INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

MESTRADO EM ENGENHARIA INFORMÁTICA

A Guide for Microservices in Greenfield Projects

JOÃO PEDRO SANTOS CARDOSO

Junho de 2021

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

A Guide for Microservices in Greenfield Projects

João Pedro Santos Cardoso

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

Supervisor: Isabel Azevedo

Porto, 2021

Dedicated to everyone who supported me through this journey and made me the man I am today.

Abstract

Microservice-based architecture usage is growing, but most of the involved applications are monoliths then migrated to microservices. Although this approach has been considered the conventional wisdom, it is still unclear if an application based on microservices can be built from scratch or must start as a monolithic application to be broken into smaller blocks later in its lifecycle.

More unconventional approaches, such as starting with microservices, are not sufficiently explored, and reported experiences lack to guide the entire process. Thus, the problem to be addressed in this work is how to start with microservices from scratch. A guide was proposed to allow obtaining successful microservices solutions in greenfield projects. Following the Technical Action Research method, this set of guidelines was used to implement a microservices-based application without migrations.

The solution was evaluated using the Quantitative Evaluation Framework and the results were analyzed. The most important aspect for that type of architecture was accomplished, such as improved development organization, the quicker release of new features and fixes, lower cost on growing the system, and increased performance and resilience. The application of the guide resulted in a value of 74% for the global quality of the system using the mentioned framework, and a score of 76 by applying the Microservices Architecture Assessment Platform.

Keywords: Microservices, Greenfield Projects, Development Guidelines

Resumo

A utilização de arquiteturas baseadas em microserviços tem vindo a crescer, no entanto a maioria das aplicações envolvidas são monólitos que foram posteriormente migrados para microserviços. Este tem sido considerado o método convencional, apesar de que uma questão que ainda não possui uma resposta consensual é se uma aplicação baseada em microserviços deva ser construída de raiz ou começar como uma aplicação monolítica que é mais tarde partida em blocos mais pequenos.

Abordagens menos convencionais como começar com microserviços desde o início do projeto ainda não foram suficientemente explorados, e as experiências existentes pecam por não guiarem o processo completo. Por este motivo o problema a ser endereçado neste trabalho é como desenvolver uma aplicação baseada em microserviços desde o seu começo.

Será possível seguir um conjunto de diretrizes no desenvolvimento de um projeto greenfield de forma a obter uma solução baseada em microserviços bem-sucedida? Esta questão foi encarada como uma oportunidade e definida como o objetivo a ser explorado nesta tese. Seguindo o método Technical Action Research (TAR), foi desenvolvido um conjunto de diretrizes para implementar uma aplicação baseada em microserviços desde o seu início, sendo posteriormente aplicado a um protótipo de forma a ser melhorado e avaliado.

A solução desenvolvida foi avaliada usando o método Quantitative Evaluation Framework (QEF) e os resultados indicam que, seguindo as diretrizes criadas, é possível alcançar com sucesso uma solução baseada em microserviços. A avaliação permite afirmar que os aspetos mais importantes deste tipo de arquitetura tal como melhoria da organização no desenvolvimento, entrega mais rápida de novas funcionalidades e correções, custo mais baixo na manutenção e crescimento do sistema, e melhoria do desempenho e resiliência são cumpridos. A utilização do guia para desenvolver o protótipo resultou num valor de qualidade global do sistema de 74% aplicando o QEF, e uma pontuação de 76 ao utilizar a Microservices Architecture Assessment Platform.

Palavras-chave: Microserviços, Projetos Greenfield, Diretrizes de desenvolvimento

Acknowledgments

Firstly, I’d like to thank my advisor and mentor teacher Isabel Azevedo who constantly showed her support during this dissertation. Thank you for your knowledge, patience, and availability who made this dissertation a much more valuable work.

I also want to express my deep gratitude to my family and friends who always believed in me and gave me the strength to finish this important chapter of my life.

Last but not least, I’d like to thank every teacher and colleague who worked with me during my academic journey in ISEP. I truly feel very lucky for the opportunities and challenges I had during the past 6 years which made me achieve professional success and will continue to help me improve every day.

Table of Contents

1 Introduction 1

1.1 Context 1

1.2 Problem Statement 1

1.3 Objectives 2

1.4 Research Methodology 2
1.4.1 Main Outcomes 3

1.5 Document Structure 4

2 Background 7

2.1 Key Concepts 7
2.1.1 Distributed Systems 7
2.1.2 Service-Oriented Architecture 8
2.1.3 Microservices 9

2.2 Challenges on Microservices Development 10
2.2.1 Granularity 10
2.2.2 Decouple Data Storage 11
2.2.3 Security 11
2.2.4 Monitoring 12

2.3 Microservices Good practices 12
2.3.1 Align microservices with business capabilities of the domain 12
2.3.2 Distinguish API microservices from core microservices 12
2.3.3 Progressive Migration 13
2.3.4 Independent Microservices 13

2.4 Microservices Bad Practices 13
2.4.1 Defining microservice boundaries incorrectly 13
2.4.2 Single Layer Team Structure 14
2.4.3 Multi-service containerization 14
2.4.4 Poor documentation 14
2.4.5 Fallacies of Distributed Computing 15
2.4.6 Chain communication between microservices 16

3 State of the Art 17

3.1 Monolith First vs Microservices from Scratch 17

3.2 Building Microservices in Greenfield Projects 18
3.2.1 Software Engineering Systematic Mapping 19
3.2.2 Summary 26

3.3 Related Work 26
3.3.1 “Microservices Decision Guides” 27
3.3.2 “Microservices From Day One” 27
3.3.3 Design Models for Microservices 27
3.3.4 Summary 28

3.4 Related Technology 28
3.4.1 Service Recognition 28
3.4.2 Microservices Development 31
3.4.3 DevOps Implementation 32
3.4.4 Data Shaping 36
3.4.5 Monitoring 37

4 Guide Artifact Design 39

4.1 Preconditions 39

4.2 Solution 40
4.2.1 Step 1 - Should this guide be used? 41
4.2.2 Step 2 – Service Identification 42
4.2.3 Step 3 - Infrastructure and Communication 43
4.2.4 Step 4 - Monitoring 45
4.2.5 Step 5 - Data Storage and Management 46
4.2.6 Conclusion 51

4.3 Design Validation 51
4.3.1 Expected effects and value 51
4.3.2 Trade-offs 52
4.3.3 Sensitivity 52

5 Guide Application 53

5.1 Application 53
5.1.1 Step 1 – Should the guide be used? 53
5.1.2 Step 2 – Service Identification 54
5.1.3 Step 3 - Infrastructure and Communication 63
5.1.4 Step 4 - Monitoring 64
5.1.5 Step 5 - Data Storage and Management 67

5.2 Step 6 - Improvements 68
5.2.1 API Standards Documentation 68
5.2.2 Service Discovery and Registration 71
5.2.3 Observability and Resilience 74
5.2.4 Deployment Infrastructure 74

6 Evaluation 79

6.1 Investigation Hypotheses 79

6.2 Indicators and Sources of Information 80
6.2.1 Microservices Assessment Platform 80

6.3 Evaluation method 81

6.4 Application of the Quantitative Evaluation Framework 82

6.5 Results and Conclusions 87

7 Conclusion 89

7.1 Outcomes 89

7.2 Results 90

7.3 Limitations and Future work 91
7.4 Contributions of the work 92

Attachment 1 - Value Analysis 101

Opportunity Identification and Problem Analysis..101
Value Proposition - VP 101
Quality Function Deployment - QFD...105

Idea Genesis and Selection 109
Idea Generation..109
Analytic Hierarchy Process - AHP...110

Attachment 2 – Pharmacy Technical Director Interview 117

Attachment 3 – QFD House of Quality 119

Attachment 4 – Guide Application Domain Model 121

Attachment 5 – Pharmacity GraphQL Stitched Schema 123

Attachment 6 – Application of Quantitative Evaluation Framework 127

Attachment 7 – QEF Requirement Evaluation Criteria 129

Table of Figures

Figure 1 – Application of Technical Action Research 3
Figure 2 - Vertical and horizontal scaling cost comparison 8
Figure 3 - Chain Microservice Communication Anti-Pattern 16
Figure 4 - Monolith First vs Microservices from Scratch 18
Figure 5 - Systematic Mapping Process 18
Figure 6 - Screening process diagram 20
Figure 7 - Distribution of research papers by publication date 23
Figure 8 - Classification scheme based on keywording 23
Figure 9 - Context Mapper Framework Architecture 29
Figure 10 - Context Map Language (CML) Syntax Sample 30
Figure 11 - Rapid Prototyping Transformation Steps 30
Figure 12 - DevOps Model 32
Figure 13 - Docker Architecture 34
Figure 14 - GraphQL Architecture 36
Figure 15 - MetroFunnel main flow 38
Figure 16 - Microservices from Start Guide Diagram 41
Figure 17 - Microservices Identification Flow 43
Figure 18 - Communication type decision flow 44
Figure 19 - Simple Monitoring Solution 46
Figure 20 - Service Persistent Data Management Solution Flow 47
Figure 21 - Database by Service Pattern 48
Figure 22 - Event Sourcing Example 49
Figure 23 - Shared Database Pattern 50
Figure 24 - Application Use-Case Diagram 55
Figure 25 - Bounded contexts and aggregates diagram 58
Figure 26 - Service Cutter System Specification 59
Figure 27 - Service Cutter Markov's algorithm results 60
Figure 28 - Service Cutter Leung's algorithm results 60
Figure 29 - Service Cutter Chinese Whispers algorithm results 61
Figure 30 - Service Cutter Girvan-Newman algorithm results 61
Figure 31 - Microservices Definition Diagram 62
Figure 32 - Buy Flow Sequence Diagram 64
Figure 33 - Monitoring Solution Sequence Diagram 66
Figure 34 - Application Components Diagram 68
Figure 35 - Swagger Example 70
Figure 36 - GraphQL Schema Stitching 71
Figure 37 - Server-side Service Discovery 72
Figure 38 - Client-side Service Discovery 72
Figure 39 - Service Registration Decision Guidance Model 73
Figure 40 - Azure DevOps pipeline example 77

Figure 41 - Microservice Architecture Assessment Platform Results 87
Figure 42 – Value Proposition Canvas - Customer Profile 102
Figure 43 - Value Proposition Canvas - Value Map 103
Figure 44 - QFD Chart 106
Figure 45 - QFD Customer Requirements 106
Figure 46 - QFD Technical Requirements, Targets, and Relationship Matrix 107
Figure 47 – QFD Technical Requirements Correlation Matrix 108
Figure 48 - QFD Customer Competitive Assessment 108
Figure 49 - QFD Technical Competitive Assessment...108
Figure 50 - AHP Hierarchy Diagram...110
Figure 51 - AHP Fundamental Scale 111
Figure 52 - Pharmacity Domain Model 121

Table of Tables

Table 1 - Systematic Research Question 1 (RQ_1) 19
Table 2 - Systematic Research Question 2 (RQ_2) 19
Table 3 - Additional sources for systematic mapping research 20
Table 4 – Systematic Mapping Screening Criteria 21
Table 5 - List of papers after the screening process 21
Table 6 - Research Topics 24
Table 7 - Greenfield microservices concerns research 24
Table 8 - Avoidable steps to start greenfield microservices 24
Table 9 - Data Flows Between Microservices 63
Table 10 - Evaluation Indicators 81
Table 11 - QEF Dimensions, Factors, and Requirements Identification 85
Table 12 - AHP Criteria Pairwise Comparison 111
Table 13 - AHP Criteria Comparison Normalized Matrix 112
Table 14 - AHP Criteria Relative Priorities 112
Table 15 - AHP Maintenance Cost Alternatives Comparison 114
Table 16 - AHP Maintenance Cost Normalized Matrix 114
Table 17 - AHP Development Effort Alternatives Comparison 114
Table 18 - AHP Development Effort Normalized Matrix 114
Table 19 - AHP Modifiability Alternatives Comparison 114
Table 20 - AHP Modifiability Normalized Matrix 114
Table 21 - AHP Scalability Alternatives Comparison 115
Table 22 - AHP Scalability Normalized Matrix 115
Table 23 - Alternatives Priority by Criteria Matrix 115
Table 24 - Overall Alternative Priority Results 116

Table of Code Snippets

Code Snippet 1 - Dockerfile example for .NET microservice 75
Code Snippet 2 - Build microservice docker image 75
Code Snippet 3 - Kubernetes Deployment Configuration File Example 76
Code Snippet 4 - Kubernetes deployment commands 76

Acronyms and Glossary

Acronyms

ADD	Attribute-Drive Design
AHP	Analytic Hierarchy Process
AMQP	*Advanced Message Queuing Protocol*
API	*Application Programming Interface*
APM	*Application Performance Monitoring*
ARs	Architectural Refactorings
AWS	Amazon Web Services
CML	*Context Mapper Language*
CPU	Central Processing Unit
CQRS	*Command Query Responsibility Segregation*
CR	*Consistency Ratio*
DDD	*Domain Driven Development*
DSL	*Domain-specific Language*
HOQ	*House of Quality*
HTTP	*Hypertext Transfer Protocol*
IBM	International Business Machines Corporation
IT	*Information Technology*
LAN	*Local Area Network*
MAP	*Microservice API Patterns*
MDSL	*Microservice Domain-Specific Language*
NCD	New Concept Development model
OS	*Operating System*
POS	*Point of Sale*
QEF	*Quantitative Evaluation Framework*
QFD	Quality Function Deployment
RAM	*Random Access Memory*
REST	*Representational State Transfer*
SOA	*Service Oriented Architecture*
SOAP	*Simple Object Access Protocol*

SQFD	*Software Quality Function Deployment*
SQL	*Structured Query Language*
VP	*Value Proposition*
VSS	*Vienna Software Seminar*

Glossary

Bounded Context	*A logical boundary that defines tangible boundaries of applicability of some sub-domain* (Samokhin, 2017)
Brownfield	*"Refers to the development and deployment of a new software system in the presence of existing or legacy software systems"* (Wade, 2018)
Business Capability	*Business capabilities are the tangible and intangible building blocks of a business that gives it the ability to do what it does*
Common Closure Principle	*States that things that change together should be packaged together to ensure that each change affects only one service* (Noback, 2018)
Containerization	*Isolation of each application of the system in a virtual environment while allowing them to share a single guest OS/host* (Pallis et al., 2018)
Event Sourcing	*"Pattern that ensures that all changes to application state are stored as a sequence of events"* (Fowler, 2005)
Go	Go programming language is an open-source project developed by a team at Google and many contributors from the open-source community (Google, 2009)
Greenfield Development	*"Software development that begins with little or no legacy code base to build upon"* (Cervantes & Kazman, 2016)
Microservices First	A microservice-oriented development approach that consists of applying software engineering techniques with a microservices mindset since day 1

Polling System	*"A system where a single server visits a set of queues in some order"*(Boxma & Weststrate, 1989)
Saga	*A pattern where business transactions span multiple microservices as a sequence of transactions. The registration of the chain of transactions allows executing a set of fallback actions when one or more of the transactions fail* (Le, 2020)
Self-Contained System	*"Software architecture approach that focuses on a separation of the functionality into many independent systems, making the complete logical system a collaboration of many smaller software systems"* (Stranghöner, n.d.)
Service-Oriented Architecture	*"An architectural style that supports service-orientation. Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services"* (The Open Group, 2016)
Software Architecture	*"Software architecture is defined as the software infrastructure within which application components providing user functionality can be specified, deployed and executed"* (Solms, 2012)

1 Introduction

To make clear the expected impact of the work on certain business areas and the engineering community, this chapter describes not only its context and objectives, but also the problem to be solved and the impacts it may have on software projects, and engineers' daily work.

This chapter ends with a summary of the structure of the document and an overview of all its chapters.

1.1 Context

The interest in distributed software systems has been growing each year. Their potential has been proved and consequentially, the study and adoption of such architectures have been considered the future of modern software. One of the most common and trendy are microservices, and they are "becoming the next big thing in modern software development" (Trajanov, 2019). However, microservices implementation is not simple. and many companies try to implement this architecture in projects where it is not a good fit at all. That is why choosing the right architecture for a project and deciding the best way to implement it is such a critical step for the success of the whole project.

1.2 Problem Statement

As more companies adopt a microservice-based architecture for their applications, there is a question that still does not have a consensual answer: should an application based on microservices be built from scratch or start as a monolithic application, to be then broken into smaller blocks? Even though the approach that follows the "conventional wisdom" (Zdun et al., 2020) is to start with a monolithic application (Fowler, 2015), building a monolithic system considering that it will later migrate to microservices and preparing it for a linear transition has been seen as a challenge (Zdun et al., 2020).

There are many examples of migrations from a monolithic architecture to a microservice architecture (Nguyen, 2020) (Bucchiarone et al., 2018) (Bjørndal et al., 2020), but very little for other approaches (Lumetta, n.d.). More unconventional approaches such as starting with microservices were not sufficiently explored, and reported experiences lack to guide the entire process. Therefore, the problem to be addressed in this work is how to start with microservices from scratch. It will contribute to the field with a documented experience available to the entire community. This will enable its replication and improvement in future developments and projects.

1.3 Objectives

The goal of this work is to facilitate the development of an application with a microservice-based architecture from its beginning.

Firstly, it is necessary to investigate how microservices are being implemented by companies around the world and understand what the major challenges have been faced by developers. With that information, a guide with a set of good and bad practices for microservices development should be implemented.

Based on the collected guidelines it will be presented an approach to develop a greenfield project based on a microservices architecture. As stated by Cervantes, greenfield projects are those that "begin with little or no legacy code base" (Cervantes & Kazman, 2016).

Thus, the objective is two-folded:

(1) Assess how the software engineering community is approaching the development of microservice oriented architectures with a focus on greenfield projects and gather the good and bad practices into a set of guidelines
(2) Propose an approach to implement microservices from scratch based on the previously collected recommendations

1.4 Research Methodology

This work is conducted by investigating the details of the problem, designing a solution (the guide), and validating it by applying it in a real case scenario. For this, the Technical Action Research (TAR) method was applied.

By definition, the Technical Action Research technique "is the use of an experimental artifact to help a client and to learn about its effects in practice" (Wieringa, 2014). It starts with the design of the artifacts applied "under idealized circumstances in a laboratory" and later scaled up to bigger scenarios with more realistic problems. This allows the designed solution to be validated (Wieringa, 2014).

Thus, two engineering cycles were identified. In the first one an artifact (a guide to start with microservices) was developed (see chapter 4), while in the second it was used in the development of an application to validate and improve the artifact (in chapter 5).

In the context of this work, multiple problems are correlated with the microservices-first development method. Given the nature of TAR, which "is technology-driven, not problem-driven", it allows the research and implementation of the solution to focus on the big problem of developing microservices from scratch while desiring "to solve a class of problems".

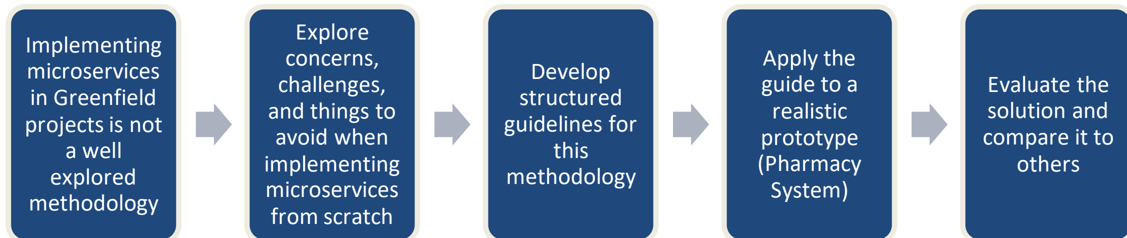


Figure 1 – Application of Technical Action Research

In summary, the research was conducted following the flow of Figure 1. After the bigger problem was identified (poor documentation on how to implement microservices from scratch on greenfield projects), a solution to enrich the development community was designed. It consists of a literature review on microservice challenges, good, and bad practices, followed by the specification of a structured guide to help developers having a base to use this methodology. According to Technical Action Research, after having designed this artifact, it is time to scale it from lab testing to a realistic scenario. For this, a business area that could benefit from this work was identified (Community Pharmacy Systems), and a prototype was implemented to apply the designed artifact. An interview with a person in the industry was conducted to understand which were the major concerns of the existing systems so that they can be tackled on the prototype (this is documented in Attachment 2).

1.4.1 Main Outcomes

Therefore, the main outcomes for the documented work are as follows:

- Identify companies and people who were involved in projects of different domains and complexity and their approaches to the development of applications that follow an architecture based on microservices. Their results will be considered, as well as the difficulties they experienced or would anticipate in starting with microservices. Depending on what is published, several professionals might also contribute through interviews or surveys. The requirements of the solution, partial or complete, to be developed for the problem will be outlined, including business area(s), application size, and resource availability, among other factors.
- Explore microservices adoption and how to enable microservices architecture in greenfield projects, the techniques and patterns (design and technological concepts) that can be considered, and what to avoid (bad practices and antipatterns).

- Analyze how the set of previously studied concepts can be aggregated and what makes sense in an initial phase and what is excessive to be considered too early. The problems that others have documented and the available antipatterns will be reanalyzed to identify possibilities and their trade-offs.
- Considering additional factors such as the timeframe for this project, the nature of the requirements, the flexibility of the technology stack, specify the steps to the microservices-first strategy considering one of the approaches previously analyzed.
- The steps will be refined with more details and additional considerations through their application in a case study that meets the applicability conditions of the approach. Thus, the approach will be used for the development of a software application that follows the microservice architecture design principles after the analysis of all its requirements. These steps, as well as their inputs and outputs, and the necessary conditions to advance to the next step (or finish at that step) will be rethought.
- Find if the followed approach was successful, namely if the project is complete and works as expected, the application is in conformance with microservice patterns. What was obtained through the approached application can be characterized as a microservices-based solution? The limitations and possible improvements will also be analyzed. The findings and alternatives previously discarded will be made available to receive feedback on the approach, in a more extended evaluation over time, and to allow its evolution even by other researchers.

1.5 Document Structure

The structure of this document consists of 7 chapters, bibliographic references and several attachments. This first chapter contains 5 sections that give the context of the document, expose the problem to be solved and the specific objectives to achieve it, and ends detailing the used research methodology and the document structure. The remaining chapters are as follow:

- **Background:** this chapter includes a brief explanation of some of the key concepts in the scope of this document and the summary of the existing literature on the challenges, and good and bad practices on microservices development.
- **State of the Art:** here is presented the most recent research about microservice development methodologies, and the specific case of building microservices in greenfield projects. The chapter ends with the research of some of the most important technologies related to microservices development.
- **Design:** it presents the requirements to be able to use the designed solution for the problem and the solution itself. This includes the guide to implement microservices in a greenfield project and the requirements to be able to follow it successfully.

- **Guide Application:** it documents the application of the designed solution in an exemplifying greenfield project.
- **Evaluation:** it presents the indicators and respective sources of information and evaluates the quality of the developed solution using the Quantitative Evaluation Framework (QEF).
- **Conclusions:** shows the global outcomes and results, identifies some limitations and future work, and ends emphasizing the main contributions of the work of the dissertation.

2 Background

This chapter is dedicated to the documentation of some of the most important concepts within the scope of this dissertation. It aims to help the reader to have a better and smoother experience throughout the rest of the document.

2.1 Key Concepts

2.1.1 Distributed Systems

In the ‘80s, authors were already investigating and publishing about the importance of studying distributed systems (Kleinrock, 1985). At that time, the motivation for this field was different but the main concern was the same, performance. Personal computers had begun to proliferate and with the adjacent technological evolution, the need for parallel processing and distributed databases became a reality. In a special issue, (Kleinrock, 1985) even compared distributed systems to natural phenomena as an example of how familiar everyone is with the topic:

> “How did the killer bees find their way up to North America? By what mechanism does a colony of ants carry out its complex tasks? What guides and controls a flock of birds or a school of fish? The answers to these questions involve examples of loosely coupled systems that achieve a common goal with distributed control.”

The computers’ hardware as we know it today went straight through that evolutionary path. A big example of that is unless you are reading this on paper, the CPU of the device you are looking at right now. Except for embedded systems and other microchip users, nearly every computer, mobile device or smart tv uses multi-core processors that take advantage of parallel computing.

As a consequence of the evolution and importance of the web, the study of distributed systems started to change its focus to systems such as clusters of computers with much lower granularity. As stated before, systems are always distributed by the need for performance to conducting

more complex and demanding tasks. And, since computers suffered a great evolution, they became powerful enough to handle increasing levels of traffic. This process of upgrading hardware is called vertical scaling, and it works well whenever it is possible. The problem is that after reaching a certain point, even the best hardware won't be capable of handling enough traffic (Kozlovski, 2018). However, the implementation of distributed systems enabled horizontal scaling. This technique consists of adding more computers communicating and working in parallel with the same objective. Besides potentially allowing to scale to infinity, it also brings other advantages such as cost reduction at a certain point (see Figure 2) and fault tolerance (Kozlovski, 2018).

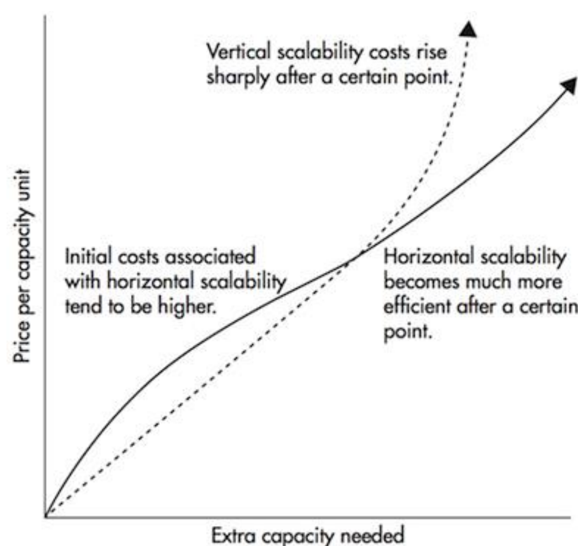


Figure 2 - Vertical and horizontal scaling cost comparison

Source: (Kozlovski, 2018)

Although using distributed systems seems to be an excellent way to create highly available and fault-proof software, for that to work, the software being executed on multiple computers in parallel needs to be capable of handling the problems that come with that type of architecture. These problems will be identified and analyzed during the rest of this document for one specific distributed architecture.

2.1.2 Service-Oriented Architecture

Service-Oriented Architecture, mostly known as SOA, has become an extremely popular approach to develop distributed applications with a deep focus on the business processes. Practically, an SOA consists of groups of software components that carry out these processes (Datz, 2004). Defined by (High et al., 2005): "The primary goal of Service Oriented Architecture (SOA) is to align the business world with the world of information technology (IT) in a way that makes both more effective".

In SOA, services should be loosely coupled, which means that temporal, technology and organizational constraints in the information system should be avoided or encapsulated. Also, each application in the system should be able to communicate and interact with the remaining without knowing their technical details (High et al., 2005).

A service in SOA is considered any task of the business process that is repeatable. Identifying the appropriate granularity and constructing the services resultant from the business design may be a complex process and therefore a few techniques were created to facilitate it. One of these techniques is offered by IBM and is called "The Service-Oriented Modeling and Analysis".

As defined by Arsanjani the service-oriented modeling “provides modeling, analysis, design techniques, and activities to define the foundations of an SOA” (Arsanjani, 2004). The functioning of this technique follows a top-down analysis that “helps by defining the elements in each of the SOA layers and making critical architectural decisions at each level”. This allows to “conduct conducts service identification through leveraging legacy assets and systems.”.

The layer-by-layer analysis results in the externalization of high-level business processes into large-grained services, the existing legacy functionalities are examined to potentially be converted into smaller-grained services, and in a final step, using goal-service modeling, there is a refining of every identified candidate service.

2.1.3 Microservices

This is one of the most discussed subjects in software architecture lately and it is a style that has been used in many projects in the last years. As summarized by Lewis and Fowler the microservice architectural style is:

[…] an approach to developing a single application as a suite of small services, each running in its process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies. (Lewis & Fowler, 2014)

Each microservice is considered by (Thönes, 2015) “[…] a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility”.

2.1.3.1 Architecture Adoption

When talking about companies that adopted microservices, the first name that must come to everyone’s mind is Netflix. This giant tech company is considered one of the pioneers of the microservice movement since it decided to break its monolith into microservices back in 2009 (Ismail, 2018). Amazon is another example of a company that started as a huge monolith and now “is perhaps the world’s most prominent advocate of microservices, with Amazon Web Services (AWS) providing the infrastructure needed for companies to launch and manage containers and microservices on the fly” (Ismail, 2018). Along with these two, many other

famous companies such as Uber, SoundCloud, Karma, and Groupon also struggled to maintain monolithic applications and ended adopting a microservice architecture.

The main reasons that led all these companies to their decision were the following monolithic architecture's issues:

- Difficult company growth
- Server outages, due to high traffic
- Long "development to production" process time

2.2 Challenges on Microservices Development

Even though the concept is not new, deeper studies on the good and bad practices of microservice implementation increased in recent years. This happened because the adoption of the architectural style by big companies like Netflix, with unseen traffic levels, revealed challenges that were inexistent until then. This section gathers the main findings of scientific research and practical cases in real companies.

Microservice growth happened at the same time that traditional SOA (using HTTP SOAP) started to decline and be replaced by HTTP REST. This would set a trend for the way microservices expose their functionalities, but it is not a challenge by itself. The issue is that it was considered as the default way to expose them rather than considering asynchronous messaging alternatives and it may lead to various challenges of performance and complex transaction management (Gupta & Palvankar, 2020). This is one of many challenges faced when developing and growing a microservice-based software solution. Next are described some of the other challenges found in this kind of software development. It is very important to understand them before deciding whether to follow the microservice approach right from the start or not. If the goal of choosing this architectural style is to allow scalability in the long term, these are challenges that the team must be ready to overcome.

2.2.1 Granularity

Software architects have been struggling with the uncertainty on the right level of granularity for a system component's scope. The maximization of the reuse of the functionalities considered for a system component and defining its boundaries correctly are considered by (Gupta & Palvankar, 2020): "[…] two of the biggest challenges for a solution designer […]" since these have two major dependencies: "[…] the domain and context of the solution space ".

The single responsibility and right size of a microservice are topics that appear in many definitions such as (Thönes, 2015) and (Pacheco, 2018), but it is important to understand what these authors mean by that.

People ask how big a microservice should be, but the answer for that is never right or wrong because it depends on several factors. When asked that question, (Thönes, 2015) affirmed that

he had seen microservices with hundreds of lines of code but others that could reach a couple thousand. The advice he often gives is that more important than the number of lines of code is that the "lines of code do one thing only" and that "everyone should be able to understand them". A recommendation to understand if the code has a single responsibility is to guarantee that there is only one reason to change it. (Thönes, 2015) leaves one last piece of advice that is related to the tradeoff between a larger microservice or breaking it into two smaller ones. When a microservice reaches the border of the single responsibility, it is a sign that "you got to the point where numbers become important" and a decision over whether what is best for your company: support bigger but fewer microservices or smaller but in larger number? This is an important question because the operational costs of each microservice are a relevant aspect to consider (it usually involves more physical resources and deployment time if the process is not automated).

2.2.2 Decouple Data Storage

This challenge comes from the need to decouple the different services in a microservice architecture on all layers. Different from a monolithic approach where a large database is shared between all the application's components, in a microservice architecture, each service should have its storage with its scope. As affirmed by (Gupta & Palvankar, 2020): "If multiple microservices are tied to the same tables in the database, then any change in the schema will result in cascading changes in other microservices, which defeats the core purpose of microservices".

It means that when developing microservices there must be special attention to the way the entities relate with each other and how often these relations need to be verified since the data of each microservice needs isolation. Thus, the retrieval of that information is no longer simple database access but a request to another service and the mapping between identifiers.

2.2.3 Security

Although the adoption of microservices has been growing at an incredibly rapid pace in the past years, Yarygina remembers that "there has been surprisingly little focus on security" (Yarygina, 2018).

Security poses a challenge in microservices since most applications are migrations from a monolith and they end up exposing code that was not designed to be accessed externally (Yarygina, 2018). This raises a lot of security concerns but can be avoided if microservices are developed from the start of the project.

Other challenges that engineers may need to face with microservice security are establishing trust between each service, dealing with distributed secret management, and making the security solutions scalable so they can "fit in with the overall approach and be adopted by industry users" (Yarygina, 2018).

2.2.4 Monitoring

The adoption of microservices leads to a "[…] more complex application topology", and involves scaling and dynamic environments. These characteristics make the traditional monitoring techniques unsuitable for this type of architecture (Golden, 2020). Since it is usual that each microservice has distinct logs and may use different technologies, it can also "exacerbate format and semantic heterogeneity of logs", which makes the monitoring process even harder for this kind of system (Cinque et al., 2019).

Since in microservices the information flows through multiple applications, it may lead to difficulties to get the wholistic view of the system. This makes monitoring even more important and without it, failures are more probable to occur without timely warning (Meng et al., 2017).

The challenge when implementing monitoring strategies for microservice-based architectures is that it is not enough to track each service separately. It is necessary to ensure that all the components and services of the system are working together and delivering the expected user experience to the user. Kowall calls this "end-to-end monitoring" and associates the concept with two requirements:

- Measuring client performance and operations on mobile and browser devices.
- Following the entire flow of system interactions to perform the transactions, "[…] from the device through the associated microservices and any third-party components that are required to fulfill the user request and drive the user experience." (Kowall, 2020).

Despite the importance of monitoring in microservices, a systematic mapping study conducted by Waseem reveals that this is still an inadequately explored area and more research effort is required. It reported that during the study only 4 out of 47 entries presented "monitoring approaches", "frameworks", and "performance monitoring challenges" (Waseem et al., 2020).

