Universidade do Minho
Escola de Engenharia
Departamento de Informática

Leandro José Abreu Dias Costa

The impact of microservices: An empirical analysis of the emerging software architecture

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Leandro José Abreu Dias Costa

The impact of microservices: An empirical analysis of the emerging software architecture

Master dissertation
Master’s in Informatics Engineering

Dissertation supervised by
António Nestor Ribeiro

December 2021

ACKNOWLEDGEMENTS

I want to start by openly recognizing that this work is the product of not only my persistence and dedication but also the support from the people I luckily have close to me. The process has been demanding to one's mental health but the encouragement and assistance I constantly received enabled me to accomplish this ambitious project. I am forever grateful for the unconditional support and patience of my family and closest friends, for understating the countless hours spent apart so this work could become what I have had envisioned from the start.

Although it has been a fairly autonomous path, I could always count on the guidance, advice, and feedback from my supervisor, António Ribeiro, to who I am deeply grateful.

Lastly, I want to show my gratitude to the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, for financing the submitted scientific paper, which is based on part of the work of this dissertation.

STATEMENT OF INTEGRITY

ABSTRACT

The applications' development paradigm has faced changes in recent years, with modern development being characterized by the need to continuously deliver new software iterations. With great affinity with those principles, microservices is a software architecture which features characteristics that potentially promote multiple quality attributes often required by modern, large-scale applications. Its recent growth in popularity and acceptance in the industry made this architectural style often described as a form of modernizing applications that allegedly solves all the traditional monolithic applications' inconveniences. However, there are multiple worth mentioning costs associated with its adoption, which seem to be very vaguely described in existing empirical research, being often summarized as "the complexity of a distributed system". The adoption of microservices provides the agility to achieve its promised benefits, but to actually reach them, several key implementation principles have to be honored. Given that it is still a fairly recent approach to developing applications, the lack of established principles and knowledge from development teams results in the misjudgment of both costs and values of this architectural style. The outcome is often implementations that conflict with its promised benefits. In order to implement a microservices-based architecture that achieves its alleged benefits, there are multiple patterns and methodologies involved that add a considerable amount of complexity. To evaluate its impact in a concrete and empirical way, one same e-commerce platform was developed from scratch following a monolithic architectural style and two architectural patterns based on microservices, featuring distinct inter-service communication and data management mechanisms. The effort involved in dealing with eventual consistency, maintaining a communication infrastructure, and managing data in a distributed way portrayed significant overheads not existent in the development of traditional applications. Nonetheless, migrating from a monolithic architecture to a microservices-based is currently accepted as the modern way of developing software and this ideology is not often contested, nor the involved technical challenges are appropriately emphasized. Sometimes considered over-engineering, other times necessary, this dissertation contributes with empirical data from insights that showcase the impact of the migration to microservices in several topics. From the trade-offs associated with the use of specific patterns, the development of the functionalities in a distributed way, and the processes to assure a variety of quality attributes, to performance benchmarks experiments and the use of observability techniques, the entire development process is described and constitutes the object of study of this dissertation.

KEYWORDS Microservices, Monolithic, Software Architectures, Inter-Service Communication, Performance Evaluation

RESUMO

O paradigma de desenvolvimento de aplicações tem visto alterações nos últimos anos, sendo o desenvolvimento moderno caracterizado pela necessidade de entrega contínua de novas iterações de software. Com grande afinidade com esses princípios, microsserviços são uma arquitetura de software que conta com características que potencialmente promovem múltiplos atributos de qualidade frequentemente requisitados por aplicações modernas de grandes dimensões. O seu recente crescimento em popularidade e aceitação na industria fez com que este estilo arquitetural se comumente descrito como uma forma de modernizar aplicações que alegadamente resolve todos os inconvenientes apresentados por aplicações monolíticas tradicionais. Contudo, existem vários custos associados à sua adoção, aparentemente descritos de forma muito vaga, frequentemente sumarizados como a "complexidade de um sistema distribuído". A adoção de microsserviços fornece a agilidade para atingir os seus benefícios prometidos, mas para os alcançar, vários princípios de implementação devem ser honrados. Dado que ainda se trata de uma forma recente de desenvolver aplicações, a falta de princípios estabelecidos e conhecimento por parte das equipas de desenvolvimento resulta em julgamentos errados dos custos e valores deste estilo arquitetural. O resultado geralmente são implementações que entram em conflito com os seus benefícios prometidos. De modo a implementar uma arquitetura baseada em microsserviços com os benefícios prometidos existem múltiplos padrões que adicionam considerável complexidade. De modo a avaliar o impacto dos microsserviços de forma concreta e empírica, foi desenvolvida uma mesma plataforma *e-commerce* de raiz segundo uma arquitetura monolítica e duas arquitetura baseadas em microsserviços, contando com diferentes mecanismos de comunicação entre os serviços. O esforço envolvido em lidar com consistência eventual, manter a infraestrutura de comunicação e gerir os dados de uma forma distribuída representaram desafios não existentes no desenvolvimento de aplicações tradicionais. Apesar disso, a ideologia de migração de uma arquitetura monolítica para uma baseada em microsserviços é atualmente aceite como a forma moderna de desenvolver aplicações, não sendo frequentemente contestada nem os seus desafios técnicos são apropriadamente enfatizados. Por vezes considerado *overengineering*, outras vezes necessário, a presente dissertação visa contribuir com dados práticos relativamente ao impacto da migração para arquiteturas baseadas em microsserviços em diversos tópicos. Desde os *trade-offs* envolvidos no uso de padrões específicos, o desenvolvimento das funcionalidades de uma forma distribuída e nos processos para assegurar uma variedade de atributos de qualidade, até análise de *benchmarks* de *performance* e uso de técnicas de observabilidade, todo o desenvolvimento é descrito e constitui o objeto de estudo da dissertação.

PALAVRAS-CHAVE Microsserviços, Monolítico, Arquiteturas de Software, Comunicação Inter-serviços, Avaliação de Performance

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ACID Atomicity, Consistency, Isolation and Durability. 1, 12, 19, 23, 24, 43, 52, 53, 57, 111, 112
AMQP Advanced Message Queuing Protocol. 1, 57, 60
API Application Programming Interface. 1, 4, 13, 18, 25, 30, 42, 69, 71, 108
APM Application-level Monitoring. 1, 30

CD Continuous Delivery. 1, 5, 17, 21, 92, 93, 94, 117
CI Continuous Integration. 1, 5, 17, 21, 92, 93, 94, 117
CLI Commmand Line Interface. 1, 28, 29, 90
CPU Central Processing Unit. 1, 30, 54, 59, 82, 83, 88, 90, 91, 95, 97, 98, 103, 104, 105, 108
CQRS Command Query Responsibility Segregation. 1, 25, 33, 54, 56, 59, 70, 102, 109, 112

DDD Domain Driven Design. 1, 13, 41, 46, 47, 51, 61, 63, 114
DevOps Development and Operations. 1, 4, 5, 15, 16, 17, 18, 20, 21, 31, 92, 117
DFS Distributed File System. 1, vii, 78, 79, 85, 95, 98, 101, 102, 106, 107, 116, 118
DNS Domain Name System. 1, 54, 68, 74, 75, 116

HTTP Hypertext Transfer Protocol. 1, 22, 53, 69, 70, 71, 81, 97, 105

I/O Input/Output. 1, 30, 59, 78, 82, 97, 98, 101
IaaS Infrastructure-as-a-Service. 1, 95

LVS Linux Virtual Server. 1, 75, 99

ORM Object Relational Mapping. 1, 42, 45, 51, 112
OS Operating System. 1, 27, 30, 87, 88

PaaS Platform-as-a-Service. 1, 4, 33, 95
PIM Platform Independent Model. 1, 37
PSM Platform Specific Model. 1, 43, 58

REST Representational State Transfer. 1, 14, 22, 69, 108
ROI Return on investment. 1, 5, 32, 119
RPC Remote Procedure Call. 1, 14, 22

SBC Single Board Computer. 1, 36, 87, 96, 100, 120
SOA Service Oriented Architecture. 1, 10, 14
SPOF Single Point Of Failure. 1, 74, 75, 78, 86, 116

Part I

INTRODUCTORY MATERIAL

1 INTRODUCTION

In this chapter, the overall context, the motivations that led to the conduction of the present dissertation, and the goals purposed are presented. The last section briefly summarizes the contents from each of the chapters that compose this document.

1.1 CONTEXT

Currently, the amount of digital users surpasses more than half of the world population [11], while in the year 2000, less than 400 million people had internet access [12]. The exponential increase of users in the last years forced multiple changes on how the web server's infrastructure work. Nowadays, most modern enterprise applications are hosted on cloud services, making use of Platform-as-a-Service (PaaS) tools, which automate a wide range of processes that assist deployment matters, enabling a higher frequency of delivering changes. Yet, software architectures used back at the beginning of the 21st century are still used up to this day, the so-called, monolithic applications. This simple traditional architectural style is defined by its singular codebase, which typically resides in one component and runs in a single process that handles the entirety of the functionalities of a system. As applications grow more complex and bigger in size, this architectural style starts to exhibit impediment limitations regarding agility and flexibility [13]. Due to the monolith's highly coupled nature, keeping up with high demand and request for new functionalities can become impractical. It happens that those characteristics are often required by modern applications. Emphasized by the current demand for continuous delivery, which combined with monolith's challenges in assuring non-functional aspects, ultimately drove to new software architectures being adopted [14].

One of the emerging architectures is Microservices [15], used by major companies with high user demand such as Netflix and Amazon. This architectural style theoretically solves all the challenges presented by monolith architectures. Contrasting with the lack of modularity of the monolithic approach, this architecture is composed of multiple small services that collaborate and run on its isolated process instead of one large single codebase. That enables better separation of concerns for both teams and actual code dependencies among different services, which only expose its Application Programming Interface (API), favoring isolation and maintainability [9].

The current state of modern development is highly focused on delivering fast and automating as many processes as possible, making Development and Operations (DevOps) (the mindset that aims to merge the development and operations team) one of the most mainstream software development ideologies today. Microservices and DevOps go hand in hand, as it provides a great degree of affinity with its core principles, bringing easy access to agile development and Continuous Delivery (CD)/Continuous Integration (CI) processes [16]. Overall, the monolith's drawbacks reported are addressed by this architectural style, which on the other hand, also has its set of costs associated, which can be underestimated.

Implementing a microservices-based architecture requires a solid understanding of its core principles, patterns, and existing trade-offs since it is no silver bullet that effortlessly solves all the monolith's limitations. Moreover, as showcased by implementations featured in academic research, microservices' development principles and guidelines are still not fully acknowledged, resulting in implementations that never reach their potentialities. Nevertheless, there's a clear and relatively recent trend in regards to modern applications development, with microservices on the headline.

1.2 MOTIVATION

It is very challenging to deny microservices' well-reasoned values that at this point are accepted by both industry and academia [17]. This makes the choice of migrating from an inflexible, traditional architecture to microservices a straightforward decision. Thus, most of the existent literature aims to specify the strategies of how to migrate from an existing monolithic application into a more decoupled one composed of multiple independent services.

For some companies, the reasons behind deciding to migrate from a traditional application to microservices are justified by its promised benefits, generally the alleged affinity with multiple quality attributes. On the other hand, a part of companies adopted a microservices architecture merely because others were doing it, and due to it being the current trend in software architectures [14]. Nonetheless, according to one of the pioneer microservices researchers, "you should only consider using microservices when you have a very good reason to" and that it "should not be the default option when choosing an architecture" [18], emphasizing that it should be a serious and conscious decision reasoned by concrete outcomes. This sort of declarations go against the current movement of undoubtedly developing applications using a microservices approach, which promises to solve all the traditional applications' issues. The contradictory nature of the statements hints that there are significant costs that should be taken into consideration, which are not as evident and documented as the alleged benefits.

The trend regarding microservices adoption keeps getting more attention from big companies, although, very little is said about potential implementation challenges and overheads, understandably, as it is promoted mainly by highlighting its desirable values. It is important to state that the adoption of microservices provides the agility to achieve its promised benefits, but in order to actually reach its values, several key principles related to implementation have to be properly respected. However, due to its newness, there is still a degree of unfamiliarity and lack of established developing principles [19] that often result in implementations conflicting with the microservices' promised benefits. Hence, the Return on investment (ROI) can easily be underestimated and poorly reported. To assure a given implementation reaches its alleged benefits, the introduction of specific patterns is required, from the communication approach used to the data management methodologies applied. Although microservices promote a wide range of fields, it also comes with their set of overheads, which extend to the overall complexity of the system and even performance [20], that can be impacted by its distributed nature.

Simply by being a distributed system, a collection of new challenges arise. The overheads extend over several topics, such as dealing with distributed data management, eventual consistency, distributed tracing and logging, complex maintenance, even changes in the team mindset, and much more, that demand a specific set of skills [20]. These processes represent concerns that are not faced with traditional applications.

Several undeniable new challenges need to be addressed when a microservices architecture is chosen. The entire server infrastructure gets inevitably more complex and the cost associated with, not only the development but also with its maintenance, must not be undervalued. On top of the unquestionable increase of complexity, there are not many studies regarding performance impact, neither there is a consensus on the subject. Moreover, most of the related research is theoretical, lacking empirical data [20] and firsthand reports on its challenges throughout the development cycle. The whole migration from monolith to microservices can be seen as a set of trade-offs in several different aspects, and awareness must be provided. The decision of whether a migration may be beneficial or over-engineering is still too blurry.

1.3 GOALS

The present dissertation is not intended to be a microservices migration research, neither it is exclusively about the development of a microservices-based application. Instead, it aims to provide insights of what are the costs of implementing such an architecture, with empirical reports and data from the development of real applications following the architectures' underlying principles. By doing so, it is intended to supply foundations on whether it is beneficial to adopt a microservices architecture and its impact on every relevant subject. Given that most of the empirical studies of the development of microservices are from surveys with pre-defined answers, there are not many insights into how is the process and the costs faced. The fact that development teams often miss both the value and the costs of microservices, which results in implementations conflicting with its promised benefits, indicates that there is still a great extent of unawareness.

That being said, the main goals of the dissertation consist of the following:

(i) Gather information about the current knowledge on what are the promised benefits and expected challenges of adopting a microservices solution as well as the current best practices and patterns for its implementation.

(ii) Provide empirical data on what are the processes for implementing microservices in accordance with the gathered best practices to achieve its alleged benefits and allow the evaluation of the costs associated with: implementing the business functionalities, assuring quality attributes, scaling the application, handling SPOFs, deployment, automation of processes, monitoring and more.

(iii) Provide concrete data on how the monoliths and different implementations of microservices-based architectures differ performance-wise under multiple load testing benchmark scenarios.

(iv) Contribute to the delimitation of in which circumstances a microservices migration is actually reasonable by addressing the practical challenges faced throughout the development phases and substantiating the alleged benefits. The impact that microservices have on multiple affairs is discussed and the costs and values experienced through the experiments conducted are evaluated and compared against the existent research.

1.4 STRUCTURE OF THE DISSERTATION

This section exposes the structure of the document and briefly describes what is discussed in each chapter by summarizing its content. The present document is composed of 10 chapters:

- Chapter 1 - In the introductory chapter, the overall context of the addressed topic is introduced and briefly explained. Additionally, the motivations behind the chosen thematic are described and the ultimate goals of the dissertation are specified.
- Chapter 2 - The State of the Art chapter, further divided into multiple sections, aggregates the current knowledge on the studied field. It starts by providing a retrospective of the evolution of software architectures towards service-oriented architectures. The theoretical impact of microservices is presented, highlighting the claimed benefits and drawbacks, and compared with traditional software architectures. Since microservices are still a new way of developing applications, the industry best practices and the existent design patterns commonly used are analyzed and discussed in this chapter. Section 2.4 presents the frequently used tools combined with microservices which are studied and compared. The last section focuses on the existent related research conducted by both industry and academia and how they relate with the present work.
- Chapter 3 - This chapter describes the selected methodology used to achieve the proposed goals. The experiment settings used for the methodical component of the dissertation are also addressed.
- Chapter 4 - The reference application used for the experiments and development insights are described in this chapter. Additionally, both functional and non-functional requirements of the reference application are specified. The modelings phases and implementation process of developing the reference application using a traditional approach are reported.
- Chapter 5 - In this chapter, the whole development process of the microservices version of the reference application is reported. Section 5.1 details the decomposition patterns and approaches taken to break up a unified domain into multiple bounded contexts. The implementation process of the selected patterns to make the architecture in accordance with the best practices are described in detail in Subsection 2.3.3. The overheads they introduce and implementation challenges are reported, providing insights from the thought processes and approaches taken to end up with the final result. The challenges associated with different types of inter-service communication mechanisms are described in Section 5.3, which provides understanding approaches used in each version. From the division of the domain to the development

challenges faced and key decisions taken, several technical aspects are reported, highlighting the impact of microservices on the development process and the costs associated with every decision made.

- Chapter 6 - Focuses on the processes of assuring quality attributes and analyzes how microservices impact them. Assuring quality attributes requires different approaches for each architectural style and each section of the chapter focuses on specific quality attributes in which the implementation process is compared between microservices and monolithic applications.
- Chapter 7 - Describes the deployment strategies for each developed solution and the used hardware in which the applications are hosted. The cluster topology used for the microservices solution is defined and the processes associated with automated deployment are described in Section 7.4.
- Chapter 8 - The performance benchmarks conducted are addressed in this chapter. Firstly the measurement methodology and workloads used for the experiments are defined in Section 8.2 followed by the featured use cases and potential validity threats. The response times and throughput results obtained are discussed throughout Section 8.5 in which the source of the potential performance bottlenecks is analyzed.
- Chapter 9 - This chapter summarizes the architectural trade-offs experienced from the development of the microservices solutions in comparison with traditional applications. Once the solutions are developed and the implementation processes are described, this section exposes insights from the most impacted topics, namely, data management, inter-service communication, and assurance of multiple quality attributes.
- Chapter 10 - An overview of the work conducted and how it relates with the purposed goals is conducted in this final chapter. A reflection about the implemented solutions and the differences between the different approaches is made. The experienced costs and values from the development of a microservices solution are conclusively assessed.

2

STATE OF THE ART

In this chapter, the theoretical knowledge, key concepts, and vocabulary required to fully understand the studied subject are presented. In a more introductory lineup, the concept of software architectures and the evolution towards microservices is addressed in Section 2.1. To be able to understand the reasons behind the current popularity around microservices, traditional applications and their limiting characteristics are discussed in Section 2.2. Introducing the microservices architecture concept and its theoretical costs and values, Section 2.3 provides the necessary information for the reader to be able to discuss and compare the insights reported throughout the dissertation. Given that microservices are not seen as the traditional way of developing applications, a review of the existing implementation approaches is presented in Section 2.3.3. The typical tools used in a microservices context are addressed in Section 2.4. The theoretical benefits and drawbacks from the implementation of a microservices-based application are discussed, enabling a comparison between what was experienced during the experiments against what is commonly described in the existent literature. The published work related to the scope of the dissertation is presented in Section 2.5. Section 2.5.1 analyzes the existent empirical research on what are the challenges of implementing microservices. Regarding the performance impact of microservices, Section 2.5.2 aggregates some of the existent performance benchmarks experiments conducted to microservices-based applications.

Most of the information presented is the result of the analysis of existent research work collected from academic resource libraries.

