes.

6. Prometheus

Prometheus is a toolkit for monitoring and alerting open-source systems created on SoundCloud. Since its creation in 2012, many companies and organizations have adopted Prometheus, and the project has a very active community of developers and users.

Prometheus collects data in the form of time series. Time series are constructed using a pull model, in which the Prometheus server consults a list of data sources at a specific search frequency. Each data source displays the current metric values for that data source in the terminal consulted by Prometheus. The Prometheus server aggregates data into data sources (Grinshteyn, 2019).

7. New Relic

New Relic is an Application Performance Management (APM) used by teams for monitoring applications. The idea is to maximize productivity and minimize downtime by monitoring the application's statistics that indicate its overall performance.

Managing modern web applications requires attention to small details, as it is usually these small things that adversely affect the user experience. The New Relic application monitoring tool reveals these parameters so that developers and website owners can take the appropriate and timely corrective actions needed to improve application performance (New Relic, 2010).

8. DataDog

Datadog is a network monitoring tool that allows customers to gain visibility into the performance of their applications.

Datadog allows the identification of performance bottlenecks in code or infrastructure and monitor hosts or containers. The platform can automatically track requests across multiple libraries and structures and allows automatic instrumentation to collect extensions from front-end to back-end. It collects performance data from infrastructure components like Redis and Elasticsearch and offers integrations with web frameworks like Django, Ruby on Rails, and Gin.

Datadog provides real-time dashboards with combined metrics and events from connected applications, hosts, containers, and services (Quinn, 2020).

9. Grafana

Grafana is an open-source visualization and analysis software. It allows querying and viewing metrics, as well as sending alerts about those metrics. It provides tools for transforming a time-series database (TSDB) data into graphs and visualizations. Grafana connects with every possible data source, commonly referred to as databases such as Graphite, Prometheus, Influx DB, ElasticSearch, MySQL, PostgreSQL, among others.

The next table shows the comparison of the tools previously presented.

Table 17 – Comparison of selected tools

	License Type	**Metrics**	**Alert management**	**Software Type**	**Availability**	**Data consistency analysis**
Monasca	Apache 2.0 Free	yes	yes	Monitoring-as-a-service	Open-source	No
Logstash	Apache 2.0 Free	yes	yes	Framework	Open-source	No
App Metrics	Apache 2.0 Free	yes	yes	Library	Open-source	No
Dynatrace	eServices	yes	yes	Application	Closed-source	No

	License Type	Metrics	Alert management	Software Type	Availability	Data consistency analysis
Nagios	Free – limited to using up to 7 hosts. It has paid versions.	yes	yes	Application	Open-source	No
Prometheus	Apache 2.0 Free	yes	Requires connection with “Alertmanager” which in turn sends alarms	Application	Open-source	No
NewRelic	Multiple licenses	yes	yes	Software-as-a-service	Partial open-source	No
DataDog	BSD License	yes	yes	Software-as-a-service	Partial open-source	No
Grafana	Apache 2.0 Free	yes	yes	Application	Open-source	No

As can be analyzed, most tools have the “Apache 2.0” license type, which is a license that allows users to use the software for any scope, including modifying the software and distributing it. All of these applications have analysis and alarm metrics associated with them, and Prometheus needs to be configured to communicate with AlertManager, which is a tool that handles alerts sent for example by the Prometheus server. This deals with the grouping and routing of alerts for the correct integration of the receiver, such as email, PagerDuty, or OpsGenie. Also, it deals with the inhibition and silencing of alerts.

They all have different types of metric collections, such as collections of metrics for RabbitMQ, Kafka, MySQL, and MongoDB.

Dynatrace is the only one that is not open-source or partially open-source.

The metrics present in the tools have several types such as:

- Server uptime
- Average load
- CPU usage
- RAM usage
- Use of disk space
- Network usage
- Queries execution time
- Ping monitoring
- HTTP Checking to check if a page is available
- Service monitoring (Up/Down)