2.3 Microservices Good practices

2.3.1 Align microservices with business capabilities of the domain

When building microservices, (Gupta & Palvankar, 2020) consider it a good practice "to think in terms of loosely coupled, event-oriented boundaries and the context mapping". This implies the identification of context-dependent interactions that each microservice needs to be involved in and therefore ensures that it is built aligned with the business capabilities of the domain.

2.3.2 Distinguish API microservices from core microservices

To avoid "REST overuse" the design of microservices should have the notion of two major types of microservice and the distinction between them must be clear for everyone involved. On one hand, there are API microservices that are responsible to expose their functionalities to the

outside of the system. These are expected to follow a REST API approach to expose all functionalities in a universal and well-documented way. On the other hand, there may exist microservices that are only expected to be used inside the system and shouldn't be public (only accessible by other microservices). These are commonly called "core microservices" and should be implemented on a reactive pattern to avoid the "REST overuse" pitfall (Gupta & Palvankar, 2020).

2.3.3 Progressive Migration

Migrating a big monolithic application into microservices is without any doubt a very complex and risky process. Thus, it is a good practice to migrate the application incrementally, each service at a time. The services to migrate should be carefully identified using DDD techniques to guarantee that there is only one bounded context for each microservice. After this identification, the services should be gradually removed from the monolith and extracted into a new smaller application. By repeating this process, the complexity of the monolith will be gradually decreasing and in the end, the monolith will be transformed into microservices (Carrasco et al., 2018).

2.3.4 Independent Microservices

By definition, microservices are small and independent applications. Thus, it is clear that independently implementing and maintaining the services must be considered a critical good practice. It promotes fast feature development and delivery, and facilitates DevOps because, being the applications autonomous, each team working on these services will also have more autonomy. It allows the development, test and release process to scale more easily (Pallis et al., 2018). In the long term this is critical to an efficient evolution of the system.

2.4 Microservices Bad Practices

2.4.1 Defining microservice boundaries incorrectly

One of the biggest struggles when designing a microservice architecture system is defining each microservice's scope. According to (Gupta & Palvankar, 2020) there are three major variations of incorrect microservice scoping:

- **Entity Scope:** orienting the microservice design to an entity level tends to cause inflexibility of each microservice which consequentially adds complexity to manage the system as a whole
- **Process scope:** since processes tend to change, microservices designed with that mindset tend to be volatile and fragile to business changes

- **Utility scope:** The notion of utilities treated as microservices causes the provisioning of operational sophistication without reaping adequate business benefits since such utilities are often non-differentiating to most enterprises

2.4.2 Single Layer Team Structure

Many organizations' structure is characterized by a division of teams by layer (e.g. UI, Business Layer, Databases, etc.). This may lead to two problems: it adds time and effort for approval whenever a change is needed and at the same time there may be a tendency for teams to force their responsibilities on multiple services and therefore make them dependent. These concerns go against one of the most important microservices principles, the independence of each service (Carrasco et al., 2018). To solve this issue, companies should adopt a structure where teams are cross-functional and each one is assigned to a specific service. That allows each team and service to be fully independent.

2.4.3 Multi-service containerization

When working around microservices, DevOps and Continuous Delivery are techniques that are usually aligned with their goals, since the deployment of highly scalable systems in an automated and fast way is a must in this type of architectural style. To accomplish this, service applications are usually packed in containers with their dependencies which avoids the complexity of management and orchestration. The problem is when multiple services are packed in the same container to simplify the setup and configurations of the system. This will make the containers heavier and remove the independence between microservices (Carrasco et al., 2018). The correct solution for this problem is to follow the container per service rule, configuring each container to include only one service and its dependencies. This way each service is independently deployed which is faster and can be separately scaled which is one of the biggest advantages of the microservices.

2.4.4 Poor documentation

Overall, documentation of software systems is a very important step to increase its maintainability. In microservice-oriented applications, this is even more important given that each service usually exposes an API which contract is crucial to assure cooperation with other services' teams. Because of time constraints, teams may tend to neglect their services' documentation, but since the number of services in a system can grow abruptly, it can have very negative impacts, such as the global vision of the application getting lost (Carrasco et al., 2018).

2.4.5 Fallacies of Distributed Computing

Some of the most common assumptions developers make when migrating to microservices are explained in (Rotem-Gal-Oz, 2008). They are known as “The 8 Fallacies of Distributed Computing”:

- **The network is reliable:** power, hardware, and software failures are real for networks, as well as security threats. This means that networks are not reliable at all, and consequentially it implies that software engineers and architects implement design restrictions in terms of infrastructure and software. There is the need to create redundancy of the hardware devices and at the same time, developers should be aware that communication messages may get lost and therefore need to implement resilience techniques on the software as well. Some possibilities to mitigate communication issues are the usage of retries, validate message integrity and duplicate messages if necessary.
- **Latency is zero:** distributed architectures like microservices tend to abstract developers on how components communicate. This takes them to think that every call is instantaneous, which isn’t probably true since most components communicate over the network. Even if they work on a LAN, the latency is still much bigger than accessing local memory.
- **Bandwidth is infinite:** to prevent the previously presented fallacy, developers frequently try to avoid making extra calls by grouping large amounts of data in one single request, but they usually forget that bandwidth has its limitations too. There should be a balance between these two fallacies, and they should be tested together by simulating the production environment during development.
- **The network is secure:** even though security evolves faster each day, the number of attacks on companies and software has also grown absurdly in the past years. That means that specific security measures should be implemented in software solutions since day one. These security measures should be applied in network, infrastructure, and application levels and be sustained by detailed risk-cost analysis.
- **Topology doesn’t change:** network topology tends to be very volatile once in a production environment, both on the client and the server-side. Therefore, the physical structure of the network should be abstracted in the software solution.
- **There is one administrator:** even if the evolution of software development is in the direction that developers tend to be less dependent on the infrastructure and its administration, there are still problems that natively are only diagnosable by IT administrators such as server-level errors. Software developers should implement monitoring techniques and provide their infrastructure administrators with tools so that they can get support when needed.
- **Transport cost is zero:** this fallacy can have two interpretations: the first resides in the fact that many developers forget the cost of serialization when moving data from the application to the transport layer. This process converts from the application format into transportable bytes, but to do that, resources are consumed, and additional time is added to the already existent latency. On the other hand, this fallacy can be interpreted as

development teams thinking that the transport layer (setup and network maintenance) is free. There are lots of costs associated with network management such as hardware, security, bandwidth lease, and other operational costs.

- **The network is homogeneous:** when architects think that all the devices connected to the network are homogeneous, they are falling into this fallacy. Lately, this is not a big problem since most of the communication protocols are universal and allow interoperation of different devices but, at a lower application level, this can be relevant and needs to be taken into account by using standard/widely accepted technologies.

2.4.6 Chain communication between microservices

Despite choosing a synchronous or asynchronous communication protocol for the services of a microservice application, the important thing is to be careful about the way these services are integrated. According to (Anil, 2020): “The fewer communications between microservices, the better”, meaning that architects should work to minimize the communication between the microservices of the application. This is because if they end up chaining too many microservices in a single request-response as shown in Figure 3, they are creating two big problems: first, dependencies between the services, and second, and even worse, if one of the services in that chain isn't performing well, the whole request/response chain will be affected (Anil, 2020).

Synchronous vs. async communication across microservices

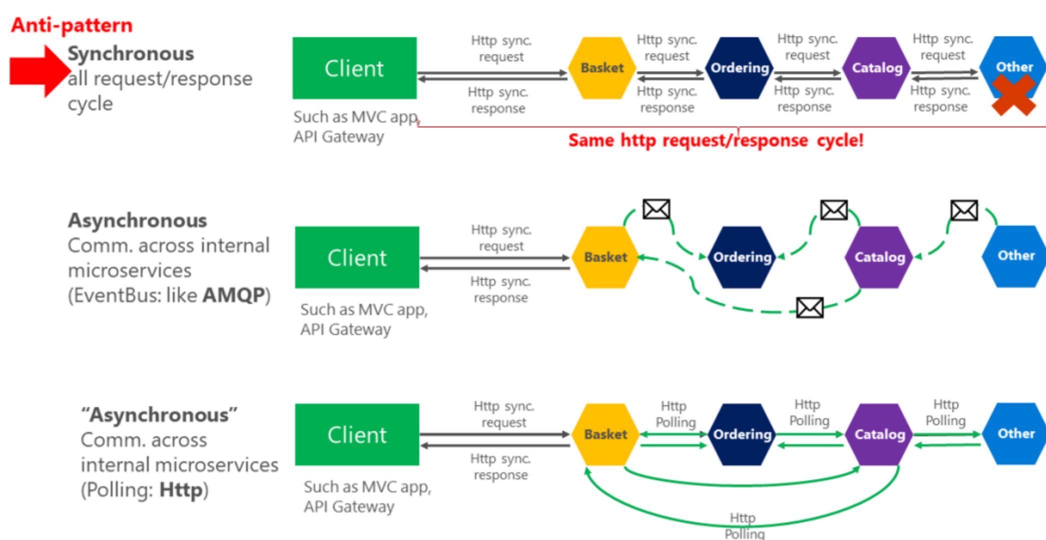


Figure 3 - Chain Microservice Communication Anti-Pattern

Source: (Anil, 2020)

3 State of the Art

This chapter details the results of a research into microservice-oriented architecture methodologies and their guidelines. This includes a literature review of how they have been implemented in greenfield projects and their known problems.

3.1 Monolith First vs Microservices from Scratch

The opinions over the best approach to develop a new microservice-based application are divided into two main approaches: Monolith First and Microservices from Scratch. Figure 4 shows a very simple comparison between the two approaches.

As explained in previous sections, almost every successful microservice stories have started with a monolith that got too big and was broken up (Fowler, 2015) and that is why it is considered the "conventional wisdom" for building microservices. There are even authors that defend that "the only way to get to a successful microservices architecture is by starting with a monolith first" (Tilkov, 2015). The logic behind this statement resides in the fact that if engineers are not skilled enough to build a monolith, they wouldn't be able to build the same project following a microservice approach from the start. It is believed that in the majority of cases it is easier to migrate an existing "brownfield" than starting with a new "greenfield" for two main reasons: firstly, it is possible to analyze a working system for what is well implemented and what can be improved, and secondly, since the system is running in production it is possible to benchmark the performance of the new approach compared to the current one (Brown, 2014).

On the opposite side, Tilkov defends that in the majority of the cases it is very difficult to migrate the pieces of a complex monolith into new separate components, but also agrees that the reference of an existing project can bring good inputs for the creation of a microservice system. He concludes that "the ideal scenario is one where you're building a second version of an existing system." (Tilkov, 2015).

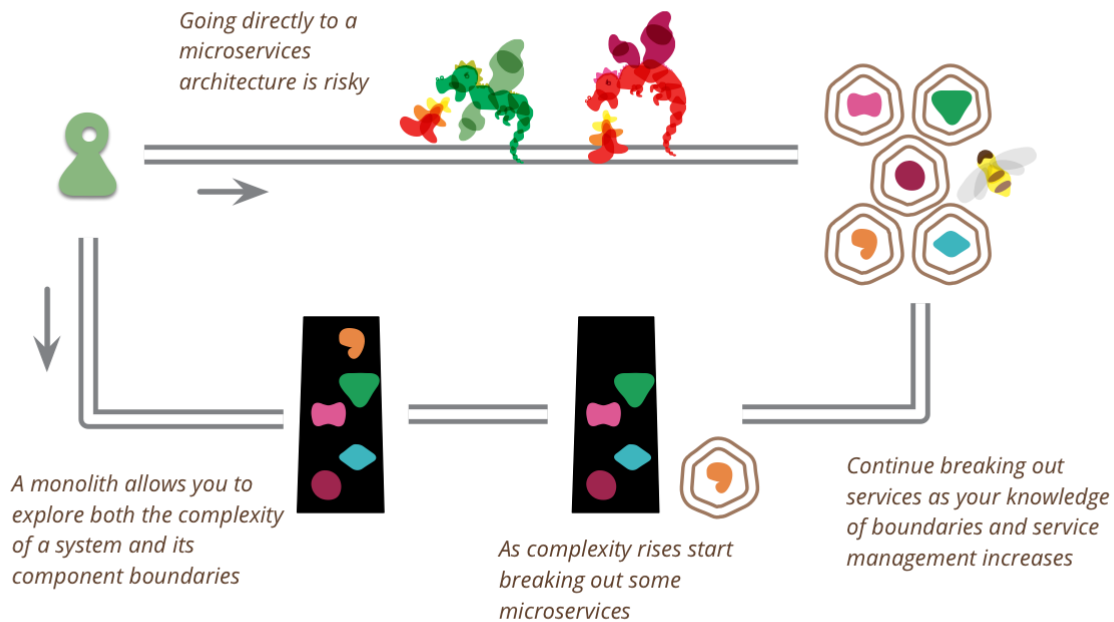


Figure 4 - Monolith First vs Microservices from Scratch

Source: (Fowler, 2015)

3.2 Building Microservices in Greenfield Projects

Greenfield projects are a synonym of launching a brand new project, which means that "everything has to be built from the ground up, from defining services, functions and product range, to deciding the technology, infrastructure, and resources required" (WaveMaker, 2019).

Additionally to the challenges, and good and bad practices detailed in sections 2.2, 2.3, and 2.4, building microservices for a small greenfield project brings more concerns. To explore microservices development focused on greenfield projects in more detail, systematic mapping research (systematized in Figure 5) was followed.

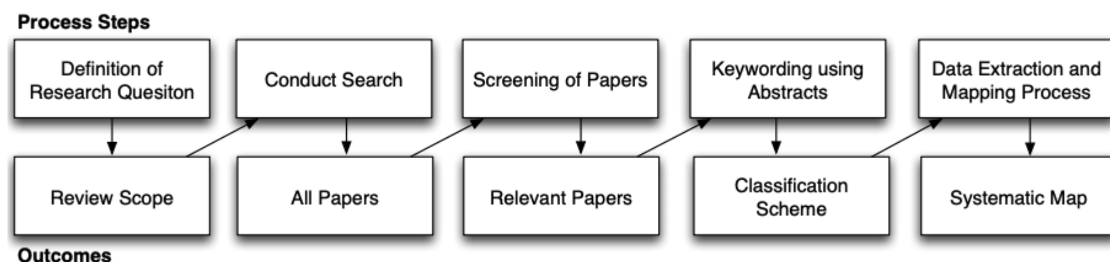


Figure 5 - Systematic Mapping Process

Source: (Petersen et al., 2008)

3.2.1 Software Engineering Systematic Mapping

In the scope of software engineering, a systematic map is a method that aims to build a classification scheme and structure a certain field of interest. This method allows determining the coverage of the research by analyzing the "frequency of publications for categories within the scheme" (Petersen et al., 2008). The main goal of this study method is to provide an overview of a specific research area while being able to quantify and qualify its results.

3.2.1.1 Research Questions

The first phase is to define what are the questions of the research. These point out what the researcher aims to answer with his study.

Table 1 and Table 2 explain the research questions for the systematic map to be performed in this document.

Table 1 - Systematic Research Question 1 (RQ_1)

Question	What are the most important concerns that developers should pay special attention to when implementing greenfield microservices?
Rationale	Microservices development involves several concerns and challenges. It is important to identify which of them are the most relevant when implementing microservices from the start.

Table 2 - Systematic Research Question 2 (RQ_2)

Question	What should be avoided in the early stages of a microservice-based greenfield project?
Rationale	Given that in greenfield projects everything is built from the ground up, and it starts as a small project, several aspects of conventional microservice migrations should be avoided on its starting phase.

3.2.1.2 Search, Sources of Information, and Databases

This study focused on digital libraries that are the most used research repository in the software engineering study area. The databases used for the search were ACM Digital Library, Google Scholar, ResearchGate, and IEEE Explore. Since the research topics are very recent, a set of specialized technical blogs (see Table 3) were also included such as Medium, Martin Fowler's blog and Microservices.io.

Table 3 - Additional sources for systematic mapping research

Blog	Url
Medium	https://medium.com
Martin Fowler	https://martinfowler.com
Microservices.io	https://microservices.io

The following search queries were performed:

- $microservices\ AND\ (challenges\ OR\ practices\ OR\ patterns)$
- $microservices\ AND$ ("from scratch" OR "from the start" OR greenfield)
- $microservices\ AND\ greenfield\ AND\ (guide\ OR\ guidelines)$

3.2.1.3 Screening Phase

After conducting the search and having a set of studies/articles it is necessary to perform a screening. This consists of applying inclusion and exclusion criteria that allow filtering those that are relevant. The process represented in Figure 6 was followed.

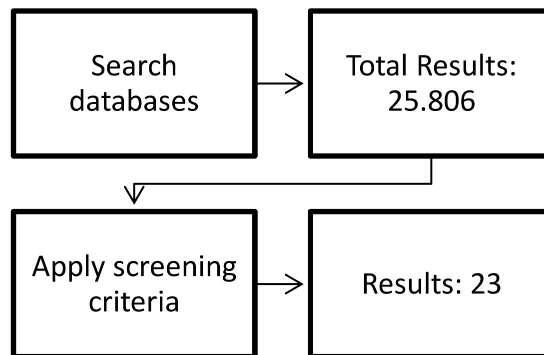


Figure 6 - Screening process diagram

After applying the criteria defined in Table 4, 23 documents (see Table 5) were obtained from the total of 25.806 papers previously identified.

Table 4 – Systematic Mapping Screening Criteria

Screening Criteria	
Inclusion	Books, scientific papers, technical reports, other grey literature, and technical blogs that describe: - *i_1*) Microservice development methodologies, patterns, and practices reported by experienced practitioners. - *i_2*) Practical case studies where microservice methodologies were applied. - *I_3*) The benefits and disadvantages of microservices when compared with other development approaches.
Exclusion	- *e_1*) Commercial publications. - *e_2*) Papers that are not written in English. - *e_3*) Papers that are not in the software engineering scope and neither refer microservices or dev-ops explicitly. - *e_4*) Study does not offer evidence for the perspective of the author

Table 5 - List of papers after the screening process

Id	Title	Reference
1	Pitfalls & Challenges Faced During a Microservices Architecture Implementation	(Gupta & Palvankar, 2020)
2	Microservices	(Thönes, 2015)
3	Microservice Patterns and Best Practices: Explore Patterns like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices	(Pacheco, 2018)
4	Exploring Microservice Security	(Yarygina, 2018)
5	Microservice architecture design: 5 key elements	(Golden, 2020)
6	A Systematic Mapping Study on Microservices Architecture in DevOps	(Waseem et al., 2020)
7	Migrating towards Microservices: Migration and Architecture Smells	(Carrasco et al., 2018)

8	Communication in a microservice architecture	(Anil, 2020)
9	MonolithFirst	(Fowler, 2015)
10	Don’t start with a monolith	(Tilkov, 2015)
11	Micro services Architecture in Greenfield Projects	(WaveMaker, 2019)
12	Emerging Trends, Challenges, and Experiences in DevOps and Microservice APIs	(Zdun et al., 2020)
13	Domain-Driven Architecture Modeling and Rapid Prototyping with Context Mapper	(Kapferer & Zimmermann, 2021)
14	Microservices Decision Guides	(Barcia et al., 2017)
15	Microservices From Day One	(Carneiro & Schmelmer, 2016)
16	Decision Guidance Models for Microservices – Service Discovery and Fault Tolerance	(Haselböck et al., 2017)
17	A logical architecture design method for microservices architectures	(Santos et al., 2019)
18	Microservices and their design trade-offs: A self-adaptive roadmap	(Hassan & Bahsoon, 2016)
19	Microservice Architecture and Model-driven Development: Yet Singles, Soon Married (?)	(Rademacher et al., 2018)
20	Microservice Architecture Aligning Principles, Practices, and Culture	(Nadareishvili et al., 2016)
21	Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study	(de Toledo et al., 2021)
22	Microservices Migration in Industry: Intentions, Strategies, and Challenges	(Fritzsch et al., 2019)
23	Sliceable Monolith: Monolith First, Microservices Later	(Montesi et al., 2021)

As shown on Figure 7, most of the selected papers (65%) were published in 2018 or after, and 34,8% were published since last year. Having such a big percentage of very recent publications is especially important in such an emerging, and trendy topic.

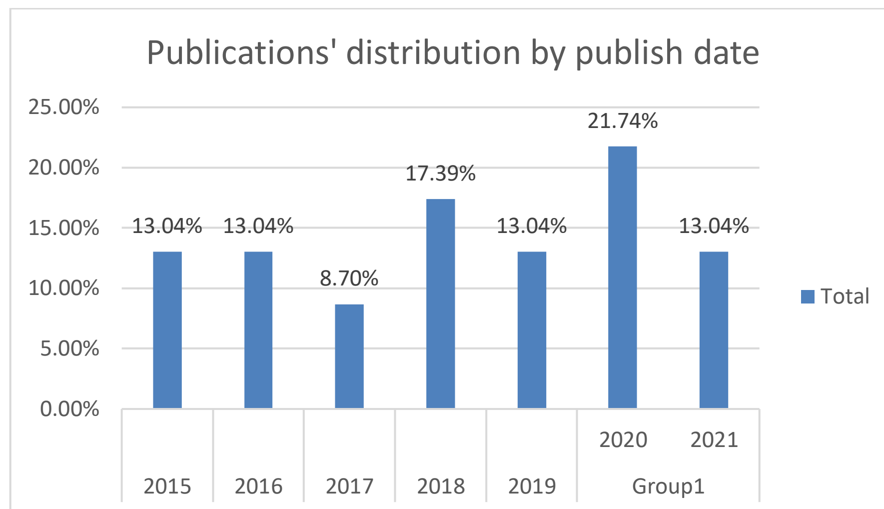


Figure 7 - Distribution of research papers by publication date

3.2.1.4 Keywording

After defining the classification categories and filtering the whole set of 25.806 papers respecting the criteria defined in Table 4, this step consists of looking for certain keywords and concepts on the abstracts of the resulting publications (listed in Table 5). In the end, the set of keywords from the different publications are combined to create an overview of the nature and contribution of the research. It will help to define a set of categories that is representative of the population and will be used to create the map. The process is summarized in Figure 8.

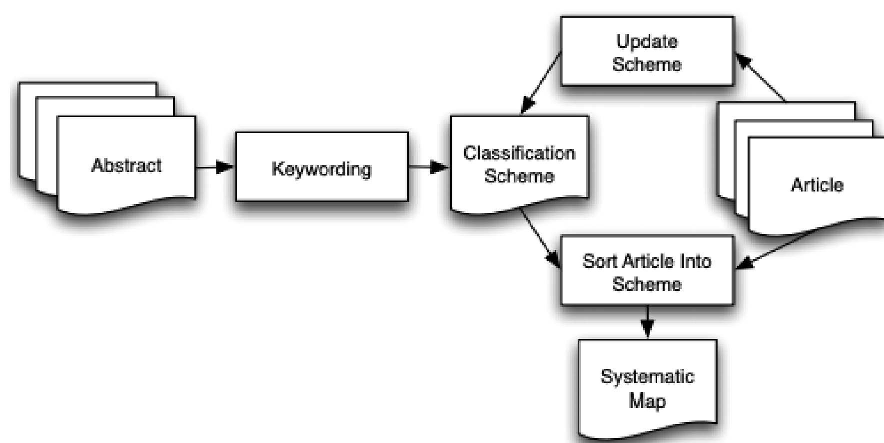


Figure 8 - Classification scheme based on keywording

Source: (Petersen et al., 2008)

Table 6 - Research Topics

Research Topics	Description
Greenfield microservices concerns	Focus on concerns of the development of microservices that have a special impact on greenfield projects.
Things that should be avoided in the early phases of microservices development	Understand which of the phases of microservice development should be avoided in Greenfield projects

3.2.1.5 Data Extraction and Mapping

In this section, the data of the publications was extracted and mapped into a set of concerns for each of the topics identified in Table 6. These concerns are reflected in the first columns of Table 7, and Table 8, and in the second column are the references that support each of them.

Table 7 - Greenfield microservices concerns research

Description	Reference Paper IDs
Define the right granularity of a microservice	1, 2, 3, 4, 5, 7, 13, 18
Size versus number of microservices trade-off	2, 13, 18
Align services with domain	1, 2, 4, 11, 14, 15, 16, 17, 19
Seek the benefit in the short term	21
Communication between services	1, 2, 3, 4, 8, 12, 20
Organizational changes and build expertise	7, 9, 15, 22

Table 8 - Avoidable steps to start greenfield microservices

Description	Reference Paper IDs
Start project without complete domain knowledge	10, 11
Services in separate repositories and pipelines	11, 23
Close microservice’s granularity too early	7, 17
Design all parts of the system	7, 9, 20

3.2.1.6 Data Analysis

Now that the data was extracted and mapped into the correct categories inside each classification system, it is possible to analyze how the collected results answer the research questions defined in section 3.2.1.1.

Systematic Research Question 1 - RQ_1

The first research question focuses on the concerns that developers should pay special attention to when implementing microservices for greenfield projects. Based on the information extracted in section 3.2.1.5 it is possible to explain in more detail some of the most relevant concerns that can impact greenfield projects based on microservices.

- **Align microservices with the domain:** a challenge referenced by many publications is the need to align microservices with the domain during design. It is considered a good practice to design microservices-based on the domain's business capabilities as it results in loosely coupled service. However, it requires the identification of every interaction that depends on the context that may be shared between services, which may become a complex task (Gupta & Palvankar, 2020). In section 3.4.1, the challenge is contextualized, and an approach based on context mapping is suggested to solve it.
- **Communication between services:** one of the key aspects to make a growable microservices system is related to how the modules will communicate. Even though each service should be autonomous in a cohesive modularized microservice architecture, the system will not be available unless there is communication between its modules (Nadareishvili et al., 2016).
- **Organizational changes and build expertise:** recently, Fritzsch et al. studied 14 microservice-based systems where only one was a greenfield development. It supports the idea that these are still rare when compared to migrations of existing systems. However, this study allowed a comparison of the challenges faced by one method and the other. The main concerns reported by the interviewees were "building the necessary expertise" in small teams, and making the necessary organizational changes because it can lead to teams being "left alone" during the transition period (Fritzsch et al., 2019).

Systematic Research Question 2 - RQ_2

Building microservices for a greenfield project has the major advantage to avoid dealing with legacy code. However, cases like (Barthel, 2017) prove that when applying this architecture, as it is, to small greenfield projects, it has several disadvantages.

From the information extracted (section 3.2.1.5), it is possible to answer the second research question defined in section 3.2.1.1. It was found a set of practices that, besides being important on microservices development, should be avoided in the early stages of a greenfield project. There are also a few recommendations that are transversal to Greenfields, and Brownfields but are considered to have more impact on projects starting from scratch.

- **Do not start a project without having a clear vision of the domain:** the definition of service boundaries can be tricky in greenfield software since there are no references to follow. This requires exceptional attention to the level of knowledge the team has over the domain before trying to separate the system into different services. Developers must have clarity about the dependencies between the services to be able to define them (WaveMaker, 2019).
- **Do not define microservice granularity too early**: the granularity level of a microservice affects directly how it will be implemented and instantiated, thus it should be an iterative process. During this whole process, more details about the service may appear and causing it to increase or decrease in dimension. By having more input, and not closing permanently the scope of a microservice in an early phase, long-term problems are less likely to occur. As Hassan & Bahsoon stated: “Splitting too soon can make things very difficult to reason about. It will likely happen that you (the software architect) will learn in the process” (Hassan & Bahsoon, 2016).
- **Services should not be separated into multiple repositories/pipelines and applications:** if the greenfield project is small enough to not create multiple teams, there is no need to divide services into separate applications. Instead, just separate them into separate software packages where it is still possible to manage the dependencies between services and develop each one independently by using interfaces for the communication. This avoids the unnecessary overhead of having different repositories, separate CI pipelines, and more complex functional testing. Once the project grows and needs more developers working on it, the migration of the packages into separate solutions/applications should be straightforward (WaveMaker, 2019).

3.2.2 Summary

Even though there is a huge number of publications about microservices, their challenges, and methodologies to migrate from monolithic applications, there is still clearly a lack of documentation regarding implementing microservices from the start in greenfields. From the total 25806 search results, only the 23 chosen publications after the screening step (detailed in section 3.2.1.3) referred to greenfield projects or challenges. Furthermore, most of these publications only refer to starting a project with microservices as an alternative, thus not referring to any specific concerns or challenges about that methodology. It proves once more the importance and contribution that this work brings to the software engineering community.

3.3 Related Work

This section aims to identify the existing work regarding guidelines for the development of microservices focused on greenfield projects. Based on the literature review performed in section 3.2.1, the publications that can be compared to this work were collected and summarized. This should help to understand what is already well documented and what needs more investigation.

3.3.1 “Microservices Decision Guides”

Similar to what is intended in this document, the work developed by Barcia et al. in 2017 aims to create a guide to help application architects deliberating decisions about the adoption of microservices. It provides “prescriptive guidance” and focuses on the following aspects: comparison between the microservices and the monolithic approach, security, deployment phase, resilience, and the adoption of good “administrative and operational” practices (Barcia et al., 2017).

Besides the similarities of the two works, they differ in the scope and detail of the contents. Unlike this work, which is focused on the identification of microservices and their implementation going in detail and suggesting approaches and alternatives for each step, “Microservices Decision Guides” has a broader scope. It is split into five smaller guides with decisions regarding the following topics: business, architecture and design, implementation, resiliency, and operations. Even though these are all very important concerns for microservice development, this work wanted to focus on the architecture, design, and implementation phases which are not very well documented for greenfield projects.

3.3.2 “Microservices From Day One”

In the book “Microservices From Day One”, the authors focused on the major topic of this work: why and how to implement microservices in greenfield projects (Carneiro & Schmelmer, 2016). Like this work, the book also recommends employing microservices from a very early stage of the application, and touches topics like: “Partitioning business into services”, “Implementing and consuming APIs the right way”, and “Monitoring Your Services” (Carneiro & Schmelmer, 2016). It also touches more on operational concerns that this guide does not, but on the other side, it does not follow a step-by-step structure and becomes too extensive in some of its sections. This makes it a worse alternative for developers that want a simpler structured approach that they can follow to implement a microservice-based application in a greenfield project.

3.3.3 Design Models for Microservices

Being the design phase one of the most critical during the development of microservices, two works (Haselböck et al., 2017) (Santos et al., 2019) are highlighted because they present structured methodologies for this process. This kind of work is exactly what is intended for this paper but, even though both works describe approaches that can be followed easily by the reader and applied to a project, its scope is restricted to one of the phases included in this work (Step 2 – Service Identification). Nonetheless, these two works are excellent supplements to the guide designed in this paper (see section 4.2.2) and should be considered as valid alternatives.

3.3.4 Summary

There is notorious work about microservice development, but, highlighted in section 3.2.1.3, there are still very few publications focused on greenfield projects.

Even though the research work on microservices is reflected in different types of publications such as scientific articles and books, there is still missing a paper that approaches not only the explanation of why, how, and in which context microservices are adequate, but also how to apply this architectural style to a greenfield project in a simple and prescriptive guide.

This work does not replace any of the work presented above but provides some of the most relevant content from many of them, with a structure that is easier to be followed by software engineers and researchers in projects. It is expected that additional sources are consulted for more details about specific concerns, but the objective is that the software engineering community can access a set of guidelines that were specifically designed for greenfield projects and tested and validated in an exemplifying project.

3.4 Related Technology

With the growth of microservice architecture adoption, new technologies were created and some of the existing ones were adapted to support and facilitate the development, testing, deployment, and monitoring of this kind of system.

Some of the most relevant technologies for the microservice-based applications' lifecycle are described in this section.

3.4.1 Service Recognition

During the Vienna Software Seminar (VSS) in 2019, many of the participants raised an important question about microservices development: "How to identify constituent services and APIs in a microservice-based application?" (Zdun et al., 2020). As it was stated in section 1.2, the conventional wisdom defends that the best practice to design microservices is to begin with a monolith and cut it to smaller services once needed. But as Zdun states: "[…] a monolith that has grown naturally (i.e., without explicit planning for future microservice migration) may often require substantial architectural redesign before nontrivial services can be cut out.", which led to some developers trying to build monolithic applications in ways that facilitate later migration. What happened is that the effort to build such applications (architectural investment and foreseeing reliability, performance, and security implications) was often considered too high for greenfield projects that are unclear if the investment will ever be paid off. This means that, if the application never grows enough to be migrated into microservices, most of the investment would have been in vain. If it is verified that this investment is relevant for the project, domain-driven design (DDD) along with domain-specific languages (DSLs) for modeling bounded contexts can be good tools for the modeling process. Context Mapper and MDSL were presented as two examples of DSLs for this purpose.

3.4.1.1 Context Mapper

In 2018, Kapferer wrote his master's thesis "A Domain-specific Language for Service Decomposition", which resulted in the starting point of the Context Mapper project. During this work, Kapferer gave big relevance to the challenge of "decomposing an application into appropriately sized services", and remembered that the strategic patterns of Domain-driven Design (DDD), with a focus on Bounded Contexts, could be a viable approach for that process (Kapferer, 2018). With this in mind, he started to implemented Context Mapper, a modeling framework, based on DDD patterns, that allows domain decomposition. The framework's architecture is presented in Figure 9.