2.1 SOFTWARE ARCHITECTURES

Before diving in-depth into two of the most discussed architectural patterns nowadays, it is important to start with the definition of software architectures and how the concept emerged. Even in the early days, decades ago, software engineers investigated ways of structuring the systems in a manner that could assure both the intended functionalities and a set of requirements [2]. Even though the software architectures' research began way earlier, the first concrete foundation of the term, as we know it nowadays, was settled in 1992 by Perry and Wolf [21]. However, there is still no full consensus on the definition of the term, but one commonly accepted description would be: "The software architecture of a computing system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both" [22]. The chosen high-level software architecture for a given system may determine the quality attributes endorsed, which, more than ever, constitute major market requirements. Once a completely established discipline, different object-oriented based architectural styles started to emerge, such as the Model-View-Controller pattern [23], designed to facilitate integration with graphical user interfaces. On a higher level, with a better separation of concerns in perspective, service-oriented computing emerged. This distributed system style is defined by integrating multiple applications that communicate with each other, seeking to provide a degree of decoupling between implementation and interfaces [24]. This form of structuring systems introduced dynamism, modularity, and the possibility of having heterogeneous implementations.

Service-oriented computing heavily relied on object-oriented principles, however, the shifting towards a service message-passing approach, combined with the notion of business capabilities, ultimately led to the emergence of Service Oriented Architecture (SOA) [2]. This "new" paradigm introduced boundaries in the workflow, splits the business logic according to an enterprise scope, introduces reusability, and promised to increase productivity based on segregation of responsibilities. Following the same base principles, microservices can be seen as an evolution of the SOA architectures, but with some key differences.

2.2 MONOLITHIC ARCHITECTURES

Monolith is a term often associated with building construction, being defined as something that is entirely made of a single piece of material. In a computer science context, monolith is the term used to refer to applications that are composed of a single deployment unit in which all codebase resides and runs in a single process [1]. Within monolithic applications all components are interdependent, lacking any degree of coupling between the several parts of the system at a high-level. This is the traditional architectural style used in most legacy applications and has been used ever since enterprise applications exist.

The tendency is for applications to grow more complex over time, which may lead to a hideous work environment for teams, due to the intimidating size that an application can reach, which is an outcome of the incremental number of new features added in the long run [9]. Monolithic applications can only grow in one direction, progressively making an application bigger in both size and complexity, possibly to a point that is overwhelming to work on [25]. Having every business capability packed in a single deployment unit constitutes a major overhead, as it rapidly becomes too big and complex to grasp. Fowler [15] defines monolith as the term nowadays used to describe systems that get too big, and adds that when a traditional application comes to a point that making any modification or extension becomes impractical and disables any attempt of practicing agile development, modularity must be introduced. The increase of complexity prevents individual developers from understanding the application as a whole [9], which greatly decreases productivity, leading to challenges when a rapid market response is targeted.

Logical separation attempts

Even though the whole codebase resides in a single deployment unit, attempts to create some degree of separation of concerns were conducted. Logic segregation patterns were developed, such as the well-known N-tier architecture or even MVC [23], which are essentially logic separation patterns that can assist with the internal organization of a given application. These architectural styles give the codebase some logical boundaries and modularity at a code level (such as Java packages), making the development process easier with the possibility of having separate teams for each layer/logical component [9]. Even though it is logically delimited, it is still the same single deployment unit, which makes it inevitable to redeploy the whole application whenever it is needed to change something in any of the components [26], which furthers a slow deployment pipeline. Efforts from the development team can be put into ensuring modularity and agility, for instance, through writing automated tests and applying the best possible practices in order to maintain the code organized and comprehensive. However, the logical modularity does not change the fact that the whole application is still packaged and deployed as a whole, nor eliminates the issues related to a team working on a hugely difficult to grasp system [9]. There is also another variation of the typical monolith, which introduces constrained modularity, the modular monolith [1], composed of multiple modules. These modules are meant to be developed independently, and it is an attempt to escape the monolith tight coupling related issues, promoting code reusability, better-organized dependencies, and maintainability. Although it introduces modularity and its associated benefits, it still lacks solutions for granular scalability, fault isolation, technology heterogeneity, and independent deployment and development pipelines for each context.

As a practical example, in an n-tier architecture, one of the most used architectural style patterns in traditional applications, teams are commonly organized across the layers (Figure 1). In this scenario, when a change is made to one of the tiers, the most common outcome would be that the scope of the change would include all the tiers and affect all the teams [1]. With a single deployment pipeline, to deploy the changes the teams have to coordinate and manage the deployment process and its ordering, lacking the ability to make changes without crossing layers and affecting other teams, which prevents the capability to have completely isolated teams. This particular architectural style adds high cohesion between the layers at a technological level rather than a business level, which is showcased by having all business functionalities encapsulated in a single layer. The end result of applying this pattern is that all applications have the same high-level infrastructure regardless of what is the context and requirements.

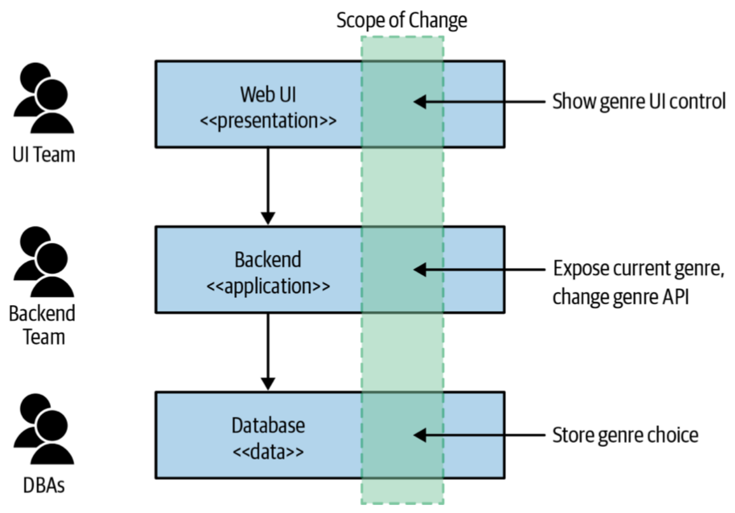


Figure 1: Scope of change in an n-tier monolithic application [1]

Drawbacks

The characteristics of a monolithic architecture highly discourage frequent innovation, not only due to the fact that there is only one shared deployment pipeline, which considerably inhibits productivity, but also as a consequence of several other factors. For instance, the adoption of new technology stacks is not particularly promoted since it would likely force the re-write of the entire application. Besides, horizontal scalability is limited as it does not enable scaling specific parts of the system, being the only option either duplicating the application servers or simply increasing the computing power of the hosting machines [26]. What happens is that when only a portion of the application actually needs to be scaled on-demand, there is no way around it besides scaling the whole application, since there is only one deployable unit.

In times where rapid adaption to the market and frequent deploys are valuable, this type of approach simply may not suit the needs, it just does not promote the quality attributes often required for modern high-demanded applications. Resulting from the lack of modularity, as monolithic applications become more complex, the harder it is to work on, leading to a point where a simple change can turn out to require a serious amount of effort [25].

Its lack of agility in both deployment and development, inflexible scaling, fixed technology stack, and increasing complexity over time caused frustration in development teams that lead them to consider alternative software architectures, which is the case of microservices [14].

Before the exponential increase of networking speeds and evolution of processing power, monoliths, later combined with logical separation patterns, were essentially the only choice for developing web applications. However, still up to this day, this architectural style is widely used and a perfectly viable option for most cases [15].

Benefits

Traditional applications also feature benefits, which in some situations can actually outshine microservices' advantages. Moreover, almost all applications started as monoliths but then evolved towards microservices due to the increase of complexity and effort to implement new features and maintain them [15]. However, when it is known that a certain application is and will remain relatively small, with no high demand for change, which can be very difficult to know beforehand, the monoliths will most likely perform well during their entire lifespan. The fact that everything in these applications resides in one single code base means there are no communication worries, there is no need for constant health checks and intensive monitoring, the deployment is relatively trivial and it is as simple to develop as it can be. Overall, this type of applications is simple to develop, test, deploy and scale, although limited [26]. Since it is the standard way of developing applications, the knowledge to build one is most likely owned by every member of a software team [9]. The ease of data management in a traditional system is one of its biggest assets, easily integrating Atomicity, Consistency, Isolation and Durability (ACID) relational databases' transactions. In contrast, with microservices it is not possible or very challenging to achieve, something that is trivial in traditional applications as there is no need to join data from different sources, nor the need for constant synchronization [18].

2.3 MICROSERVICES

Even though the concept is not particularly new, since the "microservices" term was first mentioned back in 2011 [15], its popularity started to increase exponentially fairly recently (see Figure 2).

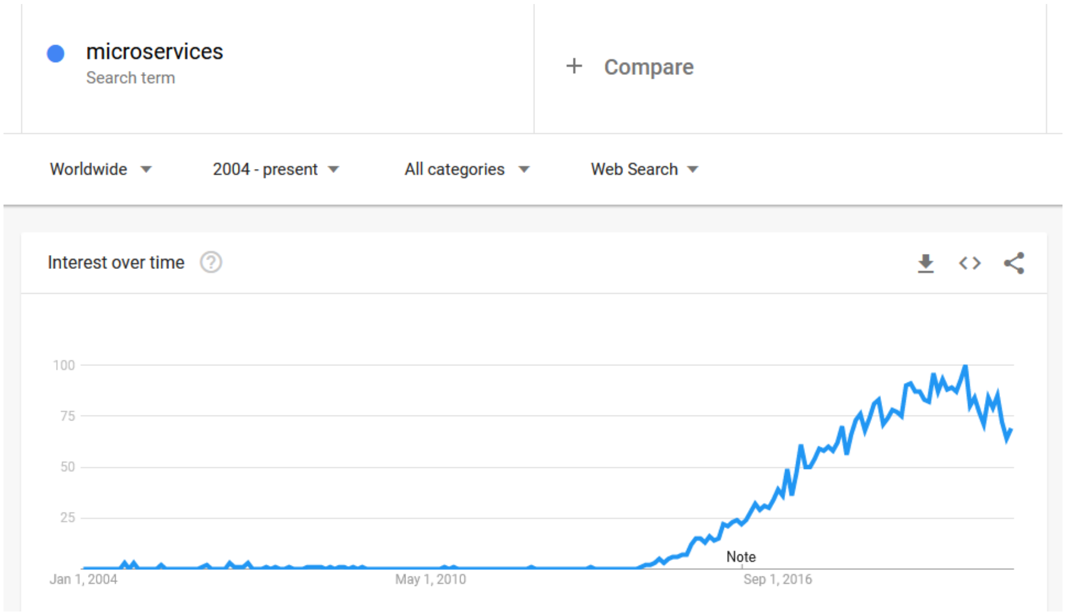


Figure 2: Google trends - Microservices (December 2020)

The ideology was firstly introduced during a conference as a claim, by Rodgers [27] back in 2005, that "software components are Micro Web Services". He states that components should be granular, loosely coupled and could rely on other components to fulfill a certain task, the so-called services, hence its nomenclature. The ultimate goal of this specific approach was to make applications more flexible and easier to change.

According to Richardson [9], one of the pioneers and most influential authors who actively contributes to the microservices' research, modularity is key when developing large-sized and complex applications, and should be introduced at an architectural level. This avoids the most likely long term outcome of complex monolithic applications, the called "big ball of mud".

Modern software development currently highly relates to the microservices architectural style and its characteristics, but it is not straightforward to define. Fowler [15] states that the best way to define it is to point out the characteristics that make a certain architecture classified as microservices. Compiled in simple terms, microservices can be defined as "independently deployable services modeled around a business domain" [1]. These services are intended to be completely independent, having every single one of them running on its own process and responsible for elementary actions [15]. Modularity is introduced by the use of the services, which can only be accessed through the use of an API, assuring impermeability. The different services are responsible for certain, well-delimited tasks, often Domain Driven Design (DDD) oriented [9], and its capabilities are exposed via network endpoints. What defines a service, although not unanimously agreed, is that it should be loosely coupled, maintainable, testable, independently deployed, owned by small teams, and be designed around business capabilities.

Microservices can promote a wide range of quality attributes, but there is a set of guidelines to follow in order to achieve its potentialities. Each service should be small and autonomous, feature well-delimited bounded contexts, isolate failures, and embrace the culture of automation [2]. Most of the guidelines exist to promote loose coupling between the services, which ultimately is the key principle behind implementing microservices.

There are different patterns and practices for implementing a called microservices architecture, and Figure 3 is an example of a simple microservices-based architecture. In the Figure, it is possible to observe that the capabilities of the system are split into multiple units that communicate via a defined protocol, and each service has its standalone database to promote decoupling. This particular solution implements the API gateway pattern, which will be further detailed in Section 2.3.3.

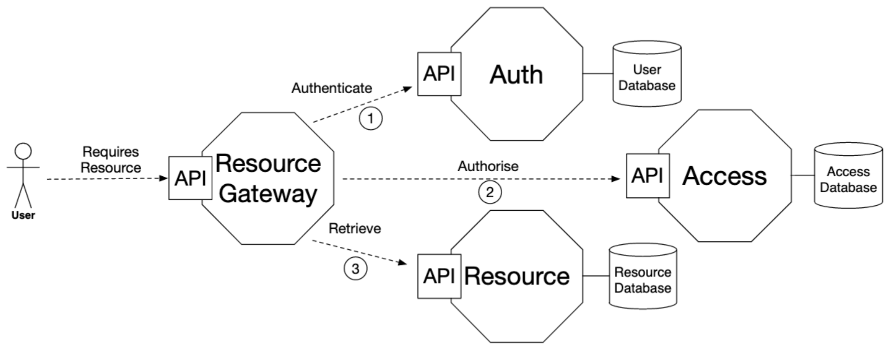


Figure 3: Microservices architecture example [2]

A successor of Service Oriented Architectures

Reassembling the well-known SOA style [28], and actually considered a sub-type of it [1], microservices often refer to a rather different approach, even though both are composed by multiple distributed services that work together as a whole [20]. The differences mainly reside in the fact that with a traditional SOA architectural design, the several services collaborate with each other directly through heavyweight technologies, such as SOAP and WS*. While the microservices aim for a collection of independent services with the minimum coordination as possible [29] using lightweight communication such as HTTP Representational State Transfer (REST), Remote Procedure Call (RPC), or message brokers. Moreover, SOA normally features shared databases and a global data model, which contrasts with the database-per-service pattern standardized within microservices architectures [30]. Another key difference is that the SOA services are typically used to integrate large applications [20], in small numbers, on the other hand, microservices aim for a greater degree of modularity, which commonly results in applications being composed of several rather small services. For those reasons, microservices are often referred to as a "fine-grained SOA" or "SOA done right" [31].

2.3.1 *Microservices promised benefits*

The reasons behind microservices' recent growth in popularity mainly reside in the multiple benefits commonly associated with it. A lot of factors led to the need for developing such an architectural style, and those factors kept getting more and more evident through the years, leading to big companies seeking change for their legacy applications [14]. With the exponential growth of online services' access and audience engagement, from online commerce to civil matters, a lot of issues started to emerge due to the increase of load and consumer expectations. The demand for both new functional and more strict non-functional requirements of a certain, high demanded application rapidly becomes extremely challenging to keep up with. Typically, traditional applications grow bigger and more complex over time, which highly decreases the overall productivity and inhibits the existence of frequent releases of new versions due to the characteristics of a monolithic application. There was an evident need for agility, independence between development teams, infrastructure automation, and evolutionary design, and those are exactly the promised values that a microservices approach enables [15].

The point of splitting the business responsibilities across multiple granular instances emerged due to the lack of agility present in legacy applications. It is specifically the monolithic application's problems that raised the need to rethink how enterprise applications were built. However, only until recently, technology evolution and processing power/cost relation enabled this architectural style to be an affordable and viable option for most companies. The adoption of a microservices architecture promises to facilitate a wide range of topics, such as maintainability, reusability, scalability, agile development, DevOps principles, and more[2][20]. From a business stand point, microservices offer to reduce a much valuable time to market and provide a competitive advantage. At this point, it is clear that opting for a microservices-based architecture promotes several commonly desired outcomes.

Teams and workflow

The many benefits associated with microservices architectures spread across multiple fields. Starting with the development teams' organization and workflow, the decentralized and modular nature of this architectural style enables the distribution of multiple autonomous cross-functional teams organized by business capabilities [15]. Hence, teams can be small and focused on specific bounded contexts.

Given that the architecture is composed of multiple independently deployable services, the deployment pipeline can be split across each one of the services, which avoids teams waiting and depending on one another to release their own updates [3]. Given that, it is possible to efficiently get code into production. A team responsible for a given service can release an update whenever it is ready since it does not affect other teams' deployment pipelines. This behavior clearly contrasts with the situation described in Section 2.2, in which changes in traditional applications likely have an impact across the whole deployment pipeline. As represented in Figure 4, with microservices, there is complete independence between deployment pipelines, in which if a "bad" update gets deployed for some reason, it will not affect the others. This degree of independence and absence of coordination allows a high frequency of deploys in short cycles, which highly relates to the current market's needs [32].

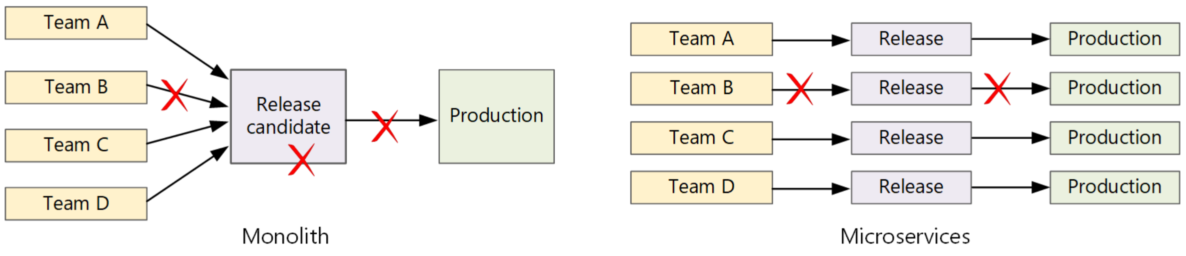


Figure 4: Deployment pipeline of a monolithic and microservices application [3]

The acceleration in the deployment of new features, assuming a deployment pipeline process is properly implemented, enables easier access to agility when it comes to keeping up with the market's changes [31]. This ideology is highly related to DevOps, which is a methodology that aims to unify the software development (Dev) and operation (Ops) teams in order to increase the frequency of new deliveries through automation of processes [33].

Availability and Fault Isolation

The independence in deployment can also enable zero planned downtime or drastically reduce it if the application is built on top of the principles of agile development and DevOps methodologies [34]. For traditional applications, the planned downtime constitutes almost a third of the whole downtime period [35], which is often due to updates, just maintenance, or other reasons, like database migrations of huge databases schemas, often found in monolithic applications. On the other hand, with microservices, there is the agility to implement automation of processes, which allows teams to deliver new versions of services in a matter of seconds [31]. However, assuring a distributed system's availability can become challenging due to the possible high number of instances to apply redundancy to [2] and cascading effects from the failure of a service.

Regardless, if services are developed targeting independence and isolation, a failure or bug in one of the services does not compromise the whole system, since each service runs on its isolated process, preventing a cascade effect of failures. While within monoliths, a failure could bring the entire application process down [36] if high availability mechanisms are not introduced.

Scalability

The agility brought by this architectural style also extends over scalability matters. As mentioned before, with monoliths the only scalability options are either the replication of the whole application or the increase of the host machine's computing power, through hardware upgrades. When it is needed to scale only certain business capabilities, monoliths lack the agility to do so, and since different tasks may demand different resource needs, it can represent an obstacle when high demand is expected. When it comes to microservices, given that each one of the services runs on its dedicated process, service-specific scalability is possible [20]. Even if the scalability needs are temporary, such as during intensive load periods, there is the agility to scale specific services. These characteristics constitute a well desirable convenience for applications with high user demand and strict non-functional requirements. Moreover, with cloud computing options, it is possible to scale resources on-demand and make use of its auto-scale capabilities [37].

Maintainability

As already stated, there are definitely additional efforts associated with a distributed system, but those are typically most perceived during the initial development phases. As represented in Figure 5, the initial productivity when developing a microservices-based application may be lower when compared to traditional applications due to its increased complexity and decentralized nature. Although, in the long term, as the application complexity increases and more features are added, productivity tends to be more or less constant over time, as modifiability and maintainability are highly favored by microservices' characteristics [4]. However, this is only true if independence between services is achieved, with as little reliance on one another as possible, which contributes to easy maintenance and extension of functionalities [2]. Each service tends to have very specific business capabilities, which makes them easy to maintain and keep track of its internal structure, which ultimately leads to effortless modifiability, a valuable characteristic in modern applications. The exponential growth in complexity and lost track of the internal organization in a traditional application, even in the absence of code smells and accordance with good practices can sometimes be unavoidable. It can still occur with microservices, but since each service is designed based on restricted domain scopes, it is not as likely to happen, at least not in short term.

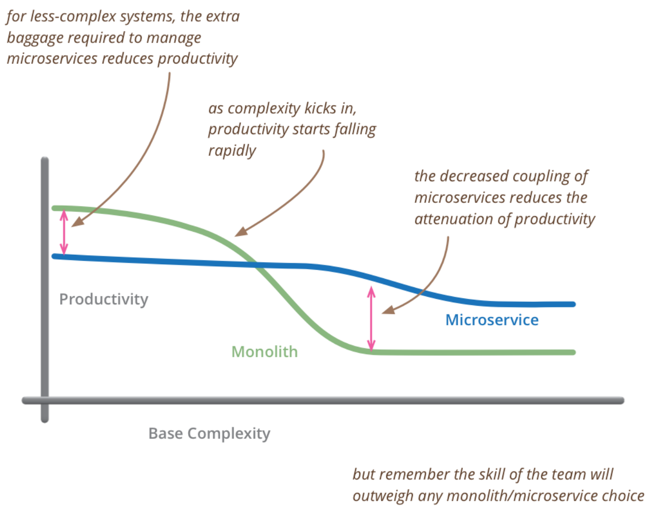


Figure 5: Productivity/effort relation over time [4]

DevOps affinity

Although promoted by having an isolated deployment pipeline for each service, frequent releases are highly dependent on how well CI/CD processes are implemented, which in its turn is dependent on the architecture in use [34]. CI/CD are approaches that provide the agility to reliably and automatically test, build and deploy software at any time, typically in short cycles [34]. Modularity in software architectures is the ideal environment for integrating a DevOps culture, which includes monitoring, testing, and deployment matters, which sympathizes with the microservices' granular nature [16]. A key fundamental of the DevOps methodology is the automation of processes, which, as much effort as it can constitute since it is almost presupposed for a microservices application, will highly decrease the time to market by making the release cycles shorter [32].

Technology Heterogeneity

Another interesting aspect of microservices architectures is that considering the system is composed of multiple services that only interact with each other via defined API, technology heterogeneity is possible [31]. Each service can have its own technology stack, which is completely self-reliant. The ability to use a different technology for each service gives teams the possibility to pick the most suitable tool or framework for each specific collection of use cases, as there is no commitment whatsoever on the chosen technology, due to the service-only scope of the decision.

2.3.2 *Microservices potential problems*

The clear trend going around microservices nowadays can be overwhelming and a lot of companies started to feel the pressure to adopt a microservices architecture for their applications, sometimes with no particular reason. Taibi et al. [14] conducted a survey directed to IT people with considerable microservices experience, and most of the interviewees stated that the fact that other companies were using microservices was one of the reasons behind the choice for the architecture. That results in teams being too eager to implement microservices, resulting in implementations that conflict with its promised benefits. This phenomenon is defined as a teams' "temptation to complicate their architecture simply because it is a fashionable architectural choice" [4]. Essentially, due to microservices' newness, there is still a degree of unfamiliarity and lack of established developing principles that often result in costly mistakes [19].

In order to implement a microservices-based application that achieves its promised benefits there are guidelines to follow. As Lewis stated [18]: "Microservices buy you options", in the sense that its adoption does not automatically enable the claimed benefits, instead, it provides the agility to achieve them. Hence, the claimed benefits highly depend on the implementation patterns used and how decoupled are the services. However, to achieve that, a lot of new concerns arise, not typically faced within the development of traditional applications. A lack of a deep understanding of the implementation processes and patterns often results in implementations that fail to reach their potentialities [31].

Newman [18] claims the migration to a microservices architecture must be "a conscious choice" that should have "some outcome you're looking for", which according to the author, does not happen often, as the industry tends to focus more on the technology rather than in its outcomes. The author also emphasizes his point by affirming that microservices should not be the default option when choosing an architecture, due to the complexity that comes from a distributed system.

The complexity of a distributed system

Being a distributed system, a lot of new challenges that require effort and deep understanding arise. The migration to a microservices architecture is all about handling a complex system that features special needs in terms of monitoring, failures' dealing, eventual consistency, and more [4]. On top of that, it inherits all the distributed system's fallacies [38]. Overall, considerable operational complexity is introduced, due to the existence of more deployable units that special needs, for which are often used tools for orchestration of the services [9]. The common complaints from teams who went through the migration process from a monolithic application to microservices, essentially reside on the extra effort, overall complexity, and added overheads [14].

Eventual consistency

Contrasting with the method level calls which traditional development teams are accustomed to, proceeding operations in a distributed way is often an unfamiliar approach [20]. Since the business functionalities and data are split across multiple independent services, some operations can no longer be processed exclusively locally, forcing the interaction between the services. This behavior introduces eventual consistency since an operation can feature transactions that may span multiple services, opening the possibility that at a given time the system is not entirely consistent. Mechanisms need to be introduced to handle an inconsistent state, which is a consequence of adopting a microservices architecture.

Moreover, as later discussed in 2.3.3, the best practices regarding the collaboration style used within microservices point towards asynchronous communication between services. This inter-communication style adds another level of eventual consistency, in which the collaboration between the services does not occur directly. This asynchronous way of communicating, using messages, may lead to data inconsistency problems, due to the eventual consistency trade-off imposed by microservices [32]. The lack of experience can constitute a barrier when deciding to adopt a microservices-based architecture, which requires a specialized set of skills [20].

Data management

Although services are meant to be independent, it is common that a given service may need resources or capabilities present in others [9], and that happens through some sort of inter-process communication mechanism. Managing and querying data in a distributed system is one of the biggest challenges faced when migrating to a microservices-based architecture [39], which almost inevitably will not feature ACID transactions and introduce eventual consistency, which has its implications. Existing patterns suggest specific ways of performing transactions that require the collaboration of multiple services, but it is always considerably more complex when compared to the simple transactions approach within traditional applications. Regarding read operations, which involve data present in different services, the queries may require distributed joins of large amounts of data. This can become impractical and patterns for solving these inconveniences were developed, which introduce even more operational complexity (see section 2.3.3).

Migration Overhead

Delimiting business boundaries in order to decide the right services' design can be challenging as there is no standard way to divide a business domain into independent services [1]. Furthermore, when migrating from an existing monolithic application, teams describe the migration process as being "impractical" [20]. There are fundamental design and modeling phases that should take into consideration independence and self-contained actions, so the application does not turn into a distributed monolith, with coupling between services. The task of rethinking an application according to a new paradigm, which is not likely mastered by development teams, introduces several new challenges regarding communication patterns between services, DevOps introduction, data management, and more [14]. Additionally, experienced teams state that the splitting of the used database constitutes one of the major issues when migrating to a microservices architecture, sometimes even resulting in keeping their legacy databases due to the overhead of restructuring it [14].

New set of skills

Fowler [32] states that microservices bring serious consequences and its adoption can be seen as a set of trade-offs. Teams working on such an architecture should have a "baseline of competencies" [20], otherwise, it should not even be considered. Specific skills, mostly DevOps related, are mandatory in order to keep the system up and running, which includes the capability of rapid provisioning, monitoring, a robust deployment pipeline [40], and more, which within a traditional monolithic application do not represent major concerns. It is a fact that microservices provide the agility to achieve multiple currently desired benefits but to achieve them, a deep understanding of the existent implementation patterns and ways to deal with its distributed nature is mandatory.

Performance

As a result of being a distributed system, services communicate with each other through a network, which contrasts with the method calls that would occur in a traditional application. The added latency existent within microservices can impact the performance negatively [20]. In the related works presented in Section 2.5.2, performance studies are addressed, but the majority of them showcase that microservices feature lower throughput and higher response time values.

Lack of return on investment

The adoption of such an architecture is not always justifiable and for less complex systems, it can be considered unnecessary over-engineering that can greatly reduce productivity in initial phases [4]. Fowler affirms that "the majority of software systems should be built as a single monolithic application", and modularity should be promoted within a monolith, putting the possible split into services as last resort [4]. He also points that one of the major factors that could lead to opting for a microservices-based architecture is when the team came to the realization that a monolith got too big and extremely hard to implement modifications or efficiently deploy them. But ultimately, the author states that if it is possible to internally organize a system in a way it does not necessarily require microservices' agility, it should be avoided. For less complex systems, extra effort regarding automation of processes for CI/CD, monitoring, and dealing with data in a distributed way, which are concerns inevitably introduced when opting for a distributed system, can represent overheads that not always excuse the extra effort. Moreover, when DevOps processes are not properly implemented, it can actually turn into a productivity bottleneck [20].

2.3.3 *Patterns for implementing microservices*

There are several different ways for implementing a called microservices architecture, which may differ in many fields, such as collaboration style, communication, data management, deployment strategies, and more [5] (Figure 6). To achieve the promised benefits of a microservices-based architecture, there are a set of basic principles to follow, mostly related to maintaining the several services decoupled. Given that microservices are not the traditional way of developing applications, there is still a degree of unfamiliarity [13] that results in implementations that fail to reach their potentialities. To assure that, the introduction of specific patterns is required, from the communication approach used to the data management methodologies applied. Some of the most impactful patterns will be briefly addressed in this section, given that the benefits from microservices depend on the way the architecture is designed and implemented.

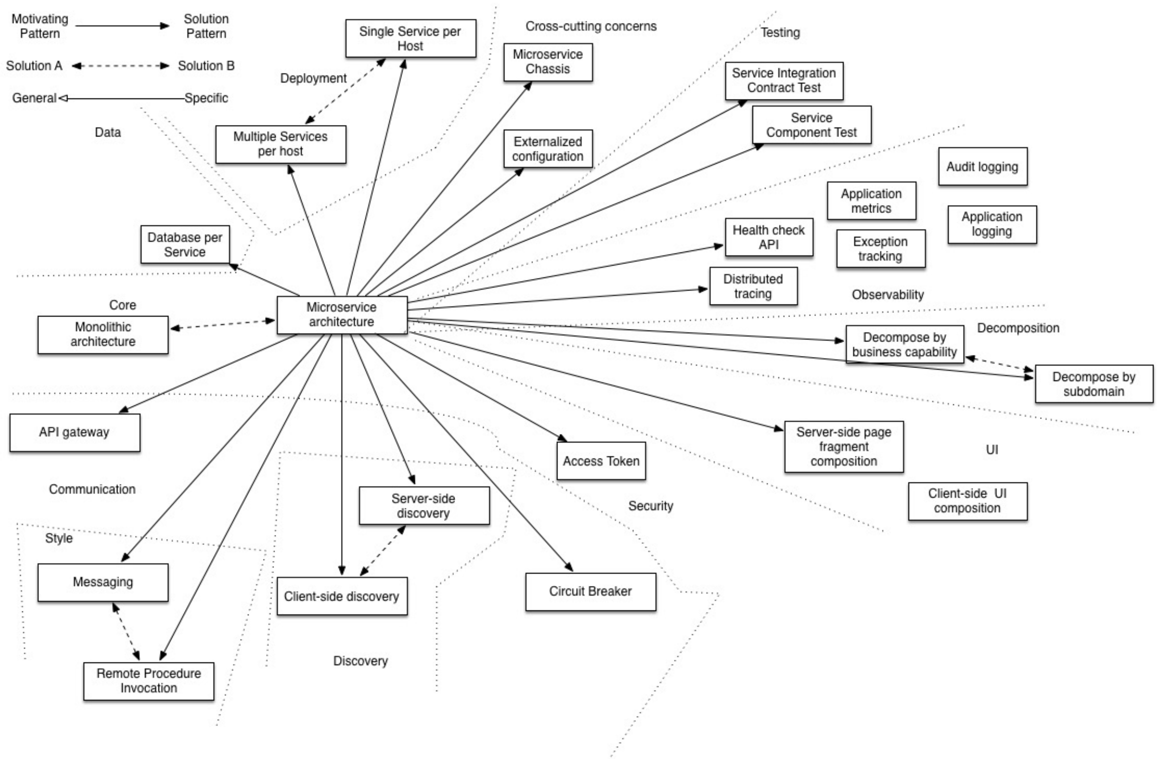


Figure 6: The multiple microservices-related patterns [5]

Collaboration styles

The collaboration style used defines how services cooperate with each other to fulfill certain tasks. This is one of the most impactful factors on microservices' quality attributes and it can essentially be divided into 2 major groups: orchestration and choreography [41].

With orchestration, there is always some conductor to initiate the requests, while the other services wait to be invoked [42]. This type of approach typically communicates through the use of synchronous request/response-based communication mechanisms, such as Hypertext Transfer Protocol (HTTP) REST or RPC. This approach ultimately leads to tight coupling between the services, as it requires all of them to be active and healthy simultaneously, which affects availability [9]. The main benefit from this approach resides in the ability to achieve real-time processing, due to the request/response approach of the communication, usually suitable for read operations. However, when multiple services are involved within the same request/response cycle, if one of them fails, is not available, or is just giving a slow response, a chain of failures occurs [43] and inconsistency needs to be handled. Perhaps only half of the operation was executed and there is no trivial way of doing a rollback of operations since it is a distributed system [44]. The direct communication between services leads to tight coupling and a high dependency degree between the services [42], which goes against the microservices principles and guidelines, which defines services as independently maintainable units.

On the other hand, choreography aims for decentralization and utilizes asynchronous messaging with publish/subscribe mechanisms to communicate through the occurrence of events [41]. To do so, a message broker component is necessary, which is composed of subscribable topics/channels that act as an intermediary and assure messages/events are delivered. The concept of events is introduced, and when an event occurs and is published to a certain topic, other services may subscribe to it and will act accordingly [44]. This way, there is no direct interaction between services, loosely coupling is achieved [42], and through the use of a message broker, there is the guarantee that messages are delivered, whether the participant services are simultaneously available or not [44]. However, this introduces eventual consistency since it is not expected that messages are delivered and resolved immediately. Consequently, the whole collaboration style can be seen as a trade-off between consistency and availability. The eventual consistency and overall collaboration approach can be challenging to manage, especially for read operations, which may retrieve misleading information. To handle the arising issues patterns were created.

The interaction styles can be categorized according to the multiplicity of the communication and whether it is synchronous or asynchronous (see Table 1).

	One to One	One to Many
Synchronous	Request/Response	-
Asynchronous	Notification Request/async response	Publish/subscribe Publish/async responses

Table 1: The various interaction styles for inter-process communication [9]

Database-per-service pattern

Concerning data management, there are several options for sharing data across services. One option is to use a single shared database used by every service, but it ultimately leads to a distributed monolith whose database only scales vertically, and is expensive to manage. The agreed best practice regarding databases and shared data in microservices is the Database-per-service pattern [9], which upholds that every service should be independent of one another and hold its own set of data. The proposition that microservices should not share data holds up to the intention of easing posterior modifications and maintaining the data schema modifications' scope within a single service [1]. It also enables the use of different types of databases, providing technology heterogeneity at this level [44]. Within a monolithic application, the fact that all data is present in the same database schema makes joining data trivial, which is not the case with distributed systems applying this pattern. Assuming each service possesses its own database if data from different services need to be manipulated in order to complete an operation, it has to be proceeded sequentially, since databases are private to the respective services. Performing ACID distributed transactions is possible, but mechanisms, such as two-phase commits, are necessary and a big part of modern NoSQL databases do not support distributed transactions, which is not ideal [44].

Saga Transactions

One solution for transaction management is the use of sagas that break down a transaction into a series of multiple local transactions (Figure 7).

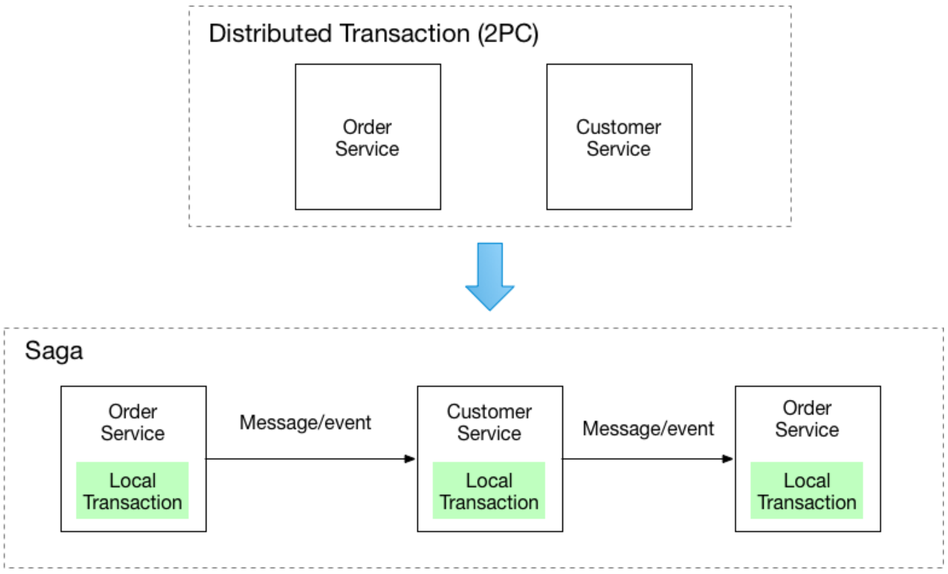


Figure 7: Saga Transactions [6]

Essentially, instead of having a distributed transaction, the steps are split and executed sequentially by each participant service, which then, if using a message broker, posts a message indicating the action performed, avoiding complex mechanisms such as two-phase commit. However, one challenge of this approach is the rollbacks. Imagining the case where the three first steps of the Saga Transaction are already committed within their own database, if the fourth step fails, then the previous steps have to be explicitly undone [44]. Another drawback of this pattern is that there is no isolation, as in ACID, since the execution of a local transaction can be interleaved by another, which may cause anomalies on a high-level, such as lost updates.

API Composition pattern

In terms of querying data present in different services, as mentioned, there is no straightforward way of doing the equivalent of a join as typically done within traditional applications [44]. Having a shared database between services would solve the issue, but, as stated previously, the election pattern for services' data is the database-per-service pattern. Considering that, related data, often required to be retrieved, is not always present within a service, so, some sort of aggregation must be done. One popular pattern to deal with this scenario is the API composition pattern [45]. With this pattern, a new component is introduced, the API gateway (see Figure 8), which acts as an intermediate between clients and the services and is responsible for invoking the services and gathering the necessary data from them [9]. This pattern is suitable when the queries are simple and basic joins are executed, however, it can struggle when complex queries are necessary. Queries that require multiple iterations of data present in different services can be very inefficient as it could require large amounts of data to be present in the API gateway, not to mention the latency involved [44].