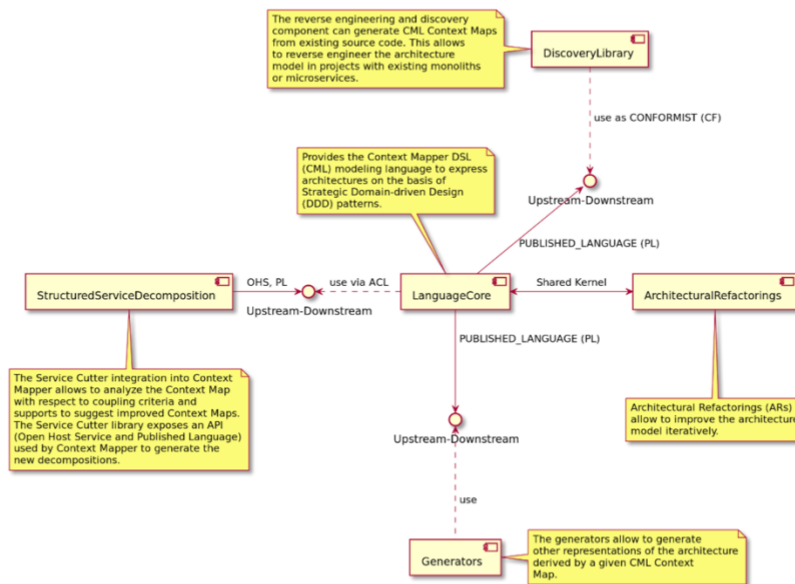


Figure 9 - Context Mapper Framework Architecture

Source: (Context Mapper, n.d.-a)

In the first iteration of the project, CML (see example in Figure 10) was developed providing a tool for a structured service decomposition integrated with Service Cutter, a tool that, based on coupling criteria, "suggests service cuts to assist an architect's decomposition decisions" (Gysel et al., 2016). As an additional feature, it was developed a proof of concept of a PlantUML generator that, from a DSL model, can generate two diagrams: a component diagram representing the bounded contexts and their relationships, and a class diagram for each bounded context that illustrates the aggregates, modules and all domain objects of a bounded context (Kapferer, 2018).

```
ContextMap {
  contains CustomerManagementContext, PolicyManagementContext

  CustomerManagementContext [U,OHS,PL]->[D,CF] PolicyManagementContext {
    implementationTechnology = "RESTful HTTP"
  }
}
```

Figure 10 - Context Map Language (CML) Syntax Sample

Source: (Context Mapper, n.d.-b)

More recently, Kapferer and Zimmermann introduced a new syntax to the framework and implemented three new model transformations (see Figure 11) that "support rapid prototyping". This new feature allows domain modelers to specify requirements in the form of user stories which will then be derived into subdomains and bounded contexts from these requirements (Kapferer & Zimmermann, 2021).

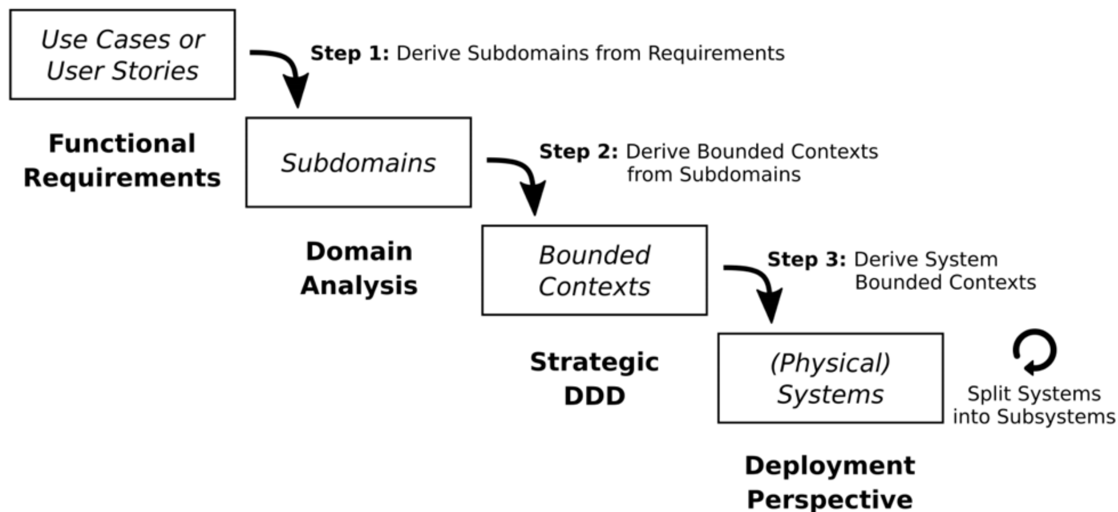


Figure 11 - Rapid Prototyping Transformation Steps

Source: (Kapferer & Zimmermann, 2021)

Currently, the framework already provides a wider set of features (Context Mapper, n.d.-a):

- **Context Mapper Language (CML):** an Xtext-based Domain-specific Language (DSL) for DDD context mapping. It "supports modeling Bounded Contexts and their relationships with tactic and strategic DDD patterns."
- **Continuous Refactoring:** allows engineers to "decompose their systems architecture by iteratively applying Architectural Refactorings (ARs). Split and merge Bounded Contexts and/or Aggregates to improve coupling and cohesion."
- **Diagram and Contract Generation:** generate PlantUML graphical representations of the created context maps' architecture as well as other representations like MDSL (micro-)service contracts, or code.
- **Context Map Discovery Library:** reverse engineer a DDD Context Map from an existing monolith or microservice application.

- **Systematic Service Decomposition:** integration with the Service Cutter allows the calculation of new Context Maps to improve coupling and cohesion “by using graph clustering algorithms and coupling criteria”.
- **Code (Microservice) Generation:** generate Microservice applications (Spring Boot) using JHipster following a provided template.

3.4.1.2 MDSL

Microservice Domain-Specific Language (MDSL) “is a Domain-Specific Language (DSL) to specify (micro-)service contracts and data representations, jointly realizing the technical part of the API Description pattern from Microservice API Patterns (MAP)” (Context Mapper, n.d.-b).

Based on CML context maps, Context Mapper can produce MSDL contracts by providing an MSDL generator tool. During these conversions, CML bounded contexts are converted into MSDL Service Specifications and Exposed Aggregates into Endpoints for example (Context Mapper, n.d.-b). Then MSDL provides generators that can convert the domain language into different technical specifications such as OpenAPI or Protocol Buffers, to a GraphQL schema, or even plain Java code (Zimmermann, 2018).

Using Context Mapper and MSDL generators it is possible to go from a DDD Context Map to a functional microservices code base in few steps.

3.4.2 Microservices Development

To develop microservices more productively, many software frameworks have been developed. Microservice applications commonly have infrastructure services responsible for the core functionalities of their ecosystem, and the functional services that contain all the business logic of the system. The major advantage of using a framework to develop an application is that it provides reusable infrastructure services that take care of the basic microservices needs so that developers can focus on what matters, the implementation of the functional services (DInh-Tuan et al., 2020).

3.4.2.1 Software Frameworks

Studies that analyze new frameworks and their evolution are becoming more popular each day because tech companies want to understand which technologies are the best fit for their microservices.

A recent research study performed by Technische Universität Berlin members compares and evaluates the usability and practicability of four commonly used open-source frameworks in cloud-based applications: Java Spring Boot, Lagom, Moleculer, and Go Micro (DInh-Tuan et al., 2020). In this specific case, there were three comparison attributes: features, design patterns, and performance. The “features attribute” was evaluated by the frameworks’ maturity in the market, development support, and its built-in features, the supported “design patterns” for

each framework were identified to fulfill that category's evaluation, and the "performance" rating relied upon two criteria: end-to-end latency performance and resource consumption.

As stated by (DInh-Tuan et al., 2020) "[…] it is clear that there is no one-size-fits-all solution when choosing a microservice framework. It depends a lot on application's specific functional and non-functional requirements". From the conducted study Spring was considered the most mature, with the best support, and the one that supports more design patterns. The most performant overall was Moleculer.

Based on this and other recent framework comparisons (Chinnasamy, 2020) (Kurmi, 2020), and having as priority criteria the maturity of the framework and the experience of the developer, were selected two candidate frameworks to be used in this dissertation prototype: Java Spring Boot and .NET Core.

3.4.3 DevOps Implementation

DevOps are defined by AWS as:

"[…] the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes." (Amazon Web Services (AWS), n.d.).

In this model, development and operation teams are not isolated, instead, they are "merged into a single team where the engineers work across the entire application lifecycle (see Figure 12), from development and test to deployment to operations" and it has several benefits such as quicker delivery, reliability, scalability, improved collaboration, and security (Amazon Web Services (AWS), n.d.).

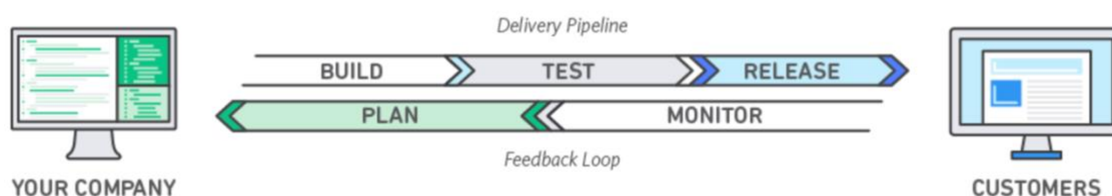


Figure 12 - DevOps Model

Source: (Amazon Web Services (AWS), n.d.)

The thing everyone must understand is that Microservices can exist without using DevOps, and DevOps may be applied to projects that are not microservice-based since one is a culture and the other an architectural style, and there is no direct relationship between them. When microservices are well implemented they may facilitate DevOps though. By definition, microservices architecture is based on small and independent applications, which promotes fast feature development and delivery to the end-user by using shorter and quicker release cycles. It helps DevOps because each team will be able to "develop, test, handle failures and scale independently" more easily. This would be nearly impossible to achieve in a monolith because

a team performing any of these tasks would be affecting the whole application instead of a single service, and consequentially all the other teams' work (Pallis et al., 2018).

Containerization is another technique that is very connected with microservice development as is detailed in the next section. Briefly, it consists of isolating each application of the system in a virtual environment while allowing them to share a single guest OS/host. This brings advantages of hosting setup and management, and at the same containers have significantly smaller sizes "which makes them easier to migrate, faster to boot, and less demanding on memory" (Pallis et al., 2018). All these characteristics of containerization and indirectly microservices, make the architecture DevOps friendly and explains why these two concepts often appear together.

Challenges

Even if it seems a simple task to implement DevOps on microservice-based projects, it is not true. Moving a microservice model from development to production environment has a lot of additional concerns that may not be straightforward for every team to deal with. Below are described some of the most important concerns that a DevOps team would need to deal with when moving their microservices to a production environment:

- **Monitoring:** even if containers simplify the deployment process, it makes monitoring a more complex task. Since multiple applications are running in the same host machine, the monitoring agent "must run through the container engine or be part of the application itself". This implies that monitoring strategies can no longer be decided after the deployment, instead, they must be included in the application design process (Pallis et al., 2018).
- **Auto-Scaling and Optimization:** microservices and containerization make it easier to scale applications "by simply creating more copies of the services overwhelmed by demand", however it requires the configuration of optimization strategies which can be difficult in a distributed deployment and when the performance of a service is affected by another's.
- **Infrastructure Management:** even if microservices are built to run anywhere, their geographical placement is critical since bad decisions in this matter may cause "severe propagation delays and other network performance issues for service communication" (Pallis et al., 2018). Following a DevOps approach, determining where the services will run and their needs in terms of infrastructural resources may become a challenging task since the management is not centralized.

3.4.3.1 Packaging and Deployment

Docker started as an open-source project written in Go in early 2013. It was a *dotCloud*'s (a technology company) internal tool developed to allow the business to run on thousands of servers (Merkel, 2014). After six to nine months the company hired a new CEO, joined the Linux Foundation, and changed its name to Docker Inc., announcing a shift of focus to the development of the Docker tool and its ecosystem (Merkel, 2014).

The main purpose of Docker is to run applications in a way that is easier to develop and distribute. When an application is built with Docker, it is packaged with all its dependencies into what is called a container (Bashari Rad et al., 2017). This will solve the *"Dependency Hell"* as it is called by (Boettiger 2015) because aside from working similarly to a virtual machine image, it provides a binary image where all the needed software has been installed, configured and tested, and differently from VMs, it shares the kernel with the host machine which makes the applications more performant and lightweight (Boettiger, 2015).

Figure 13 shows the Docker client-server architecture. When a developer types docker commands or interacts with the Docker UI to run, stop or build containers, the client communicates with the Docker Daemon via REST API, this entity is responsible for the processes of building, running and distributing the containers (Docker, 2018).

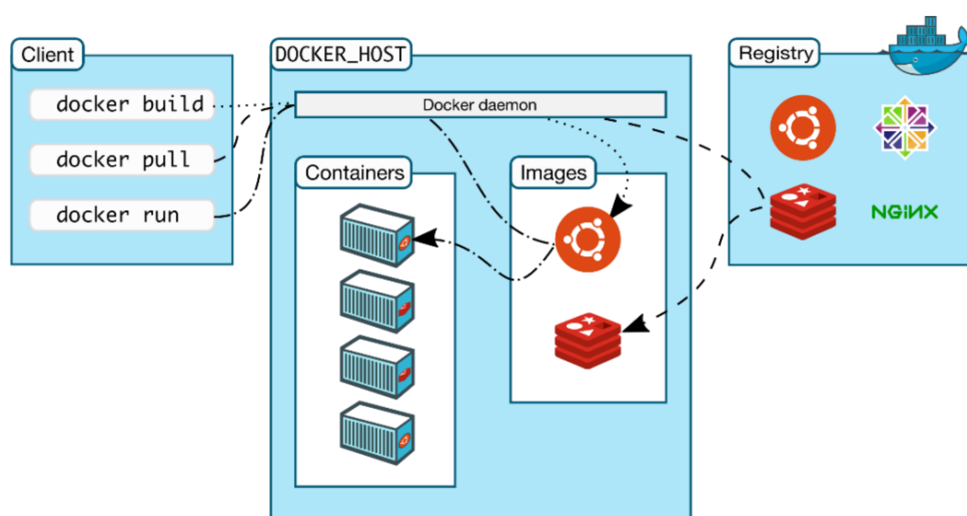


Figure 13 - Docker Architecture

Source: (Docker, 2018)

3.4.3.2 Docker in microservices

There is a set of characteristics associated with Docker that responds directly to some of the microservice concerns. Apart from solving the dependencies issue, Docker also promotes isolation and security since containers are not accessible unless they are explicitly configured to allow it.

Docker containers are small and can be deployed in multiple physical servers and cloud platforms which simplifies the development and testing and provides scalability (Bashari Rad et al., 2017).

Docker also promotes SOA and microservice architectures since each container should only run a single application or process. The system will follow a distributed model composed of a series of inter-connected containers (James, 2019).

3.4.3.3 Kubernetes as an orchestrator

As it was seen, Docker is a tool that is highly compatible with microservice architectures because it allows encapsulating every microservice in a separate container. However, it is important to

put together these separate entities to create a real microservice-based system. Kubernetes is an open-source orchestrator, originally developed by Google, that deals with the deployment of containerized applications and provides all the necessary software to build and deploy reliable, scalable distributed systems (Burns et al., 2019).

There are many benefits on Kubernetes adoption, some of the most relevant are:

Development Velocity

One of the most common reasons for adopting a microservices architecture is the need for speed when it comes to sustainably develop new features, always assuring the stability of the system. Modern web applications are expected to be highly available 24/7, which means that every new feature release must be performed with no downtime. Kubernetes provides tools for this, by applying the concepts of Immutability, Declarative Configuration, and Self-Healing Systems (Burns et al., 2019).

Scalability

Another reason why companies commonly adopt microservices is that their products grow, and inevitably end up needing to scale both their software and the teams developing it. Given the nature of Kubernetes which uses immutable containers, the scaling process becomes simple as it consists of just changing the number of container replicas. This can be a manual process, but it is also possible to set Kubernetes for autoscaling mode, which will take care of upscaling and downscaling the application automatically depending on the demand (Burns et al., 2019).

Efficiency

As stated by (Burns et al., 2019): "Efficiency can be measured by the ratio of the useful work performed by a machine or process to the total amount of energy spent doing so.". In the case of Kubernetes, this is related to two main characteristics:

- On one side, abstracting the infrastructure allows much simpler and faster deployment and management processes. It allows to reduce the costs of running servers unnecessarily (e.g. when traffic is low) and the human cost of the management;
- The quick and cheap creation of testing environments is another increase in efficiency provided by Kubernetes. This can be configured as a single cluster in a virtual environment, which may be shared by multiple developers for tests. The main difference in terms of cost reduction is that instead of measuring the costs associated with a certain number of complete running VMs, they are simple containers that may even be all running in the same machine, and therefore have a much lower cost.

3.4.4 Data Shaping

As defined by Wittern: “GraphQL is a query language for APIs and a runtime to execute queries. Using GraphQL queries, clients define precisely what data they wish to retrieve or mutate on a server, leading to fewer round trips and reduced response sizes.” (Wittern et al., 2019).

In microservices, communication happens through a network, that can be local or through the internet. This may become a challenge since it is necessary to deal with latency and other network concerns such as bandwidth for example. As it was shown in section 2.4.5, the amounts of data that flow from and to microservices must be carefully managed and should be as reduced as possible because larger payloads may slower communications.

To avoid transferring large amounts of data between services and an orchestrator for example, a set of low granularity REST endpoints could be implemented to respond to each of the necessary properties, but this would lead to another problem, the number of requests to get all the needed properties would rapidly rise, increasing the application’s dependencies exponentially, making it much harder to manage and ultimately resulting in an even slower response time.

GraphQL addresses this concern. As it is shown in Figure 14, the client sends a query request to the server with the list of desired properties from the defined schema. Each one is assigned a resolver that is responsible for defining how to get the needed information (it can be a request to an API or another service, make an internal function call, or querying a database).

Abstracting how data is fetched is also an advantage because it reduces the coupling between services and their dependencies. It makes it easier to switch the sources of information without needing to perform complex changes to the code.

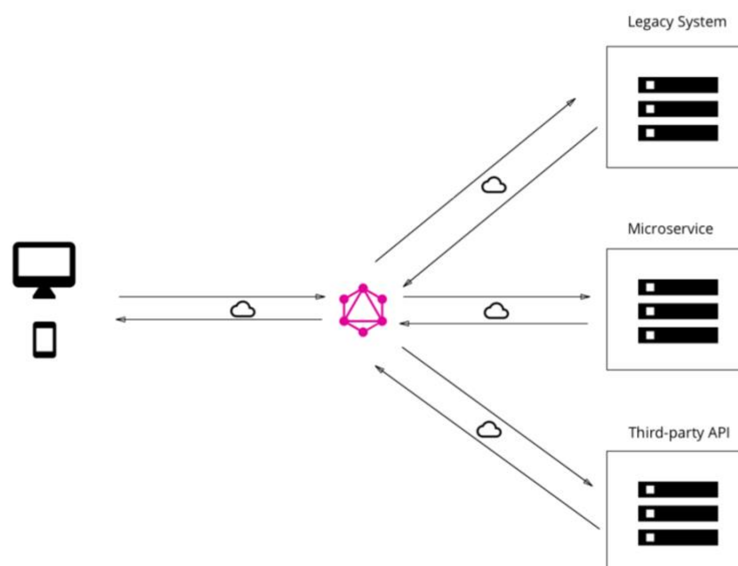


Figure 14 - GraphQL Architecture

Source: (GraphQL, n.d.)

In a recent study, Brito describes some of the GraphQL key concepts and compares them directly to the REST API approach. One of the fundamental differences is that "GraphQL allows clients to query a database represented by a schema" while REST server applications "implement a list of endpoints" (Brito et al., 2019). Besides presenting some key concepts of the technology, the study also focuses on describing its major benefits and disadvantages. Some of the most relevant are presented below.

Benefits:

- **Strongly Typed:** since the objects and fields have specific types, queries may be syntactically verified before their execution and it allows GraphQL "to provide descriptive error messages" (Brito et al., 2019).
- **Query Specification:** as explained before, the main advantage of this technology is that "the specification for queries are encoded in the client rather than the server" which allows it to return "exactly what a client asks for and no more" (Brito et al., 2019).
- **Objects as a graph and APIs combining** since GraphQL models objects like graphs, and "makes it easy to combine multiple APIs into one", it can obtain data from multiple sources in a single request and have each part of the schema implemented in different independent services. This is an especially important benefit for microservices (Brito et al., 2019).

Disadvantages:

- **Private Fields:** by default, private fields are not supported. This makes it harder for clients to implement queries since they must have a detailed view of a much larger data schema. Although GraphQL natively does not allow hiding properties, some frameworks provide this support.
- **Expensive Queries:** GraphQL queries may be complex to process by the server if they have deep nesting. The flexibility is a benefit but requires the server to implement restriction strategies to avoid expensive queries that can consume excessive amounts of resources and potentially bring it down. Having this in mind most GraphQL server frameworks implement simple features to set the maximum nesting level allowed.

3.4.5 Monitoring

Monitoring consists of looking at metrics and detect anomalies, it should provide simple views of the system status and alert when there is a problem (Meng et al., 2017).

There are two main types of monitoring and both should be implemented for microservices systems:

- **Infrastructure:** based on an agent that is installed in the machine where the system is running, it monitors several of its parameters such as CPU, RAM, Disk, and Network.

- **Application:** monitors the application and the metrics that it publishes. The source of data is the application's logs and events and may include received and sent requests as well as any other warnings or errors happening during the application execution.

Even though microservices may bring high availability to a system, Li defends that for microservice-based applications to achieve high availability they need to implement "Continuous Monitoring". That way "their health can be analyzed to automatically and responsively react to failures with minimal human intervention" (Li et al., 2021).

Monitoring systems for microservices usually consist of agents running on each service. These are responsible for tracking and logging a set of relevant metrics of that instance as well as capturing logs created by the application. This information is often stored in a centralized database, so it is possible to correlate data between services and allow to configure alerts or analyze event data manually (Golden, 2020).

Since understanding and correlating the logs from different microservices can be a significantly complex process, some tools that simplify it studies have been conducted and tools created to facilitate it (Cinque et al., 2019). The authors present the state of the art in the area of monitoring microservices and describe the implementation of their own monitoring tool (MetroFunnel) specially designed for this type of architecture. Even though this tool is a prototype and is not one of the most commonly used in the industry, it is part of scientific work and reflects a few important aspects that are transversal to most monitoring tools for microservices.

MetroFunnel consists of "a multithreaded Java program aimed at capturing the network packets and generating the trace". Its main component is based on the Java sniffing tool PacketSniffer which is "responsible for capturing, filtering, and inferring HTTP messages' field" (Cinque et al., 2019).

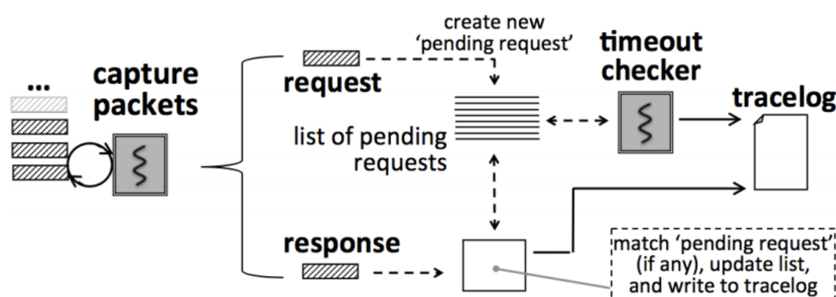


Figure 15 - MetroFunnel main flow

Source: (Cinque et al., 2019)

The rationale behind the approach is that REST is the most used architecture in APIs, and is often implemented over HTTP. Since microservices communication usually also occurs using these techniques, the proposed approach (see Figure 15) captures network packages, analyzes the responses, and provides the trace of the transaction (Cinque et al., 2019).

4 Guide Artifact Design

In this chapter, a proposal of a guide to building a software solution with a microservices mindset since the beginning of the development process is described. For each guideline, a solution is proposed along with a set of alternatives that for some reason were rejected or unfavored.

To prove the value of developing a greenfield project following the proposed guidelines, and to improve the initial version of the guide, a prototype of a pharmacy management system will be implemented using that approach.

4.1 Preconditions

It is possible to build a successful software solution starting with a microservice-based architecture from the start, but there are some requirements. Fowler states that “if you don't have certain baseline competencies, you shouldn't consider using the microservice style” (Fowler, 2014). Similarly, this guide also defines a set of requirements to be able to be used successfully:

- The responsible development team must be very knowledgeable of microservices and had a reasonable contribution to one or more microservice-based projects. That is a guarantee that decisions will take into account at least some of the challenges of section 2.2. This is one of the most remarkable pieces of information collected by Ponce et al., who states that: “[…] distributed system development needs skilled developers.” (Ponce et al., 2019);
- The domain must be well known by all the stakeholders of the project, including internal ones, such as developers and testers (WaveMaker, 2019). This is especially important because one of the first steps to specify microservices is the identification of the system’s operations and the extraction of domains and subdomains. If this step is not carefully performed, the whole project can go wrong (Newman, 2015b);
- Developers should have at least basic experience with DevOps and tools like Docker and Kubernetes to allow them to cost-effectively scale microservice applications. The DevOps

culture also ensures that the provisioning and deployment can be done quicker, which is especially important when monitoring identifies a problem that needs a rapid fix (Fowler, 2014). This experience is known as "operational readiness maturity" (WaveMaker, 2019);

4.2 Solution

When all the previously described requirements are satisfied, the process of developing microservices from the start of the project shouldn't be too different from what has been documented as good practices of "the conventional way" of developing microservices. Microservices are microservices, and their definition shouldn't change whichever methodology is used to generate them. Whether they result from breaking a larger entity or created from a "white canvas", the methodologies to identify and design services are very well documented and therefore included in our guide to successful microservices from scratch.

The construction of the following guide was inspired by attribute-drive design (ADD). Alike this method, the most important inputs are "functional requirements, design constraints, and quality attribute requirements that system stakeholders have prioritized according to business and mission goals" (Wojcik et al., 2006).

ADD consists of a design process that applies strategies and patterns to each system's component in a recursive way to satisfy its requirements (Wojcik et al., 2006). The developed guide follows a similar strategy by analyzing each of the defined microservice concerns and applying the patterns that may fit best the project in development.

The provided guidelines are a proposal and are not supposed to be a "silver bullet" for developing microservices from scratch since they can't be applied to every single project. They should be a good starting point for development teams that are interested in taking risks and trying a new microservices' development approach in a more structured way. To summarize the guide decisions and steps it is simplified in Figure 16.

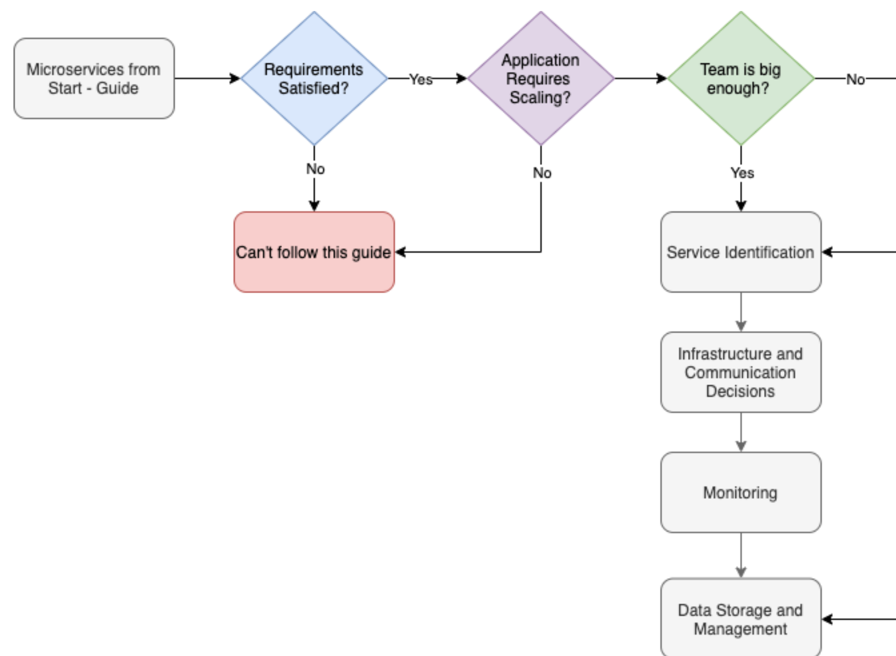


Figure 16 - Microservices from Start Guide Diagram

4.2.1 Step 1 - Should this guide be used?

Before moving to the first step of this guide, it is crucial to understand if it is a good fit for the project needs. As it was stated in the RQ_1 section, microservice architectures may bring a heavy overhead, not only to the development of the whole system but also to its design because it involves complex steps such as identifying services and boundaries. As explained in section 3.4.1, there are cases of engineering teams who have followed the complex and time-consuming service identification process so they could build an application that could be easily migrated to microservices but ended up never needing to perform such migration, which led to a waste of resources. Thus, it is essential to follow the "Business decisions guide" section by (Barcia et al., 2017) to understand the impact of the necessary investments in finance, culture, and DevOps.

Some factors that may indicate that microservices is not the fittest approach for the project are:

- If the business is not complex, the microservices building process can be even more complex than handling the business needs. If an application is simple by nature and functionality, unnecessary complexity is undesirable;
- Some additional points to considerer are scalability and the number of qualified human resources, namely:
  - Will the system need to scale? In what measure?
  - Does the team have the right size to handle the tasks involved with microservices development?
- If there is no need to scale the application or there are not enough people to handle it, microservices will not solve the problem.

If the business is indeed intricate and the application must have the potential to scale, but the team is too small, a mixed approach can be adopted.

As previously mentioned, developing microservices in a greenfield project can be hard for small teams. The same group of people will be working in different repositories, running separate pipelines to deliver value, and dealing with less simple integration tests.

This guide can be partially followed even if the two points are unfavorable. The suggestion is to start building the services in a unique software solution and apply service identification as it is described in the guidelines and deal with a future-proof data management solution. Infrastructure and monitoring concerns should be delayed to a point in the future when more teams become ready to work on their services. This implies that these teams will follow the steps that were not performed before and separate the service from the main application.

4.2.2 Step 2 – Service Identification

Some of the many advantages of the microservices architecture are to simplify testing and allowing independent deployments for each isolated service. But these benefits are not guaranteed if the created services do not conform to the Common Closure Principle. This principle states that things that change together, should be packaged together to ensure that each change affects only one service (Noback, 2018).

A pattern is to be followed to decide which microservices to create. Developers should follow one or more common techniques to guarantee that every service is self-contained and has a correctly bounded context. Each team must change the code of their microservices without knowing the internals of any of the others.

To obtain microservices with the right level of granularity (see section 2.2.1), the good practices referred to in sections 2.3.1, and 2.3.4 are applied and the bad practice of section 2.4.1 is avoided.

Domain and business capabilities-driven microservices with Inverse Conway Maneuver

The main driver of the first step of our guide (microservice identification) is Domain-Driven Design (DDD) with the use of the "Decomposition by Subdomain" and the "Decomposition by Business Capability" patterns (Richardson, 2019c). They get confused and found ambiguous, and they can be if the context of the business matches the domain. (Conway, 1968) states that: "Any organization that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure", which means that if the organization's development teams are aligned with business units, the two patterns are automatically applied.

This alignment is exactly what is trying to be achieved in this solution. Thus, "Inverse Conway Maneuver" will be used. Defined by (Parsons et al., 2014):

"Conway's Law asserts that organizations are constrained to produce application designs which are copies of their communication structures. This often leads to unintended friction points. The 'Inverse Conway Maneuver' recommends evolving your team and organizational structure

to promote your desired architecture. Ideally, your technology architecture will display isomorphism with your business architecture."

Resorting to this technique, the organization's structure should map the business units to more effective apply the decomposition by business capability.

In summary, the solution for the microservices definition starts by analyzing and modeling the business domain and, then, identifying the bounded contexts for the domain. The second step is to adapt the organization's structure to the business domain by creating a development team for each of the bounded contexts. Using the decomposition by business-capability pattern and considering the Conway's Law, the list of microservices for the application should be generated, and finally, it is refined to assure that none of the services should be merged or separated (see Figure 17).

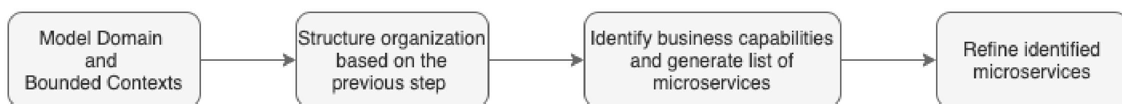


Figure 17 - Microservices Identification Flow

Alternative:

Self-contained services

Self-contained services can be built to respond to synchronous requests without waiting for a response from any other service (Richardson, 2020). The CQRS pattern is used to maintain replicas of the data that the service needs to fetch. At the same time the SAGA pattern is applied to send all the needed asynchronous requests to other services. Even though availability and response time are improved, there are a lot of drawbacks in this possibility, such as increased cost and complexity of using CQRS, increased complexity of using sagas, less straightforward API when using sagas, and larger service due to functionality being implemented in the service instead of as a separate service (Richardson, 2020). This alternative was discarded as a starting point to microservices to avoid adding too much complexity to the fresh application, but can be a valid approach in a later phase.

4.2.3 Step 3 - Infrastructure and Communication

Unlike monolithic applications that run on a single process and their components communicate by calling methods and functions, microservices run separately and, therefore, use different communication mechanisms. One of the biggest challenges that developers find when migrating a monolith to microservices is exactly finding an efficient alternative (Anil, 2020).

When starting with microservices from scratch there is no need to struggle with this issue, but it is a good practice to not fall into the fallacies described in Fallacies of Distributed Computing in section 2.4.5. The approach described in Figure 18 is proposed and should be applied to every

service identified in the previous step to avoid the pitfall described in section 2.2 is avoided. Moreover, the problems associated with “Chain communication between microservices” (see section 2.4.6) are less likely to occur since every transaction between the components of the application is analyzed.