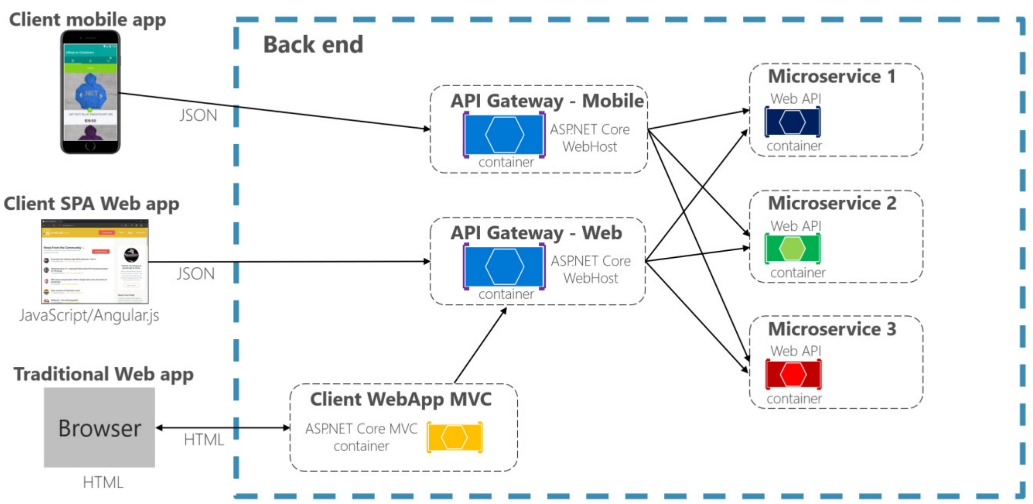


Figure 8: API Composition/Backends for frontends patterns [7]

The API composition pattern can also be useful when it is intended to provide a custom set of endpoints for each different type of client application. Assuming there are desktop and mobile versions of a given application, one same view on the desktop version could require different data to be displayed, while the mobile version of that view could possibly require less information, and using the same endpoints for different purposes can add unnecessary load. Having different entry points, through the introduction of another API Gateway component, provides the ability to have client-specific API's, which enables a great degree of agility and customization. That specific use case of API gateways is defined as another pattern, which is called Backends for Frontends [46].

Command Query Responsability Segregation

The API composition pattern does not solve the challenges faced when complex join queries are required [9], but those issues are addressed by another pattern, Command Query Responsibility Segregation (CQRS) [47]. This pattern upholds that read (queries) and write (commands) operations should be separated within a service. As a result, replicated data, usually, read-only views of frequently requested together data, are maintained in separated dedicated services, or even within an existent service [44]. Those services that hold the replicated views (read models) subscribe to events emitted that affect the owned data in order to keep it updated, even if only eventually. Being able to keep a replica of high demanded data provides the agility of adopting alternative NoSQL databases, which could increase lookup's performance and enable horizontal scalability [44].

The implementation of this pattern avoids the need of having to perform inefficient in-memory joins or a high cadency of requests when complex joins of data that require iteration of large amounts of data [44]. Moreover, there is the positive aspect of not having write and read operations competing for the same resources [48]. On the other hand, besides the extra complexity added, when consulting data present in the read models, there is no assurance that the data retrieved is entirely up to date since the replicas are usually updated through asynchronous events, leading to a possible inconsistent state.

Event Sourcing

Contrasting with the direct calls to code-level methods present in traditional monoliths, the event-driven way of structuring business logic, through the use of events and publish/subscribe mechanisms, is a rather different approach. With CQRS, which specifies that the read and write models are segregated, when a service performs a certain function and changes its state, only afterward the event is published to the message broker. Subsequently, the read model will consume the event triggered and update its state, eventually.

There is no simple way of assuring atomicity between the execution of the operation and the event publication to the message broker [44]. Moreover, the lag associated could compromise the user experience to the extent that a user could update some entities' state and check it right away and the changes have not been committed yet, due to the write and read models inconsistency. This misleading presentation of data happens due to the lack of atomicity within the update and publication of operations of a given service, and a workaround for this issue is the introduction of event sourcing.

Integrated with an event-driven architecture and CQRS, event sourcing enables the persistence of the occurred events, having the entities' state stored as a traceable sequence of events, which composes the only source of truth [9]. Given that, at any time the state of the application is completely reconstructable through the "sum" of the events present in the event store. So, by having the state persisted through a subscribable sequence of events, atomicity is achieved, since the data inserted is the event itself, there is no lag in between [44]. Having a backlog of events also provides relevant data regarding statistics by analyzing the stored history of states, which can be extremely valuable nowadays. The downside of using event sourcing is the drastic change on how to store domain objects, which is most likely atypical for teams familiar with monoliths combined with traditional databases [9].

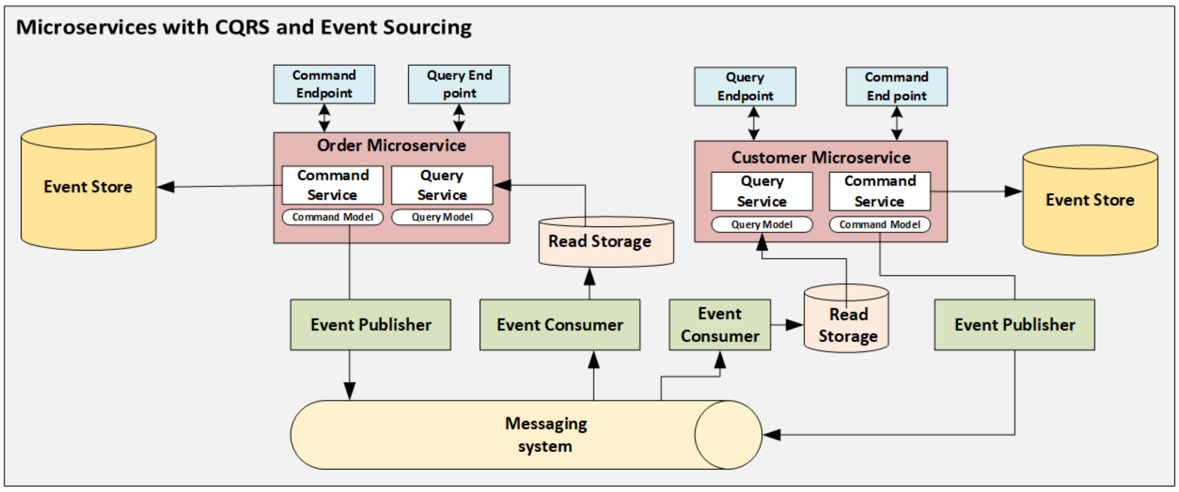


Figure 9: CQRS with Event Sourcing [8]

2.4 TECHNOLOGIES

Given that microservices operate differently from traditional approaches and feature different needs, there are tools and technologies that are used to address them.

Assuming the monoliths are the standard way of developing applications, the technologies and tools specified in this section are focused on microservices, as monoliths do not have special needs in terms of communication, monitoring, health checks, etc. As far as microservices are concerned, the choice of a lightweight framework for the services, the best message brokers, and a way of monitoring the multiple services' health are some of the newcomer worries coming from a monolith background, and the following tools specified throughout this section cover those matters.

2.4.1 *Containers*

Using containers for microservices is the established standard due to their great affinity [49], being Docker[1] the most used containerization tool. According to the official Docker website, "Containers are a standardized unit of software that allows developers to isolate their app from its environment, solving the 'it works on my machine' headache".

Containers are one of the most commonly used existing types of virtualization, but not the only option. For better resource exploitation and the ability to run multiple environments in a single machine, server virtualization was explored. Virtualization is the abstraction on top of hardware that enables division into multiple computers [50].

1 https://www.docker.com/

The two main ways to achieve virtualization are the use of virtual machines or containers. Before the exponential growth of containers' popularity, hardware virtualization with the usage of virtual machines was the used approach for most infrastructures. Even though both constitute an abstraction from the environment they run on, they are rather different approaches. Both pack together all the necessary packages and dependencies, providing an isolated run-time environment for applications. However, in the case of virtual machines, they have their own guest Operating System (OS) and emulate a machine's hardware, in which applications run, providing the flexibility of having multiple OS in one same machine. Virtual machines use special layers for interaction with the hardware, the hypervisors, which are also responsible for the allocation of processes, storage, and memory between them. Despite all its undeniable benefits, virtual machines are large-sized, big resource consumers, lack portability, and feature extra configurations overhead. The big-sized images that are packed with the respective OS were not the ideal scenario for the virtualization of a high number of instances. Its "successor", containers, run independently from the host OS, and that is done by virtualizing a shared OS and packing it together with all the dependencies and code, instead of the hardware virtualization. Its great degree of portability, fast initialization, small size, and independence from the target environment are ideal characteristics for a microservices architecture.

Envisioning a microservices-based application with thousands of moving parts, even though it is possible to use the partition provided by virtual machines, having a separate OS for each one of the services can be too resource-consuming and substantially affect performance negatively. By using containers the overhead is drastically reduced, providing low-cost isolation and better resource utilization. Although, multiple containers running and communicating with each other, adding to a dynamic arrangement of the number of instances, can generate a considerable amount of pressure on the machinery used.

Summing up, containers are portable, enable fast deliveries, scale with ease and carry less overhead when hundreds of running instances are required [51]. Performance-wise, several studies conclude that containers out-perform virtual machines [52].

2.4.2 *Message brokers*

As discussed in Section 2.3.3, asynchronous communication aligned with a choreography style is the agreed best practice for inter-service communication in a microservices context. That being said, a message broker for asynchronous communication is required, which acts as an intermediate in which messages are passed through to be later consumed. With this approach, the sender of the message does not know the receiver's location, and even if the receiver is not active at a given time, the messages are buffered until it gets consumed. Every message broker has its own implementation of a message channel, whose main principles generally stand for the same. Regarding asynchronous communication, there are several different styles, such as notifications, request/async response or a publish/subscribe approach [53]. When an asynchronous one-to-one communication is intended, the commonly used model is queuing, in which messages are stored and eventually consumed by one or more consumers. For a one-to-many communication, the used pattern is the publish/subscribe approach, in which a published message to a certain message channel is broadcasted and consumed by multiple services that subscribed to it.

There are many implementations of a message broker, but when deciding which to choose, some of the main influent factors are the following [9]: broker scalability, messages persistence, supported messaging approaches, delivery guarantee, supported integrations, and latency. Some of the most popular asynchronous messaging technology options are Apache Kafka, RabbitMQ, and ActiveMQ.

Apache Kafka[2], created by LinkedIn in 2011, is a complex to master but a highly performant and scalable distributing streaming platform, rather than a message broker. Its strongest trait is the overwhelming numbers of throughput and latency, promised to be lower than 2ms, according to the official documentation. Additionally, Kafka is highly scalable, since its message channels within a cluster, the topics, can have multiple partitions, which enable parallel messages' consuming with load balancing. The messages/events sent through Kafka's distributed commit log are persisted and order is guaranteed. Regarding messaging models, Kafka only supports the one-to-many publish/subscribe model.

Released in 2007, RabbitMQ[3] is more of a general use implementation of message broker based on queues. Nonetheless, it supports both queuing and publish/subscribe styles. Unlike Kafka, which only supports primitives and binary messages, RabbitMQ works with multiple queue protocols such as STOMP, AMQP, HTTP, and more. Regarding message persistence, this technology provides support for both persistent messages, as well as temporary and also allows message prioritization, which is not supported by Kafka.

Esmaili et al. [54] conducted an empirical study that compares Kafka and RabbitMQ. It was concluded that both of them do not differ too much in terms of latency, throughput, or scalability, and both are stated as perfect suitable tools for pub/sub mechanisms, which is the typical use case for microservices.

2.4.3 *Orchestration tools*

Managing a complex microservices-based application that makes use of containers deployed across multiple different hosts benefits from the use of orchestration tools [55]. These tools assist the integration, control, and management of distributed containers across multiple instances during the development and deployment phases. The two most popular and widely used orchestration tools are Docker Swarm and Kubernetes.

Developed by the Docker team, Docker Swarm[4] is a cluster management tool integrated with the Docker Engine and is used for managing containers at scale, spread across multiple machines. Since it is part of the Docker ecosystem (since version 1.12.0) and shares its Commmand Line Interface (CLI), being the standard way of packaging and distributing container-based applications. When familiarized with the Docker CLI commands, a transition to Docker Swarm is simplified as a lot of the knowledge is shared with core Docker. Another important aspect of orchestration tools is monitoring and tracing, which Docker Swarm lacks natively, having to make use of third-party tools. In terms of scalability, services from clusters managed using Docker Swarm are easily scaled by adding new instances on demand.

2 https://kafka.apache.org/
3 https://rabbitmq.com/
4 https://docs.docker.com/engine/swarm/

Introduced by Google in 2014, the most popular orchestration tool, Kubernetes[5], is a robust tool that also enables container scheduling, auto-scaling, self-healing, native monitoring, and health checks. It has its own CLI, which is not shared with Dockers', providing a wide variety of functionalities not present in other orchestration tools, which also leads to a bigger learning curve. Just like with Docker Swarm, with Kubernetes, it is possible to define how an application should work and how the different components should interact with each other.

The main key differences between the two most used orchestration tools are presented in Table 2.

Table 2: Direct comparison between Kubernetes and Docker Swarm [10]

	Kubernetes	Docker Swarm
GUI	Inbuilt dashboard	Requires third-party dashboards
Scalability	5000 node clusters with 150,000 pods. More mature auto-scaling	Up to 5 times more scalable than kubernetes
Logging and Monitoring	Inbuilt tools available	Requires third-party tools
Node Support	Up to 5000 nodes	2000+ nodes
Availability	Health checks are performed directly on the pods	Containers are restarted on a new host if an existing host fails

2.4.4 *Observability*

Since microservices compose a distributed system, specific monitoring requirements arise, for which there are tools that assist in assuring the observability of the system. This topic within microservices is a much more critical concern since, contrasting with monolithic applications, there are several instances to keep track of the state and metrics, which benefits from being centralized.

Observability techniques provide measurements about a system that enables the inferring of its state and the reasons why something goes wrong. It is often known as being sustained by three main pillars: logs, metrics, and traces [56]. Applied to a distributed system context, centralized logging, monitoring, and distributed tracing tools are meant to collect and aggregate all the necessary data to be able to have a continuously observable system.

Monitoring tools provide and aggregate the system's hardware metrics that are fundamental to enable the dynamic arrangement of the infrastructure. through the collection of metrics, it is possible to avoid unexpected downtime periods by triggering specific actions when metrics reach specific intervals.

5 https://kubernetes.io/

For the overall infrastructure monitoring, tools like NewRelic[6] provide an inclusive view of the host machines involved by collecting Central Processing Unit (CPU), Input/Output (I/O), network, and memory metrics that enable teams to infer the current state of the system and act accordingly. Besides infrastructure monitoring, there is also Application-level Monitoring (APM), which provides information regarding lower-level statistics, such as, the most invoked methods, and many other statistics, which can provide useful information.

There are also open-source alternatives, although they require more configurations. In order to gather applications and OS metrics from the services, Prometheus[7], an agent installed on the system, can make use of the exposed API's by the services and infer their current state. It is usually combined with Grafana[8], which provides a robust and customizable dashboard for better visualization of the collected metrics (see Figure 36).

In the context of microservices, when an error occurs there is no simple manual way of identifying its placement or root cause due to its distributed nature. Contrasting with traditional applications, in which the log files are related to the single existent running instance, with microservices the logs are distributed across the multiple services. Collecting and aggregating the logs from the multiple services into a single location brings easy access to a better perspective that assists the identification of the cause of the problems that occur. Centralized logging tools, such as the ELK stack[9] (composed of Elastic Search, Logstash, and Kibana), aggregate the logs from the services into a database in which it is possible to search and apply filters over the collected data.

Tracing is another relevant observability pillar, but since within a distributed system operations may span multiple services, there is no straightforward way of tracing the end-to-end request flow that takes place. With distributed tracing tools it is possible to trace the path taken by each request and identify the placement of potential performance bottlenecks or bugs that might reside in one of the spanned components. Zipkin[10] is a popular distributed tracing tool that provides traceability through a visual interface with information about the course of the request and also the processing time taken on each spanned service, enabling performance monitoring.

2.5 RELATED WORK

Due to its relatively recent appearance and the fact that most of its applications are developed in an industry context, there is not a lot of research on the subject. The existent trade-off associated with the implementation of microservices is not a particularly sustained topic by empirical validation. This shortage of empirical research work was also perceived by Di Francesco et al. [17], which emphasizes the still low number of studies that address the topic and fill the gap of evaluation research.

Although the existent research roughly agrees on the existent challenges, without strong empirical validation it is difficult to decide whether a migration towards microservices is justifiable. Moreover, there is a clear lack of understanding regarding correct implementations and their influence on quality attributes, composing a rather unexplored topic [57]. The existent empirical reports of the development of microservices come mostly from

6 https://newrelic.com/
7 https://prometheus.io/
8 https://grafana.com/
9 https://elastic.co/elk-stack
10 https://zipkin.io/

surveys with pre-defined answers, hence, there are not many first-hand reports related to how is the process and the difficulties experienced.

2.5.1 *Empirical research on the challenges presented by microservices*

Research conducted by Ghofrani and Lubke [58] is an attempt to supplement the lack of practical data regarding the implementation of microservices-based architectures. The used methodology was to perform a survey directed to developers/software architects who have previously implemented or worked on microservices applications. According to the survey, one of the main difficulties faced was the distributed nature of microservices architecture, which introduces multiple challenges, such as: "Too many repositories to maintain", "Hard to find issues in a distributed system. Rarely any benefit", "networking between dockers" and more. Another challenge described was the skill and knowledge required as it was stated that "changing people's minds that are used to traditional monoliths" represented a major difficulty. The survey also asked what were the reasons that led them to opt for a microservices architecture and the most popular answer was scalability, followed by agility.

Baškarada et al. [20] conducted another survey, which similarly to the previously mentioned, is directed to interviewees with experience in microservices development (with a minimum of 5 years of ICT Architecture experience). The interviews made were composed of questions related to how was the experience of working with microservices, how would they compare it to a monolith and the challenges faced. The majority of the interviewees complained about the lack of agility on traditional applications and confirmed that it is one of the main reasons that justified the migration. All of them praised the possibility of using heterogeneous technology stacks for each service, pointing it as an enabler of innovation, contrasting to the inability of updating the technology used in monoliths, which would force the rewriting of the entire application. Scalability was also mentioned by most interviewees, as monolith's incapability of scaling specific high demanded modules was described as another big reason behind considering the migration to microservices. Some challenges were also pointed, and one of the major ones came from the complexity related to distributed systems. Furthermore, it is stated that this architectural style requires specialized skills and a DevOps mindset adoption. The knowledge for managing data in a distributed way was also considered an obstacle. Since it is not a good practice to share data models between services, coming from a monolithic applications' development background, there is typically little understanding on the subject. The interviewees seemed to disagree on the communication approach to use in a microservices architecture, which clearly, according to several authors, choreography over orchestration is highly encouraged. In terms of testing, there was a consensus that it becomes challenging when using a microservices architecture since integration testing between the services is needed every time a service is updated.