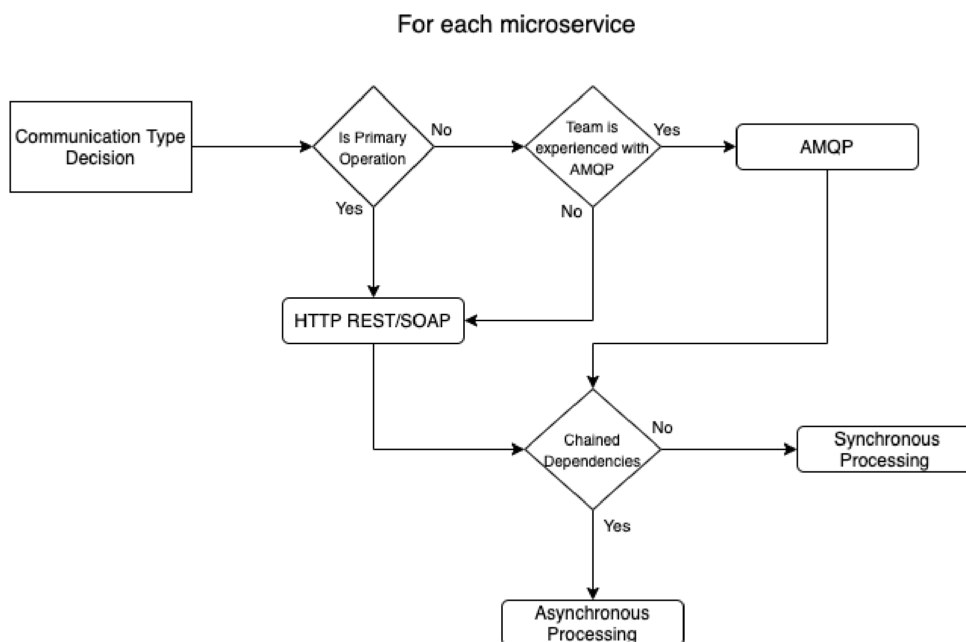


Figure 18 - Communication type decision flow

Choosing the type of communication

Two major types of communication can be used in microservices: HTTP, which is a synchronous protocol, and AMQP, an asynchronous alternative. Synchronous protocol clients send requests and wait for responses from the services. This doesn’t mean that the application’s execution necessarily needs to block the thread to wait for a response since the client code is independent of the rest of the application, but the client code is blocked and can only run after the response is received. On the other side, AMQP is message-based and its client message sender doesn't need to wait for a response.

The decision of the best alternative must reside in the type of operations the service will need to perform. For primary operations such as data fetching or storing, HTTP REST is the best overall option thanks to its simplicity. If, on the contrary, the service needs to dispatch lots of messages to other services and trigger heavy processes that don’t need to be awaited, AMQP is a better option. The other two criteria that should be considered in this decision are the experience of the development team with each technology and also the compatibility of the application language with each type of client.

To prevent the problem described in the “Chain communication between microservices” section, there shouldn’t be any synchronous dependencies between the service and its peers. The approach described by (Anil, 2020) in Figure 3 is recommended. Each of its dependent services should implement mechanisms to receive asynchronous messages or a polling system

must be created to allow sending the request to multiple other services asynchronously. However, if it is not possible to implement an asynchronous solution for the request/response operation and, the service relies on the data of one or more of its peers, (Anil, 2020) proposes an alternative approach: "[…] replicate or propagate that data (only the attributes you need) into the initial service's database by using eventual consistency.". However, it implies database sharing, which is to avoid. A solution for this challenge is proposed later in this chapter.

Alternatives:

In this step, the guidelines are based on ponderation over the type of application to develop and its development team necessities and skills. Therefore, there is no specific solution and alternatives to this point. The described process already includes solutions and alternatives to the specific problem of choosing how infrastructure and communication will work. Thus, the decision should consider all the previously described aspects.

4.2.4 Step 4 - Monitoring

A solution with microservices tends to grow rapidly, it can easily become hard to have an overview of what is happening in each service at a specific moment or during a certain period in the past. Implementing monitoring is especially important in this kind of application (Shahin & Ali Babar, 2020). It allows identifying and tracing problems, that otherwise could get unnoticed until potential drastic consequences were revealed. The question is: "Should these techniques be implemented right from the start of the application, or later in the future?". The right answer to this question will be explored from this point on.

Dealing with effective and efficient monitoring is one of the main challenges to be tackled in this work as specified in section 2.2.4. The sooner it is implemented the better since it is much easier to create a foundation to be implemented in each service as they are created by development teams than do this when a lot of services already exist and may have created their heterogeneous monitoring tools. Creating a framework to be used by all services at the same time the first components are being implemented is important.

The monitoring solution should of course be adapted to the expected growth and needs of the project without limiting the expansion of services to avoid higher maintenance costs if for some reason the system grows beyond the initial expectations. In Figure 19, a very simple solution for this problem is presented. It consists of a tool that allows at any time to know if a service is up or down and if any of its dependencies are unavailable. This can be as simple as implementing an endpoint that answers with a 200 Status Code if the application is running, and another that calls the same endpoint on its dependencies' addresses. It means that every service must register their dependencies addresses and can set an importance level for each one (e.g. critical, important, additional). With the set of responses, the service can generate a response with all the necessary data and send it to the monitoring tool.

Additional features like alerts and notifications in case of unavailability of a service, or more complex status responses can be implemented at this starting point or in the future, to fulfill the project needs.

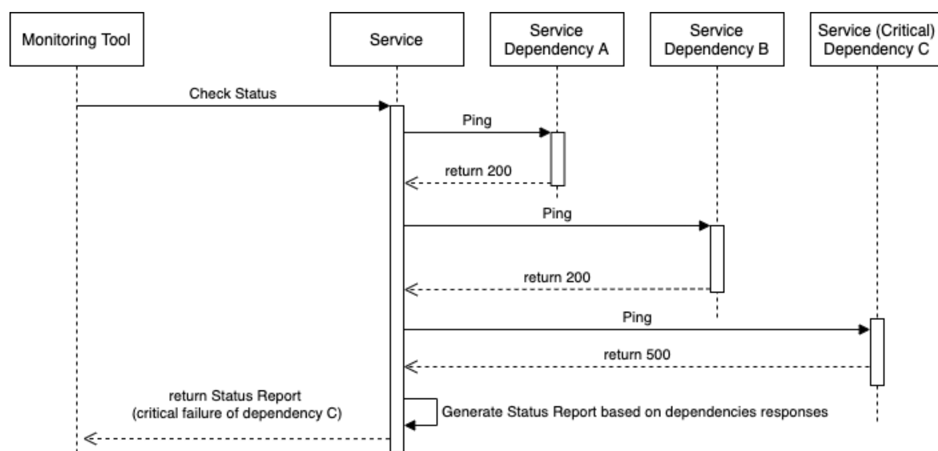


Figure 19 - Simple Monitoring Solution

Alternatives:

If the solution being implemented needs more generic or complete monitoring features, there are lots of paid and open-source projects that can be adopted. For example, if the solution is being built for a "heterogeneous virtualization environment" and performance metrics are required, (Noor et al., 2019) present a specific monitoring framework.

Recently, (Mayer & Weinreich, 2017) analyzed some of the current monitoring requirements on microservices, compare some of the most used solutions in the market (Dynatrace APM, New Relic, and Zipkin). The authors proposed "an experimental dashboard for microservice monitoring and management".

4.2.5 Step 5 - Data Storage and Management

This step presents a solution along with alternatives for one of the most complex challenges in microservices identified in section 2.2.2.

Microservices must be loosely coupled, but at the same time, there are business transactions that need to be spread by multiple services. This can be a challenge if these transactions need to query or update data that is also owned by more than one service, or even worse, join data that comes from different services.

To avoid dependencies between services, a separate data store should be created for each microservice. When a single data store is used, it is common that to reduce duplication of work, microservices owned by different teams share the same database. This practice creates major problems on the application because, if a team needs to update the database structure, all

other services that share the same database need to be adapted. Besides removing dependencies, forcing each service to create their database also allows each team to choose the technologies that best suit their specific service's requirements (Mauro, 2015). It would not benefit the application development because for some services, a relational database is the best choice while other services might need a NoSQL database to store unstructured data or more specific alternatives like Neo4J, designed to efficiently store and query graph data (Richardson, 2019b).

The solution for this problem is represented in Figure 20 and can involve simple or more complex alternatives for the data storing problem. It mostly depends on the type of transactions the service needs to perform and may result in a combination of the following patterns: Database by Service, Saga, Event Sourcing, and CQRS.

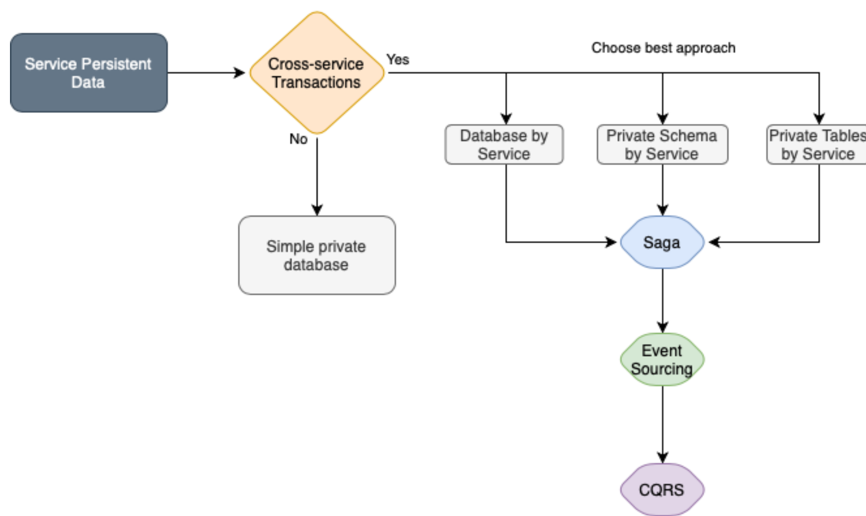


Figure 20 - Service Persistent Data Management Solution Flow

Firstly, the privacy of each microservice's persistent data must be assured. Data should only be directly accessible to the service itself. This can be achieved by applying the Database by Service pattern shown in Figure 21.

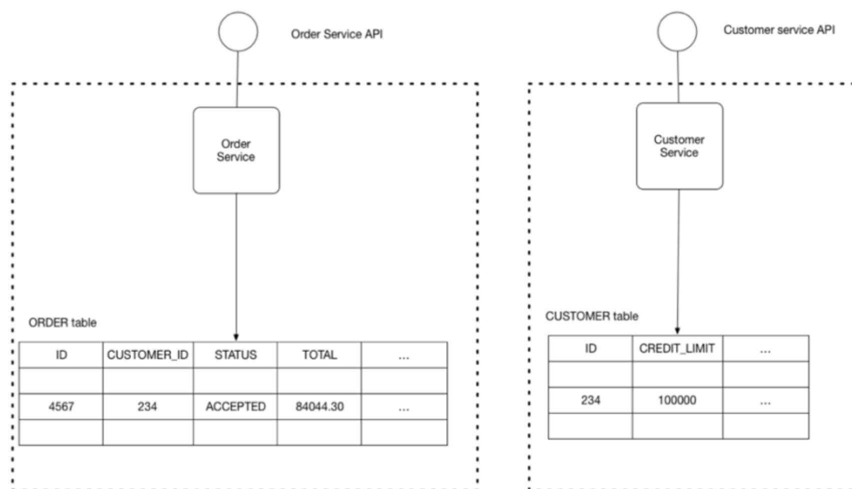


Figure 21 - Database by Service Pattern

Source: (Richardson, 2019b)

As defined by (Richardson, 2019b), there are multiple options to keep the privacy of the service's persistent data:

- Private-tables-per-service – each service owns a set of tables that must only be accessed by that service
- Schema-per-service – each service has a database schema that is private to that service
- Database-server-per-service – each service has its database server

The first two approaches are simpler, but the second has the advantage of making the ownership of each part of the data clearer, so it is better. However, if the service database has a very high throughput, the third approach is the best since the throughput is distributed by multiple servers.

Once the privacy of the persistent data is assured for each service, there may still exist challenges that need to be dealt with. If the service has transactions that need to reach multiple other services, there is also the need to implement a mechanism that can deal with transactions that span multiple services. Since now a complex data and transactions management is needed, a proposed solution may rely on the Saga pattern.

(Richardson, 2019e) explains that the Saga pattern consists of:

> "a sequence of local transactions where each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions."

The main concept of this pattern is that each service involved in the transaction is responsible for updating its database when a message is published. That solves the problem of the multi-

service transactions while implementing a database by service approach, but it also deals with another problem that is the consistency of data. If for some reason, one of the services involved in the transaction fails, all the other services need to revert their changes to maintain consistency (Richardson, 2019e). Nevertheless, when a service needs to perform a Saga, it involves two actions, the update of its database and the publishing of the messages for the consumption of another service. This state update and publish needs to be an atomic operation, otherwise, inconsistencies can occur, which is why the Event Sourcing pattern will be part of our solution.

To be able to solve inconsistencies and revert a business entity to its previous state there is the need to implement a way of storing the list of states of each of the entities. That is what Event Sourcing is responsible for, it persists the state of the business entity as a sequence of state-changing events. Whenever the entity state changes, a new event is added to a list of states (Event Store), and since saving an event is a single operation, it is inherently atomic. If a transaction needs to be reverted, the application reconstructs the entity's current state by replaying the events registered before (Richardson, 2019d). This flow is shown in the example given by Martin Fowler in Figure 22, in which any time a ship's location changes, a corresponding arrival/departure event is registered in the Event Store.

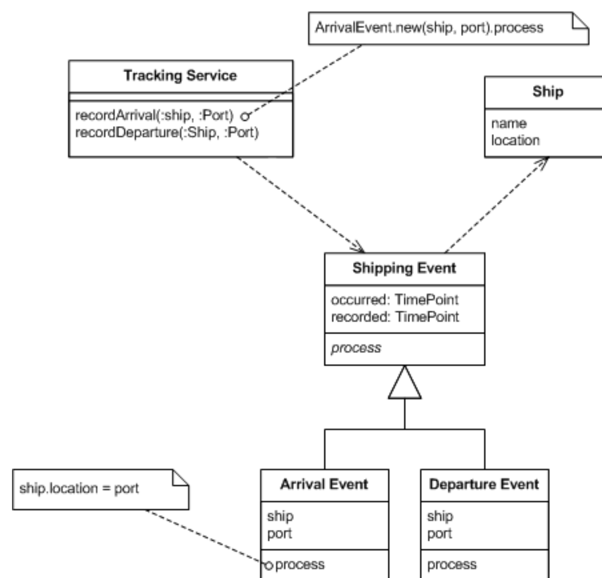


Figure 22 - Event Sourcing Example

Source: (Fowler, 2005)

As explained by (Richardson, 2019d): "The event store is difficult to query since it requires typical queries to reconstruct the state of the business entities." and the suggestion to solve the problem is to use Command Query Responsibility Segregation (CQRS) to implement queries.

In this specific case, CQRS will be used since the service needs to create a way to facilitate read-only access to its data. For this, the service should create a CQRS View and subscribe to all the domain events that may require it to change. That view consists of a replica of data from one or more services and is usually optimized for a particular set of queries (Richardson, 2019a).

Alternative:

Shared Database Pattern

An alternative to the previously presented solution is to use a shared database for the set of services involved in the necessary business transactions. This was previously referred to as a limited solution in section 2.4, but it still can be used if the teams find it suitable for their project.

This pattern consists of having multiple applications using the same database and therefore share data.

With this approach, where a set of integrated applications rely on the same database, if they are running in the same physical environment since the time taken to access the database is so small, the consistency of reading data can be very high. And even if the database gets simultaneous updates from multiple sources to the same data, it must rely on transaction management systems to handle the concurrent write requests. Therefore, since, the time between updates is short, any errors are easier to find and fix. However, if the frequency of the read and modification operations increases, it “can turn the database into a performance bottleneck and can cause deadlocks as each application locks others out of the data” (Hohpe & Woolf, 2008).

If the applications are distributed across multiple locations the communication with the database will typically be too slow to use this approach. And, if people think that distributing the database to be accessed via a local network connection is a good solution for that problem, (Hohpe & Woolf, 2008) refers that: “A distributed database with locking conflicts can easily become a performance nightmare”.

The major disadvantage of this approach is that the independence between microservices is lost which will consequentially limit the growth and scaling of the system as a whole.

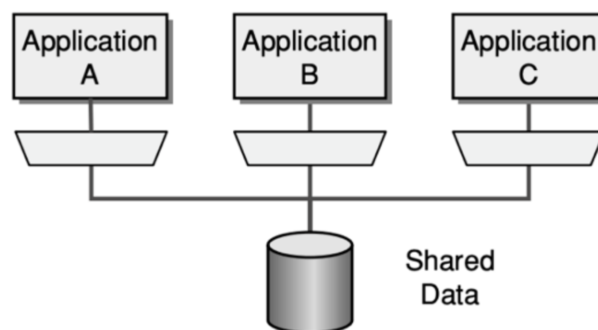


Figure 23 - Shared Database Pattern

Source: (Hohpe & Woolf, 2008)

4.2.6 Conclusion

In this phase, after following all the previous steps, it is expected to have a base design for a functional, easily maintainable, scalable, and resilient microservices solution. It is certainly not the same solution running a year from now as it is not the perfect solution, and it will need to suffer a lot of work to reach the desired level of perfection. The main objective of following these guidelines is to avoid common pitfalls and creating potential bottlenecks in the long term. Without these concerns, the evolution of your microservices application should be smoother and consequentially involve fewer maintenance costs.

4.3 Design Validation

During this chapter, the problem of implementing microservices in greenfield projects was investigated, and a treatment was designed. In the scope of the Technical Action Research method, the treatment consists "of an artifact interacting with a problem context", which in this case is the guide being applied to a greenfield project (Wieringa, 2014). However, before implementing the designed treatment there is still one step remaining in the engineering cycle, the design validation.

The design validation process includes four main knowledge questions that allow understanding what is expected from the solution, and assess the robustness of the designed artifact under changes of the problem and the context:

- Expected effects: What will be the effects of the artifact in a problem context?
- Expected value: How well will these effects satisfy the criteria?
- Trade-offs: How does this treatment perform compared to other possible treatments?
- Sensitivity: Would the treatment still be effective and useful if the problem changes?

4.3.1 Expected effects and value

The expected effect for the designed artifact is that its application to a greenfield project can lead into a satisfactory/successful microservice-based or hybrid software. It is also expected that the result promotes new iterations for improvement of the solution.

One of the most important questions regarding the validation of the design is whether it will really satisfy the expected effects or not. Furthermore, it is crucial to measure how likely are the expectations to be satisfied.

To assess the expected value of the artifact, the Quantitative Evaluation Framework (QEF) was applied (see section 6.4). This method used, as input, the results of the application of the Microservices Architecture Assessment Platform which can assess some of the most important concerns about microservice architectures.

4.3.2 Trade-offs

During the design of the treatment, for each step, a solution was presented along with alternatives. That way, it is possible to understand which solution can be applied in the best-case scenario, but also have access to different approaches that, despite having their flaws exposed, can be followed when the project has development restrictions such as budget, time, or human resources.

Certain steps of the guide do not include the design of a perfect solution because they may depend on external factors (e.g. size of the project or size of the development team). In these cases, presenting multiple alternatives while providing information about each one's trade-offs is even more important because the artifact is expected to fit as many project types as possible.

4.3.3 Sensitivity

The designed treatment is considered most effective when applied to the preconditions described in section 4.1. However, a hybrid approach is also suggested when the problem or the context slightly changes. This allows the artifact to be more flexible and be used and improved by more people.

Although the guide presents five major steps to build microservices from scratch, it is expected that they need to be adjusted and improved. It is also expected that by applying the artifact to more problems, new steps appear, and the existing ones are iterated.

5 Guide Application

The application of the designed guide not only allows its evaluation in chapter 6 but also provides relevant inputs to improve its contents. During the development of the project, new challenges, problems, and questions are likely to arise, which can reveal the need to update the proposed guide.

5.1 Application

A greenfield software solution, based on microservices, was developed using the guide. This project consists of a platform to manage the main processes of a community pharmacy. It was found as an opportunity given that there is still a lack of alternatives in this area, and, as it was detailed in Attachment 1, there is still a margin for improvement and addition of new functionalities. Given the timeframe of the project, only the top priority requirements identified in the same attachment are going to be implemented. These consist of improving the inventory management and introducing the business into the online market by creating an ecommerce solution. The developments of this prototype as well as additional documentation can be found in (Cardoso, 2021).

5.1.1 Step 1 – Should the guide be used?

The first step before applying the guide was to understand whether it would fit the project to be implemented. Even though the prototype to implement in the scope of this thesis is just part of the whole project and is reduced when compared to a whole platform for community pharmacies, there is still a lot of complexity in the processes to implement.

The main part of the prototype is an ecommerce web application that should allow a community pharmacy to manage and sell their products online. This module by itself involves business needs such as managing customers, the products, and their stocks, creating orders, managing

payments, and handling deliveries. Therefore, the complexity of the application is adequate to apply the guide, but also within the scope of the academic work under discussion.

Two other concerns that may indicate that the guide is not adequate for a project are the need for the system to scale, and the size of the development team. Being the prototype implemented to a small specific client it would not require high scalability by itself. However, the application is intended to grow and be adopted by other pharmacies around the country, and in the future by pharmacies of other countries. This will require that the system can scale, which should be implemented from the start of the project, therefore microservices, and especially this guide should be a good fit. Regarding the size of the team, it could be a dealbreaker implementing microservices and using this guide since in the scope of this application there is only one developer. In a normal scenario, the team would be bigger but, due to its scope, only one person worked on it. In this special scenario, the hybrid approach specified in section 4.2.1 will be adopted. The components of the microservices application will be implemented in a single solution, even though they will be developed as they were isolated. This will simplify the development process while the team does not grow and allow an easy migration when it does. Besides that, monitoring and infrastructure concerns will not be implemented just yet, even though a solution should start to be designed.

5.1.2 Step 2 – Service Identification

After verifying the validity of the guide for the project it was necessary to understand its domain and use microservice decomposition techniques so that the microservices could be defined. As a starting point, the domain model, business capabilities, and the use cases were designed so that the identification of bounded contexts and aggregates was easier. With these, it was possible to apply tools to define the services to implement.

5.1.2.1 Domain Modeling, Business Capabilities, and Use Cases Identification

In this section is analyzed the purpose, structure, and areas of expertise of the business to be prototyped. The results of this process are the domain model (see Figure 52 in Attachment 4), the set of business capabilities, and, to describe how these capabilities will be materialized, the use cases are represented in a use-case diagram (in Figure 24). All the artifacts focus only on the necessary entities to satisfy the requirements to be implemented in the prototype (e-commerce related). It is expected that these artifacts help to build the bounded-contexts and aggregates proposal in the next section.

The business capabilities are what a business needs to achieve its goals. In the context of e-commerce, the following capabilities were identified for the Pharmacity prototype:

- Manage and show real-time pricing for products
- Manage current and future product stocks
- Track orders and deliver them to the customers
- Isolation and quantification of the e-commerce sales and revenue
- Offer customer specific personalization

- Guarantee customer information security

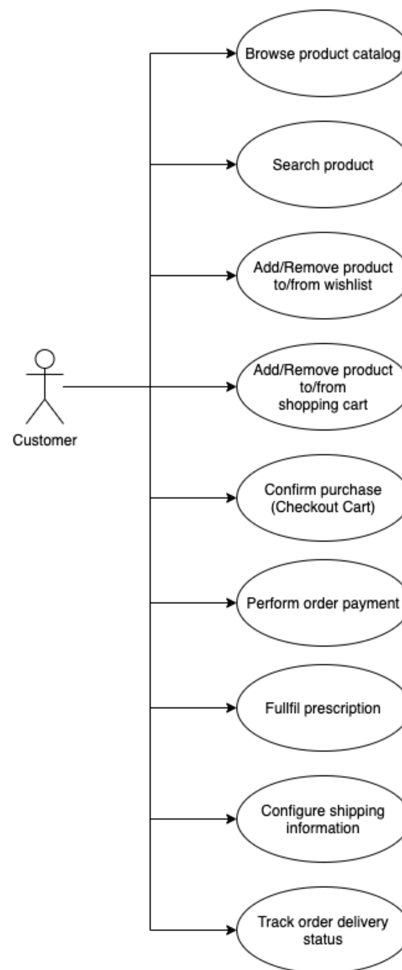


Figure 24 - Application Use-Case Diagram

5.1.2.2 Bounded Contexts

Bounded context identification can be performed following an approach suggested by Nick Tune. It starts by using DDD to draft a set of experimental models, they are then iterated to identify and fix possible development bottlenecks that may appear in the organization (Tune, 2017).

Based on the domain design of section 5.1.2.1, considered the interactions and dependencies of the domain entities, the bounded contexts represented in Figure 25 were thought:

- **Prescriptions Context**

  Since the prescriptions are a completely different kind of product in the context of a pharmacy's sales and have very different business rules and requirements it makes sense to have a specific context for them.

- **Orders Context**

The concept of order in a pharmacy is similar to order in any sales/market context. It is associated with the concretization of a sale of one or more products of the reseller. Consequentially the order will be one of the most important concepts for the company since it is the money generator, and for the client since it will be associated with a payment and a delivery that contains sensitive customer's information. Given the complexity of the whole order entity, this was considered to be a separate context.

- **Deliveries Context**

When an order is created and a payment is received, it needs to be delivered to the customer. It is a requirement to be able to track the order delivery status which will imply integrating external services of the partner shipping companies. Therefore, this should be the scope of a single context.

- **Product Browsing Context**

Products sold in the pharmacy can have different origins and may require integrations with existing stock management systems. Besides that, even though the concept of product in this scope does not involve very complex logic, it contains a lot of information and is shared by essentially every other scope (orders, shopping bag, etc.). Therefore, it should be centralized in a single context that is responsible to synchronize the products' data across the whole system.

- **Payments Context**

Payment is one of the most important entities in the ecommerce domain. It is critical during the order concretization, otherwise the order should not be fulfilled. Given that the payments result during the checkout of an order, and the checkout by itself does not seem to require a specific context, it makes sense to create a context that is responsible for the payment and the process of checking out an order.

- **Shopping Cart Context**

During the customer's buying journey, he needs to save the products that he wants to buy, similar to a real shopping basket in a physical store. Since these products are not real stock items and are a mere reference to the real product it makes sense to be in a separate context than the orders and the browsing context. Therefore, the shopping cart context seems to be the best solution to gather the actions of storing the desires of the customer until he decides to complete the order.

- **Wishlist Context**

While the customer is browsing the products catalog, he may like a product but not want to buy it right away. A wishlist is a place where the customer can save products without reservation so that they can be bought in the future if still available. Even though the context is related to a specific customer, it also involves storing products' information. Since it would not make sense to attach products directly with the Customer's context, the wishlist context is needed to create the connection.

- **Customer Context**

  Customers that visit Pharmacity's ecommerce solution are free to browse the products catalog anonymously, but once they want to save items in a permanent (cross-device) bag and check out an order, they will need to be authenticated in the system. For that, the physical customer needs to be translated into a "customer entity" in the application. Thus, this new entity requires storing data of different types: payment information, personal data such as contacts or physical attributes for a personalized experience, and address information for order deliveries. Such amounts of personal data with specific logic should have their context.

5.1.2.3 Aggregates

One of the most important patterns in DDD is the definition of aggregates, they consist of a group of objects that is not supposed to interact with the end-user separately (Joshi, 2018).

Based on the use cases defined in section 5.1.2.1, it is easier to understand the user interactions. Consequentially, it helps to identify boundaries around entities involved in each interaction.

To implement the aggregates represented in Figure 25, each aggregate root is attached to one or more value objects that contains an identifier of the corresponding entity in its original aggregate. For example, the Order entity belongs to an aggregate that does not involve any other entity, however, there are use cases where it is necessary to associate the Order to other entities outside its aggregate such as Payment, Customer, or Delivery. To be able to identify the status of the Order's delivery, the Order entity contains a Value Object "Delivery" which consists of the Delivery Id, with that value it is possible to access the Delivery aggregate containing all the relevant information to the use case. The same applies to the other entities associated with the Order.

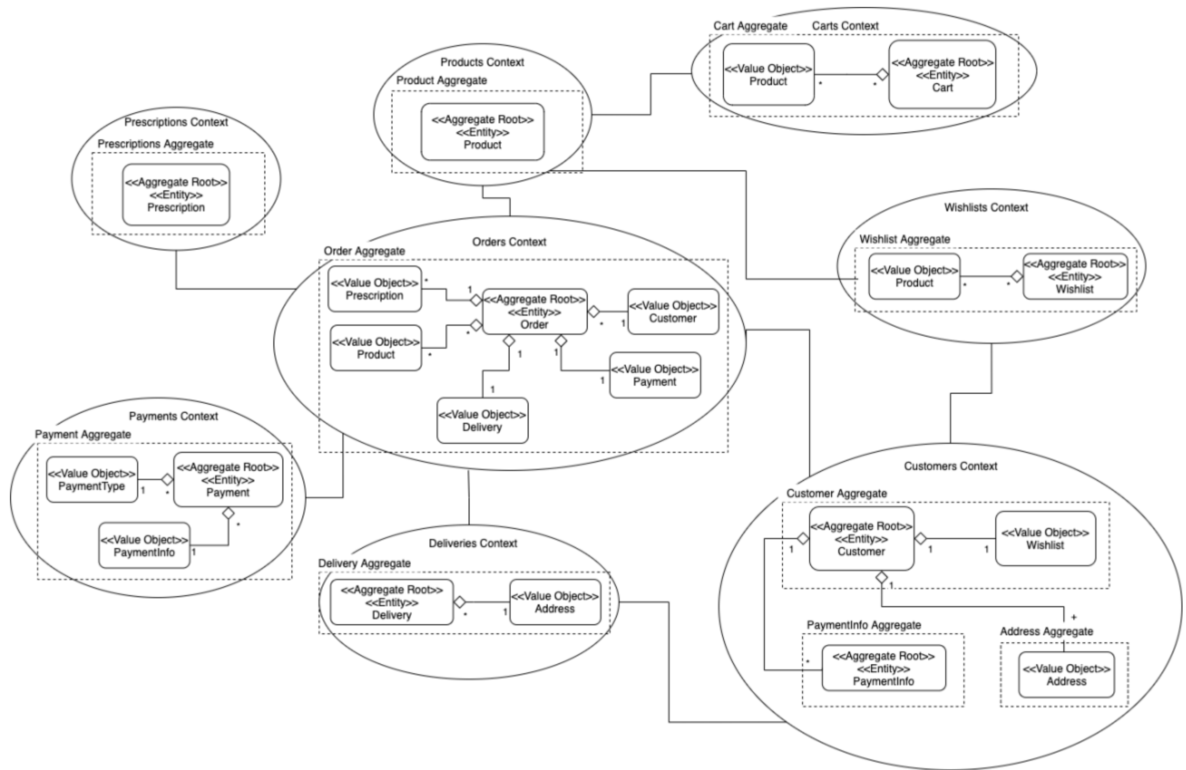


Figure 25 - Bounded contexts and aggregates diagram

5.1.2.4 Service Cutter Service Identification

Based on the specified domain model and use cases specified in section 5.1.2.1, the domain model and user representation JSON files were created to be used as input for the Service Cutter tool. These were uploaded to the tool, resulting in the “System Specification” which an excerpt is shown in Figure 26.