Another survey-based study was performed by Taibi et al. [14], which used a similar approach and collected answers from multiple participants whose requirement was having at least 2 years of experience on a migrated microservices architecture. The results provided a detailed ranking of motivations, reports related to issues dealt with, and experienced benefits. Starting with the motivations behind deciding to adopt a microservices architecture, a ranking system showed that the main desired outcome is maintainability, followed by scalability, the delegation of responsibilities to independent teams, and the ease of adopting DevOps methodologies. Intriguingly, the motivation "Because everybody does it" was also one of the most selected. Concerning issues faced when migrating from a monolith, the complexity of decoupling an existing monolith, the splitting of data from a single legacy database into several, and the communication among services, were stated as the major obstacles. Regarding the issue related to efforts in splitting databases, some of the interviewees admitted that old legacy databases are still connected to their microservices architecture, which goes against the database-per-service pattern discussed in Section 2.3.3. Another chosen migration-related issue was the lack of ROI in the long run, which emphasizes the efforts and overheads when adopting this architectural style, even though it was mostly experienced at initial phases.

Although the surveys are not directly related to the conducted research, they provided crucial information for the purpose of the dissertation. Since one of the goals was to provide empirical data that substantiate the costs and values associated with microservices, the reports from specialized teams can be compared with the results of the present work.

A different empirical study conducted by Villamizar et al. [37] evaluates a real scenario of both monolithic and a microservices implementation. The purpose of the paper was to compare how the two architectural styles differ in several fields: development, deployment, performance, and overall efforts. Some challenges and benefits were stated through the study, providing empirical data regarding the general development of both styles. However, the used application only featured two services, not accurately representing what a realistic scenario for implementing microservices would be, which is preferably composed of more services. Moreover, asynchronous communication was not applied, which is an established best practice for implementing microservices that follow the correct guidelines (see section 2.3.3). Furthermore, the used application for the evaluation was also very simplistic and did not feature more than two operations.

2.5.2 *Existing benchmark experiments*

Numerous studies analyze the performance impact of microservices, but there is not a consensus on the subject. Research work contributions still contradict each other in terms of the performance results of a microservices architecture, potentially due to the distinct implementation patterns used. Some experiments conducted feature a direct synchronous inter-service communication approach that has been discussed as being a bad practice since it is an obstacle in achieving loose coupling between the services [2]. Coming from a traditional application's development background, there is unawareness regarding the implementation process of a microservices-based application that can result in solutions that do not achieve its promised benefits. In order to achieve them, there are patterns and methodologies that need to be introduced which can have an impact on performance, and the interest resides in analyzing specifically those cases.

However, it is difficult to find a performance analysis that features the correct guidelines of a microservices-based application regarding the isolation between the services. From the inter-service communication methodology implemented to the approach used for data management and querying, several factors have an influence on how independent from each other are the services. The analysis of a microservices application that features the practices necessary to achieve its alleged benefits seems yet to be tested and compared to a corresponding monolithic application.

Nonetheless, comparative performance analysis between monolithic and microservices applications have been conducted [59][60][61], but none of them featured the best practices in terms of communication style and data management, all of them used a synchronous request/response approach.

As an example, Flygare and Holmqvist [61] conducted an experiment in order to compare the performance between monoliths and microservices, but, similarly to most empirical studies, it also did not feature an asynchronous collaboration style between the services. Moreover, the setting was not ideal as it featured only two hosts, lacking the amount of communication that would be present in a realistic scenario.

Conducting an experiment with ideal conditions, Bjørndal et al. [62] provided useful appropriate data regarding the performance impact of microservices. The article compares the performance between a monolithic application and one implementation of a microservices architecture, which makes use of asynchronous messaging between the services. Even though it is in accordance with the microservices guidelines, featuring asynchronous messaging, it lacks patterns such as CQRS, which means replicas of data are not maintained, something that can have an impact on performance. Moreover, it makes use of Azure App Services, a PaaS functionality that takes out some control of the equipment used for the experiments. Regardless, this article portrays the closest research to the current dissertation, regarding performance analysis. The study ultimately concluded that the monolith outperforms the microservices application, in both response time and throughput.

The experiment conducted by Akbulut and Perros [63], analyzes three different microservices implementation patterns, featuring different results for each one of them. The three used patterns are the API composition pattern, the chain of responsibilities pattern, and the use of asynchronous messaging. However, it features different types of applications for each pattern, which may provide misleading data to compare results between the implementations.

In general, most of the studies point to worse performance numbers for the microservices architectures, however, comparative studies which compare the same monolithic and microservices applications that follow the best practices and patterns are remarkably scarce. Therefore, the research currently available on the matter is very limited and can be misleading due to the experiment settings and collaboration patterns chosen for the experiments.

Part II

CORE OF THE DISSERTATION

3

METHODOLOGIES

This chapter describes the research methodologies used to reach the proposed goals defined in Section 1.3. The selected methodologies applied to this dissertation aim to evaluate and validate the research results using a scientifically known approach.

3.1 EMPIRICAL RESEARCH METHODS

As mentioned in Section 1.3, this dissertation aims to contribute with empirical data to what is the impact of the adoption of a microservices-based architecture and its multiple implementation patterns in several affairs. As showcased in Section 2.5, empirical research work that evaluates the impact of microservices is limited and can be misleading due to the improper implementations patterns featured. The decision of whether a migration to a microservices architecture would be actually justified and beneficial is not particularly sustained by empirical studies, neither the associated costs are properly addressed.

The used approach to fill the existent gap of empirical reports and answer the initially purposed goals is the conduction of a case study research, which focuses on studying a real project. This empirical research methodology is suitable for the evaluation of software engineering methods and tools [64] and comparison of results from different approaches. It can be used to evaluate the impact of changes compared to some reference scenario [65], in which the outcome is identified and the related activities are documented [66].

Hence, the base strategy used to collect data is to undergo the development of a reference monolithic application and distinct microservices implementations with the same set of functional and non-functional requirements. The purpose is to analyze and report the necessary processes to implement the requirements, and evaluate the effort and costs associated and compare them with the reference application. A comparison between the development of the applications using different architectural styles targets to substantiate, in an empirical way, the vaguely described challenges presented by microservices. Furthermore, evaluate the existent trade-offs associated with the implementation of different design patterns and their impact on quality attributes.

In that sense, the case study includes: the development of a monolithic application, microservices with synchronous orchestration-based communication featuring the data management's best practices, and microservices with synchronous request/response-based communication often featured in existent researches. The discussion of the results from this research methodology is aggregated in Chapter 9.

Regarding the performance benchmarks, the selected research methodology is the conduction of experiments. This empirical research methodology composes a more formal and controlled methodology in which a limited set of factors is manipulated [64]. The performance benchmarks are conducted over multiple implementations of the same application, in which the effect of the use of different architectural styles and design patterns is measured and analyzed. Multiple load testing scenarios are defined and used to evaluate the influence of the defined variables in terms of response times and throughput values. The experiments definition, setting, validity threats, and performance measuring methodologies are described in Section 8.3.

Preceding the empirical research, a literature review is used to evaluate the current best practices and technologies for the development of a microservices architecture, a considered unfamiliar approach of developing applications.

That being said, the methodologies are organized over the 5 following steps:

- The conduction of a literature review to gather information about microservices' promised benefits, associated costs, guidelines, and the best practices for developing microservices-based applications. Since it still is an emerging and different way of developing applications, the existent implementation patterns are addressed.
- The development of a robust ready-to-run in production test application to be implemented according to three approaches (3 case studies): monolith, microservices with synchronous orchestration-based communication, and microservices with asynchronous event-based communication featuring the best patterns gathered during the literature review phase.
- The reporting of the whole development cycle, quality attributes assurance processes and evaluation of the promised benefits and challenges faced during all the development and deployment phases.
- The realization of benchmarks experiments to the three versions of the same application developed. The applications are deployed in a cluster of local Raspberry Pi Single Board Computer (SBC), which enables full control and tuning of the infrastructure. Observability techniques and stress testing tools were used to analyze a set of relevant metrics and resulting throughputs from different test scenarios and workloads.
- The analysis and discussion of the impact that microservices had on the development processes and provision of empirical data that support the decision of whether the migration would be beneficial and justified.

4

REFERENCE APPLICATION

For the research, a reference monolithic application was designed and developed from scratch[1], providing full control and freedom, which would be unlikely with the use of a proprietary system, a common experimental scenario. The test application used for the conduction of the case studies is an e-commerce platform. Thousands of businesses depend on their online shopping platform and, consequently, on its availability and agility and reduced time to market, characteristics allegedly favored by microservices. The choice for the e-commerce domain also resides in the fact that when microservices are theoretically introduced, the practical example given to highlight its promised benefits is very frequently an e-commerce scenario. Moreover, since it is a domain that can be easily split into multiple sub-domains, makes it the perfect candidate for the introduction of a microservices architecture. Furthermore, a complex flow of events that could span multiple services was targeted because the communication between services is a key topic for the case studies conducted. For instance, in an e-commerce scenario, the creation of orders forces inventory checks, followed by payment confirmation, which portrays a decent candidate for that aimed complex flow of events, among others.

Another reason behind the choice for this context is based on the multiple existing suggestions of domain boundaries delimitation for an e-commerce platform. Given that the strategies and patterns for breaking a monolith by itself could compose a standalone research, which is already featured in several publications, the scope for this dissertation is not specifically focused on that particular subject. Nonetheless, there was an effort put into splitting the domain into granular independent services, which is addressed in Section 5.1.

An e-commerce context can feature multiple different ways to solve the same business problems with several distinct implementation approaches and functionalities. As stated, the one chosen is ideally composed of several different entities with some sort of interaction between them so it could portray an ideal test application. The designed reference application was structured to be production-ready and a completely functional e-commerce platform featuring all the core functionalities commonly found on these types of applications. The reasons for the effort put into developing a robust web application reside in the fact that comparisons between traditional applications and microservices are often conducted using very simplistic reference applications. In order to properly analyze the impact of microservices in development, deployment and benchmarks a small application does not emphasize the costs and values associated.

The approach taken and the entities featured are represented in a Platform Independent Model (PIM) (Figure 10), which showcases all the elementary entities and the relationships between them.

1 https://github.com/leandrocosta16/gama-monolith

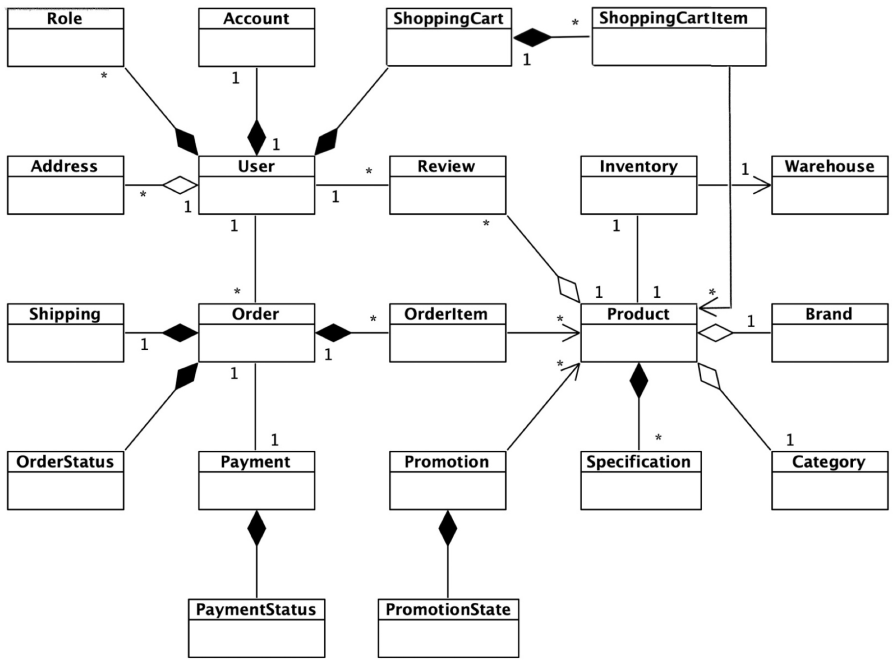


Figure 10: Domain Model diagram

4.1 REQUIREMENTS SPECIFICATION

The requirements for the e-commerce application were defined as an attempt to compose a robust application with capabilities that could originate as many services as possible. The goal is for the microservices version of the application to approximate the closest to a realistic environment, composed of several services, contrasting with the simplistic infrastructures found in existent research work.

The intention behind the specification of a broad set of requirements was to have multiple different scenarios that would require the interaction between different services. This emphasizes the possible challenges of maintaining inter-service communication and distributed transactions, rather than having only limited-scope operations. Another reason behind the implementation of a wide range of functionalities was to provide both more complex workloads featuring intensive operations and also simpler workloads, to have multiple test scenarios for the performance experiments.

The requirements defined are divided into functional requirements, which describe the core functionalities of the system, and non-functional requirements or quality attributes, which include constraints and qualities of the system [67]. The functional requirements can be further divided into user and system requirements, but the main purpose of specifying them was merely to define what are the capacities of the system to be built.

With no intention of providing a full requirements specification document, the defined functional requirements of the reference application are the following:

- **FR001:** The user must be able to register in the system.
- **FR002:** The user must be able to log in.
- **FR003:** The user must be able to edit their personal data.
- **FR004:** The user must be able to consult its personal data.
- **FR005:** The user must be able to remove their account and all associated personal data.
- **FR006:** The user must be able to see the catalog of products.
- **FR007:** The user must be able to filter the products from the catalog according to its multiple specifications.
- **FR008:** The user must be able to order the products by price and name.
- **FR009:** The user must be able to see the available promotions.
- **FR010:** The user must be able to add/remove items to their shopping cart.
- **FR011:** The user must be able to create reviews of a product.
- **FR012:** The user must be able to delete their product reviews.
- **FR013:** The user must be able to create an order.
- **FR014:** The user must be able to pay for an order.
- **FR015:** The user must be able to consult their orders.
- **FR016:** The admin must be able to add/remove stock to the inventories.
- **FR017:** The admin must be able to insert new warehouses.
- **FR018:** The admin must be able to create new promotions.
- **FR019:** The admin must be able to schedule new promotions.
- **FR020:** The admin must be able to delete existing promotions.
- **FR021:** The admin must be able to add/remove products from existing promotions.
- **FR022:** The admin must be able to check inventory information.
- **FR023:** The admin must be able to edit inventory information.

- **FR024:** The admin must be able to create new products.
- **FR025:** The admin must be able to delete existing products.
- **FR026:** The admin must be able to edit existing products.
- **FR027:** The admin must be able to manage brands.
- **FR028:** The admin must be able to manage categories.
- **FR029:** The admin must be able to consult the registered customers.
- **FR030:** The admin must be able to filter the orders according to its multiple specifications.
- **FR031:** The system must be able to check if there is enough stock for a given product.
- **FR032:** The system must be able to reserve stock whenever a purchase proceeds.
- **FR033:** The system must be able to generate the payment form when an order is confirmed.
- **FR034:** The system must be able to alter the orders' status whenever an order is paid.
- **FR035:** The system must be able to calculate the orders' shipping value.
- **FR036:** The system must be able to automatically start/end the scheduled promotions when the defined date is reached.
- **FR037:** The system must be able to change the product's price when a promotion starts/ends.

4.2 QUALITY ATTRIBUTES ENDORSED

For the scope of the study, what are the system's functionalities is not a major influential factor. Instead, the system's quality attributes' assurance represents the biggest concern, since the choice of the architectural style does not have a direct impact on functionality.

The choice of adopting microservices is usually driven by the promoted quality attributes often advertised, which are present in every microservices' search result. However, there are not many studies that make a direct correspondence between microservices and how it affects quality attributes in a practical way, being a rather unclear and unexplored topic [57]. A mapping between quality attributes and patterns in microservices is still not clear. Having the introduction of quality attributes facilitated and actually assure them are very distinct processes, which entirely depend on how the microservices application is developed. The several design patterns often applied to microservices architectures have a considerable influence on quality attributes, yet, there is no clear understanding of exactly how. Therefore, assuring the quality attributes through the introduction of the commonly used patterns is an object of study for the present dissertation.

Quality attributes are not easily tested and can be ambiguous, but the approach taken was to promote the quality attributes that microservices can have the most impact on: maintainability, extensibility, scalability, and availability [17][68]. These quality attributes will be assured across all developed versions of the application and the process is discussed in Chapter 6. The effort for its introduction is analyzed and compared across several implementations.

4.3 DEVELOPMENT OF THE MONOLITHIC SOLUTION

As part of the planned methodology, a monolithic version of the application, that would later origin a microservices-based one, was developed. In most cases, applications start as monoliths, and the typical scenario is that problems start to emerge and then architectures are later evolved towards a microservices approach. Some authors uphold that every application should start as a monolith that endorses modularity, which would result in an easier shifting towards microservices [69]. The reason behind this is that starting with microservices can be overwhelming and negatively impact productivity at early stages, and having the context and domain previously explored, would facilitate the development of microservices.

Regardless, the purpose of developing the monolithic version of the application was to be able to directly analyze the development process of implementing both functional and non-functional requirements of a monolithic application and compare it to microservices. Being the traditional way of developing software, the monolithic solution serves as the reference application for comparing efforts and overheads between the two architectural styles. Evaluating the different methodologies during the development of an application with the same functionalities following different infrastructure organizations aims to highlight what is the impact of adopting a microservices-based architecture.

4.3.1 *Architecture*

In order to structure the base application in a more modular way, a DDD approach was used to design its internal structure. This approach upholds that modeling software by using the same terminology as the real entities of the domain can highly assist the development of business software that solves real-world problems [70].

Although it is a monolithic application, its internal organization was structured following an n-tier architecture. This architectural design suggests the division of the elements of the application into multiple logically separated layers. Each of the several components that compose the application integrates one of the layers which group specific types of entities according to responsibilities. The interactions between the components part of the layers are unidirectional, so each layer is only allowed to interact with a lower layer but not the other way around. The application of this architectural style aims to avoid software that lacks structure and organization, by providing logical boundaries and some degree of modularity.

Figure 11 illustrates the n-tier architecture used for the development of the monolithic solution. Essentially, there are controller components, which are responsible for handling the incoming requests, that are processed by the service components, which may also rely on each other. To interact with data, service components use repositories, which are the elements exclusively responsible for interacting with the database.

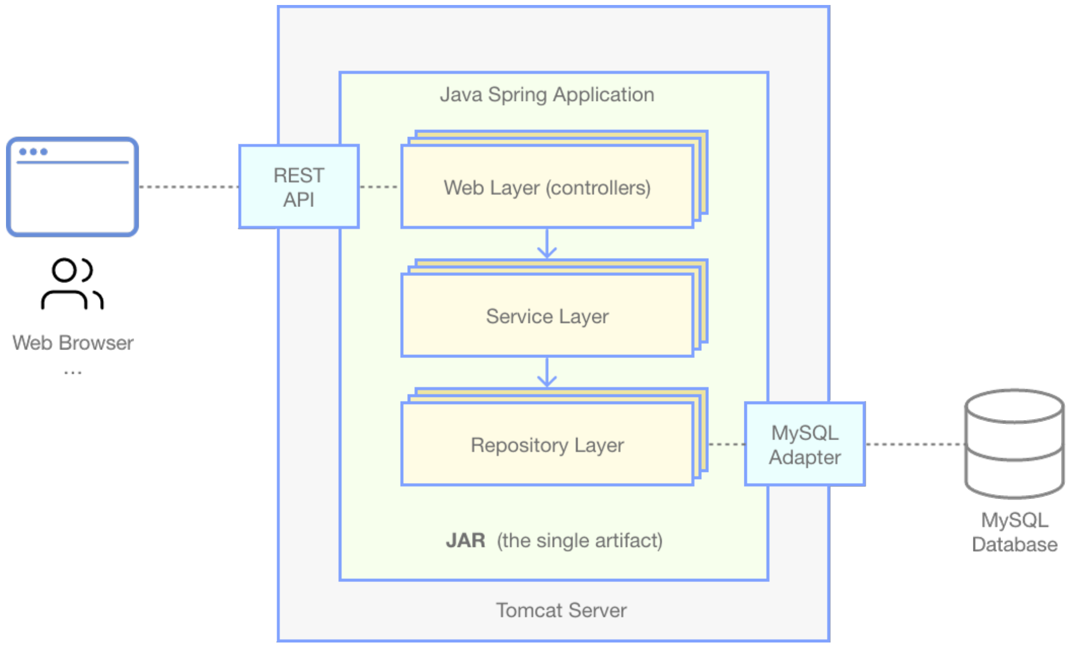


Figure 11: Base architecture for the monolithic solution

4.3.2 *Implementation*

The monolithic application essentially runs in one process that is coupled to one fixed technology stack and is connected to a single database. The entirety of the application was developed using Java combined with the Spring framework (version 2.4.4), Hibernate as the Object Relational Mapping (ORM) through which the database operations are performed, and the database used in production was MySQL. Other technology stacks could be used but the important factor was to use the same stack across all the implementations to avoid additional validity threats. Thus, the decision behind the chosen stack was merely based on acquired experience.

The described functionalities in Section 4.1 were implemented, and everything is packed in a single deployable unit ready to run in production. The developed application was intentionally made stateless, so scalability could be facilitated and multiple instances could be running at a given time, promoting high availability and boosting performance. To assure the absence of a state, for instance, JSON Web Tokens were used instead of server-side sessions for authentication, which enables the identification of a user by decoding a self-contained generated token. Additionally, since the server-side rendering of HTML pages is out of the scope of this work and affects performance, a JSON-based RESTful API was implemented as the backend.

Other tools, typically present in modern applications, such as automatic documentation of the APIs was introduced by using Swagger[2], which is useful for the frontend development and also helps to track the APIs versioning.

Since the goal was to compare two different implementations of the same application and not to have the best possible performance, neither denormalized data nor caching systems were introduced. When the read rate is

2 https://swagger.io/

superior to the write rate, maintaining denormalized data could be beneficial, avoiding the need for extra joins, and also an explicit caching system, that could avoid unnecessary database accesses. However, those performance boosts were not considered for any of the implementations because they could impact the performance tests differently since monoliths and microservices lead with data distinctly. The approach was to optimize the implementation and database accesses as much as possible across all versions of the application and analyze the results without extra tooling.

Regarding implementation challenges, there is not much to report since it is the traditional way of developing applications. Nonetheless, effort was put into making it maintainable by promoting modularity from an internal organization's perspective, in order to promote quality attributes. For instance, by using Spring's Inversion of Control and Dependency Injection features, code-level dependencies are abstracted and delegated to the framework, as an attempt to further loose coupling. Moreover, the SOLID design principles [71], were followed to promote the maintainability of the codebase, since one of the most reported issues of monoliths is the difficulty in maintaining the application and adding new features in long term. Each component has a specific set of functionalities, which are designed to avoid overlaying dependencies, that could negatively impact the extensibility of the application.

Concerning the implementation of the functionalities defined, since with traditional applications there is only one component and a single database with all the data, all business functionalities are proceeded locally at a method-invocation level. The entirety of the operations is easily inserted in ACID transactions, which introduce every mechanism necessary to assure the integrity of the data. However, having all the codebase residing in one single component can be both convenient and problematic. Although in theory, implementing functionalities in a distributed way is a more challenging process, at least modularity is inevitably promoted to a certain extent. While with traditional applications, if care is not constantly taken to promote an organized structure of the several components, the end result, in a long term, is the "big ball of mud", as commonly described.

Having all team members working in one single codebase, despite modularity efforts, will become gradually harder if the application keeps stacking more functionalities over time. It is very difficult for a single developer to be capable of grasping the internal structure of huge monolithic applications, which is something that can deeply affect productivity, just as stated in several surveys conducted (see Section 2.5).

The end result of the development of the monolithic application is a Java project with dozen of classes, which, although are organized and only feature the strictly necessary dependencies between the modules, can rapidly become a too complex workspace to work on. In order to develop a maintainable monolithic application, it is strictly necessary to promote modularity through the creation of as independent as possible modules.

With that in mind, the final structure of the application developed is illustrated in a Platform Specific Model (PSM), a class diagram with the main participating classes (Figure 12) with the different layers represented as packages.

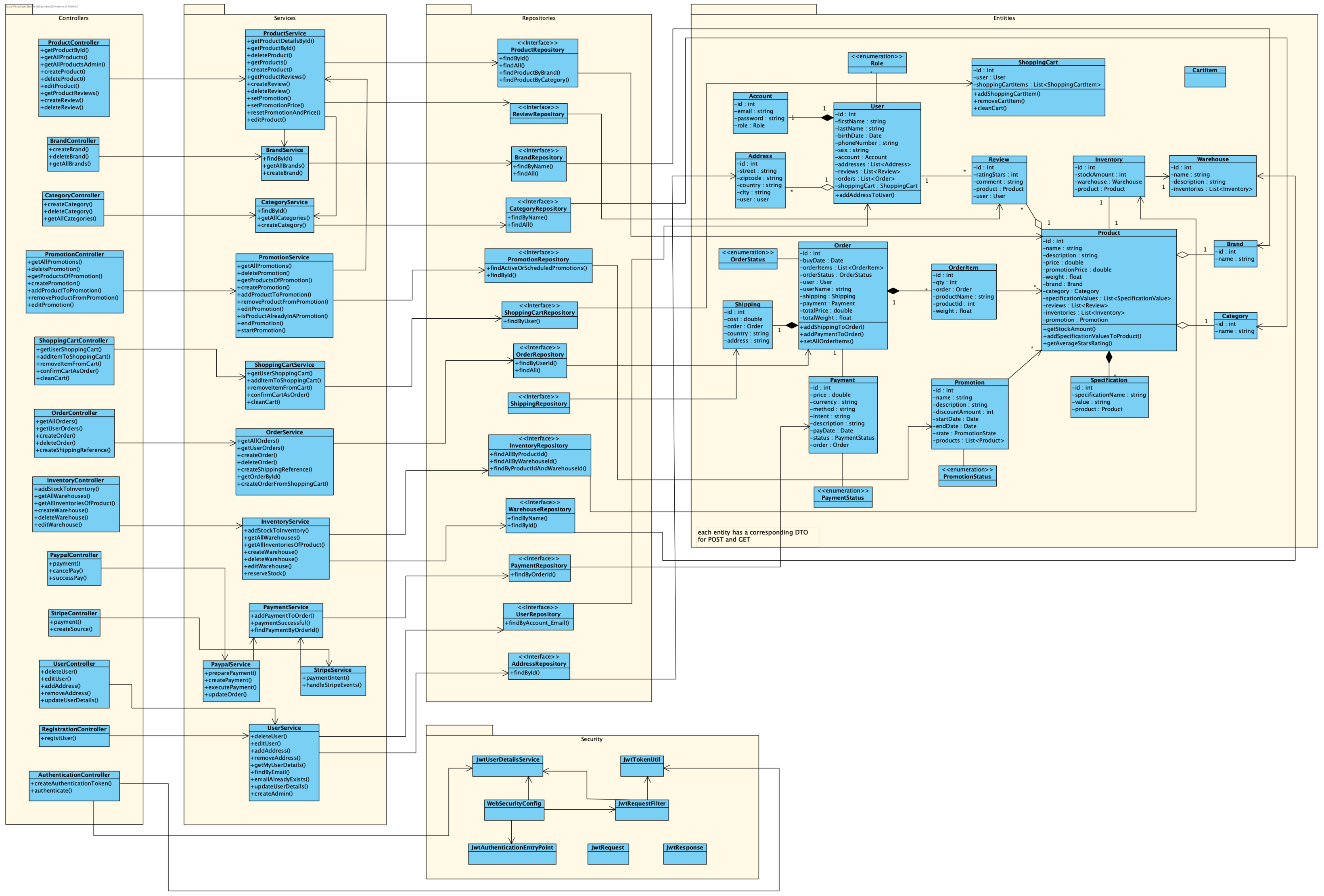


Figure 12: *Platform Specific Model of the monolithic solution*

The data model, generated by the used ORM, is composed of the multiple entities featured in the domain model and the constraints between them, represented in Figure 13.

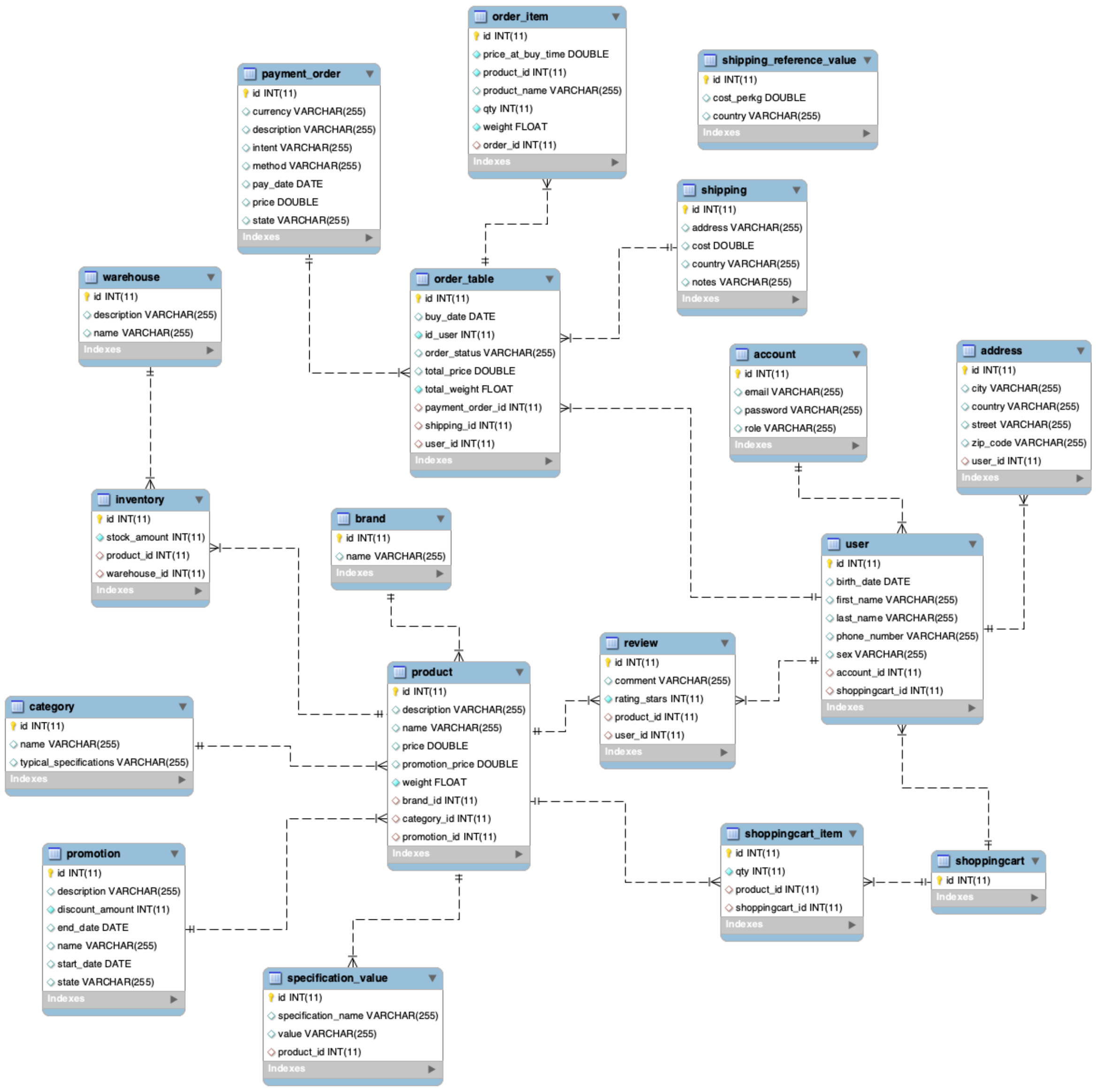


Figure 13: Data model of the monolithic solution

5

DEVELOPMENT OF THE MICROSERVICES SOLUTIONS

Assuring that a microservices application achieves its promised benefits has significant implications on how the application is designed and built, and that constitutes one of the biggest costs of its adoption. The development of a microservices solution features many challenges that were just not present in the monolithic version, which will be discussed in this section.

As described in the defined methodology, two microservices-based applications were developed: one makes use of a message broker, featuring an asynchronous event-driven approach, and the other makes use of a synchronous request/response-based approach for inter-service communication. Through this chapter, implementation patterns, key decisions, challenges faced, thought processes, and some insights from the development are discussed.

5.1 BREAKING THE MONOLITH

There is no standard way of defining the domain boundaries in order to split the domain into multiple services, but there are strategies that assist the decomposition process. A correct decomposition of the domain structures multiple services that are focused on well-delimited tasks, favoring separation of concerns, and to preserve granularity, services should be small and autonomous. Microservices are a perfect candidate for the introduction of a DDD methodology, which defines bounded contexts, aligned with real-world entities, which portray the multiple business processes that compose the application [70].

Having a unified model in a microservices context is challenging since the domain is split across multiple independent instances. Within a monolithic application, as it grows more complex, what often happens is that one same class ends up having different meanings for different contexts, becoming gradually more challenging to maintain. One way of dealing with the increase of complexity and lack of structure is the introduction of bounded contexts, a key principle of the DDD methodology.

Since the monolithic version of the application was already modeled using a DDD approach, with real-world entities and processes, the division into multiple services was made easier. Having a base monolithic version of the application highly assisted the development of the microservices solution because most of the domain-related complexity has already been addressed. The thought processes of implementing the functionalities in a distributed way from scratch would be much more challenging if the development of the reference monolithic application did not precede the microservices version. Splitting the domain into multiple separated services requires knowledge of the domain boundaries which was mostly acquired during the development of the monolith. Although, if the only reason behind developing an entire monolithic application from start to end is to assist the development of another application, it may not be entirely justifiable due to the amount of time and effort taken. Nonetheless, the difficult path of developing microservices became more straightforward, and although some of the key challenges still persisted, it provided a solid base to start with.

As mentioned in Chapter 4, one of the reasons behind choosing the context of an e-commerce platform as the reference application was due to the existing suggestions of decomposition of this domain. This way, there is assurance that the domain is correctly decomposed and does not constitute a validity threat to the architecture, since an incorrect delimitation of the business processes can have a negative impact on several quality attributes.

In order to decompose the monolithic application developed decomposition patterns were analyzed. The most common approach is to either decompose by business capabilities or subdomains [9]. Decomposition by subdomains is highly related to DDD, whose principles were already implemented for the monolithic version, in which the several domains can be easily identified.

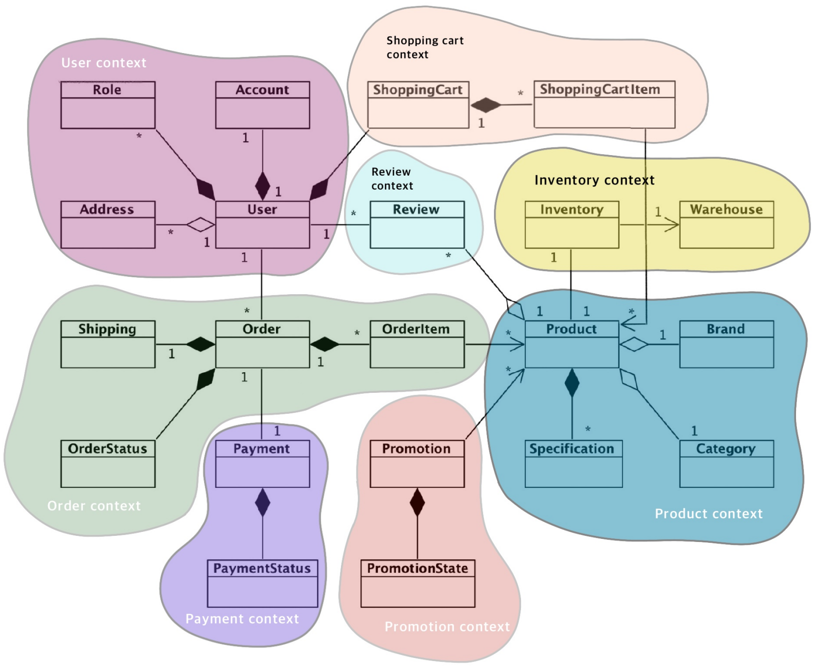


Figure 14: High-level division into sub-domains

A high-level division of domains using the sub-domain decomposition approach is represented in Figure 14, and the several colors illustrate the defined sub-domains. Since the monolithic version has been developed at this point, there was a great degree of understanding of the overall domain, so its splitting was fairly straightforward.

The shared entities from the application of bounded contexts are not yet represented since the first step was exclusively to define the several existing sub-domains that would originate a service in the microservices solution. The thought process of defining a service essentially resides in whether it would benefit from growing separately and having an isolated deployment pipeline.

Microservices and bounded contexts highly relate since each service is meant to have its completely isolated context that does not rely on others. Every entity within a bounded context has a specific meaning independent from other contexts, enabling them to evolve independently. In essence, bounded contexts try to delimit boundaries of a complex domain, which relates to the fundamentals of microservices.

Despite the division into different contexts, manifestly isolated, there are related entities that were split. For example, the product entity has a relationship with several other entities of now different domains. Some of them would need product information to complete their operations, which is the case of the order and inventory contexts. Although product information is required in different contexts, inside each one of them it has a different meaning and purpose. For instance, the order context would need the price information of the products so it can calculate the total price of an order, and also, the weight and dimensions information for calculating shipping costs (Figure 15). Although all of that is information about a product, seemingly part of the product context, it is directly related to other contexts.

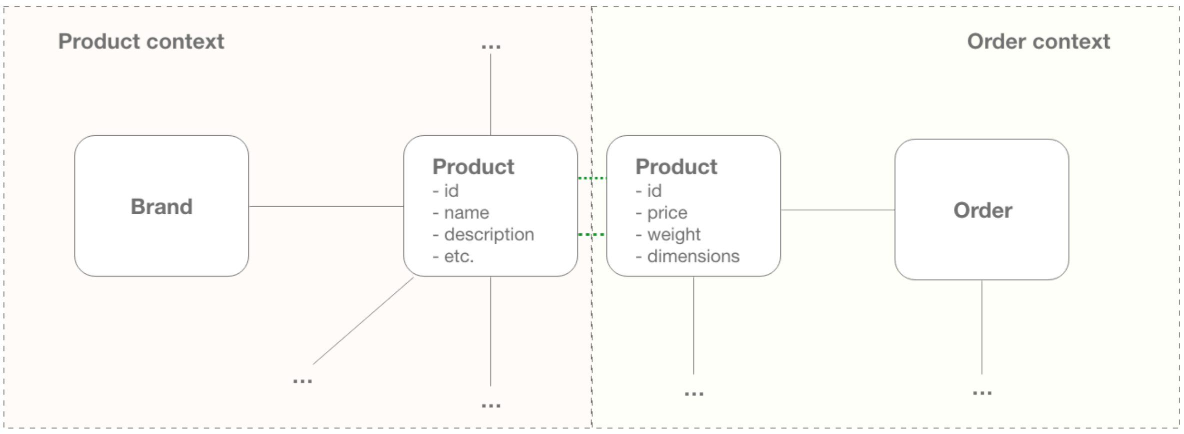


Figure 15: Example of a bounded context

The development of the monolithic version of the application did not feature any of these concerns, since every model was locally available. Regardless, having split contexts could also still be beneficial within a monolith in order to promote maintainability and reduce the scope of changes.