System #1

Name: **PharmaCity E-Commerce**

Nanoentities

Order.trackingId Order.value Prescription.number Prescription.accessCode Prescription.optionCode Prescription.attachment Prescription.comments Payment.amount Brand.name Brand.description DeliveryMethod.name DeliveryMethod.cost Category.name Wishlist.id Cart.id Customer.id Customer.name Customer.email Address.deliveryAddress Address.phone Address.postalCode Address.city Address.country PaymentType.name CartItem.productId Promotion.percentage Promotion.type Inventory.name Product.name Product.sku Product.description Product.price Product.quantity Product.color Product.size Catalog.name WishlistItem.productId

Coupling

Filter Coupling Criteria instances

Content Volatility - CHARACTERISTIC - **Often**

price quantity

Content Volatility - CHARACTERISTIC - **Rarely**

deliveryAddress phone postalCode city country id name email name name sku color size

Content Volatility - CHARACTERISTIC - **Regularly**

trackingId value number accessCode name optionCode productId attachment comments amount description name cost name id id name productId percentage type name description

Figure 26 - Service Cutter System Specification

Based on the provided input files and by using the same priority criteria, the tool generated four different suggestions (based on different algorithms) of the service division for the application:

- Markov Cluster Algorithm

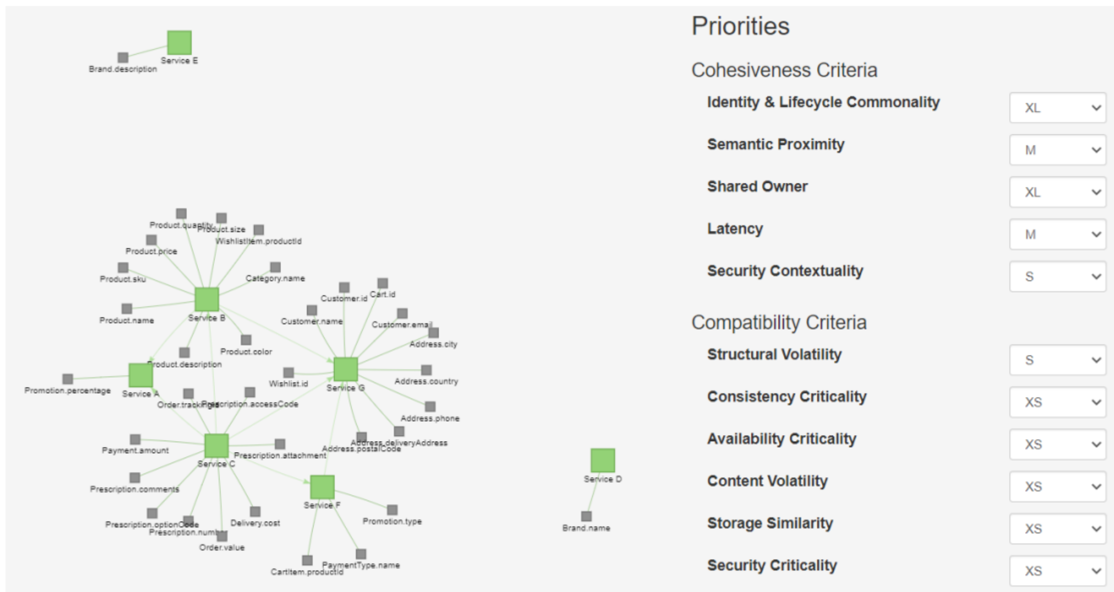


Figure 27 - Service Cutter Markov's algorithm results

- Leung's Epidemic Label Propagation

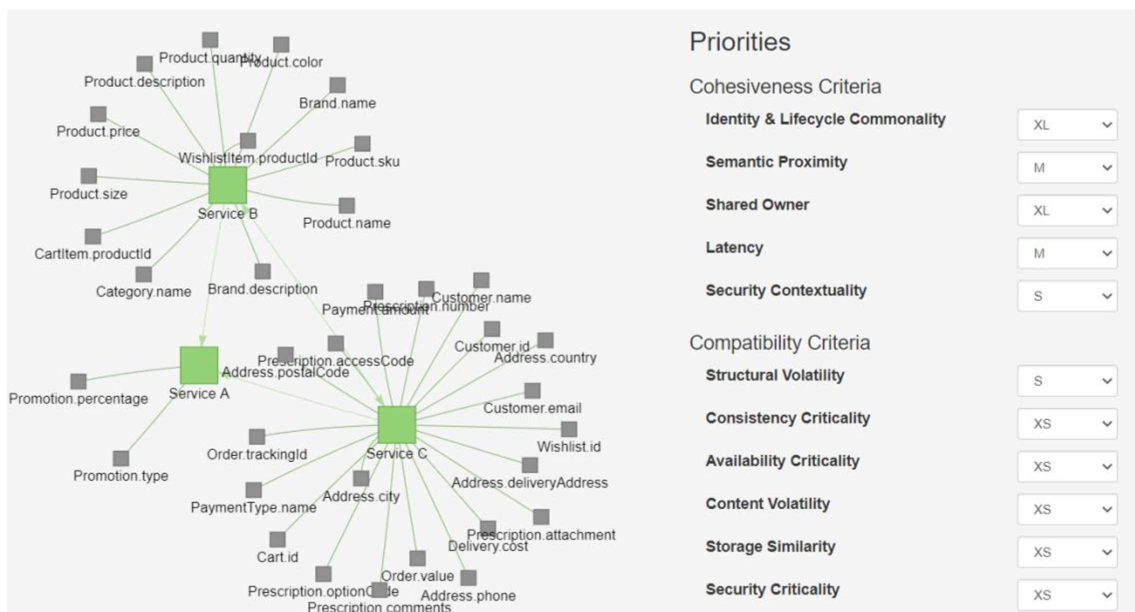


Figure 28 - Service Cutter Leung's algorithm results

- Chinese Whispers

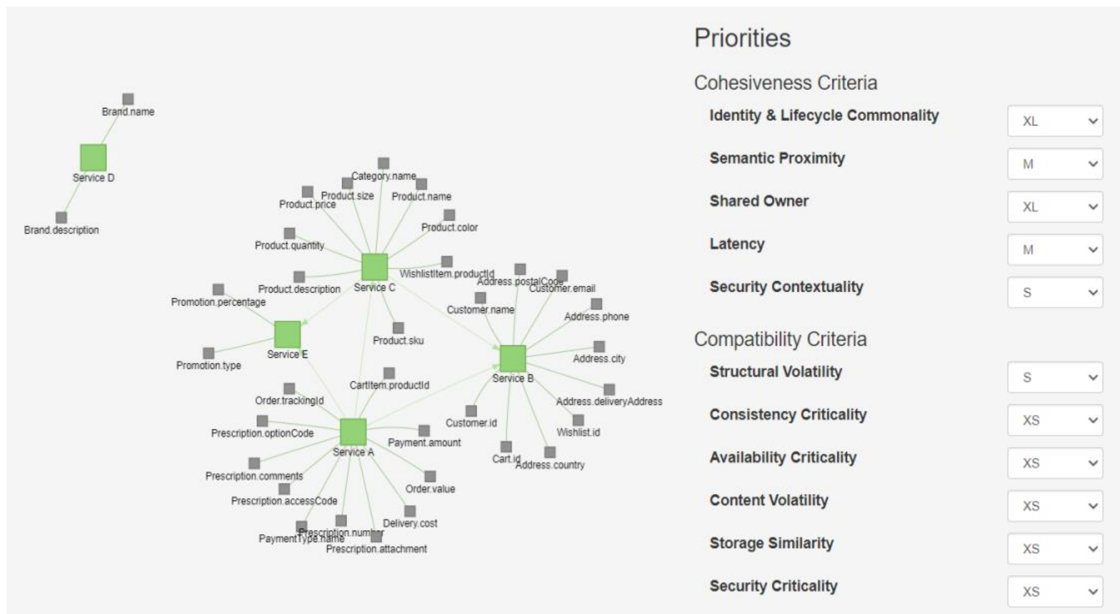


Figure 29 - Service Cutter Chinese Whispers algorithm results

- Girvan-Newman

Figure 30 - Service Cutter Girvan-Newman algorithm results

The results of the Service Cutter evaluation are a lot different for each algorithm although there are a few similarities between Chinese Whispers and Markov (MCL) tending to isolate the "brand" entity in an isolated service. Also, every algorithm suggested separating "product" related logic from "prescriptions", "addresses", "customers", "wishlist", and "carts".

The suggestions that seem to make more sense for the microservice decomposition are the following:

- Separation of "customer" and "address" information from the rest of the entities in a single service is suggested by Chinese Whispers and MCL algorithms.
- Markov, Chinese Whispers, and Girvan-Newman algorithms tend to suggest the isolation of "product" details from the rest of the model, even though it may make sense to include "category" and "brand" information in the same service.

Although none of the algorithms mentioned the isolation of "cart" or "wishlist" from other entities, it can bring more flexibility to the application growth since it is an area where a lot of product-related requirements are likely to appear in the future so these will be implemented as separate services.

5.1.2.5 Microservices Definition

After developing a domain model and identifying the use cases it was easier to propose a division of bounded contexts and business capabilities in 5.1.2.2. Moreover, the Service Cutter tool was used to provide different suggestions for the microservices definition. Figure 31 shows the resultant microservice division for the first iteration of the application to be implemented. However, as recommended in RQ_2 section, the microservices' division will not be closed yet, and may still suffer changes during the next steps of the guide.

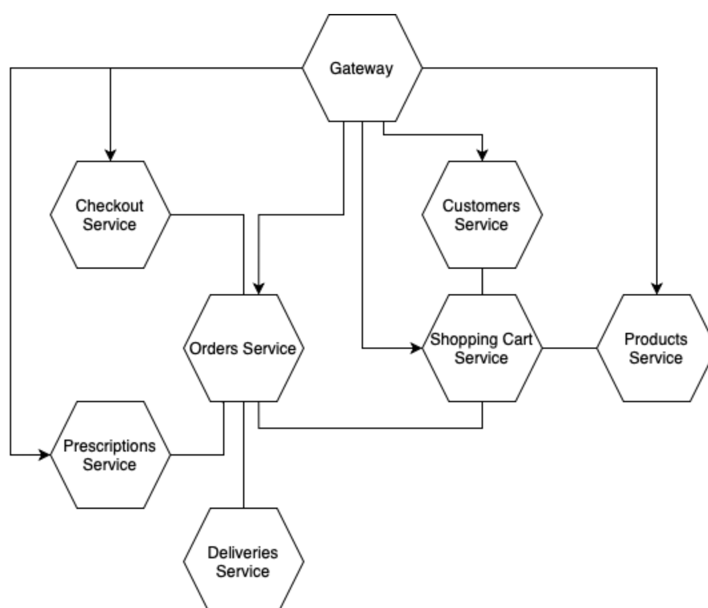


Figure 31 - Microservices Definition Diagram

5.1.3 Step 3 - Infrastructure and Communication

This section aims to understand at a lower level which will be the data transactions between the identified services. In Table 9, for each interaction are represented the associated data flows and the type of transaction to be used.

Table 9 - Data Flows Between Microservices

Source	Destination	Transaction Type	Data Flows
API Gateway	Products Service	GraphQL Query	• Search Product • List Products
API Gateway	Carts Service	GraphQL Query	• Cart Items
		GraphQL Mutation	• Add product to cart
		GraphQL Mutation	• Remove product from cart
API Gateway	Orders Service	GraphQL Query	• Order Details
API Gateway	Checkout Service	GraphQL Mutation	• Pay and checkout order
API Gateway	Customer Service	GraphQL Query	• See customer details
API Gateway	Deliveries Service	GraphQL Query	• Check delivery status by tracking id
Orders Service	Deliveries Service	Async Message	• Create new delivery
Orders Service	Carts Service	Async Message	• Clear ordered products from cart
Checkout Service	Orders Service	REST POST	• Create order

5.1.3.1 Main Buy Flow

Recalling the defined transaction types for each data flow, it was designed the interaction between components for each user story. In this section the design for one of the main flows in the application is documented: the one that allows a customer to search for a product in the catalog, add it to his shopping bag, and checkout the order by performing a payment. This will then trigger the creation of a delivery in an asynchronous way.

Figure 32 presents the sequence diagram that shows how the different components of the application interact to complete the application’s buy flow. It involves 9 different components excluding data stores. However, most of the interactions are completely independent and

therefore do not require special consistency concerns. The most important transaction in terms of consistency during this flow is the creation of a new order when the payment is assured. Since the order can only be considered when the payment is complete, the two services are dependent in terms of flow (order service is triggered by checkout service). However, regarding data, these services are completely independent having their private databases and therefore accessing only their data. This kind of autonomy is also expected across other application use cases. Regarding transactions that do not have critical consistency requirements, such as creating a delivery, or clearing the cart after an order is complete, asynchronous messages were used to improve the overall performance of the application.

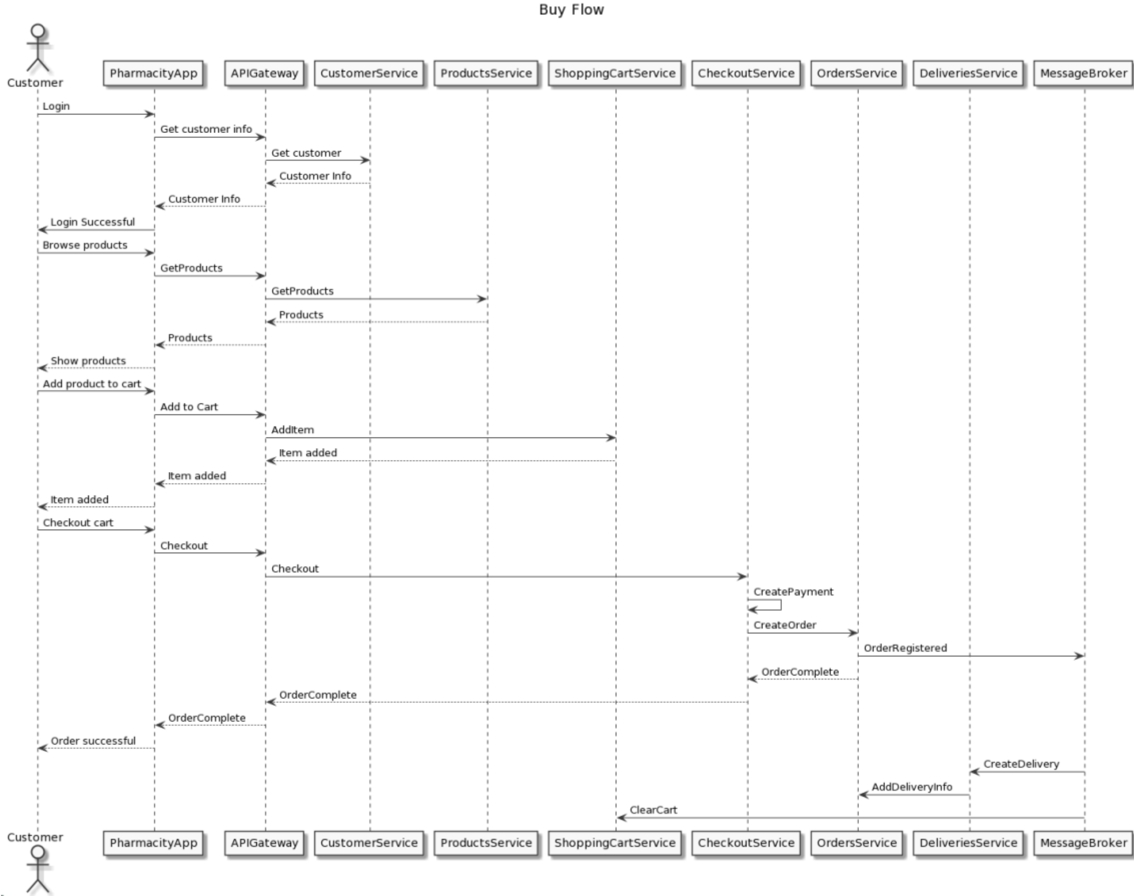


Figure 32 - Buy Flow Sequence Diagram

5.1.4 Step 4 - Monitoring

Given the importance of monitoring microservices (documented in section 4.2.4) and the relevance of its implementation in the early stages of the project, the suggestion of applying a simple solution that allows knowing at each moment the status of every service was followed.

Since the application's microservices are based on GraphQL APIs, the monitoring solution should be adapted to this technology.

A simple GraphQL system consists of four main layers (schema definition, query routing, resolvers and data loaders) and the monitoring solution can be applied to any or all of these places. Unlike REST, monitoring endpoints in GraphQL does not give much information about the service. Since there is usually a single endpoint, it simply verifies if the service is up and running. To have a broader vision of what is happening in the service, the monitoring needs to be applied at the resolver level, this can give more information about the status of different operations inside each service. This approach brings a concern though, since in GraphQL, resolvers and data loaders are not attached to queries it becomes harder to trace the entire flow of a request. This makes it harder to identify the source of a problem when it is detected at the data loading level. Although it may be a complex task, a solution for this concern may be tagging each query with a tracing identifier that is passed along until it reaches resolvers and data loaders. The monitoring solution to be implemented in the Pharmacity application is based on this approach and is represented in Figure 33.

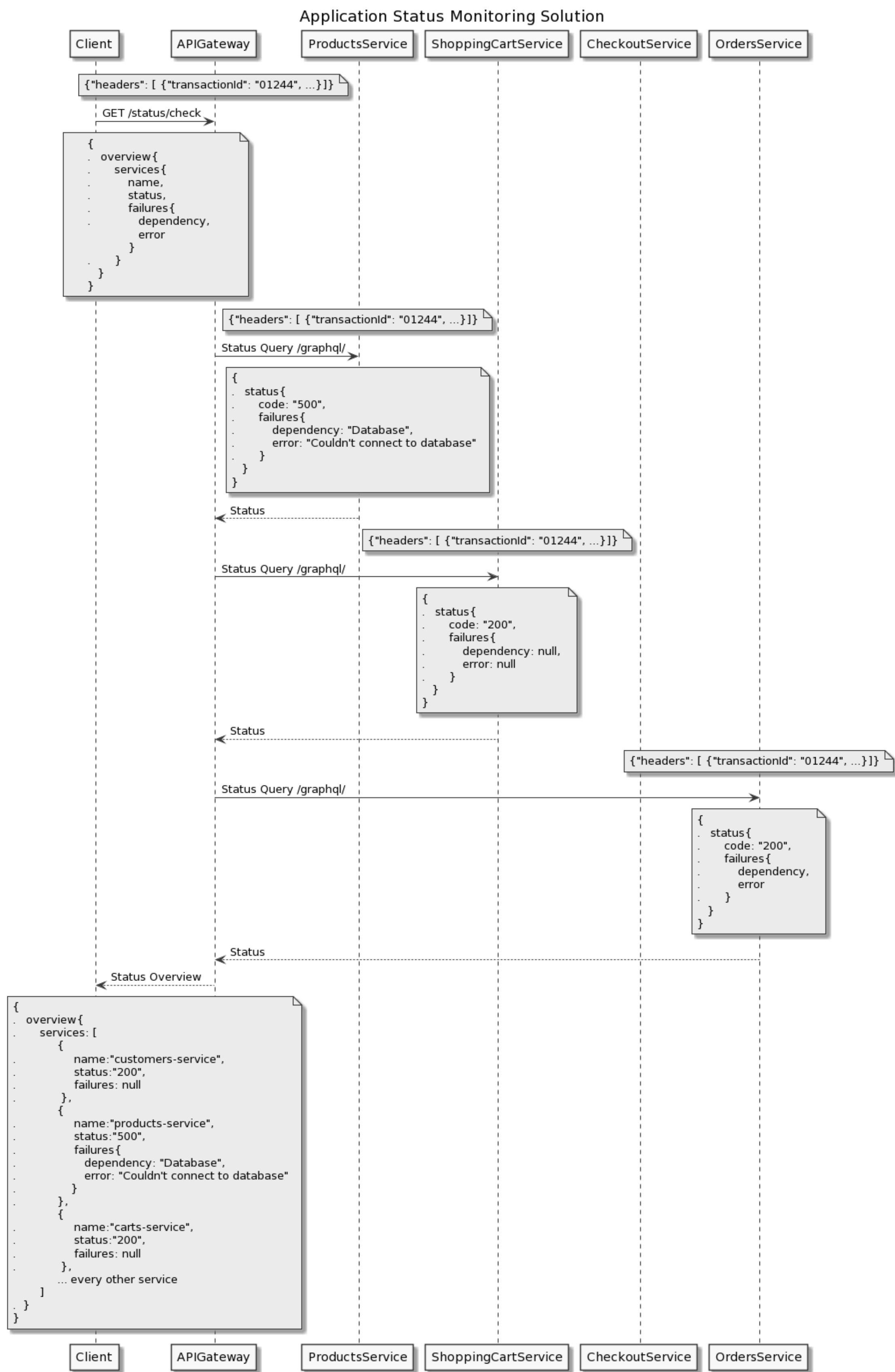


Figure 33 - Monitoring Solution Sequence Diagram

This solution is two-folded:

a) Provides a tool that checks the health of each service under the gateway based on the status of their dependencies. This does not give detailed information about what is happening in each service but allows to identify any critically failing service so that the problem can suffer a deep dive investigation and be mitigated.

b) Allows identifying every transaction in the system (except for asynchronous messages) from the time it is created by the client until it reaches the deepest service where data is processed or stored. The transaction identifier is sent through an HTTP header "TransactionId" which is set by the client every time a request to the API Gateway is performed. This header is then propagated to the services below no matter the used communication protocol. Since the two protocols used in the application (GraphQL and REST) are HTTP-based, there is full compatibility with this approach. And, if in the future any other HTTP-based protocol is used for the microservices communication, this will still be a valid solution.

It is important to have into consideration that the presented monitoring solution is not perfect, but, following what is suggested in section 4.2.4, given the needs of the solution in this initial phase, the main objective was to create a framework to be used by every service of the application, and that could give a simple but wide overview of the whole system. These two aspects provide a scalable solution, and at the same time fulfill the project needs.

5.1.5 Step 5 - Data Storage and Management

In section 4.2.5, the importance of maintaining data-independent between services is highlighted. After designing an application where service cutting is aimed to minimize shared contexts, there are still some operations that are executed across multiple contexts, and therefore require synchronization between the services involved. Another concern that leads to the adoption of microservices is the need for high scalability. Since relational databases struggle with horizontal scaling, it could be a bottleneck for this application in the future, therefore it was chosen a document-based non-relational database (MongoDB) for the application's services. This choice was based on three main aspects:

- Since the data is not structured and the database is document-based it allows to easily migrate from a single database for multiple services to database by service without breaking any compatibility on the data access.
- MongoDB offers high data consistency and allows horizontal scaling which will consequentially provide scalability to the whole application.
- For simple queries such as the ones to be implemented in this application, MongoDB is high performing.

However, applying a database by service approach allows that in the future different technologies are adopted in each service of the application. Below in Figure 34 is represented the application services with the implemented data stores.

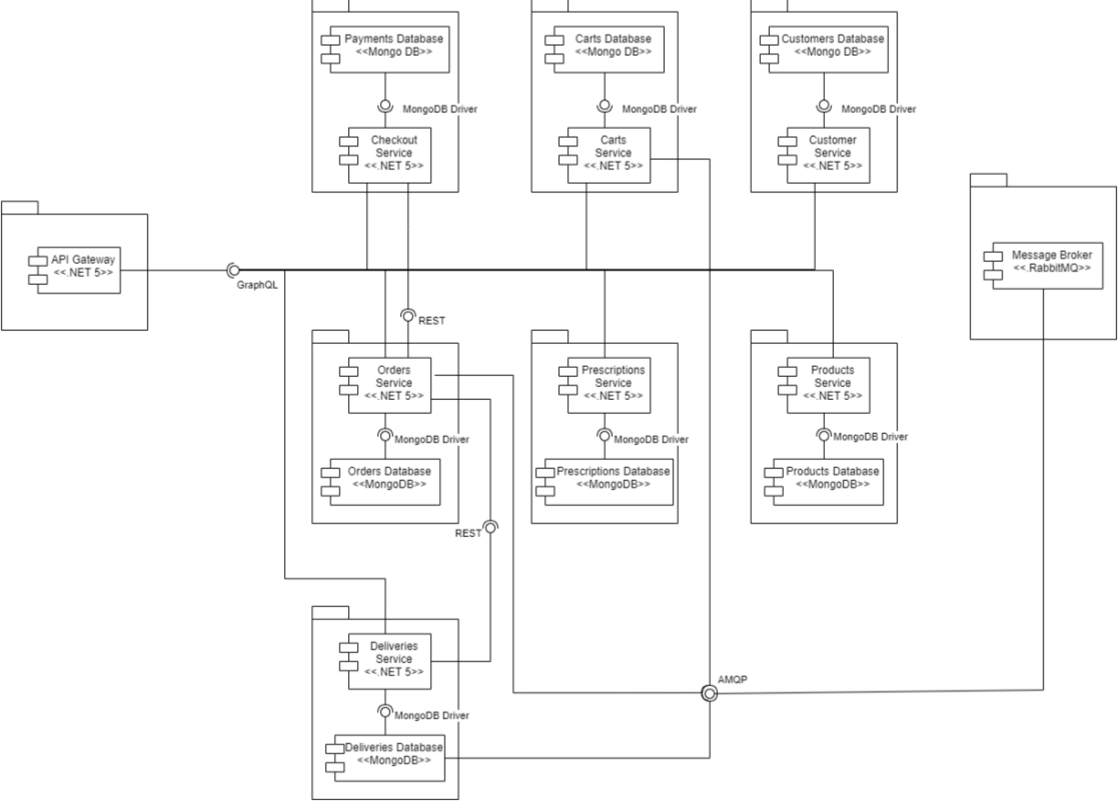


Figure 34 - Application Components Diagram

5.2 Step 6 - Improvements

During the application and evaluation of the guide were found gaps in some of its steps and a few that were missing. In this section are described all the improvements that lead to a more mature version of the guide. The more the guide is used and iterated for projects in the future, the more mature the solution will be.

5.2.1 API Standards Documentation

In microservice architectures, components are supposed to be independent which gives autonomy to the different development teams. The communication technologies may vary a lot from application to application, and even inside each application depending on the type of operation to be performed. However, for the components to be isolated and still establish communication they need to define a communication protocol. The most common way of communication between microservices are REST APIs, these interfaces expose a set of operations that are known and can be used by the other services.

Given that teams are isolated, and their services tend to grow and change quickly in this type of architecture, every decision and change must be documented and available to every consumer. What is also very important is that a documentation standard is defined and used by every team involved so that it is easier to maintain documentation organized.

This concern of the guide provides an example of how teams can document their development:

a) Decision logging

Behind every API there is in most cases, business logic that may limit or influence how data is provided to the clients. During the development of most APIs there are meetings with stakeholders to discuss constraints and requirements and generally involve a certain negotiation until the final version of each endpoint is defined. Even though this logic should not be known by the consumers, it is still very important to be documented so that both stakeholders and development teams are aware of the reasons why the API was built the way it was. This will likely help in future developments, and if the scope of the API moves to a different team it makes the handover much smoother and less time-consuming.

A good way to guarantee that all these decisions are recorded for future reference is to enforce the creation of a document on a shared space (e.g. Atlassian Confluence) every time a decision is made.

b) API Protocol/Contract Documentation

While decisions are made and the API is developed, a contract with the future consumers is being defined. To make it easier for clients to understand each of the API's features public documentation should be developed in parallel. The technology and methodology to use will vary significantly depending on the technology used in the project, however, in this step an example is shown for REST APIs.

Swagger is a set of tools that along with many other features, provides the capability to generate and maintain API documentation using "OpenAPI Specification" which is the *de facto* standard for RESTful APIs specification. One of its tools allows to auto-generate documentation from an API definition, and if the definition is not yet implemented, there is another tool that is capable of generating the OpenAPI definitions during runtime (Swagger, n.d.).

Even though Swagger is a powerful tool and supports long-term advanced usage, it also provides simple yet useful tools that can be used without spending much precious time during the beginning of the development of a microservices-based project. Figure 35 shows an example of an API specification and the corresponding visualization on Swagger UI.

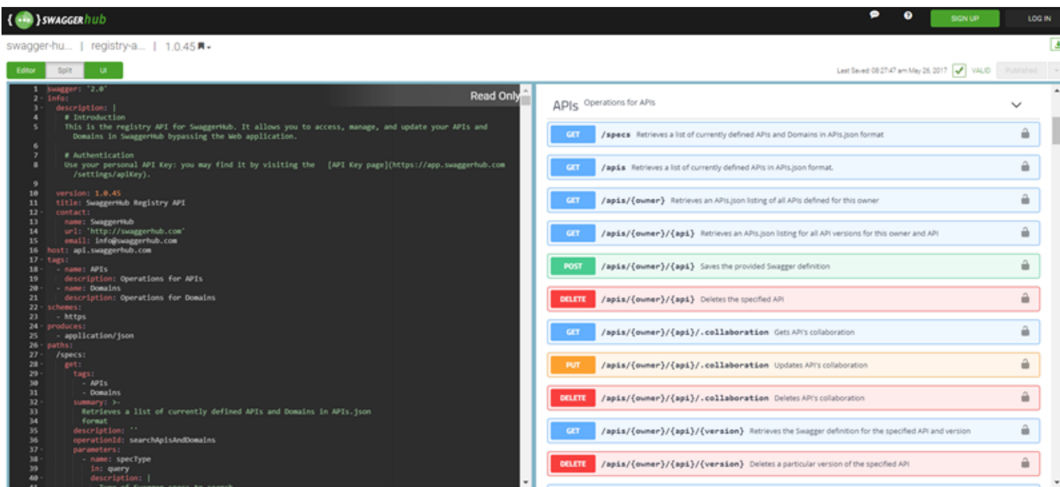


Figure 35 - Swagger Example

Source: (Swagger, n.d.)

Solution:

By choosing GraphQL as the standard communication protocol of the application, API standardization becomes simpler. Unlike REST APIs, which require documentation to understand how to use them, with GraphQL, developers only need to learn the syntax of the querying language and can work with any application that supports it. Since a GraphQL server requires the predefinition of the schema, it becomes available for the consumers natively. REST APIs would require to use a tool like Swagger (as suggested above) to generate and manage documentation.

The services and API Gateway were implemented using .NET 5 integrating Chilicream's HotChocolate Framework for GraphQL support. The development followed an annotation-based approach which ultimately results in an automatically generated schema in Schema Definition Language (SDL) to be used by clients. In Attachment 5 is represented the schema for the prototyped application which is published by the Gateway component.

Even though one of the best parts of GraphQL is having access to all data in a single place, until recently it was necessary to implement the whole application schema in a unique codebase. Since GraphQL is mostly used in microservice-based applications, a new concept "Schema Stitching" appeared to solve this problem. It allows each independent service to develop and expose their schema and combine all of them in a single API (see Figure 36). That way, the development and deployment of the services can be completely isolated (Stubailo, 2017).

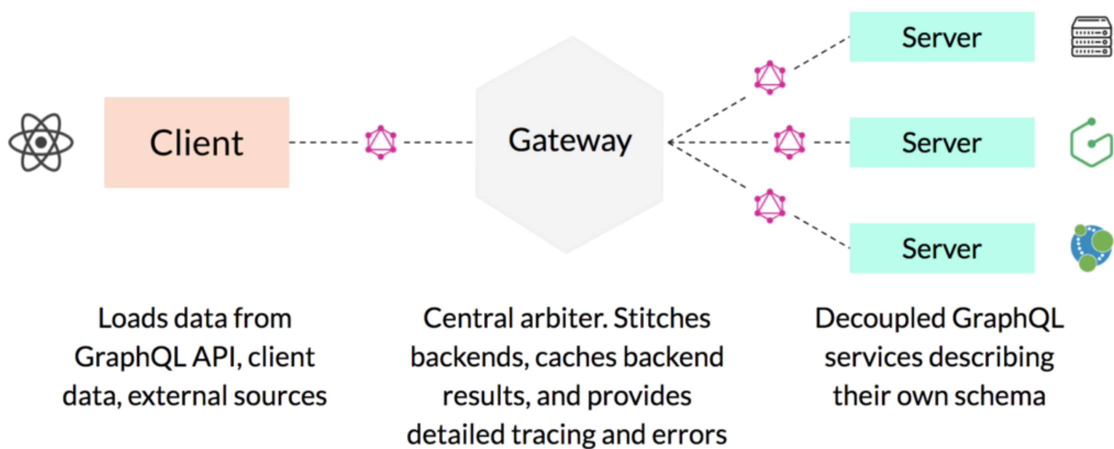


Figure 36 - GraphQL Schema Stitching

Source: (Stubailo, 2017)

5.2.2 Service Discovery and Registration

Since microservices need to communicate, they need to know each other's location in the network. In traditional applications that run on physical hardware, their location is usually static which makes it possible to be registered in configuration files for example. However, in modern applications that are executed inside virtual environments, such as Docker containers, their location is typically dynamic. Furthermore, each service application usually runs multiple instances, and these can vary dynamically in number by autoscaling tools. This problem makes service discovery a very important component in the microservices architecture since it may allow an effective way to access each service's location, and consequentially reducing their coupling (Tang et al., 2019).

When choosing to implement service discovery in the microservices architecture the main decision to be made is which type to use:

- **Server-side Discovery**

This kind of service discovery is based on a router that provides the functionality to the services that call it with the information about the required downstream service. It has the advantage of reusability since it does not require the usage of any client libraries which will allow working along with cross-technology services. However, as it is shown in Figure 37, it requires a new component with high availability which may impact performance negatively (Haselböck et al., 2017).

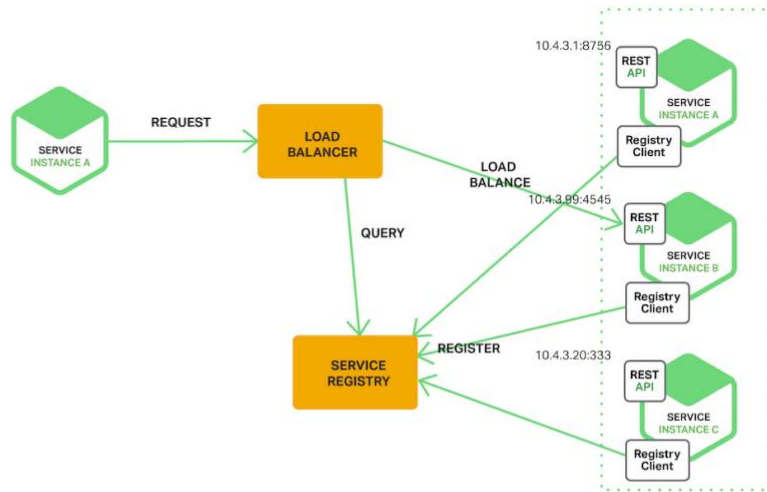


Figure 37 - Server-side Service Discovery
Source: (Richardson, 2015)

- **Client-side Discovery**

The client-side discovery (shown in Figure 38) does not require a specific component to perform the service discovery, instead, it queries the service registry directly to get the location of the necessary downstream services. It requires that each microservice uses a library that is specific for its language/technology. Contrarily to server-side alternative, this approach has better impacts on performance but demands that microservice deployment integrates the client library (Haselböck et al., 2017).

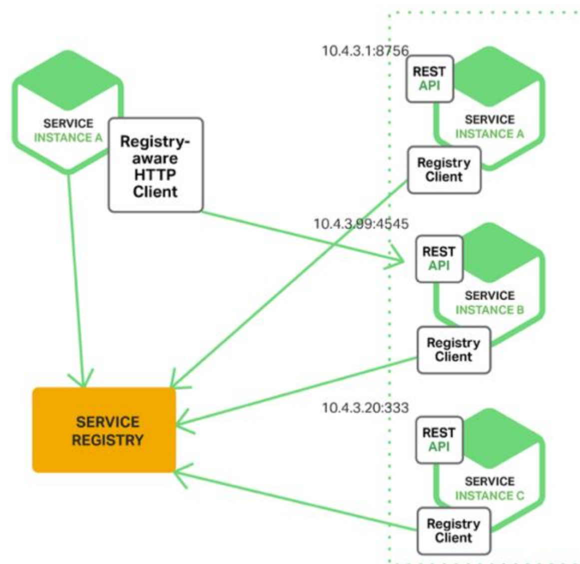


Figure 38 - Client-side Service Discovery
Source: (Richardson, 2015)

When service discovery is implemented, there is the need for a service registry where the location of all services' instances is stored. The storing process is called service registration and

according to Haselböck et al. there are three major and transversal concerns associated with it: “microservices should be registered automatically after launching”, “they should be deregistered automatically after shut down”, and the implementation of the service registration should be “technology-independent” (Haselböck et al., 2017).

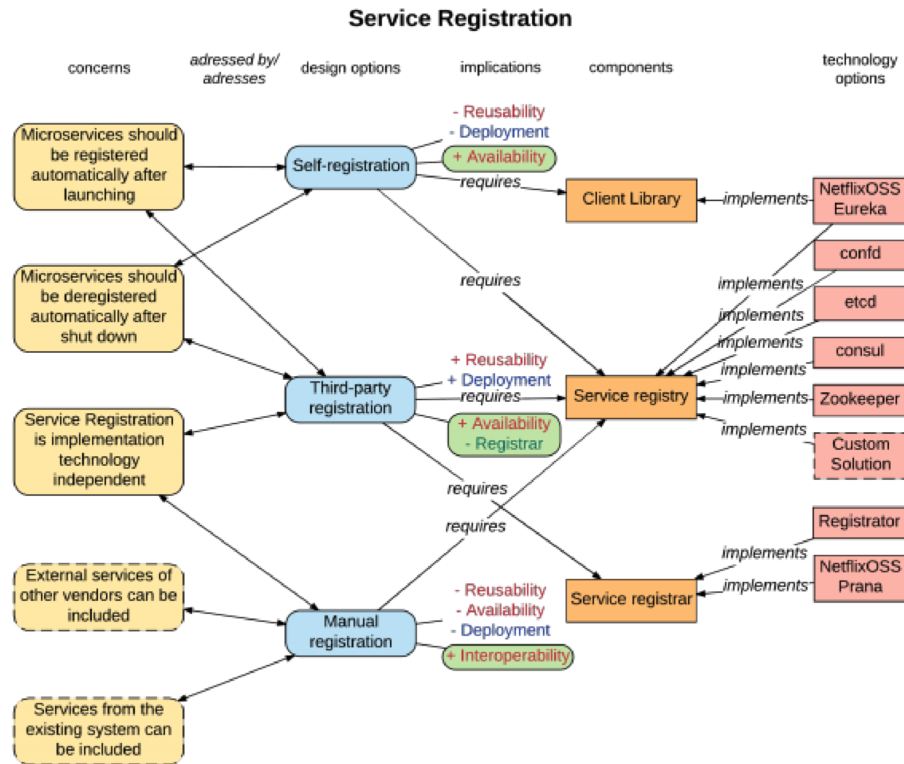


Figure 39 - Service Registration Decision Guidance Model

Source: (Haselböck et al., 2017)

Solution:

Since the services are all implemented in the same technology (.NET Core) it would be possible to implement a service discovery solution that would fit the whole application. However, to allow implementing new services using other technologies, or integrating external services, a language-agnostic solution brings more flexibility.

Given that the designed architecture potentiates containerization, a good solution for the service discovery and registration concerns may be Kubernetes. This open-source tool allows to automate the deployment, scaling, and management of containerized applications. It not only deals with all the communication between service instances (pods), but also provides configuration of automatic scaling for certain levels of traffic. During the deployment process, Kubernetes gives all Pods a specific IP address and creates unique DNS name for the set of Pods. This deals with the service registration and discovery, and allows load-balancing through the set of instances.

5.2.3 Observability and Resilience

The monitoring concern was described in section 4.2.4, but needs to be complemented with observability. Monitoring is the process of collecting data regarding how the application is behaving, on the other side observability refers to what can be done with that information.

One of the most important features brought by microservices architectures is high scalability. This means that in a production environment multiple instances of each service could be running in parallel which may have different statuses at a given point. Thus, it is very important to be able to know the whole time if any of the instances has a problem and implement mechanisms to prevent it from affecting the system and the client. If a certain instance of the service has an internal problem and is not able to respond successfully, it should stop receiving requests for a while so it can try to recover.

During this step, the development team must focus on guarantying two things:

- Create a mechanism to identify error patterns on services responses and top requesting failing instances.
- Configure alerts when error rate thresholds are achieved.

The status of a service is already provided by the implemented monitoring solution, however, as it is, it is not prepared to deal with multiple instances of the same service. Once service discovery and registration solutions are implemented, it will be easier to implement resilience on that layer since it should know all the services' instances. Most of the solutions that support service discovery and provide tools for scaling and load balancing, also deal with the management of "broken" instances and automatically stop sending requests to them. That is the case with Azure Kubernetes Services.

Solution:

The implemented monitoring solution (see section 5.1.4) allows detecting a failing instance of any service, however that feature is only being used for alerting in the prototype. An important improvement of the application would be circuit-breaking the failing instance so that traffic is redirected to the remaining working instances. Also, tools for monitoring application metrics and collecting logs should be integrated.

To implement this step of the guide the best option is to follow the approach suggested above.

5.2.4 Deployment Infrastructure

As mentioned in section 3.4.3, DevOps implementation can bring quicker delivery to the development of microservice-based architectures. One of the most important practices that allow it is the definition of deployment infrastructure.

This step will consist of two phases: configure the application services to run in containers with all the necessary communication and data stores and build a continuous delivery pipeline that

can compile the code, test it, build the containers using the previously defined configuration, and deploy the code into development and production environments by applying all the necessary configurations.

a) Configure Containerization

To take advantage of the features specified in section 3.4.3, it is necessary to configure the application to use containers during the deployment. Moreover, to build a resilient and cohesive microservices system it is also important that these containers are managed by an orchestrator such as the one described in section 3.4.3.3. Below is shown an example of a solution for this step, it consists of using Docker containers and configuring Kubernetes clusters to deal with the management of the services and their instances.

The first step is the creation of the Docker image for each microservice in the application. The simplest way to do it is to add a Dockerfile to the SCM repository, it will be different depending on the service technology stack, in Code Snippet 1 there is an example for a .NET Core application using .NET 5.

```
FROM mcr.microsoft.com/dotnet/aspnet:5.0
COPY bin/Release/netcoreapp3.1/publish/ App/
WORKDIR /App
ENTRYPOINT ["dotnet", "aspnetapp.dll"]
```

Code Snippet 1 - Dockerfile example for .NET microservice

Then, build the docker image and tag it with the desired name by running the command shown in Code Snippet 2.

```
docker build -t pharmacity/products-service:1.0
```

Code Snippet 2 - Build microservice docker image

After building the images for every microservice of the application it is time to configure their deployment to Kubernetes. To use Azure Kubernetes Service (AKS), the images must be pushed to an image registry which can be Azure Container Registry or DockerHub. After doing that, one deployment configuration YAML file like the example of Code Snippet 3 should be created for each service.

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: products-service
  name: products
spec:
  replicas: 1
  selector:
    matchLabels:
      service: products
  template:
    metadata:
      labels:
        app: products-service
        service: products
    spec:
      containers:
        -
          env:
            -
              name: ASPNETCORE_URLS
              value: "http://+:80"
          image: "exploredocker.azurecr.io/products-service:1.0"
          imagePullPolicy: IfNotPresent
          name: products-service
          ports:
            -
              containerPort: 80
              protocol: TCP
```

Code Snippet 3 - Kubernetes Deployment Configuration File Example

Then a Kubernetes cluster resource must be created in Azure, its credentials must be retrieved, and the commands listed in Code Snippet 4 should be executed to deploy the application to that cluster.

```
kubectl apply -f deploy-products-service.yml
kubectl apply -f deploy-other-service.yml
...