With bounded contexts defined, it is possible to maintain shared entities that could have a completely different meaning to each context, avoiding the need to constantly communicate with other services to retrieve needed information. However, when different contexts share one same entity, synchronization becomes necessary. With shared entities, one of them is the parent of the relation, which applied to an event-driven microservices scenario, would mean that the product replicas have to subscribe to events that may affect its data, for instance, if a product gets deleted or created. More information on data replication and communication between the services is described in Section 5.3. Another approach could be to query the service that owns the necessary data every time it is needed. However, within a bounded context, all required data to fulfill a services' operations should preferably be present locally. With that methodology in mind, each service becomes even more self-contained and less coupled to one another.

Since the domain is now split into multiple services, there is not just a single component packing all functionalities anymore. The microservices version of the application was built based on the same domain model and since one of the objectives is to compare the process of developing the application using different architectures, the same technological stack was used for the main business processes. The business domain services were also built using Java and the Spring framework but each one of them run independently on its own process. All the business capabilities are the same (specified in Section 4.1) and are exposed via the same endpoints specification, being the internal implementation the only distinction.

The several services are responsible for a portion of the business capabilities, which may even require communication with each other, as will be described in the following sections. A correspondence between the services and the functional requirements (see Section 4.1) they are responsible for is represented in Table 15.

Table 3: Mapping between the services and their functional requirements

Service	Functional Requirements
Product Service	FR024, FR025, FR026, FR027, FR028, FR037
Order Service	FR013, FR031, FR034, FR035
Inventory Service	FR016, FR017, FR022, FR023, FR032
Review Service	FR011, FR012
Shopping Cart Service	FR010
User Service	FR001, FR002, FR003, FR005
Promotion Service	FR009, FR018, FR019, FR020, FR021, FR036
Payment Service	FR014, FR033
Catalog View	FR006, FR007, FR008, FR028
Orders/Users View	FR004, FR015, FR029, FR30

The final result of the decomposition and definition of the boundaries of the system is represented in Figure 16 which showcases the microservices responsible for the business capabilities, specifically the event-driven version. Other developed components are missing in the diagram, which is the result of the application of several patterns, detailed in Section 5.2.

Each service was originated out of the division of domains from the decomposition phase. Some shared entities are featured in the diagram, which is the case of the product and order models, whose information is necessary across multiple services. The internal organization of each individual service is based on an n-tier architecture, in which components are part of layers with restricted interactions, furthering modularity at the services' level.

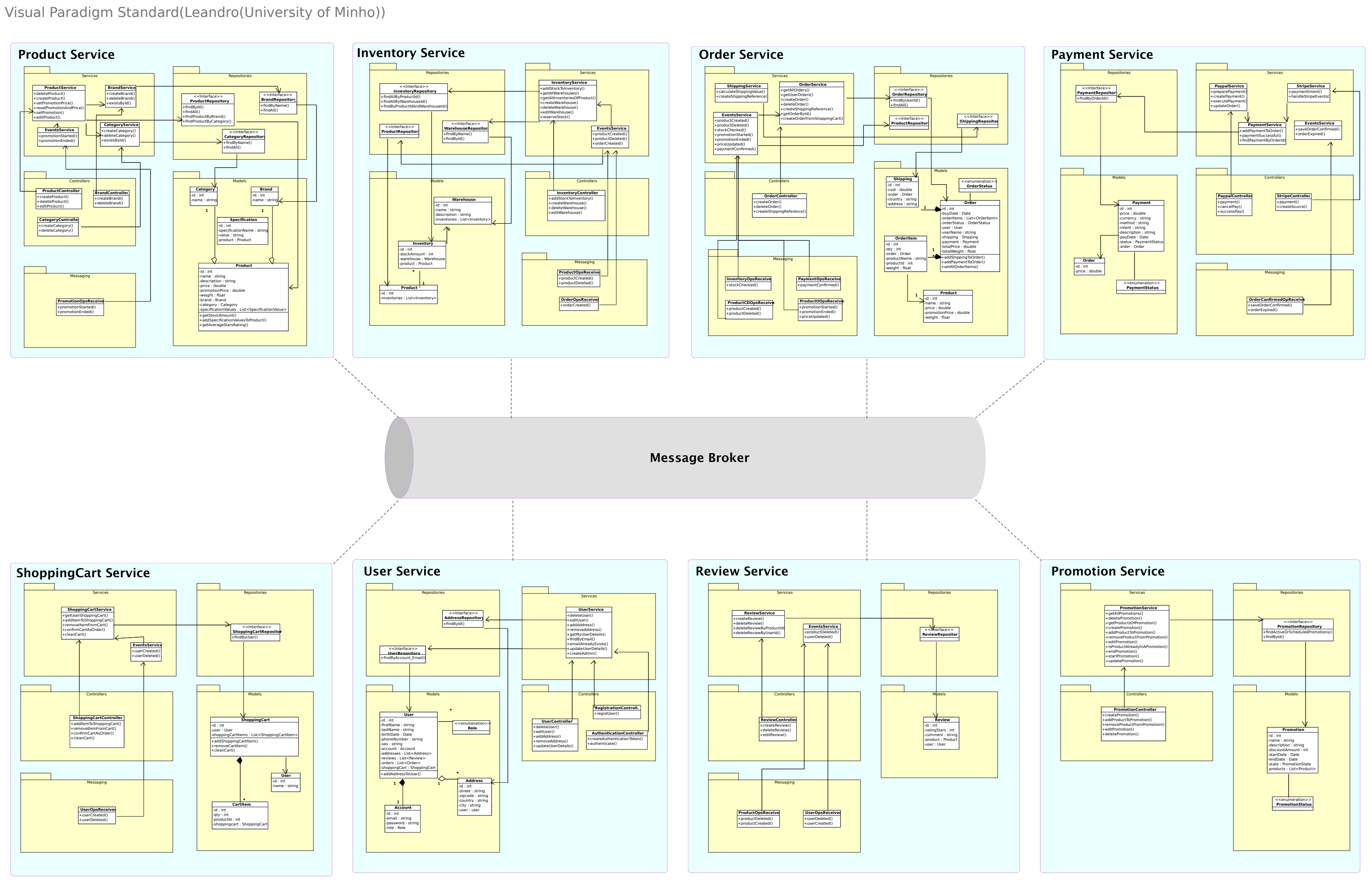


Figure 16: Class Diagram of the business services

5.2 APPLIED PATTERNS

The introduction of a pattern in a system's architecture always comes down to a set of trade-offs in which it purposes to solve something but with some cost associated [72]. Regarding microservices, there are all sorts of patterns that help to mitigate some of its many challenges. The principles and patterns are designed solutions that are proposed to promote the promised values of microservices architectures and avoid improper implementations that could actually detriment them. If the projected values of microservices are not achieved due to incorrect implementations, then whether a given architecture is even considered microservices becomes debatable. What often happens is that the result of not applying the correct patterns and methodologies is something that is not quite microservices, as it conflicts with its base principles. One way of telling if microservices were properly implemented is by analyzing whether it is possible to make changes and deploy a service without modifying any of the others, which confirms loose coupling was achieved.

In order to be able to accurately discuss what are its benefits and challenges, every fundamental of microservices architectures have to be properly respected. Contrasting with the familiarity when developing a monolithic application, microservices' characteristics demand a set of specific patterns and workarounds to achieve the same functionalities and similar behavior.

Through this section, the applied patterns and the problem they purpose to solve are addressed and the challenges faced and the approaches to tackle them are addressed.

Database-per-service

Each service should be as independent to each other as possible, and one way of contributing to further isolation is through the use of a separate database for each service. As a result, any unexpected change on a schema would be completely transparent to the other services, as it does not affect them. The absence of coupling between the services and a single database layer is a key factor when implementing microservices with the goal of having completely isolated development and deployment environments.

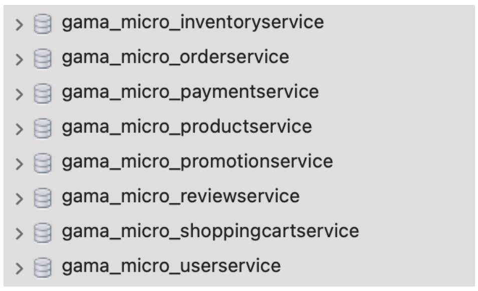

gama_micro_inventoryservice
gama_micro_orderservice
gama_micro_paymentservice
gama_micro_productservice
gama_micro_promotionservice
gama_micro_reviewservice
gama_micro_shoppingcartservice
gama_micro_userservice

Figure 17: Databases per service

Given that a combination of an object-oriented language with an ORM was used, the splitting of the domain using a DDD approach also translated into the splitting of the database models (Figure 17). However, there were no shared entities, aligned with bounded contexts, so that needed to be manually added.

With the data split and only directly accessible by an owning service, querying for data present in different services does not occur directly to their databases. Operations that require data from different services become more complicated with the introduction of the database-per-service pattern since it requires cooperation with other services. Moreover, when a single operation requires the manipulation of data from different services, it introduces several potential problems since, with data split, all the integrity mechanisms and constraints of an ACID transaction do not take place. Furthermore, queries that require to join data in multiple different databases are not as straightforward as with the monolithic version, requiring additional aggregation mechanisms.

To access or share data between services, while avoiding inconsistency issues and communication coupling, specific patterns have to be applied. The communication processes between the services and implemented strategies are addressed in Section 5.3.

When migrating from a legacy application a common scenario is to maintain the same database due to the redesign overhead, but it highly goes against the isolation between services principle which is a core fundamental of microservices. Although having a shared database would solve the inconveniences that come from when data from different services is required, it would introduce coupling between the services and the database layer. The easy path of using a shared database should be avoided as it constitutes a clear anti-pattern.

Since a relational database would be used for every business service, a variant of this pattern, schema-per-service, was applied. A database-server-per-service not only would be impractical but could represent an "unfair" advantage against the monolithic counterpart for the performance benchmarks later conducted.

With the introduction of the database-per-service pattern, different database technologies could be used for each service, but again, since the performance between the monolithic version and microservices will be compared, the same technology stack was used to prevent results variation from the use of different technologies. Nonetheless, in the case of using different database technologies, there is more work in managing multiple database systems.

Saga Transactions

By applying the database-per-service pattern some transactions cannot be entirely processed within a single service's scope, since the data is distributed across different services. Even though the services should own all the data needed to fulfill its operations, some business processes inevitably require the span of different services, from different contexts. With the inability to easily perform ACID transactions between the multiple services, one way of dealing with the need for a sequence of events is to break up a complex operation into multiple local transactions that take place sequentially.

Combined with a message broker, events published on it are used to implement the sequence of transactions that span multiple services. Given the asynchronous nature of using a message broker, there is eventual consistency between an operation and the trigger of the next step, which is something that needs to be handled in the case of the failure of some intermediary step.

One example of the application of Saga Transactions is the flow of creating an order which requires an inventory check followed by pre-processing payment (Figure 18), all operations being part of different contexts.

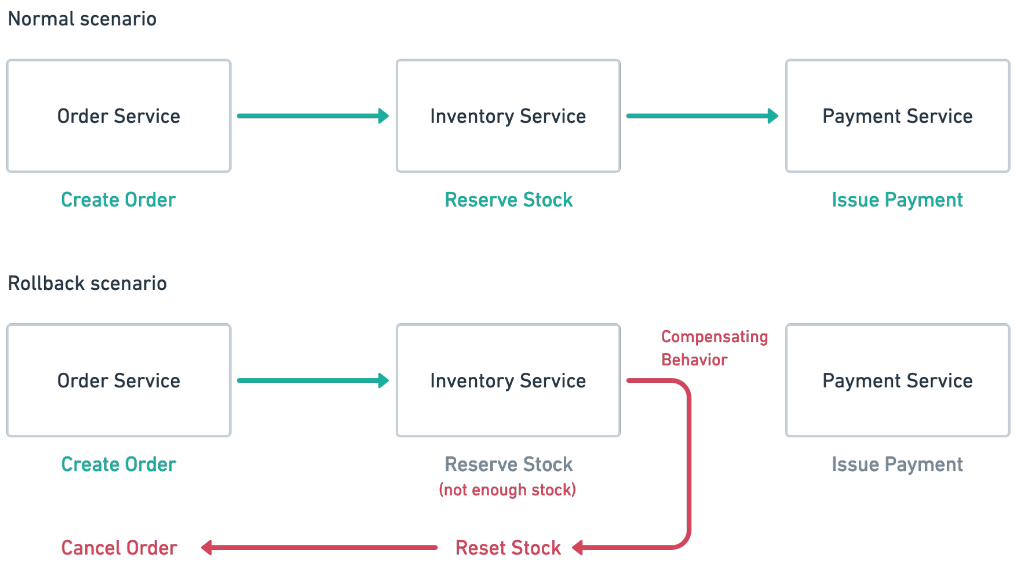


Figure 18: Saga Transactions - Compensating behavior on failure

Due to the eventually consistent nature of this approach, if one of the steps fails, for instance, the inventory check operation, which can be out of stock, a specific event is published and the Order Service will act accordingly, marking the order as canceled. With Saga Transactions the failure of an operation and consequently, the possible rollbacks need to be addressed, otherwise, the system could permanently stay at an inconsistent state. The compensation on failure is something exclusive to the microservices version and has proven to be a considerable overhead due to the need of defining compensating operations for each distributed transaction. There are more possible scenarios that require handling, which contrasts with the straightforward use of ACID transactions within traditional applications.

The introduction of this pattern changes the way of how business operations are implemented, becoming more complex when compared with the traditional way of developing applications. The implementation of the *create order* operation was a much more direct process on the monolithic version, simply requiring the invocation of one method that could easily rollback if necessary. The same principles are applied across all operations that require interactions with other services.

Saga Transactions were also applied to the microservices version that uses synchronous inter-service communication. Although its implementation was made easier, without the need of defining the asynchronous communication channels, it was also necessary to implement compensation methods that are triggered when something goes wrong. In this scenario, if a service is down the operation will fail, and everything prior to it will have to rollback, otherwise, the system becomes inconsistent. Spring's ecosystem provides a relatively simple way of performing rollbacks whenever specific exceptions are thrown, through the use of the *Transactional* annotation. If a HTTP request-related exception is thrown, then all the database operations completed before the exception will rollback.

API Gateway

Whereas the monolithic version was composed of one single unit that received all the incoming requests, microservices are composed of multiple services, each with its set of grained endpoints. To avoid exposing the internal organization of all the infrastructure and hiding implementation details, a component responsible for routing incoming requests to the corresponding services was introduced, the API Gateway.

This component is often added to a microservices architecture for several different purposes. As mentioned in Section 2.3.3, the introduction of an API Gateway could be for the application of the API Composition pattern, in which this component would orchestrate requests to multiple services individually and aggregate the results. However, this pattern was not applied due to its problems regarding more complex querying, which could cause slow responses and resources exploitation due to in-memory joins of data from different services. The elected pattern to solve complex querying was CQRS and its implementation is further explained in Section 5.2.

The developed API Gateway is the single entry point for the application and is responsible for redirecting all the incoming requests to the correspondent destination. Another use case for the API Gateway is to provide a custom set of endpoints for different types of client applications. This is one of the differences between using an API Gateway and an API Proxy, which is an alternative approach. The main reason behind introducing this component, in this case, besides hiding the internal structure of the application, was to use it as a load balancer for services with multiple running instances, combined with a discovery service. Although, in the final application the load balancing capability of the API Gateway was disabled since the used orchestration tool provides built-in server-side load balancing and an internal Domain Name System (DNS) server. If an orchestration tool was not used, the load balancing capabilities and a discovery service would be mandatory, and those were implemented in an initial test version.

In the end, the API Gateway was with the purpose of acting as a single entry point for the application, hiding the infrastructure's implementation details and as an initial layer of filtration for unauthorized requests. Additionally, its tracking may provide interesting analytics from the users' usage of the system.

Regarding possible unauthorized accesses, being the entry point for the application it was decided to add a simple token verification at this level, then the real authentication is done by the requested services. This initial filtration avoids unauthorized requests to pass through, eliminating unnecessary load.

The Spring Cloud Gateway[1] library was used to implement this component mainly because the authentication code was already written in Java so it could be directly applied to it. Despite that, this library offered every desired capability and makes use of a reactive programming model, built on top of Webflux, which is beneficial for these high rates of non CPU intensive requests. This library enables the declaration of custom filters and predicates, load balancing, and easy integration with discovery services.

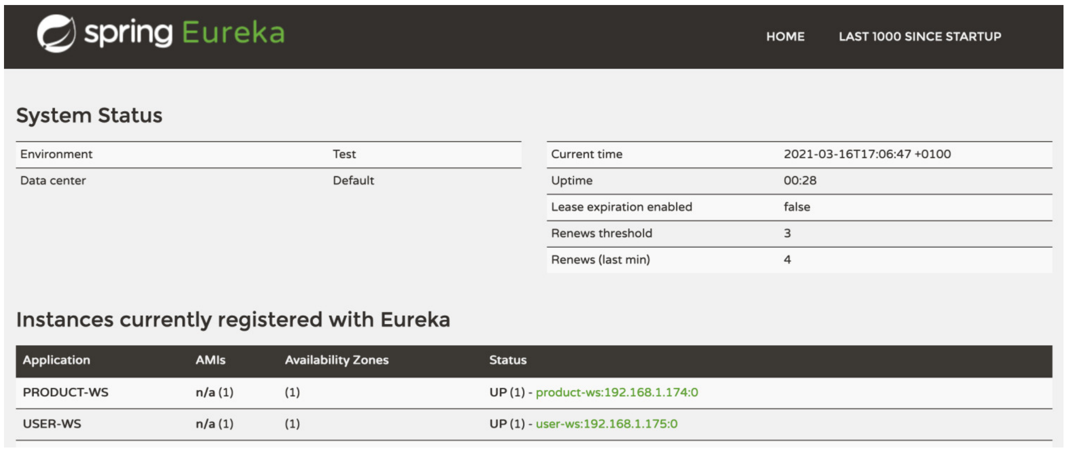
Discovery service

In a microservices context, some services could scale and have multiple replicated running instances with dynamically assigned network locations, which makes static routing impossible since the services could restart with

1 https://spring.io/projects/spring-cloud-gateway

a different address. Therefore, hard coding the address of a given service is not a viable option. As a solution, a discovery service is added to the infrastructure. This component registers the address of the multiple services and when requested, it retrieves the list of their addresses.

The need of knowing what are the addresses of the services is only necessary if a brokerless approach is used, which is the case of the synchronous microservices version developed. Hence, all the implemented services in that version were initially integrated with Netflix Eureka discovery service[2]. During the initialization of each service, a random port number is assigned and a request is sent to the discovery service, becoming registered as an available service. The replicated services have the same application name, so Eureka can identify the registrations as one same service. Whenever the API Gateway asks for the address of a service, the list of the available addresses is provided so the request can be load-balanced across the available instances. Eureka also provides a dashboard in which it is possible to see all the registered services and their locations (Figure 19).