kubectl get all
```

Code Snippet 4 - Kubernetes deployment commands

b) Build CD/CI Pipeline

The configuration of a continuous delivery/integration pipeline is one of the best ways to shorten the distance between development and release of code to production. It not only standardizes processes between development teams but also allows developers to focus on

their specific tasks without needing to worry about preparing testing environments or deploying the applications manually.

There are many alternative tools to implement pipelines being Jenkins the most common and used. Despite that, since the suggested deployment infrastructure is based on Azure, the simplest way to integrate it with the rest of the pipeline is using the Azure DevOps pipelines in a similar way to what is represented in Figure 40.

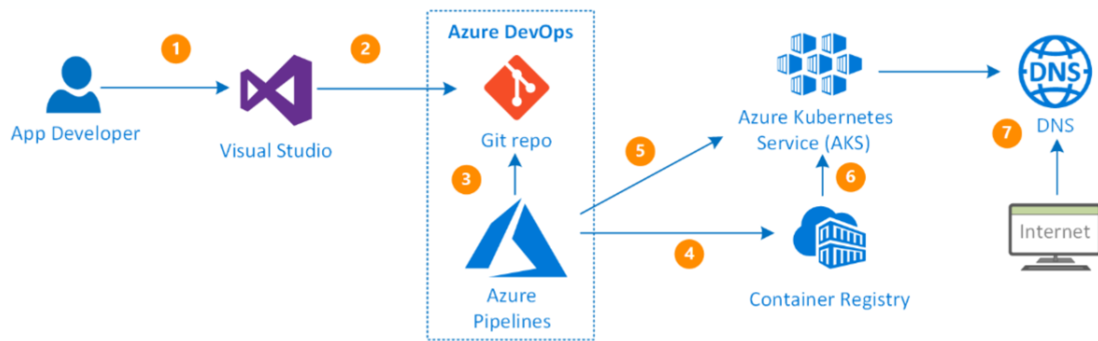


Figure 40 - Azure DevOps pipeline example

Source: (Kumar, 2020)

This tool allows integrating all the needed steps to deploy containers to a Kubernetes cluster in a few clicks, avoiding what is described in the previous step after the image building process.

Solution:

The application was developed towards service isolation allowing containerization. It makes horizontal scaling easier and more efficient, however it is still necessary to implement the process for scaling automation. Aside from the scaling process it is also important that the application's configurations are externalized so that it is easier to deploy to different environments and sensible information such as credentials are stored safely. These features are covered by the solution presented above and, therefore, it can be used to implement these features in the prototype.

6 Evaluation

In software engineering, the definition of a good quality evaluation process is extremely important because, unlike many others, software products can be very hard to measure in terms of quality, even though it is deeply important. As stated by (Boehm et al., 1976):

> "Suppose you receive a software product which is delivered on time, within budget, and which correctly and efficiently performs all its specified functions. Does it follow that you will be happy with it? For several reasons, the answer may be "no"."

The authors then explain why software products can satisfy their users but still have bad quality: there can be high costs of maintenance, usability problems, or issues related to integration with other software and hardware.

This section describes the evaluation of the software solution documented in chapter 5, but also the guide proposed in this dissertation. First, the research hypotheses are defined, and the indicators and information sources are presented, followed by the method to perform the evaluation, and ends by listing the results of such evaluation and the corresponding conclusions.

6.1 Investigation Hypotheses

The objectives of this work (see section 1.3) are to investigate what has been considered by the developers' community as good and bad practices for successful microservice-based software solutions, and with the results of the investigation, create a guide to help developers implementing a greenfield project based on this type of architecture in a more structured way.

To evaluate if the implemented guidelines are valuable for the implementation of microservices in greenfield projects the following hypotheses were defined:

$$H_0 : Using\ the\ guidelines\ of\ this\ work\ in\ a\ greenfield\ project$$
$$does\ not\ allow\ to\ achieve\ a\ sucessful\ microservices\ solution$$

$$q < 60\%$$

$$H_1 : Using\ the\ guidelines\ of\ this\ work\ in\ a\ greenfield\ project$$
$$allows\ to\ achieve\ a\ satisfatory\ microservices\ solution$$

$$q \geq 60\%$$

Where q is the quality of the system when applying the QEF method (explained in section 6.3). This value reflects the quality of the entire solution by evaluating each of the identified dimensions and considering their relative importance.

6.2 Indicators and Sources of Information

Here are identified the indicators and the information sources of the evaluation to perform. They are important to understand what information needs to be collected to perform the desired evaluation and where it should come from.

Two major aspects were analyzed, the quality of the overall architecture as well as the proximity of the solution with the documented microservice patterns in section 4.2.

6.2.1 Microservices Assessment Platform

To understand if the chosen approach for this project was successful, the Microservices Assessment Platform was used. This platform was developed by Chris Richardson and provides a tool to assess the architecture of microservices-based applications and identify areas that need to be improved (Richardson, 2021).

The assessment is based on two parts:

- **Application Assessment**

The first part evaluates the application's overall architecture based on the following aspects: key software delivery outcomes, general architecture, inter-service communication, deployment and reliability, observability, externalized configuration, supporting infrastructure, libraries and frameworks, documentation, and organization and process.

- **Service Assessment**

The second part assesses specific aspects of each service: key software delivery outcomes, service design, externalized configuration, inter-service communication, service reliability, deployment/release, observability, documentation, automated testing, and organization and process.

Therefore, the indicators to be used in this evaluation are presented in Table 10.

Table 10 - Evaluation Indicators

Indicators	Description
Application architecture follows the right patterns for microservices	The application architecture should follow the recommended patterns for microservices development.
Compliance with microservices non-functional requirements: - Performance - Cost - Time Cost - Deployment Risk	The resulting solution should be compliant with the requirements for a successful microservice software application described throughout this document

The sources of information for the indicators above will be the defined criteria for each requirement identified during the application of the Quantitative Evaluation Framework (QEF).

6.3 Evaluation method

To measure quality, specific and quantifiable measurement units and criteria must be defined. However, it is frequent that quality characteristics are not directly measurable through specific metrics, in those cases, it is necessary to decompose these characteristics into measurable ones (Escudeiro & Bidarra, 2008).

Quantitative Evaluation Framework

In the software evaluation model proposed by (Escudeiro & Bidarra, 2008), the Quantitative Evaluation Framework could be used to validate and evaluate the software quality in any phase of its lifecycle and consequentially allow to detect and correct the identified problems. This model is based on several definition phases: dimensions, factors, requirements, criteria, and measurement levels.

This framework seems to be appropriate to be applied to the solution developed in this dissertation since it was specifically designed to evaluate the quality of software solutions and it is simple to perform since the quality value is proportional to the percentage of accomplished requirements which was one of the chosen aspects to be analyzed in this section.

- **Dimensions:** to perform a QEF, the first step is to identify the dimensions to be evaluated in the developed software solution. These match some of the most important characteristics that were identified as indispensable to microservice-based applications throughout this document.

- **Factors:** for each dimension, there is the need to define which will be the factors that will allow to measure it. A factor is nothing more than an aggregator of requirements with a certain percentage of the total weight of the dimension.
- **Requirements:** inside each factor is included a set of requirements that are assigned a relevance value of one of the following values: 2, 4, 6, 8, and 10.
- **Criteria and measurement levels:** to be able to set an evaluation value for each of the identified requirements, criteria and measurement levels need to be defined for each one.

Quality Calculation

After having defined the dimensions, factors, requirements, and criteria, it is possible to calculate a percentage value for the quality of the system.

The quality of each factor is obtained by summing the multiplication of each of its requirements' weights by its fulfillment and dividing it by the sum of the requirements' weights. It is translated to the following formula:

$$Q_{factor} = \frac{\sum_m (pr_m \times pc_m)}{\sum_m pr_m}$$

Where m is the number of relevant criteria for the factor, pr_m the weight of the criterium *m*, and pc_m the fulfillment percentage of the criterium *m*.

With all the factors' quality value calculated it is possible to calculate the value for the dimension that contains them. It is given by the sum of the products of each of its factors' weight and their quality (Q_{factor}).

The global deviance of the system (Euclidian Distance) relative to the ideal system is given by:

$$D = \sqrt{\sum_j \left(1 - \frac{Dim_j}{100}\right)^2}$$

Finally, the percentage value for the quality of the system is given by:

$$q = \left(1 - \frac{D}{\sqrt{n}}\right) \times 100$$

6.4 Application of the Quantitative Evaluation Framework

This section identifies the dimensions to be evaluated along with their factors and requirements, these are based on the Microservices Architecture Assessment Platform questionnaires and summarized in Table 11.

General Architecture and Software Delivery Outcomes

This dimension assesses if the architecture consists of multiple distributed services and if it enables the rapid, frequent, and reliable delivery of software.

Factors:

- Key software delivery outcomes
  - Release/Deployment frequency
  - Frequency that a deployment leads to outage
  - Time to recover from an outage
  - Tracking and review of outage metrics
- General Architecture
  - Microservice architecture must consist of two or more independently deployable/executable components
- Inter-service communication
  - Documented API Standards and style guides
  - External client access to service mechanisms
  - Service discovery mechanism
  - Service registration patterns

Deployment and Reliability

This dimension measures the capacity of the services in the solution to be deployed independently and in a reliable way. It is highly connected to the coupling level between the services and implemented mechanisms for the process.

Factors:

- Deployment process
  - Deployment options definition
  - Horizontal scaling support
- Deployment reliability
  - Failure detection and short-circuiting mechanisms
  - Fault injection in production prevention

Monitoring and Observability

The monitoring and observability dimension evaluates how simple and efficient it is to understand the state of the application at any time based on its outputs.

Factors:

- Audit and tracing
  - Usage of audit logging
  - Distributed tracing
  - Errors/exceptions tracking

- Application metrics
    - Collection of application metrics
    - Health check mechanism
- Tracking Deployments
    - Track configuration changes
    - Log deployments

Infrastructure and Configurations

This dimension is related to the application's strategy regarding configurations and the infrastructure that supports it.

Factors:

- Configurations
    - Externalized configuration strategy
    - Sensitive properties safe storage
- Supporting infrastructure
    - Development/testing environment provisioning time
    - Similarity between testing and production environments
    - Deployment pipeline infrastructure support

Libraries and Frameworks

Developing a microservice-based application takes requires exceptional effort to implement cross-cutting concerns such as logging, monitoring, metrics, and distributed tracing. This dimension refers to using helper tools for the development and testing and implementing strategies to minimize the effort of cross-cutting concerns. One way to do it is by creating microservices using a "microservice chassis framework", a framework that takes care of the cross-cutting concerns, allowing the developers to focus on the core of the application.

Factors:

- Definition of microservice chassis frameworks
- Use testing and development tools/frameworks

Documentation, organization, and processes

This dimension refers to the documentation of the architecture and every important decision, it also focuses on assessing the development organization and the processes.

Factors:

- Documentation
    - Accessible architecture documentation
    - Technical decisions are logged and documented
- Organization
    - Weekly working hours

        - Scope of the development staff
    - Process
        - Percentage of uninterrupted technical work
        - Percentage of time for training
        - Percentage of time reducing technical debt

Table 11 - QEF Dimensions, Factors, and Requirements Identification

Dimension	Factors	Requirements
General architecture and software delivery outcomes	Key software delivery outcomes	Release/Deployment frequency
		Frequency that a deployment leads to outage
		Time to recover from an outage
		Tracking and review of outage metrics
	Inter-service communication	Documented API Standards and style guides
		External client access to service mechanisms
		Service discovery mechanism
		Service registration patterns
	General Architecture	Microservice architecture must consist of two or more independently deployable/executable components
Deployment and reliability	Deployment Process	Deployment options definition
		Horizontal scaling support
	Deployment Reliability	Failure detection and short-circuiting mechanisms
		Fault injection in production prevention
	Audit and tracing	Usage of audit logging

Monitoring and observability		Distributed tracing
	Application metrics	Collection of application metrics
		Health check mechanism
	Tracking Deployments	Track configuration changes
		Log deployments
Infrastructure and configurations	Configurations	Externalized configuration strategy
		Sensitive properties safe storage
	Supporting infrastructure	Development/testing environment provisioning time
		Similarity between testing and production environments
		Deployment pipeline infrastructure support
Libraries and frameworks	Microservices chassis	Definition of microservice chassis frameworks
	Testing/development tools	Use testing and development tools/frameworks
Documentation, organization, and processes	Documentation	Accessible architecture documentation
		Technical decisions are logged and documented
	Organization	Weekly working hours
		Scope of the development staff
	Process	Percentage of uninterrupted technical work
		Percentage of training time
		Percentage of time reducing technical debt

6.5 Results and Conclusions

It is important to notice that the evaluation of the work developed in this dissertation would be much harder without it being applied to a project. That is why the evaluation process directly assesses some of the characteristics of the developed prototype. The quality of the prototype was found as one of the best indicators for the consequent success of the guide even though it can only be considered valid for the same kind of project and having only one dedicated developer.

The QEF was applied very much inspired by the criteria of the Microservices Architecture Assessment Platform which allows to evaluate some of the most important concerns regarding microservice architectures.

The results of the application of that platform are schematized in Figure 41 which gives a score of 76 out of 100. The most positive aspects of the developed solution are the general microservices architecture, the documentation which is favored by the nature of the work (a structured guide), the inter-service communication, and the externalized configurations. The worse aspects of the solution are the lack of tracking and review over the release/deployment metrics by management and observability because of the missing audit logging features.

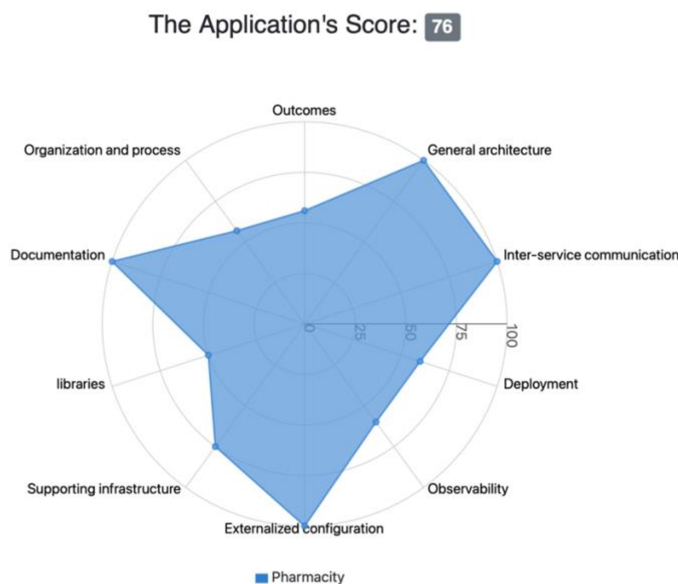


Figure 41 - Microservice Architecture Assessment Platform Results

Using the concerns raised by the mentioned platform and listed in Table 11, the QEF method was applied. The resultant global deviance of the system (D) value was 0,94, and the global quality of the system (q) was 74%. That value corresponds to Hypothesis 1 (H_1) which concludes that the designed artifact allows obtaining a satisfactory microservices solution. The complete set of results is in Attachment 6.

Due to the restrictions of this academic work, where only one person developed the whole solution, it was not possible to satisfy all the recommended factors to use the guide.

Consequentially the guide was only applied partially. During the evaluation was considered not only the implemented prototype but also developments such as infrastructure configurations and deployment processes, that were specified in the solution (see section 4.3) but will only be materialized in future iterations of the project.

To assess the level of fulfillment of each defined requirement were specified concrete criteria presented in Attachment 7.

7 Conclusion

This chapter summarizes what was achieved for every outcome defined in section 1.4.1. It also presents the results of the work and how it contributes to the software engineering community.

The Systematic Mapping (see section 3.2.1) showed a lack of documentation regarding microservices in greenfield projects. This makes this thesis an important contribution to the subject, namely in what is important to guarantee from the beginning, but also what to avoid.

Some of the limitations and potential improvements of the followed approach are clearer after the application of the guide in the Pharmacy prototype and applying the QEF method. The application of the proposed guide by different development teams will allow it to be improved.

7.1 Outcomes

Some key points to be achieved with this dissertation were defined in section 1.4.1. For each of them are presented the results and how they were accomplished.

1. **Research on microservice architecture development experiences and challenges:** it was possible, through the satisfactory amount of literature about the subject, to identify some of the biggest and most common challenges found when developing and maintaining this kind of software solutions (see section 2.2).
2. **Distinguish good patterns and approaches from bad ones:** most of the literature regarding microservice development is based on real use cases, which best reflect the true positive and negative aspects of the adoption of the microservice-based architecture in the real world. Nevertheless, there is also a set of researchers that invest a big part of their work in this type of architecture (Newman, 2015a) (Richardson, 2021) (Fowler, 2019) (Tilkov, 2015). From the literature review, it was possible to define some good and bad patterns and practices for microservices development (see sections 2.3 and 2.4 respectively).
3. **Identify from what was investigated what should be applied to greenfield projects and what should only be a concern in the longer term:** even though there is a lot of literature

regarding microservices, there is too little documentation on how to implement this type of architecture in greenfield projects, and those that exist do not document the whole development process. Given the shorter amount of literature for this specific topic, it was performed literature systematic mapping research. Based on the sources of information listed in 3.2.1.2, this research is conducted in section 3.2. The results of the research allowed identifying what are the most important processes when implementing microservices from scratch, and what are the steps that should not have a big focus during the initial phase.

4. **Design a solution to develop software using a microservices-first approach:** after conducting the research mentioned in the previous item, a set of guidelines were defined with multiple alternatives for each of the major steps for microservices implementation in greenfield projects. Even though some technologies were chosen to demonstrate the application of the patterns/methodologies, the solution does not aim to be technology-oriented, so that it can be adopted by the widest range of people.
5. **Iterate over the artifact to refine the existing steps with additional details and add new ones:** after designing the artifact, it was applied to a prototype to find limitations in the original solution and improve it. The concerns described in section 5.2 resulted from the reiteration of the design process, having as input the findings in the application of the guide to the exemplifying web application.
6. **Evaluate the designed solution based on the results of the QEF and identify limitations and improvement opportunities:** after designing the solution, applying it to a prototype, and adjusting some missing concerns, the Microservices Architecture Assessment Platform was used to assess if the solution can result in successful microservice-based software. The results of this assessment, along with the formalization of an evaluation based on the QEF method, also allowed to perform a last iteration over the guide. The results of this evaluation are documented in chapter 6 and allowed verifying that the usage of that version of the guide can originate satisfying results.

7.2 Results

The use of TAR methodology in this dissertation makes it important to guarantee that three roles were played and kept isolated by the researcher (Wieringa, 2012):

- **Technique developer:** by playing this role, the researcher investigated and developed a new set of guidelines to help to implement microservices in greenfield projects. Applying the new technique to a project not only contributes to the project itself but also to learn more about the technique and improve it.
- **Client helper:** this role can be seen from two different angles, with the designed guide the researcher contributed to the engineering community that can be seen as the major stakeholder of the project. Nevertheless, the application of the guide to a prototype in a business area that lacks evolution regarding software solutions can also be seen as a help to a specific client (community pharmacies).

- **Empirical researcher:** the research was concluded with the application of the developed technique and the investigation of its effects and utility in practice. Given that the project was based on TAR, similarly to other case studies, it is not possible to apply statistical inference to generalize the usage of the technique (Wieringa, 2012). In this case, after assessing the results of the application of the technique, only case-based inference can be used. This means that it is possible to admit that for similar cases of application, the technique should also have similar results. Thus, it is possible to infer that for medium-sized e-commerce projects, having only one working developer, it is possible to achieve a satisfactory result. Consequentially, it is probable that for the same project, having a team of several developers (most cases), the result would be different.

To assess the guide designed in chapter 4, it was applied to an exemplifying project in chapter 5 and the results were evaluated and documented in chapter 6.

The application of the Quantitative Evaluation Framework resulted in a value of 74% for the global quality of the system. It concludes that using the guidelines designed in this work in a medium-sized greenfield project with only one developer allows achieving a satisfactory microservices solution.

Since the improvements listed in section 5.2 were only identified during the application and evaluation of the guide, they were not initially considered steps of the guide. That can indicate that without these concerns, the result for the global quality of the system could be even better.

7.3 Limitations and Future work

This work focused on the backend of microservices solutions architectures. Given that microservices are gaining a lot of relevance in the Web Development domain it can be considered a limitation of the work and makes sense that frontend concerns receive more attention in future iterations of this guide. Since the frontend of microservice applications is becoming much more complex than in the early days, more challenges are likely to appear. To answer all of these a completely new guide must be created and, once it is ready, both guides should be adopted either isolated or in parallel.

It is also known that during this dissertation the guide application was limited. To improve the existing set of guidelines and identify new potential problems the guide needs to be applied to other projects with different characteristics and the development team should conform with all the recommendations defined in section 4.2.1.

7.4 Contributions of the work

As more and more companies are creating products that have the potential to grow, they usually end up needing to implement approaches that allow the projects to grow sustainably. Companies find microservice-based architectures as one of the most common solutions for a system's sustainable growth.

Even though migrations from monolithic architectures to microservices have been the most common method to develop big projects, it is an approach that usually brings high costs even for Greenfield projects that are expected to be migrated right after the creation of a monolith. The costs are higher since the total cost of the project will be at least the sum of the monolith creation plus the cost of the migration itself.

The microservices-first approach is an alternative that seeks to reduce the cost of the development of a big project while creating a solid foundation based on carefully analyzed assumptions. The work of this dissertation is a big contribution to this methodology since it gives a structured set of guidelines for the whole development and maintenance process which was not found so far in the literature.

The designed guide and the prototype available in (Cardoso, 2021) intend to provide a solid and versatile solution that developers can use to start their microservices greenfield projects. These contributions are also expected to benefit the guide with the improvement of its guidelines and the addition of new ones.

Even though the guide was applied to a prototype aiming to help the management of a community pharmacy, there may be other business areas lacking software architecture innovation. This prototype should only be seen as an example of how to adapt the designed guidelines to a specific set of technologies and a specific domain. The guide is not technology-driven and therefore can be applied to projects of any domain, using any technologies that support the principles it defines.

References

Amazon Web Services (AWS). (n.d.). *What is DevOps?* AWS DevOps. Retrieved February 24, 2021, from https://aws.amazon.com/devops/what-is-devops/

Anil, N. (2020). *Communication in a microservice architecture*. Microsoft Docs. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture

Arsanjani, A. (2004). Service-oriented modeling and architecture. *IBM Developer Works*, *January*, 1–15. https://www.ibm.com/developerworks/library/ws-soa-design1/

Barcia, R., Brown, K., Chen, G., Daya, S., Martins, M., Osowski, R., & Verwaayen, J. (2017). *Microservices Decision Guides*. https://www.ibm.com/cloud/architecture/files/Microservices-Decision-Guides-FINAL-1.pdf

Barthel, S. (2017). *Backpedal a microservice architecture of a greenfield project*. Medium. https://medium.com/hackernoon/backpedal-a-microservice-architecture-of-a-greenfield-project-77432905d90a

Bashari Rad, B., John Bhatti, H., & Ahmadi, M. (2017). An Introduction to Docker and Analysis of its Performance. *IJCSNS International Journal of Computer Science and Network Security*, *17*(3), 228–235.

Bjørndal, N., Mazzara, M., Bucchiarone, A., Dragoni, N., & Dustdar, S. (2020). *Migration from Monolith to Microservices : Benchmarking a Case Study*. *March*, 1–18. https://www.researchgate.net/publication/339749917_Migration_from_Monolith_to_Microservices_Benchmarking_a_Case_Study

Boehm, B. W., Brown, J. R., & Lipow, M. (1976). Quantitative evaluation of software quality. *Proceedings - International Conference on Software Engineering*, 592–605.

Boettiger, C. (2015). An introduction to Docker for reproducible research. *Operating Systems Review (ACM)*, *49*(1), 71–79. https://doi.org/10.1145/2723872.2723882

Boxma, O. J., & Weststrate, J. A. (1989). *Waiting Times in Polling Systems with Markovian Server Routing* (Issue R 8905). CWI. https://doi.org/10.1007/978-3-642-75079-3_8

Brito, G., Mombach, T., & Valente, M. T. (2019). Migrating to GraphQL: A Practical Assessment. *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, 140–150. https://doi.org/10.1109/SANER.2019.8667986

Brown, S. (2014). *Distributed big balls of mud*. Blog Article. http://www.codingthearchitecture.com/2014/07/06/distributed_big_balls_of_mud.html

Bucchiarone, A., Dragoni, N., Dustdar, S., Larsen, S. T., & Mazzara, M. (2018). From Monolithic to Microservices: An Experience Report from the Banking Domain. In *IEEE Software* (Vol. 35, Issue 3, pp. 50–55). IEEE Computer Society. https://doi.org/10.1109/MS.2018.2141026

Burns, B., Beda, J., & Hightower, K. (2019). *Kubernetes: Up and Running* (2nd ed.). O'Reilly Media, Inc.

Cardoso, J. (2021). *Applying "A Guide for Microservices in Greenfield Projects."* GitHub; João Cardoso. https://github.com/k4rd050-GitHub/pharmacity-ecommerce

Carneiro, C., & Schmelmer, T. (2016). Microservices From Day One. In *Microservices From Day One*. Apress. https://doi.org/10.1007/978-1-4842-1937-9

Carrasco, A., Van Bladel, B., & Demeyer, S. (2018). *Migrating towards Microservices: Migration and Architecture Smells*. https://doi.org/10.1145/3242163.3242164

Cervantes, H., & Kazman, R. (2016). *Designing Software Architectures: A Practical Approach*. Addison-Wesley Professional. https://www.oreilly.com/library/view/designing-software-architectures/9780134390857/

Chinnasamy, V. (2020). *5 Best Technologies To Build Microservices Architecture*. Clarion. https://www.clariontech.com/blog/5-best-technologies-to-build-microservices-architecture

Cinque, M., Della Corte, R., & Pecchia, A. (2019). Microservices Monitoring with Event Logs and Black Box Execution Tracing. *IEEE Transactions on Services Computing*. https://doi.org/10.1109/TSC.2019.2940009

Context Mapper. (n.d.-a). *A Modeling Framework for Strategic Domain-driven Design*. Contextmapper.Org. Retrieved February 23, 2021, from https://contextmapper.org/

Context Mapper. (n.d.-b). *MDSL (Micro-)Service Contracts Generator*. Contextmapper.Org. Retrieved February 24, 2021, from https://contextmapper.org/docs/mdsl/

Conway, M. E. (1968). How do committees invent. *Datamation*, *14*(4), 28–31.

Datz, T. (2004). What You Need to Know About Service-Oriented Architecture. *CIO*. https://www.cio.com/article/2439859/what-you-need-to-know-about-service-oriented-architecture.html

de Toledo, S. S., Martini, A., & Sjøberg, D. I. K. (2021). Identifying architectural technical debt, principal, and interest in microservices: A multiple-case study. *Journal of Systems and Software*, *177*, 110968. https://doi.org/10.1016/j.jss.2021.110968

DInh-Tuan, H., Mora-Martinez, M., Beierle, F., & Garzon, S. R. (2020). Development Frameworks for Microservice-based Applications: Evaluation and Comparison. *ACM International Conference Proceeding Series*, 12–20. https://doi.org/10.1145/3393822.3432339

Docker. (2018). *Docker overview | Docker Documentation*. Docker.Com. https://docs.docker.com/get-started/overview/%0Ahttps://docs.docker.com/get-started/overview/#docker-objects%0Ahttps://docs.docker.com/get-started/overview/%0Ahttps://docs.docker.com/engine/docker-overview/

Escudeiro, P., & Bidarra, J. (2008). Quantitative Evaluation Framework (QEF). *Conselho Editorial/Consejo Editorial*, 16.

Fowler, M. (2005). *Event Sourcing*. 18. https://martinfowler.com/eaaDev/EventSourcing.html

Fowler, M. (2014, August 28). *MicroservicePrerequisites*. Martinfowler.Com. https://martinfowler.com/bliki/MicroservicePrerequisites.html

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

Fowler, M. (2019). *Microservices Guide*. Martinfowler.Com. https://martinfowler.com/microservices

Fritzsch, J., Bogner, J., Wagner, S., & Zimmermann, A. (2019). Microservices Migration in Industry: Intentions, Strategies, and Challenges. *Proceedings - 2019 IEEE International Conference on Software Maintenance and Evolution, ICSME 2019*, 481–490. https://doi.org/10.1109/ICSME.2019.00081

Golden, B. (2020). *Microservice architecture design: 5 key elements*. TechBeacon. https://techbeacon.com/app-dev-testing/5-fundamentals-successful-microservice-design

Google. (2009). *Documentation - The Go Programming Language*. The Go Programming Language Official Documentation. https://golang.org/doc/

GraphQL. (n.d.). *GraphQL Architecture & Big Picture*. How to GraphQL. Retrieved February 27, 2021, from https://www.howtographql.com/basics/3-big-picture/

Gupta, D., & Palvankar, M. (2020). Pitfalls & Challenges Faced During a Microservices Architecture Implementation. *Cognizant*, *February 2020*. https://www.cognizant.com/whitepapers/pitfalls-and-challenges-faced-during-a-microservices-architecture-implementation-codex5066.pdf

Gysel, M., Kölbener, L., Giersche, W., & Zimmermann, O. (2016). Service cutter: A systematic approach to service decomposition. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *9846 LNCS*, 185–200. https://doi.org/10.1007/978-3-319-44482-6_12

Haag, S., Raja, M. K., & Schkade, L. L. (1996). Quality Function Deployment Usage in Software Development. *Communications of the ACM*, *39*(1), 41–49. https://doi.org/10.1145/234173.234178

Haselböck, S., Weinreich, R., & Buchgeher, G. (2017). Decision guidance models for microservices - Service discovery and fault tolerance. *ACM International Conference Proceeding Series*, *Part F1305*. https://doi.org/10.1145/3123779.3123804

Hassan, S., & Bahsoon, R. (2016). Microservices and their design trade-offs: A self-adaptive roadmap. *Proceedings - 2016 IEEE International Conference on Services Computing, SCC 2016*, 813–818. https://doi.org/10.1109/SCC.2016.113

High, R. J., Kinder, S., & Graham, S. (2005). IBM's SOA Foundation, An Architectural Introduction and Overview. In *IBM Systems Journal* (No. 1; Vol. 44, Issue 4). http://www-128.ibm.com/developerworks/views/webservices/libraryview.jsp?type_by=Articleshttp://researchweb.watson.ibm.com/journal/sj/444/ferguson.html

Hohpe, G., & Woolf, B. (2008). Enterprise Integration Patterns. In *Geriatric Nursing* (Vol. 29,

Issue 5). http://linkinghub.elsevier.com/retrieve/pii/S019745720800311X

Ismail, K. (2018). *7 Tech Giants Embracing Microservices*. CMSWire. https://www.cmswire.com/information-management/7-tech-giants-embracing-microservices/

James, T. (2019). *The Docker Book: Containerization Is the New Virtualization by James Turnbull* (18th ed.). James Turnbull. https://books.google.pt/books?id=4xQKBAAAQBAJ&printsec=copyright&redir_esc=y#v=onepage&q&f=false

Joshi, U. (2018). *Aggregate Oriented Microservices*. Medium. https://medium.com/@unmeshvjoshi/aggregate-oriented-microservices-d314eb04f2b1

Kapferer, S. (2018). *A Domain-specific Language for Service Decomposition* [UNIVERSITY OF APPLIED SCIENCES OF EASTERN SWITZERLAND (HSR FHO)]. http://eprints.hsr.ch/id/eprint/722

Kapferer, S., & Zimmermann, O. (2021). Domain-Driven Architecture Modeling and Rapid Prototyping with Context Mapper. In *Communications in Computer and Information Science* (pp. 250–272). Springer. https://doi.org/10.1007/978-3-030-67445-8_11

Kleinrock, L. (1985). Distributed systems. *Communications of the ACM*, *28*(11), 1200–1213. https://doi.org/10.1145/4547.4552

Kowall, J. (2020). *How microservices broke monitoring (and how to fix it)*. TechBeacon. https://techbeacon.com/app-dev-testing/how-microservices-broke-monitoring-how-fix-it

Kozlovski, S. (2018). *A Thorough Introduction to Distributed Systems*. FreeCodeCamp. https://www.freecodecamp.org/news/a-thorough-introduction-to-distributed-systems-3b91562c9b3c/

Kumar, P. (2020). *Azure DevOps CI/CD pipeline with AKS*. Medium. https://prakashkumar0301.medium.com/azure-devops-ci-cd-pipeline-with-aks-409fd2af52a0

Kurmi, A. (2020). *Top 10 Microservices frameworks for 2020 - Microservices Architecture - Medium*. Medium. https://medium.com/microservices-architecture/top-10-microservices-framework-for-2020-eefb5e66d1a2

Le, T. (2020, March 30). *What is SAGA Pattern and How important is it?* Medium. https://medium.com/swlh/microservices-architecture-what-is-saga-pattern-and-how-important-is-it-55f56cfedd6b

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

Li, S., Zhang, H., Jia, Z., Zhong, C., Zhang, C., Shan, Z., Shen, J., & Babar, M. A. (2021). Understanding and addressing quality attributes of microservices architecture: A Systematic literature review. In *Information and Software Technology* (Vol. 131, p. 106449). Elsevier B.V. https://doi.org/10.1016/j.infsof.2020.106449

Lucidchart. (2019). *How to Build a House of Quality (QFD)*. Lucidchart Blog. https://www.lucidchart.com/blog/qfd-house-of-quality

Lumetta, J. (n.d.). Microservices for Startups: Should you always start with a monolith? In *Microservices for Startups*. Retrieved November 29, 2020, from https://buttercms.com/books/microservices-for-startups/should-you-always-start-with-a-monolith

Mauro, T. (2015). Adopting Microservices at Netflix: Lessons for Architectural Design. In *Nginx Blog* (pp. 1–5). https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/

Mayer, B., & Weinreich, R. (2017). A dashboard for microservice monitoring and management. *Proceedings - 2017 IEEE International Conference on Software Architecture Workshops, ICSAW 2017: Side Track Proceedings*, 66–69. https://doi.org/10.1109/ICSAW.2017.44

Meng, F. J., Zhang, X., Chen, P., & Xu, J. M. (2017). DriftInsight: Detecting Anomalous Behaviors in Large-Scale Cloud Platform. *IEEE International Conference on Cloud Computing, CLOUD*, *2017-June*, 230–237. https://doi.org/10.1109/CLOUD.2017.37

Merkel, D. (2014). Docker: lightweight Linux containers for consistent development and deployment. In *Linux Journal* (Vol. 2014, Issue 239, p. 2). https://www.seltzer.com/margo/teaching/CS508.19/papers/merkel14.pdf

Montesi, F., Peressotti, M., & Picotti, V. (2021). *Sliceable Monolith: Monolith First, Microservices Later*. https://parken-und-laden.de/

Nadareishvili, I., Mitra, R., McLarty, M., & Amundsen, M. (2016). Microservice Architecture Aligning Principles, Practices, and Culture. In *Microservice Architecture*. O'Reilly Media, Inc. https://www.oreilly.com/library/view/microservice-architecture/9781491956328/

Newman, S. (2015a). Building Microservices. In *O'Reilly*. http://safaribooksonline.com

Newman, S. (2015b). *Microservices For Greenfield?* Samnewman.Io. https://samnewman.io/blog/2015/04/07/microservices-for-greenfield/

Nguyen, C. D. (2020). *A Design Analysis of Cloud-based Microservices Architecture at Netflix*. Medium. https://medium.com/swlh/a-design-analysis-of-cloud-based-microservices-architecture-at-netflix-98836b2da45f

Nicola, S. (2020). FAST and QFD Techniques. *Análise de Valor (Value Analysis)*, 25.

Noback, M. (2018). *The Common Closure Principle BT - Principles of Package Design: Creating Reusable Software Components* (M. Noback (ed.); pp. 171–184). Apress. https://doi.org/10.1007/978-1-4842-4119-6_8

Noor, A., Jha, D. N., Mitra, K., Jayaraman, P. P., Souza, A., Ranjan, R., & Dustdar, S. (2019). A framework for monitoring microservice-oriented cloud applications in heterogeneous virtualization environments. *IEEE International Conference on Cloud Computing, CLOUD*, *2019-July*, 156–163. https://doi.org/10.1109/CLOUD.2019.00035

Pacheco, V. F. (2018). Microservice Patterns and Best Practices: Explore Patterns like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices. In Packt

Publishing (Ed.), *Microservice Patterns and Best Practices: Explore patterns like CQRS and event sourcing to create scalable, maintainable, and testable microservices*. https://dl.acm.org/doi/book/10.5555/3208677

Pallis, G., Trihinas, D., Tryfonos, A., & Dikaiakos, M. (2018). DevOps as a Service: Pushing the Boundaries of Microservice Adoption. *IEEE Internet Computing*, *22*(3), 65–71. https://doi.org/10.1109/MIC.2018.032501519

Parsons et al., R. (2014). *Inverse Conway Maneuver*. TW Technology Radar. https://www.thoughtworks.com/radar/techniques/inverse-conway-maneuver

Peter A. Koen, Greg M. Ajamian, Boyce, S., Allen Clamen, Fisher, E., Fountoulakis, S., Johnson, A., Puri, P., & Seibert, R. (2002). *The PDMA ToolBook for New Product Development* (P. Belliveau, P. B. Associates, A. Griffin, & S. Somermeyer (eds.)). John Wiley & Sons, Inc.

Petersen, K., Feldt, R., Mujtaba, S., & Mattsson, M. (2008, June 1). Systematic mapping studies in software engineering. *12th International Conference on Evaluation and Assessment in Software Engineering, EASE 2008*. https://doi.org/10.14236/ewic/ease2008.8

Pigneur, Y., Osterwalder, A., Bernada, G., & Smith, A. (2014). *Value Proposition Design: How to Create Products and Services Customers Want*. Wiley.

Ponce, F., Marquez, G., & Astudillo, H. (2019). Migrating from monolithic architecture to microservices: A Rapid Review. *Proceedings - International Conference of the Chilean Computer Science Society, SCCC*, *2019-Novem*. https://doi.org/10.1109/SCCC49216.2019.8966423

Rademacher, F., Sorgalla, J., Wizenty, P. N., Sachweh, S., & Zündorf, A. (2018). Microservice architecture and model-driven development: Yet singles, Soon Married (?). *ACM International Conference Proceeding Series*, *Part F1477*. https://doi.org/10.1145/3234152.3234193

Richardson, C. (2015). *Service Discovery in a Microservices Architecture*. NGINX Tech Blog. https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

Richardson, C. (2019a). *Command Query Responsibility Segregation (CQRS)*. Microservices.Io. https://microservices.io/patterns/data/cqrs.html

Richardson, C. (2019b). *Database per Service*. Microservices.Io. https://microservices.io/patterns/data/database-per-service.html

Richardson, C. (2019c). *Decompose by subdomain*. Microservices.Io. https://microservices.io/patterns/decomposition/decompose-by-subdomain.html

Richardson, C. (2019d). *Event sourcing*. Microservices.Io. https://microservices.io/patterns/data/event-sourcing.html

Richardson, C. (2019e). *Saga*. Microservices.Io. https://microservices.io/patterns/data/saga.html

Richardson, C. (2020). *Decomposition - Self-contained service*. Microservices.Io. https://microservices.io/patterns/decomposition/self-contained-service.html

Richardson, C. (2021, January 4). *Microservices Assessment Platform*. Microservices.Io. https://microservices.io/microservices/2021/01/04/microservies-assessment-platform-ga.html

Rotem-Gal-Oz, A. (2008). Fallacies of distributed computing explained. *Doctor Dobbs Journal*, 11. https://www.researchgate.net/publication/322500050_Fallacies_of_Distributed_Computing_Explained

Saaty, R. W. (1987). The analytic hierarchy process-what it is and how it is used. *Mathematical Modelling*, *9*(3–5), 161–176. https://doi.org/10.1016/0270-0255(87)90473-8

Samokhin, V. (2017, October 5). *DDD Strategic Patterns: How To Define Bounded Contexts*. Codeburst. https://codeburst.io/ddd-strategic-patterns-how-to-define-bounded-contexts-2dc70927976e

Santos, N., Salgado, C. E., Morais, F., Melo, M., Silva, S., Martins, R., Pereira, M., Rodrigues, H., Machado, R. J., Ferreira, N., & Pereira, M. (2019). A logical architecture design method for microservices architectures. *ACM International Conference Proceeding Series*, *2*, 145–151. https://doi.org/10.1145/3344948.3344991

Shahin, M., & Ali Babar, M. (2020). *On the Role of Software Architecture in DevOps Transformation: An Industrial Case Study* (Vol. 10). ACM. https://zookeeper.apache.org/

Solms, F. (2012). What is software architecture? *ACM International Conference Proceeding Series*, 363–373. https://doi.org/10.1145/2389836.2389879

Stranghöner, R. (n.d.). *SCS: Self-Contained Systems*. Self-Contained Systems: ASSEMBLING SOFTWARE FROM INDEPENDENT SYSTEMS. Retrieved March 4, 2021, from https://scs-architecture.org/

Stubailo, S. (2017, November 7). *The GraphQL stack: How everything fits together*. Apollo Blog. https://www.apollographql.com/blog/backend/the-graphql-stack-how-everything-fits-together-35f8bf34f841/

Swagger. (n.d.). *Documenting Your Existing APIs: API Documentation Made Easy with OpenAPI & Swagger*. Swagger.Io. Retrieved June 11, 2021, from https://swagger.io/resources/articles/documenting-apis-with-swagger/

Tang, W., Wang, L., & Xue, G. (2019). Design of High Availability Service Discovery for Microservices Architecture. *Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences - ICMSS 2019*. https://doi.org/10.1145/3312662.3312676

The Open Group. (2016). *What Is SOA?* The Open Group. https://web.archive.org/web/20160819141303/http://opengroup.org/soa/source-book/soa/soa.htm

Thönes, J. (2015). Microservices. *IEEE Software*, *32*(1). https://doi.org/10.1109/MS.2015.11

Tilkov, S. (2015). *Don't start with a monolith*. Martinfowler.Com. https://martinfowler.com/articles/dont-start-monolith.html

Trajanov, T. (2019). *Why Most Startups Don’t Need Microservices*. Adevait.Com. https://adevait.com/software/why-most-startups-dont-need-microservices-yet

Tune, N. (2017). The Art of Discovering Bounded Contexts. *Devoxx*.

Wade, J. (2018, September 27). *Greenfield vs. Brownfield Software Development*. Synoptek. https://synoptek.com/insights/it-blogs/greenfield-vs-brownfield-software-development/

Warwick Manufacturing Group. (2007). Quality Function Deployment. In *Product Excellence using Six Sigma*. Warwick Manufacturing Group. https://www.coursehero.com/file/12615334/QFD-Warwick/

Waseem, M., Liang, P., & Shahin, M. (2020). A Systematic Mapping Study on Microservices Architecture in DevOps. *Journal of Systems and Software*, *170*, 110798. https://doi.org/10.1016/j.jss.2020.110798

WaveMaker. (2019, November 12). *Micro services Architecture in Greenfield Projects*. Enterprise Application Development. https://www.wavemaker.com/considerations-when-adopting-microservices-architecture-in-greenfield-projects/

Wieringa, R. (2012). Technical action research. *RCIS*. http://rcis-conf.com/rcis2012/document/slides/RCIS12_TechnicalActionResearch.pdf

Wieringa, R. (2014). Technical Action Research. In R. Wieringa (Ed.), *Design Science Methodology for Information Systems and Software Engineering* (pp. 269–293). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-662-43839-8_19

Wittern, E., Cha, A., Davis, J. C., Baudart, G., & Mandel, L. (2019). An Empirical Study of GraphQL Schemas. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, *11895 LNCS*, 3–19. https://doi.org/10.1007/978-3-030-33702-5_1

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., & Wood, B. (2006). *Attribute-Driven Design (ADD), Version 2.0*. 55.

Yarygina, T. (2018). *Exploring Microservice Security*. http://bora.uib.no/handle/1956/18696

Zdun, U., Wittern, E., & Leitner, P. (2020). Emerging Trends, Challenges, and Experiences in DevOps and Microservice APIs. *IEEE Software*, *37*(1), 87–91. https://doi.org/10.1109/MS.2019.2947982

Zimmermann, O. (2018). *Microservice Domain Specific Language (MDSL) Tools*. Microservice DSL (MDSL). https://microservice-api-patterns.github.io/MDSL-Specification/tools

Attachments

Attachment 1 - Value Analysis

The main purpose of value analysis is to assess how the value of a product or idea can be explored at its maximum while keeping the cost at the minimum without decreasing the quality. To do this, it evaluates three main aspects of the product: its usefulness (utility value), the aesthetic and subjective value given by the customer (esteem value) and the price the market is willing to pay for the product (market value).

The value analysis reflected in this chapter has three phases: (1) Identify and analyze opportunities, (2) generate and select ideas to take advantage of the identified opportunities and finally (3) develop the concepts and technologies for the selected idea. These steps are mainly inspired by the NCD model (Peter A. Koen et al., 2002).

Opportunity Identification and Problem Analysis

Migrations from monolithic architectures to microservices are the most common solution that companies find for a system's sustainable growth, however this is a high-cost process. Microservices-first approach is an alternative that aims to reduce that cost of the migration by investing it in microservices development since the beginning of the project.

To evaluate the implementation of a microservice architecture following a new methodology, a prototype software solution is to be developed. Since the market of pharmacy management systems is lacking innovation, the prototype will try to add value to this industry by providing a more modern and versatile solution. This system will be web-based and will focus on maintaining the market concurrence best features while creating and improving the missing or not satisfactory ones. In the next sections, the value proposition for the system will be defined and a market comparison will be performed to identify the most important requirements in the business.

Value Proposition - VP

To analyze the real value of the identified opportunity, a value proposition canvas based on the Osterwalder Value Proposition Design (Pigneur et al., 2014) was developed.

Value Proposition Design is a very powerful tool that aims to understand the patterns of value creation, leverage the experience and skills of the team and minimize the risk of failure. It may be used in two different lifecycle phases of a business: invent a new business by building its brand-new value proposition or renew an existing value proposition and business model. In this case, the value proposition for a new concept was applied.

The creation of the Value Proposition Canvas followed two steps:

Designing the Customer Profile

Jobs: The customer profile needs to identify the "Jobs" which can be translated to what the customers are trying to achieve, or what their need is. There are four types of job: functional, social, personal/emotional, and supporting (Pigneur et al., 2014).

Pains: The pains reflect the problems that may occur before, during, and after trying to perform a job. These can be undesired outcomes (the solution doesn't work, works poorly), obstacles (when the customer is prevented from even getting started with a job), and risks (negative consequences of performing the job) (Pigneur et al., 2014).

Gains: The gains consist of the specific achievements the customers seek in their jobs. There are four types of gain: required, expected, desired, and unexpected (Pigneur et al., 2014).

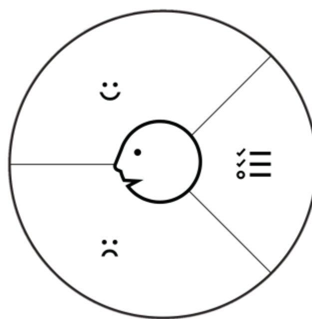


Figure 42 – Value Proposition Canvas - Customer Profile

Source: (Pigneur et al., 2014)

Constructing the Value Map

Products and Services: this refers to the list of all products and services that the business will have to offer to their customers and that the value proposition will be built on (Pigneur et al., 2014).

Pain Relievers: they exist to eliminate or reduce the identified customer pains. The pain relievers do not necessarily need to tackle every pain but should focus on those that matter most to the customer, this is called pain reliever relevance (can be a nice to have or an essential). Usual pain relievers tend to reduce money, effort, frustrations, and annoyances in the customer (Pigneur et al., 2014).

Gain Creators: should be the drivers to satisfy the identified customer needs in the gains section. They should describe how the products and services can help to produce outcomes and benefits to the customer. Like the pain relievers, gain creators do not need to address every identified gain, there is also the relevance factor which will translate into satisfying the customer expectations and desires or on the other side surprising him with unexpected utility, social gains, positive emotions, and cost savings (Pigneur et al., 2014).

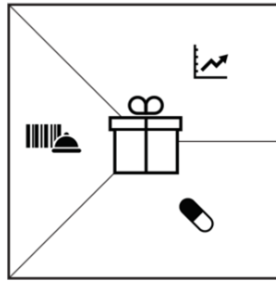


Figure 43 - Value Proposition Canvas - Value Map

Source: (Pigneur et al., 2014)

Pharmacy Management Software Value Proposition

Now that the main concepts of the value proposition process are described, the application of each one in the specific case of pharmacy management software will be reviewed. To identify the following points, an interview with the Technical Director of a local pharmacy was conducted and is in Attachment 2.

Customer Jobs

- Sell product / Fill drug prescriptions
- Create product orders to suppliers
- Receive order (order reception process)
- Store received products
- Analyze sales (sell-out focused management)
- Control temperature and humidity (generate a report)
- Process billing and payments
- Validate prescriptions (for legal and reimbursement purposes)
- Measure customer health parameters (blood pressure, glucose)
- Administer vaccines and other injectables
- Control product expiry dates
- Return products (expired, damaged, etc.)
- Intermediate hospital drugs delivery

Pains

- Not being able to differentiate the expiry dates inside the stock of a product automatically (only one date available for the product as a whole) leads to the need for manual verification or the sale of a longer expiry date item instead of the shorter one
- Manual processing of the order reception takes a lot of time
- New stocks storage in shelves takes a big amount of time
- Finding errors during the monthly prescription validation leads to money loss

- Product returns request needs to be performed with anticipation of one month which requires a frequent verification of expiring stocks
- Order creation can take a lot of time because supplier prices need to be checked for each product

Gains

- Provide the customer with valuable advice and drug information will give a better image to the pharmacy and consequentially retain customers (e.g. side effects of drug mixing);
- Grow the customer base will increase significantly the rentability of the pharmacy
- Having an efficient supply chain leads to a reduction of costs and better purchase margins
- A faster and easier order receiving process gives pharmacy workers more time to spend on more productive tasks
- Processes automatization may lead to staff cost reduction
- Have the patients' medical registries can allow the pharmacy to predict customer needs patterns and improve stock's management

Products and Services

- Online Store
- Point of sale (POS) module
- Orders management module
- Payment registration module
- Employee schedule manager
- Inventory management service
- Integration with official drugs database
- Integration with robots and automation devices
- Customer fidelity service

Pain Relievers

- Unlike many other pharmacy management software solutions, our inventory management service allows controlling the amount of stock for each expiry date. This will allow the seller to know the correct item to pick from the storage
- Order management module provides functionalities to automatically create order templates based on configurations (e.g. use the cheapest supplier for every product)
- Robot integration allows the pharmacy to use an automatic dispenser that will reduce the storage and order reception processes' time and human error rate

Gain Creators

- The POS module is integrated with several databases that will provide the pharmacist with all the important information about the products he is selling
- Implementing an online web store will allow the pharmacy to reach new markets and increase sales volume without adding a big, fixed cost in the sale process
- Automatic employee schedule will reduce the time spent with that recurring task and consequentially the associated costs
- The customer fidelity service allows registering the customer purchase history that will allow to give a more personalized experience and help to improve stock management
- Inventory management service can help to have a more efficient stocks management with more effective orders from the right suppliers for the business

Quality Function Deployment - QFD

QFD is a technique that involves three main concepts: Quality - achieving customer quality requirements, Function - understanding what needs to get more focus to achieve quality, and Deployment - identify the entities who will be responsible for the function tasks and when these will be performed (Nicola, 2020).

One of the most common problems that software manufacturing companies face for decades is associated with the incorrect, incomplete and inconsistent specification of user requirements. This usually leads to increased costs in the development and testing of these companies' products (Haag et al., 1996). To solve this problem, software manufacturers needed to find techniques that could facilitate the correct specification of user requirements and that is how SQFD (Software Quality Function Deployment) was born.

House of Quality - HOQ

House of Quality contributes to the customer quality requirements identification, as well as understanding how these can be achieved. It gathers the customers' desires and maps them to engineering characteristics. It requires cooperation and communication across marketing and technical functional areas (Nicola, 2020). The house of quality is separated in the sections described in Figure 44.

The first step is to identify the customer requirements and, since a value proposition was previously implemented based on an interview with a pharmacy's technical director, this task is a simple review of the customer profile and value map. After this, the importance of each requirement needs to be defined. For this, a second interview was arranged to discuss and measure the degree of importance of each requirement. Both requirements and each one's importance degree are visible in Figure 45.

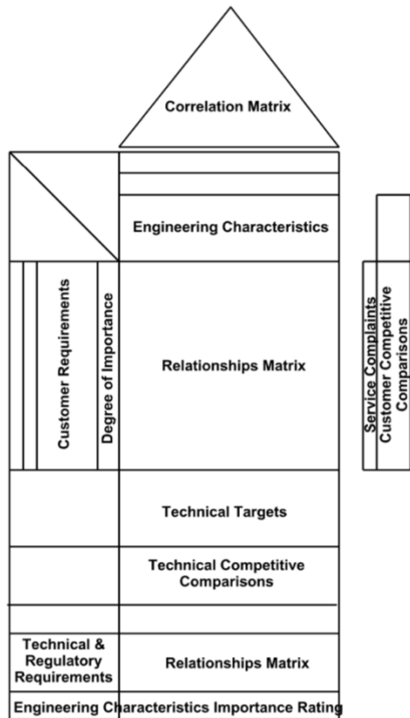


Figure 44 - QFD Chart

Source: (Warwick Manufacturing Group, 2007)

Row #	Weight Chart	Relative Weight	Customer Importance	Maximum Relationship	Customer Requirements (Explicit and Implicit) / Functional Requirements
1		9%	3	9	Low price
2		13%	4	9	Access to relevant information about the products
3		16%	5	9	High availability/resillience
4		16%	5	3	Simple and intuitive user inteface
5		13%	4	9	Stock's expiry dates control
6		13%	4	9	Grow customer base
7		13%	4	3	Employee working hours optimisation
8		9%	3	9	Sales analysis (Sell-Out)

Figure 45 - QFD Customer Requirements

Then, the engineering characteristics of the pharmacy management systems are listed. These work as enablers for the previously identified requirements and a technical target is set for each one. The next step is to develop a relationships matrix of customer requirements and engineering characteristics, which defines the level of relationship between these two parameters on a scale of four qualitative values: None, Weak, Moderate, and Strong (See Figure 46).

Customer Requirements (Explicit and Implicit) \ Technical Requirements	System is available whenever na employee needs it	Ability to sell products online	Manage inventory	Use Open-Source Tech Stack	Access to national health system's database	Time taken to use major functionalities
Low price		○		●		
Access to relevant information about the products					●	
High availability/resillience	●	▽		○		
Simple and intuitive user inteface		○				●
Stock's expiry dates control			●			
Grow customer base		●				
Employee working hours optimisation	○		○		○	●
Sales analysis (Sell-Out)			●			
Target	Availability between 5 a.m. and 2 a.m.	Grow cosmetics and no-prescription products sales in 50%	Reduce costs of expired products by 25%	Reduce system overall costs by 25%	Employees find relevant information about the product they are selling 85% of the time	Employee takes less than 5 minutes to complete a sale (time of interaction with the system only)

Figure 46 - QFD Technical Requirements, Targets, and Relationship Matrix

The two missing steps of the HOQ are the correlation matrix and the technical comparison of the competitors. These two points do not affect the importance ratings but help to understand which are the engineering characteristics and customer requirements that matter most (Lucidchart, 2019).

The correlation matrix is visible in Figure 47 and refers to the correlations among the technical attributes, which is a positive, negative, or neutral value on how a technical attribute helps, hinders or doesn't have any influence on another.

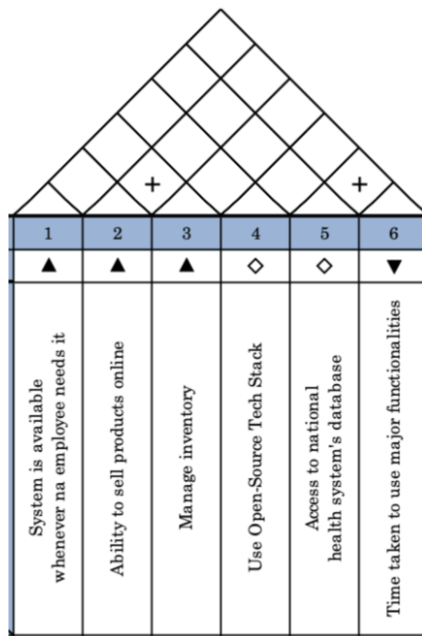


Figure 47 – QFD Technical Requirements Correlation Matrix

The final part of the house of quality is the competitive assessment, this consists of a research that shows how each of the competitors ranks in every customer need and technical target (Figure 48 and Figure 49). It allows determining what needs more improvements and consequentially what are the best areas to gain competitive advantage (Lucidchart, 2019).

The complete House of Quality diagram is attached to this document in Attachment 3.

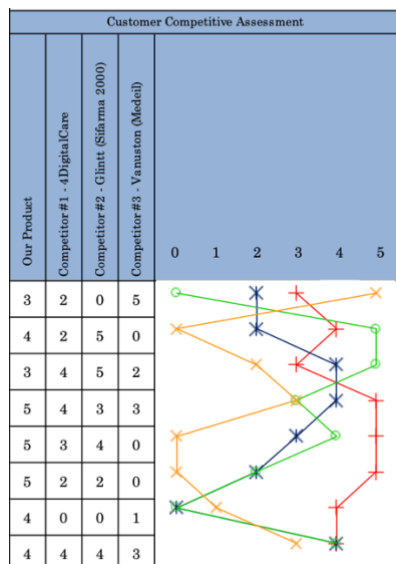


Figure 48 - QFD Customer Competitive Assessment

Figure 49 - QFD Technical Competitive Assessment

Conclusions

After analyzing the results of the QFD it is possible to verify that exists a positive correlation between the ability to sell products in an online marketplace and inventory management. This happens because to have an online store that can scale, an automated inventory management is required.

Based on the calculated "Technical Importance Rating" it is visible that the most relevant technical requirements are the time taken to use major functionalities, inventory management and the ability to sell products online. While the first is explained by the terrible assessment of the competitors in the "Employee working hours optimization" requirement, the other values are mostly explained by the importance given to the control over stock expiry dates, sell-out functionalities and optimization of employee productivity.

Since employee work optimization is not a very specific requirement, the customer assessment could have been too depreciated. Therefore, the focus of the work from this point on will be on the other two most rated requirements, improving inventory management and introducing the business into the online market.

Idea Genesis and Selection

Now that the opportunity was identified and its value was verified, this section organizes the ideas to take advantage of that opportunity. This involves two phases: the generation of ideas and the selection of the fittest. While the idea generation is based on a brainstorming process, the selection of the most reliable solution uses the multicriteria method AHP (Analytic Hierarchy Process).

Idea Generation

Following the NCD model, a brainstorming session was performed to gather a set of ideas that may help to solve the problem identified before.

1. **Microservices from scratch:** this approach consists of creating an application with a microservice mindset since project's day one. The whole engineering process is focused on designing the application based on microservice architecture and all the needs around it, like infrastructure and development processes.
2. **Start with a monolithic application and migrate it to microservices according to its growth process:** when applying this method, the application starts to be developed as a big block of code. When a problem or need starts to emerge (scalability, too many responsibilities for a single application), the big application begins to be broken into smaller pieces, each one with fewer responsibilities.
3. **Stick with a monolithic application during the whole application lifecycle:** similarly, to the previous approach, the application is built in a single piece of code. This application will inherit all the responsibilities of the system in a single place and every developer will be

working over the same code. The difference is that in this case there is no place for breaking the big monolithic application into smaller blocks until it is no longer maintained/used.

4. **Create a hybrid solution with a main monolithic service and other microservices around it:** this method is a mixture of 1 and 2 since the application is built with a microservices mindset from day one but is more conservative about the granularity of the decomposition. The analysis of each microservice responsibility is carefully performed to understand which ones can be aggregated and minimize the nano-services pitfall.

Analytic Hierarchy Process - AHP

The idea generation process resulted in four alternatives to implement a prototype for a pharmacy management system. Now there is the need to find the approach that has more potential to be successful and to perform this evaluation it was used the Analytic Hierarchy Process.

Defined by (Saaty, 1987): "The Analytic Hierarchy Process (AHP) is a general theory of measurement. It is used to derive ratio scales from both discrete and continuous paired comparisons.". Since one of the most common applications of this theory is multicriteria decision-making, this seems like a good tool to help to decide the best alternative for this project strategy.

As a starting point for the application of AHP in the decision process, there is the need to create a hierarchic structure to represent the objective, the criteria, and the alternatives and pairwise comparisons to establish the relations within that structure (Saaty, 1987). In Figure 50, there is the hierarchy for the Pharmacy Management System development approach decision making. Four important software architecture criteria were chosen and each of the previously identified alternatives will need to suffer a pairwise comparison.

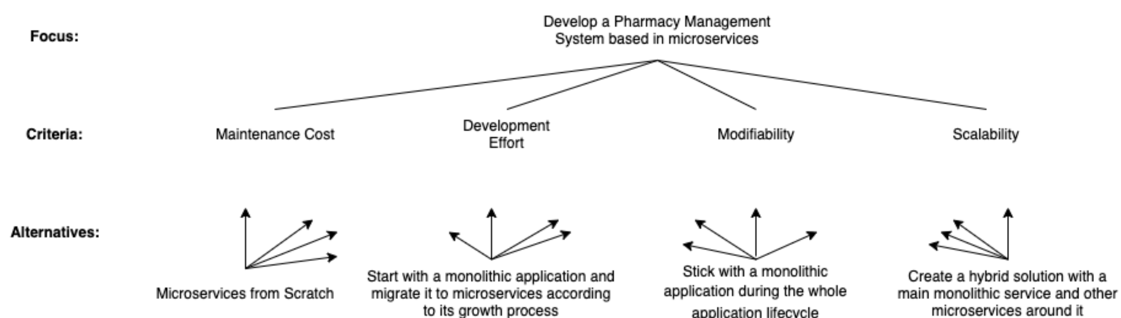


Figure 50 - AHP Hierarchy Diagram

Pairwise Comparison

Now it is time for the second phase of AHP, the pairwise comparison of the chosen criteria and alternatives. A pairwise comparison generally is the process of comparing entities in pairs to evaluate which one is preferred or has a greater amount of some quantitative property. The scale to be used in the pairwise comparison is the fundamental scale proposed by (Saaty, 1987) and is detailed in Figure 51.

Intensity of importance on an absolute scale	Definition	Explanation
1	Equal importance	Two activities contribute equally to the objective
3	Moderate importance of one over another	Experience and judgment strongly favor one activity over another
5	Essential or strong importance	Experience and judgment strongly favor one activity over another
7	Very strong importance	An activity is strongly favored and its dominance demonstrated in practice
9	Extreme importance	The evidence favoring one activity over another is of the highest possible order of affirmation
2,4,6,8	Intermediate values between the two adjacent judgments	When compromise is needed

Figure 51 - AHP Fundamental Scale

Source: (Saaty, 1987)

The result of the pairwise comparison is listed in Table 12.

Table 12 - AHP Criteria Pairwise Comparison

Criteria	Maintenance Cost	Development Effort	Modifiability	Scalability
Maintenance Cost	1	2	3	4
Development Effort	0,33	1	0,25	0,50
Modifiability	0,33	4	1	3
Scalability	0,25	2	0,33	1
Total	**1,91**	**9**	**4,58**	**8,50**

Criteria Comparison Normalization

Now that the comparison for each pair of criteria was made in Table 12, that matrix needs to be normalized to obtain the result of each criteria priority. The resultant matrix is represented in Table 13.

To calculate the normalized matrix each entry of the matrix is divided by the sum of the values in the correspondent column.

The formula for this operation is:

$$A' = \left[a'_{ij}\right] = \frac{a_{ij}}{\sum_{k=1}^{n} a_{ik}} \ para\, 1 \leq i \leq n \ e\, 1 \leq j \leq n$$

Table 13 - AHP Criteria Comparison Normalized Matrix

Criteria	Maintenance Cost	Development Effort	Modifiability	Scalability	***Relative Priority***
Maintenance Cost	0,524	0,222	0,655	0,471	*0,468*
Development Effort	0,173	0,111	0,055	0,059	*0,099*
Modifiability	0,173	0,444	0,218	0,353	*0,297*
Scalability	0,131	0,222	0,072	0,118	*0,136*

The relative priority of each criterion is the average of the values in the corresponding row. These values were converted into a percentage and ordered in Table 14.

Table 14 - AHP Criteria Relative Priorities

Criteria	**Result**	***Priority***
Maintenance Cost	46,8%	*1st*
Modifiability	29,7%	*2nd*
Scalability	13,6%	*3rd*
Development Effort	9,9%	*4th*

To evaluate the consistency of the calculated relative priorities the Consistency Ratio (CR) is used. It is calculated by multiplying the initial criteria pairwise comparison's matrix by the priorities' vector, as follows:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 0.33 & 1 & 0.25 & 0.5 \\ 0.33 & 4 & 1 & 3 \\ 0.25 & 2 & 0.33 & 1 \end{bmatrix} \times \begin{bmatrix} 0.468 \\ 0.099 \\ 0.297 \\ 0.136 \end{bmatrix} = \begin{bmatrix} 2.101 \\ 0.396 \\ 1.255 \\ 0.549 \end{bmatrix}$$

The average consistency ratio is given by the average of the results for the consistency ratio calculated previously.

$$\boldsymbol{\lambda}\mathbf{max.} = \mathbf{Avg}\left(\frac{\mathbf{2{,}101}}{\mathbf{0{,}468}}, \frac{\mathbf{0{,}396}}{\mathbf{0{,}099}}, \frac{\mathbf{1{,}255}}{\mathbf{0{,}297}}, \frac{\mathbf{0{,}549}}{\mathbf{0{,}122}}\right) = \mathbf{4,187}$$

Now that the average of the consistency ratio is given by the following formula:

$$\boldsymbol{CI} = \frac{\boldsymbol{\lambda max - n}}{\boldsymbol{n-1}} = \frac{\mathbf{4{,}187-4}}{\mathbf{4-1}} = \mathbf{0{,}062}$$

The consistency ratio is calculated by dividing the previously calculated consistency index by a table value that, since this comparison involves four different alternatives, is 0,9.

$$\mathbf{CR} = \frac{\boldsymbol{CI}}{\mathbf{0{,}9}} = \frac{\mathbf{0{,}062}}{\mathbf{0{,}9}} = \mathbf{0,069} < \mathbf{0,1}$$

Since the value of the Consistency Ratio (CR) is smaller than 0,1, it is possible to conclude that the relative priorities are consistent.

It is possible to conclude that the highest priority criteria are the Maintenance Costs and the Modifiability of the software solution.

Alternatives Comparison by Criteria

The next step in the Analytic Hierarchy Process is to perform a pairwise comparison of the four identified alternatives for each of the defined criteria. This allows identifying the best alternative for each criterion, and consequentially find its overall priority crossing that result, with the previously determined importance of each criterion.

The priority results will be determined by following a similar process to the criteria comparison. It begins with the pairwise comparison matrix followed by the calculation of the priority vector by normalizing that matrix.

In the end, the priority of each alternative will be established by crossing the resultant priority vector of each criterion with its relative priority.

Table 15 - AHP Maintenance Cost Alternatives Comparison

Maintenance Cost	Alt. 1	Alt. 2	Alt. 3	Alt. 4
Alternative 1	1	6	9	2
Alternative 2	0,17	1	3	0,25
Alternative 3	0,11	0,33	1	0,14
Alternative 4	0,5	4	7	1
Total	**1,78**	**11,33**	**20,00**	**3,39**

Table 16 - AHP Maintenance Cost Normalized Matrix

Maintenance Cost	Alt. 1	Alt. 2	Alt. 3	Alt. 4	*Priority Vector*
Alternative 1	0,562	0,530	0,450	0,590	*0,533*
Alternative 2	0,096	0,088	0,150	0,074	*0,102*
Alternative 3	0,062	0,029	0,050	0,041	*0,046*
Alternative 4	0,281	0,353	0,350	0,295	*0,320*

As it is shown in Table 16, for the maintenance cost criterion, the highest priority alternative is to start with microservices from scratch (Alternative 1).

Table 17 - AHP Development Effort Alternatives Comparison

Development Effort	Alt. 1	Alt. 2	Alt. 3	Alt. 4
Alternative 1	1	2	0,17	0,33
Alternative 2	0,5	1	0,33	3
Alternative 3	6	3	1	4
Alternative 4	3	0,33	0,25	1
Total	**10,50**	**6,33**	**1,75**	**8,33**

Table 18 - AHP Development Effort Normalized Matrix

Development Effort	Alt. 1	Alt. 2	Alt. 3	Alt. 4	*Priority Vector*
Alternative 1	0,095	0,316	0,097	0,040	*0,137*
Alternative 2	0,048	0,158	0,189	0,360	*0,189*
Alternative 3	0,571	0,474	0,571	0,480	*0,524*
Alternative 4	0,286	0,052	0,143	0,120	*0,150*

Regarding the development effort criterion, the best alternative is by a large margin Alternative 3 (see Table 18), since building a Monolithic Application is a much simpler process than any of its alternatives. However, the simplicity of the development process will compromise all the other criteria.

Table 19 - AHP Modifiability Alternatives Comparison

Modifiability	Alt. 1	Alt. 2	Alt. 3	Alt. 4
Alternative 1	1	3	8	2
Alternative 2	0,33	1	5	0,33
Alternative 3	0,125	0,2	1	0,33
Alternative 4	0,5	3	3	1
Total	**1,96**	**7,20**	**17**	**3,66**

Table 20 - AHP Modifiability Normalized Matrix

Modifiability	Alt. 1	Alt. 2	Alt. 3	Alt. 4	*Priority Vector*
Alternative 1	0,512	0,417	0,471	0,546	*0,486*
Alternative 2	0,169	0,139	0,294	0,090	*0,173*
Alternative 3	0,064	0,028	0,059	0,090	*0,060*
Alternative 4	0,256	0,417	0,176	0,273	*0,281*

From the comparison of the alternatives in the scope of modifiability, Alternative 1 is the winner again. As seen in Table 20, Microservices from Start alternative has the highest priority 48,6%, followed by Alternative 4 with 28,1%, which consists of a hybrid solution of main monolithic service and smaller services working with it

Table 21 - AHP Scalability Alternatives Comparison

Scalability	Alt. 1	Alt. 2	Alt. 3	Alt. 4
Alternative 1	1	4	8	2
Alternative 2	0,25	1	3	4
Alternative 3	0,125	0,33	1	0,17
Alternative 4	0,5	0,25	6	1
Total	**1,88**	**5,58**	**18**	**7,17**

Table 22 - AHP Scalability Normalized Matrix

Scalability	Alt. 1	Alt. 2	Alt. 3	Alt. 4	***Priority Vector***
Alternative 1	0,533	0,717	0,444	0,279	*0,493*
Alternative 2	0,133	0,179	0,167	0,558	*0,259*
Alternative 3	0,067	0,059	0,056	0,024	*0,051*
Alternative 4	0,267	0,045	0,333	0,139	*0,196*

In Table 22, it is possible to see that, for the scalability concern, the alternatives with the best result were Alternative 1 with 49,3% followed by Alternative 2 with 25,9%.

Table 23 - Alternatives Priority by Criteria Matrix

	Maintenance Cost	Development Effort	Modifiability	Scalability
Relative Priority	0,468	0,099	0,297	0,136
Alternative 1	0,533	0,137	0,486	0,493
Alternative 2	0,102	0,189	0,173	0,259
Alternative 3	0,046	0,524	0,060	0,051
Alternative 4	0,320	0,150	0,281	0,196

With these values, it is possible to calculate the real priority of each alternative by multiplying a matrix where each column is a criteria's alternative priority vector listed in Table 23, by the criteria's relative priorities vector from Table 14.

$$\begin{bmatrix} 0{,}533 & 0{,}137 & 0{,}486 & 0{,}493 \\ 0{,}102 & 0{,}189 & 0{,}173 & 0{,}259 \\ 0{,}046 & 0{,}524 & 0{,}060 & 0{,}051 \\ 0{,}320 & 0{,}150 & 0{,}281 & 0{,}196 \end{bmatrix} \times \begin{bmatrix} 0.468 \\ 0.099 \\ 0.297 \\ 0.136 \end{bmatrix} = \begin{bmatrix} 0{,}474 \\ 0{,}153 \\ 0{,}098 \\ 0{,}275 \end{bmatrix}$$

Table 24 - Overall Alternative Priority Results

Alternative	**Overall Result**	***Priority***
1 - Microservices from Scratch	47,4%	*1st*
2 - Create a hybrid solution with a main monolithic service and other microservices around it	27,5%	*2nd*
4 - Start with a monolithic application and migrate it to microservices according to its growth process	15,3%	*3rd*
3 - Stick with a monolithic application during the whole application lifecycle	9,8%	*4th*

Results

The final result of the AHP (see Table 24) leads to the selection of Alternative 1, followed by Alternative 2, and a close ending for Alternatives 4 and 3. It is possible to conclude that sticking with a monolithic application during the whole application lifecycle is the worst alternative taking into account some of the most important criteria of software solutions in modern environments.

The only criterion in which alternative 4 has the best priority is the development effort. In the short term, the development process is much simpler when working on a monolithic application since it has less complexity than the other alternatives. All the challenges delineated in section 3.2 are therefore avoided. Except for the development effort, Microservices from Scratch is the top priority alternative in all criteria. That approach is followed by the hybrid alternative of microservices which is the second-best approach in the remaining criteria except in scalability where Alternative 2 had a smaller advantage (25,9% against 19,6%).

Attachment 2 – Pharmacy Technical Director Interview

Question 1

EN - Which are the main tasks in the daily routine of a pharmacist?

PT – Quais são as principais tarefas desenvolvidas por um farmacêutico na sua rotina diária?

Answer: Atendimento, receção de encomendas, arrumação de medicamentos, realização de encomendas, análise de vendas (por princípio ativo/ por laboratório) de forma a aumentar a rentabilidade (Sell Outs), registo de temperatura e humidade (já automatizado), faturação mensal (automática, mas é necessária a validação das receitas físicas), medições de glicémia, administração de vacinas, devolução de medicamentos fora de validade.

Encomendas de medicamentos hospitalares (em tempo de pandemia)

Question 2

EN – From the previous answer, which are the tasks that are most time consuming and why?

PT – Das tarefas acima mencionadas, quais as que demoram mais tempo e porquê?

Answer: A receção e arrumação de medicamentos, assim como a realização de encomendas. A primeira porque é um processo manual de entrada no sistema informático e de arrumação nas prateleiras/gavetas, o segundo porque o Sistema não permite de forma eficiente definir os critérios de seleção de fornecedor (ex.: selecionar automaticamente o fornecedor com preço mais baixo para cada item da encomenda).

Question 3

EN – Which are the tasks that would take a lot more effort/time to perform manually?

PT – Quais as tarefas que demorariam mais / teriam um maior esforço em serem realizada de forma manual (não informatizada)?

Answer: Tal como foi dito anteriormente, a receção e arrumação de medicamentos atualmente é feita de forma manual, seria bom o investimento no future num robot dispensador de medicamentos, mas é muito dispendioso. Todo o processo de venda e faturação que já é atualmente informatizado, seria bastante mais demorado caso tivesse de ser feito de "forma manual".

Question 4

EN – What data do you need to store?

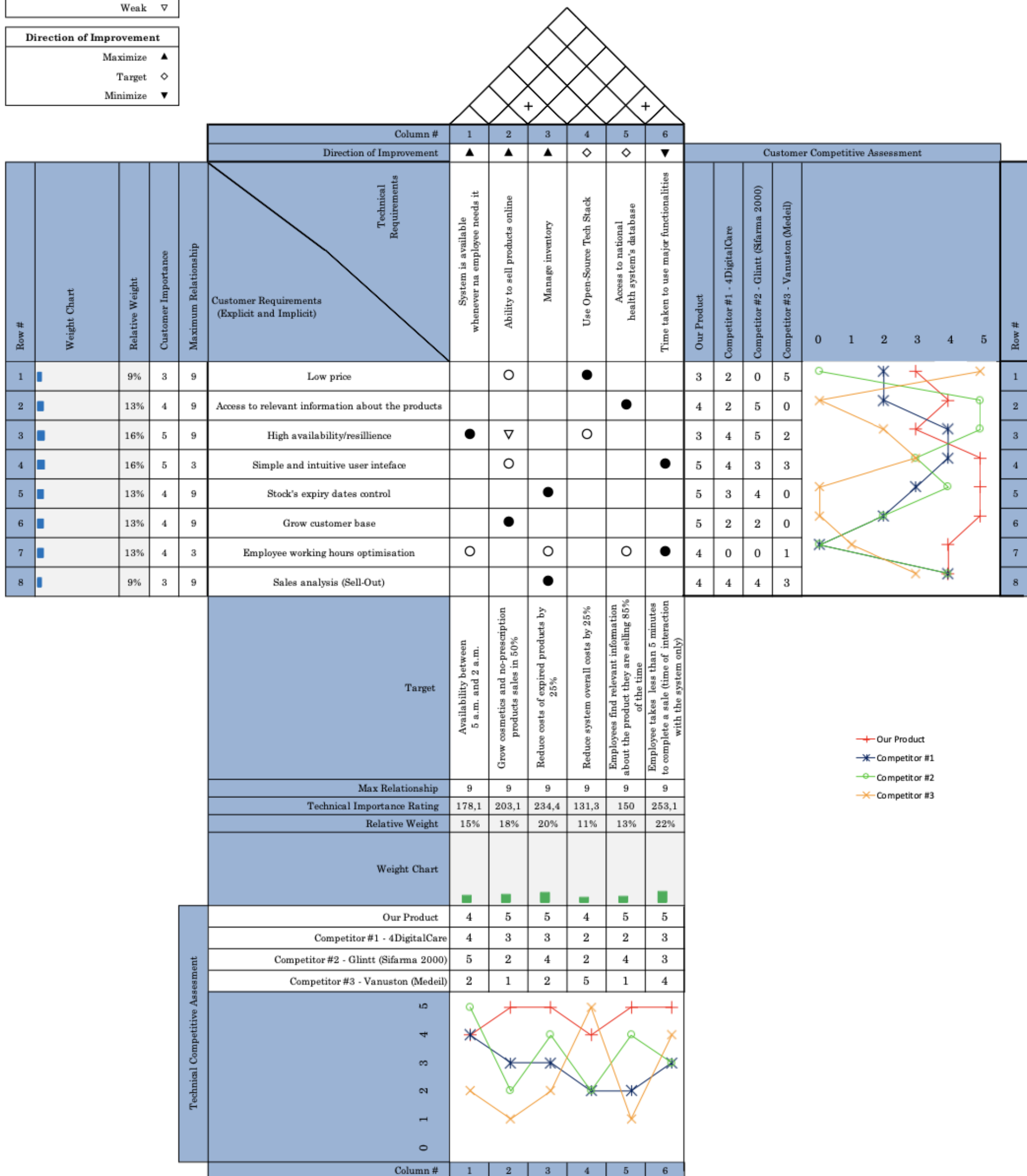
PT – Que informação necessitam de armazenar?

Answer: Apenas é necessário registar os dados pessoais básicos do cliente (nome, número de contribuinte, etc.). Com o aumento do número de administrações de vacinas, também se tornou um processo mais complexo o agendamento das mesmas e podia ser informação relevante a armazenar no Sistema.

Outras Notas Relevantes sobre o Sistema de informação atual:

- Receitas ao serem registadas devem ser enviadas para o organismo correto ARS, seguradoras, etc.
- Listagem de medicamentos que vão expirar no mês seguinte
- No processo de venda avisar quando os produtos têm stock em fim de validade.
- No processo de venda, ao identificar o medicamento seria importante mostrar toda a informação sobre contraindicações, etc. de forma simples e rápida.

Attachment 3 – QFD House of Quality



Attachment 4 – Guide Application Domain Model

Figure 52 - Pharmacity Domain Model

Attachment 5 – Pharmacity GraphQL Stitched Schema

```
schema {
    query: LocalQuery
    mutation: LocalMutation
}

interface Order @source(name: "Order", schema: "orders") {
    id: String
    customerId: String
    addressId: String
    value: String
    date: String
    status: [OrderStatus]
    paymentId: String
    deliveryId: String
}

type Address @source(name: "Address", schema: "customers") {
    id: String
    deliveryAddress: String
    zipCode: String
    city: String
    country: String
    phone: String
}

type Cart @source(name: "Cart", schema: "carts") {
    cartId: String
    productIds: String
    items: [Product] @delegate(schema: "products", path: "productsById(id
s: $fields:productIds)")
}

type Customer @source(name: "Customer", schema: "customers") {
    id: String
    name: String
    email: String
    discountCard: String
    cartId: String
    wishlistId: String
    addresses: Address @delegate(schema: "customers", path: "addresses(id
s: $fields:addresses)")
    orders: [Order] @delegate(schema: "orders", path: "orders(ids: $field
s:orders)")
```

```
    cart: Cart @delegate(schema: "carts", path: "cart(cartId: $fields:car
tId)")
}

type LocalMutation {
    addItemToCart(cartId: String productId: String): Cart @delegate(schem
a: "carts")
    checkoutCart(cartId: String addressId: String prescriptionId: String
productIds: String customerId: String paymentInfoId: String amount: Strin
g): Payment @delegate(schema: "checkout")
}

type LocalQuery {
    products: [Product] @delegate(schema: "products")
    productsById(ids: String): [Product] @delegate(schema: "products")
    customer(id: String): Customer @delegate(schema: "customers")
    payment(id: String): Payment @delegate(schema: "checkout")
    customers: [Customer] @delegate(schema: "customers")
    cart(cartId: String): Cart @delegate(schema: "carts")
    addresses(ids: String): Address @delegate(schema: "customers")
    orders(ids: String): [Order] @delegate(schema: "orders")
    orderById(id: String): [Order] @delegate(schema: "orders")
}

type Mutation {
    checkoutCart(cartId: String addressId: String prescriptionId: String
productIds: String customerId: String paymentInfoId: String amount: Strin
g): Payment @delegate(schema: "checkout")
    addItemToCart(cartId: String productId: String): Cart @delegate(schem
a: "carts")
}

type OrderItem @source(name: "OrderItem", schema: "orders") {
    productId: String
    quantity: String
}

type OrderStatus @source(name: "OrderStatus", schema: "orders") {
    date: String
    value: String
}

type Payment @source(name: "Payment", schema: "checkout") {
    id: String
    customerId: String
    paymentInfoId: String
    amount: String
```

```
    date: String
}

type Prescription @source(name: "Prescription", schema: "prescriptions")
{
    number: String
    accessCode: String
    optionCode: String
    attachment: String
    comments: String
    orderId: String
}

type PrescriptionOrder implements Order @source(name: "PrescriptionOrder"
, schema: "orders") {
    prescriptionId: String
    id: String
    customerId: String
    addressId: String
    value: String
    date: String
    status: [OrderStatus]
    paymentId: String
    deliveryId: String
}

type Product @source(name: "Product", schema: "products") {
    id: String
    name: String
    description: String
    price: String
    quantity: Int!
    color: String
    size: String
}

type ProductsOrder implements Order @source(name: "ProductsOrder", schema
: "orders") {
    items: [OrderItem]
    id: String
    customerId: String
    addressId: String
    value: String
    date: String
    status: [OrderStatus]
    paymentId: String
```

```
    deliveryId: String
}

type Query {
    products: [Product] @delegate(schema: "products")
    productsById(ids: String): [Product] @delegate(schema: "products")
    customers: [Customer] @delegate(schema: "customers")
    customer(id: String): Customer @delegate(schema: "customers")
    addresses(ids: String): [Address] @delegate(schema: "customers")
    address(id: String): Address @delegate(schema: "customers")
    orders(ids: String): [Order] @delegate(schema: "orders")
    order(orderId: String): Order @delegate(schema: "orders")
    prescription(id: String): Prescription @delegate(schema: "prescriptio
ns")
    payment(id: String): Payment @delegate(schema: "checkout")
    carts: [Cart] @delegate(schema: "carts")
    cart(cartId: String): Cart @delegate(schema: "carts")
}
```

Attachment 6 – Application of Quantitative Evaluation Framework

q	D	Qi	Dimension	Q_j	W_{ij} (Factor Weight j in Dim i) [0,1]	Factor	rw_{jk} (requirement weight k in Factor j) {2, 4, 6, 8, 10}	Requirement	wf_k % requirement fulfillment k) [0,100]
72%	0,92	82,6	General architecture and software delivery outcomes	60,9	0,44	Key software delivery outcomes	6	GAK01 - Release/Deployment frequency	25
							8	GAK02 - Frequency that a deployment leads to outage	100
							10	GAK03 - Time to recover from an outage	100
							8	GAK04 - Tracking and review of outage metrics	0
				100	0,44	Inter-service communication	6	GASC01 - Documented API Standards and style guides	100
							10	GASC02 - External client access to service mechanisms	100
							8	GASC03 - Service discovery mechanism is implemented	100
							8	GASC04 - Service registration patterns are applied	100
				100	0,11	General Architecture	10	GAGA01 - Microservice architecture must consist of two or more independently deployable/executable components	100
		75	Deployment and reliability	100	0,50	Deployment Process	6	DRP01 - Deployment options definition	100
							10	DRP02 - Horizontal scaling support	100
				50	0,50	Deployment Reliability	6	DRR01 - Failure detection and short-circuiting mechanisms	100
							6	DRR02 - Fault injection in production prevention	0
		37,6	Monitoring and observability	57,1	0,33	Audit and tracing	6	MOAT01 - Usage of audit logging	0
							8	MOAT02 - Distributed tracing	100
				55,6	0,33	Application metrics	8	MOAM01 - Collection of application metrics	0
							10	MOAM02 - Health check mechanism	100
				0	0,33	Tracking Deployments	6	MOTD01 - Track configuration changes	0
							6	MOTD02 - Log deployments	100
		80	Infrastructure and configurations	100	0,40	Configurations	6	ICC01 - Externalized configuration strategy	100
							8	ICC02 - Sensitive properties safe storage	100
				66,7	0,60	Supporting Infrastructure	8	ICS01 - Development/testing environment provisioning time	100
							8	ICS02 - Similarity between testing and production environments	0
							8	ICS03 - Deployment pipeline infrastructure support	100
		50	Libraries and frameworks	50	0,50	Microservices chassis	8	LFC01 - Definition of microservice chassis frameworks	50
				50	0,50	Testing/development tools	6	LFT01 - Use testing and development tools/frameworks	50
		71,4	Documentation, organization and processes	100	0,29	Documentation	8	DOPD01 - Accessible architecture documentation	100
							8	DOPD02 - Technical decisions are logged and documented	100
				100	0,29	Organization	4	DOPO01 - Weekly working hours	100
							10	DOPO02 - Scope of the development staff	100
				33,3	0,43	Process	6	DOPP01 - Percentage of uninterrupted technical work	50
							6	DOPP02 - Percentage of training time	0
							6	DOPP03 - Percentage of time reducing technical debt	50

Attachment 7 – QEF Requirement Evaluation Criteria

General architecture and software delivery outcomes

		Wfk - Fullfilment (%)		
Requirement	**Metric Evaluation**	**0**	**50**	**100**
GAK01 - Release/Deployment frequency	How often are releases/deployments?	Less than a monthly release	Weekly release	More than a release per day
GAK02 - Frequency that a deployment leads to outage	How often does a deployment result in na outage?	Often	Sometimes	Rarely
GAK03 - Time to recover from an outage	How long doest it take to recover from na outage?	Longer than a day	Less than a day	Less than na hour
GAK04 - Tracking and review of outage metrics	Are these metrics tracked and reviewed by management?	No	-	Yes
GASC01 - Documented API Standards and style guides	Are there defined and documented API standards and style guides?	No	-	Yes
GASC02 - External client access to service mechanisms	How do external API-based clients of the application access the services?	API-based clients access the individual services directly	-	The application has one or more API gateways (e.g. Backends for Frontends) OR No external API-based clients
GASC03 - Service discovery mechanism is implemented	Is there a defined service discovery mechanism?	No - the application uses static, configuration files	-	Yes - the application uses dynamic service discovery: either the Server-side discovery pattern or Client-side discovery pattern
GASC04 - Service registration patterns are applied	Is there a defined service registration pattern mechanism?	No - the application uses static, configuration files	-	Yes - the application uses dynamic service registration: either the Self registration pattern or the 3rd party registration pattern
GAGA01 - Microservice architecture must consist of two or more independently deployable/executable components	Does the application consist of multiple services?	No	-	Yes

Dimension	Deployment and Reliability

		Wfk - Fullfilment (%)			
Factor	**Requirement**	**Metric Evaluation**	**0**	**50**	**100**
Deployment Process	DRP01 - Deployment options definition	Are the application's deployment options defined?	No chosen strategy	Yes - Multiple services per host	Yes - Some combination of the following patterns: Service per Container, Service per VM, or Serverless
	DRP02 - Horizontal scaling support	Does the deployment infrastructure support horizontal scaling	No	Yes - run command (or click) to scale up or down	Yes - fully automated based on load or other metrics
Deployment Reliability	DRR01 - Failure detection and short-circuiting mechanisms	Is there a mechanism to stop routing requests to failed service instances	No	-	Yes
	DRR02 - Fault injection in production prevention	Are faults regularly injected into the production environment?	No	-	Yes

Dimension	Monitoring and Observability

		Wfk - Fullfilment (%)			
Factor	**Requirement**	**Metric Evaluation**	**0**	**50**	**100**
Audit and tracing	MOAT01 - Usage of audit logging	Is there a defined audit logging mechanism?	No	-	Yes
	MOAT02 - Distributed tracing	Is there a defined distributed tracing infrastructure?	No	-	Yes
Application metrics	MOAM01 - Collection of application metrics	Is there a defined application metric collection infrastructure?	No	-	Yes
	MOAM02 - Health check mechanism	Is there a defined health check endpoint and response format?	No	-	Yes
Tracking Deployments	MOTD01 - Track configuration changes	Is there defined mechanism for logging and viewing changes (e.g configuration changes, deployments) to infrastructure and applications?	No	-	Yes
	MOTD02 - Log deployments	Are there any deplyment registry with release versioning?	No	Yes for production environment	Yes for development and production environments

Dimension	Infrastructure and Configurations

		Wfk - Fullfilment (%)			
Factor	Requirement	Metric Evaluation	0	50	100
Configurations	ICC01 - Externalized configuration strategy	Is there a standard mechanism for supplying externalized configuration?	No	-	Yes
	ICC02 - Sensitive properties safe storage	Are sensitive configuration properties (e.g. credentials) stored securely?	No	-	Yes
Supporting Infrastructure	ICS01 - Development/testing environment provisioning time	How long does it take to provision an environment for development and testing?	More than a day	Within than a day	Within minutes
	ICS02 - Similarity between testing and production environments	Do development and test environments reflect production?	No	-	Yes
	ICS03 - Deployment pipeline infrastructure support	Is there infrastructure to support a deployment pipeline (e.g. Jenkins CI server)?	No	-	Yes

Dimension	Libraries and Frameworks

		Wfk - Fullfilment (%)			
Factor	Requirement	Metric Evaluation	0	50	100
Microservices chassis	LFC01 - Definition of microservice chassis frameworks	Are there defined microservice chassis frameworks for each supported language?	None	Some	All
Testing/development tools	LFT01 - Use testing and development tools/frameworks	Are there defined development support frameworks, such as testing frameworks, for each supported language?	None	Some	All

Dimension	Documentation, Organization and Process

Factor	Requirement	Metric Evaluation	Wfk - Fullfilment (%) 0	50	100
Documentation	DOPD01 - Accessible architecture documentation	Is there architecture documentation that is stored in an easily accessible, searchable location, e.g. wiki?	No	-	Yes
	DOPD02 - Technical decisions are logged and documented	Does the architecture documentation document all major technical decisions including selection of patterns?	No	Yes without rationale	Yes with rationale
Organization	DOPO01 - Weekly working hours	How many hours per week do team members normally work?	40-45	45-50	More than 55
	DOPO02 - Scope of the development staff	Are staff dedicated to the application?:	Staff split their time between multiple, unrelated applications and projects	Staff are assigned long term to the application	Staff are assigned long term to a service team
Process	DOPP01 - Percentage of uninterrupted technical work	What percentage of time do team members spend do uninterrupted technical work?	Less than 50%	Between 50 and 79%	80% or more
	DOPP02 - Percentage of training time	What percentage of time do staff spend learning/teaching?	Less than 5%	Between 5 and 19%	20% or more
	DOPP03 - Percentage of time reducing technical debt	What percentage of time is spent reducing technical debt?	Less than 5%	Between 5% and 10%	At least 10% or None (There is no technical debt that reduces productivity)