System Status

Environment	Test
Data center	Default

Current time	2021-03-16T17:06:47 +0100
Uptime	00:28
Lease expiration enabled	false
Renews threshold	3
Renews (last min)	4

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
PRODUCT-WS	n/a (1)	(1)	UP (1) - product-ws:192.168.1.174:0
USER-WS	n/a (1)	(1)	UP (1) - user-ws:192.168.1.175:0

Figure 19: Dashboard from Eureka Discovery Service

However, as mentioned previously, it was later discovered that the used orchestration tool provided built-in load balancing by hostname, invalidating the need for a discovery service in this scenario. If an orchestration tool was not used, then a discovery service would be a very important component for the request/response-based microservices version, which uses the service's address to communicate and complete its operations.

Configuration server

Within a distributed system composed of several components and moving parts, having a centralized configuration server could highly assist the process of automatically deploying configuration changes to all services.

Initially, each one of the services had its properties configuration file, and whenever configuration details needed to be changed, it had to be done manually involving the redeploy of the services. This impractical process is solved by introducing a standalone configuration server that is responsible for supporting externalized configuration in a distributed system. The configuration server enables the utilization of centralized storage for

2 https://github.com/Netflix/eureka

mutual configurations that are dynamically updated without needing to redeploy applications. The configurations were placed in a private git repository, so it can keep a version history of the configuration changes. Each service connects to the configuration server during initialization and fetches the configurations from it.

CQRS

When data from different services is required it has to be aggregated from different sources as the services tend to be as granular as possible and not share a database. If a client application requires data that is present in multiple services it could simply request data from each service individually and then perform a client-side join. However, this is not ideal as it would require multiple requests to be performed. These inconveniences can be solved by applying the API Composition pattern, which provides the agility of unifying multiple requests into a single one through the specification of custom endpoints on the API Gateway component. However, as referred previously, this pattern is not appropriate when complex querying is necessary. As a concrete example, if an Admin wants to retrieve all the orders from customers over 25 years of age, it would force a new request to the User Service to fetch the user details for each existing order. This becomes a very costly operation from a network traffic and application load perspective.

To solve these inconveniences, the CQRS pattern suggests that a view of frequently joined data should be maintained and queried whenever a client requires data. Also promoted by the ideology that write and read operations should preferably be segregated, this pattern enables the separation of the load of both types of operations. As an outcome, being each read and write model a separate service, it is possible to scale both independently, according to the needs.

The application of this pattern solves the complex querying-related issues through having replicas of the data that otherwise would have to be distributively joined. The views maintained are part of new services whose only purpose is to provide a read model for the client applications. In order to maintain the replicated data, however, the services responsible for the read operations have to somehow get notified whenever its data changes, so it can be consistent. If an event-based approach is used, then these services have to subscribe to all events that affect the maintained data, which requires the definition of an event whenever any entity changes its state.

This pattern was only applied to the event-driven version of the microservices architecture. The version that communicates using synchronous mechanisms simply requests data directly from other services. Maintaining views of replicated data using a request/response-based approach would not be ideal since a failure in the views services would cause an inconsistent state as it would highly depend on their availability.

There are implementations of this pattern that feature a read model for each individual service. That contrasts with the definition given by Richardson [47], who states that the maintained views are supposed to be composed of frequently joined data. By having a separate read model for each service, all of them would have to contain data from other read models, otherwise, it would not make sense to use this pattern rather than the API Composition, which is simpler. That specific way of applying this pattern introduces too much redundancy, synchronization overhead, and unnecessary complexity.

The approach used introduced only two read models, divided according to domain contexts that feature some degree of relation. The read models maintain replicas of data from multiple services through the subscription of events that affect its data (mechanisms that are further explained in Section 5.3.1). By having all necessary data for every possible query present locally, there is no need for additional communication between the services. When a client application requests data it is fetched using a request/response approach, instead of publishing an event, which is preferred for read operations so the user experience is not sacrificed.

Having that said, all the data requested from the client applications come from the read models, which are eventually consistent, since all the operations to the data are originated from the consumption of events. In some contexts, getting not assuredly up-to-date data from the application can be an unacceptable scenario, in which misleading information is critical. In such scenarios, that cannot bear the presentation of inconsistent data, asynchronous communication should not be used, consequently, even adopting microservices in the first place. One of the costs associated with this architectural style is dealing with eventual consistency due to the lack of ACID transactions and the fact that the read and write models are segregated. The implementation of Saga Transactions means that, at a given time, an operation can be partially completed, but it is certain that all the participant services will fulfill their roles, and the system becomes eventually consistent. Furthermore, by having read models completely separated from the write models, another type of eventual consistency exists. At a given time the write and read models may hold different information. Consequently, the only assurance is that, eventually, everything will be in sync. Therefore, the whole situation can be described as a set of trade-offs since, even though the replicas are only eventually consistent due to the asynchronous flow of events, the availability of the system as a whole is enhanced, and loose coupling is promoted.

The existent services responsible for managing the entities make use of a SQL-based database, MySQL. However, the SQL constraints and referential integrity mechanisms are not needed for the views since there is no business logic applied to this level. Moreover, SQL solutions present challenges regarding horizontal scalability, and since in an e-commerce scenario the majority of operations would be reads from fetching the catalog of products, it could benefit from that type of scalability. Therefore, and driven by the fact that each service can have its technological stack, a NoSQL alternative was considered. MongoDB was selected due to its flexibility and easy access to scalability.

Regarding the read models implemented, there are many entities related to products that reside in different services, namely reviews, promotions, brands, categories, and inventories. When a product's information is requested, all those entities would have to be joined in order to provide all the information about a given product. With every related entity present within one same database, it would be close to what we have on the monolithic version of the application. An independent service containing the products' data and every related entity was introduced, the *Products View*. Its only responsibility is to perform database queries when requested and update the data it holds through the subscription of certain events. The other read model defined aggregates data from Users, their Orders, and associated payments, which is also information that is frequently requested together, originating the *Orders/Users View*. These Views are essentially standalone services that act as the read models, which means that every read operation is directed to these services that hold replicated data, constantly being synchronized. Regarding its implementation, since the code for integrating the services with the Advanced Message Queuing Protocol (AMQP) was already developed for Java for the other services, the same programming language was used. The internal structure of these services is very similar to the main business services (Figure 16), except these do not have any business logic applied. Figure 20 features a PSM in the form of a class diagram which illustrates the classes defined inside each read model and the relations between them.

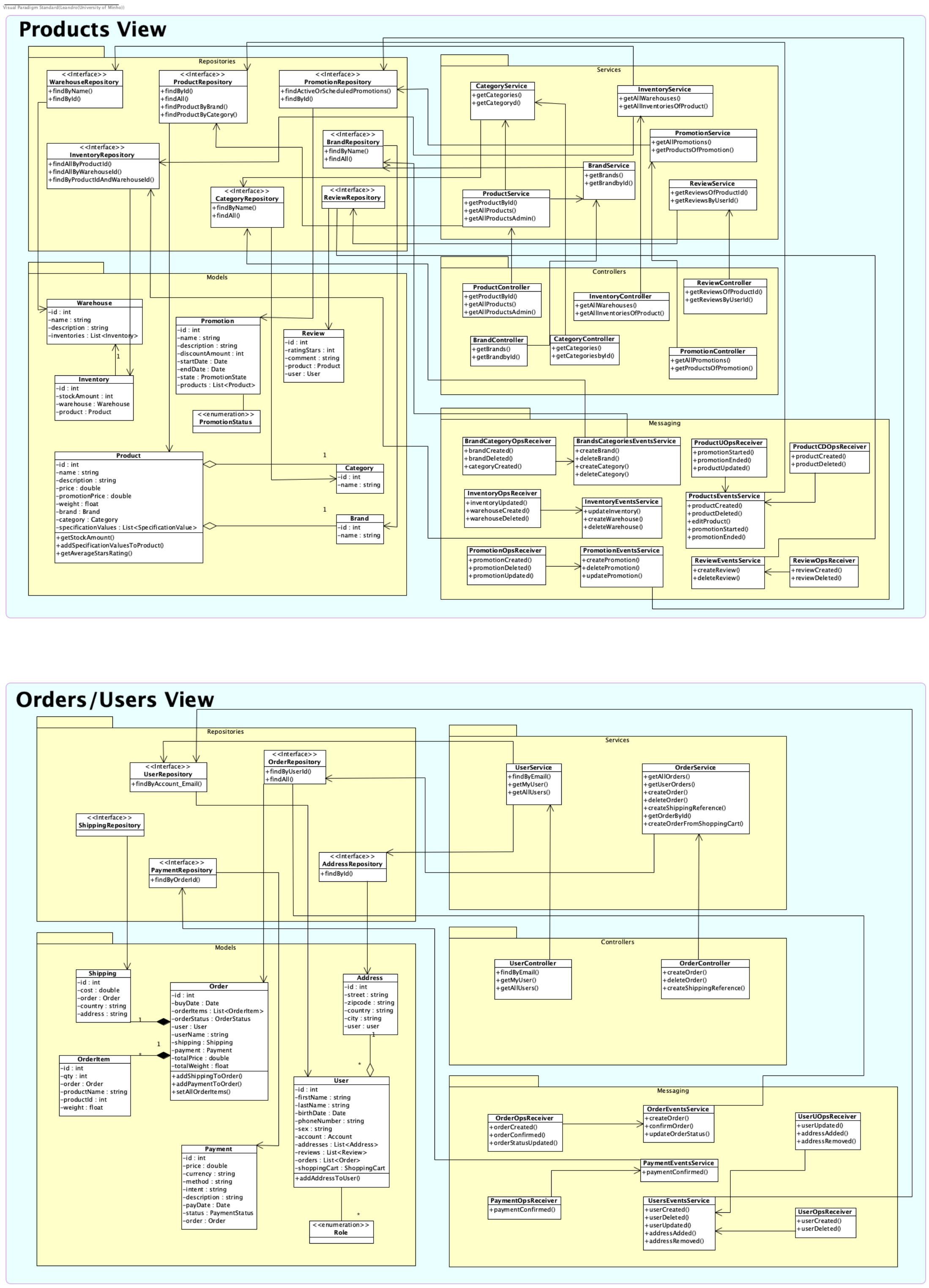

Visual Paradigm Standard(Leandro(University of Minho))
Products View
Repositories
<<Interface>> WarehouseRepository +findByName() +findById()
<<Interface>> ProductRepository +findById() +findAll() +findProductByBrand() +findProductByCategory()
<<Interface>> PromotionRepository +findActiveOrScheduledPromotions() +findById()
<<Interface>> BrandRepository +findByName() +findAll()
<<Interface>> InventoryRepository +findAllByProductId() +findAllByWarehouseId() +findByProductIdAndWarehouseId()
<<Interface>> CategoryRepository +findByName() +findAll()
<<Interface>> ReviewRepository
Services
CategoryService +getCategories() +getCategoryd()
InventoryService +getAllWarehouses() +getAllInventoriesOfProduct()
PromotionService +getAllPromotions() +getProductsOfPromotion()
BrandService +getBrands() +getBrandbyId()
ReviewService +getReviewsOfProductId() +getReviewsByUserId()
ProductService +getProductById() +getAllProducts() +getAllProductsAdmin()
Models
Warehouse -id : int -name : string -description : string -inventories : List<Inventory>
Promotion -id : int -name : string -description : string -discountAmount : int -startDate : Date -endDate : Date -state : PromotionState -products : List<Product>
Review -id : int -ratingStars : int -comment : string -product : Product -user : User
Inventory -id : int -stockAmount : int -warehouse : Warehouse -product : Product
<<enumeration>> PromotionStatus
Product -id : int -name : string -description : string -price : double -promotionPrice : double -weight : float -brand : Brand -category : Category -specificationValues : List<SpecificationValue> +getStockAmount() +addSpecificationValuesToProduct() +getAverageStarsRating()
Category -id : int -name : string
Brand -id : int -name : string
1
1
1
Controllers
ProductController +getProductById() +getAllProducts() +getAllProductsAdmin()
InventoryController +getAllWarehouses() +getAllInventoriesOfProduct()
ReviewController +getReviewsOfProductId() +getReviewsByUserId()
BrandController +getBrands() +getBrandbyId()
CategoryController +getCategories() +getCategoriesbyId()
PromotionController +getAllPromotions() +getProductsOfPromotion()
Messaging
BrandCategoryOpsReceiver +brandCreated() +brandDeleted() +categoryCreated()
BrandsCategoriesEventsService +createBrand() +deleteBrand() +createCategory() +deleteCategory()
ProductUOpsReceiver +promotionStarted() +promotionEnded() +productUpdated()
ProductCDOpsReceiver +productCreated() +productDeleted()
InventoryOpsReceiver +inventoryUpdated() +warehouseCreated() +warehouseDeleted()
InventoryEventsService +updateInventory() +createWarehouse() +deleteWarehouse()
ProductsEventsService +productCreated() +productDeleted() +editProduct() +promotionStarted() +promotionEnded()
PromotionOpsReceiver +promotionCreated() +promotionDeleted() +promotionUpdated()
PromotionEventsService +createPromotion() +deletePromotion() +updatePromotion()
ReviewEventsService +createReview() +deleteReview()
ReviewOpsReceiver +reviewCreated() +reviewDeleted()
Orders/Users View
Repositories
<<Interface>> UserRepository +findByAccount_Email()
<<Interface>> OrderRepository +findByUserId() +findAll()
<<Interface>> ShippingRepository
<<Interface>> PaymentRepository +findByOrderId()
<<Interface>> AddressRepository +findById()
Services
UserService +findByEmail() +getMyUser() +getAllUsers()
OrderService +getAllOrders() +getUserOrders() +createOrder() +deleteOrder() +createShippingReference() +getOrderById() +createOrderFromShoppingCart()
Models
Shipping -id : int -cost : double -order : Order -country : string -address : string
Order -id : int -buyDate : Date -orderItems : List<OrderItem> -orderStatus : OrderStatus -user : User -userName : string -shipping : Shipping -payment : Payment -totalPrice : double -totalWeight : float +addShippingToOrder() +addPaymentToOrder() +setAllOrderItems()
Address -id : int -street : string -zipcode : string -country : string -city : string -user : user
OrderItem -id : int -qty : int -order : Order -productName : string -productId : int -weight : float
User -id : int -firstName : string -lastName : string -birthDate : Date -phoneNumber : string -sex : string -account : Account -addresses : List<Address> -reviews : List<Review> -orders : List<Order> -shoppingCart : ShoppingCart +addAddressToUser()
Payment -id : int -price : double -currency : string -method : string -intent : string -description : string -payDate : Date -status : PaymentStatus -order : Order
<<enumeration>> Role
1
1
*
*
1
*
*
Controllers
UserController +findByEmail() +getMyUser() +getAllUsers()
OrderController +createOrder() +deleteOrder() +createShippingReference()
Messaging
OrderOpsReceiver +orderCreated() +orderConfirmed() +orderStatusUpdated()
OrderEventsService +createOrder() +confirmOrder() +updateOrderStatus()
UserUOpsReceiver +userUpdated() +addressAdded() +addressRemoved()
PaymentEventsService +paymentConfirmed()
PaymentOpsReceiver +paymentConfirmed()
UsersEventsService +userCreated() +userDeleted() +userUpdated() +addressAdded() +addressRemoved()
UserOpsReceiver +userCreated() +userDeleted()

Figure 20: Class Diagram of the read models

Since there is flexibility with MongoDB collections, denormalized data, such as brands and categories' names, which are originally present in different collections, were introduced inside the products collection, avoiding frequent aggregations of the data. Those entities could have also been fully embedded inside the products collection, however, retrieving the list of all distinct brands and categories would have become a much more I/O intensive query. Whenever a client application requests the data of a product, all the necessary details are present inside one same document as every information is either embedded or denormalized. The same principles were applied to the Orders/Users view, which contains every entity related to orders and users.

Having dedicated read models facilitated the read operations and provided a more organized infrastructure by segregating the read and write operations. If CQRS was not implemented, then complex queries would be highly inefficient due to the distributed nature of the system, which makes data joins across services impractical. However, it adds a considerable amount of complexity and overheads due to the need for data synchronization and definition of the events and their handlers for each case. By delegating all the read operations to dedicated services it was necessary to define events for every single update, creation, or deletion of all entities. As a consequence, since the read models maintain replicas from different services, it is almost constantly performing changes on its data, so the CPU is never idle, which is highlighted when a high load of writes occurs.

A failure in the reception of an event would cause a permanent state of inconsistency which could be very challenging to track and could rapidly turn into a critical situation. In the case of failure on the services that hold the read models, all the events are stored in the message broker until they are healthy and online again so operations still take place. However, to assure data is consistent, it largely depends on the availability of the message broker. To further mitigate potential inconsistencies on the replicated data scheduled jobs can be defined to periodically verify if everything is properly synchronized.

5.3 INTER-SERVICE COMMUNICATION

Ideally, services should not need data from each other, but when there is no viable alternative, the way services share information is an extremely relevant topic. The communication approach used in microservices is often one of the main reasons why it fails to achieve the promised benefits of using such an architecture. One key principle of using microservices is the independence and isolation between the services, which is directly affected by the inter-service communication approach selected.

While querying data or performing business operations in a monolithic application is as simple as invoking locally accessible methods, microservices require more complex mechanisms. When a service needs data from another service there are several ways to do it, and the decision for which to use is not always straightforward. Each case is a different case and there is no standard correct way to do it and determining the best approach for each scenario should be properly deliberated as it can have a significant impact on quality attributes.

Two versions of the microservices architecture were developed, only differing on the collaboration style between the services[3] [4]. As mentioned in Section 2.3.3, direct synchronous communication between the services

3 https://github.com/leandrocosta16/gama-microservices
4 https://github.com/leandrocosta16/gama-microservices-direct

should be avoided since it causes tight coupling between services. However, as mentioned in Section 2.5, this approach is still often used so it is relevant to understand if its particularities and the ease it provides overcomes its drawbacks.

5.3.1 *Event driven*

Asynchronous communication with a choreography style between the services is generally accepted to be the preferred approach of transferring information among the multiple services without conflicting with the microservices values. A publish/subscribe methodology was used and messages are published to a topic that one or more services may subscribe to. The published messages are events that correspond to actions that have been performed. Those events when consumed by the subscriber services will trigger a set of actions, portraying the behavior of a typical Saga Transaction (further detailed in Section 5.2).

For the asynchronous microservices version of the application, this collaboration style was implemented using a message broker, through the use of RabbitMQ. Although there are several implementations of a message broker, RabbitMQ was chosen mostly due to its well-documented integration with Java and because of the existent previous experience of integrating it with the AMQP protocol. Either way, any of the message brokers would perfectly suit the needs of the case studies as there are no significant differences in terms of performance or functionality in this scenario. The use of a message broker enables a great degree of independence between the services, allowing them to be completely self-contained and with a limited scope of change since interactions between the services do not happen directly. The consumer services are not even aware of which queues it is sending the messages to, which promotes isolation between the services. Furthermore, with its introduction, there is the guarantee that if a message is indeed received by the broker, then it will eventually get consumed by the subscriber services, even if offline at the moment, promoting the availability of the system.

The isolation from the failure of the upstream services comes at the cost of possibly not having an immediate response. While read operations should preferably use a request/response approach, which is true across all implementations, most of the business operations can bear being eventually completed. Regardless, despite its asynchronous nature, the processing time is not substantially different when compared with the synchronous microservices version. The differences mainly reside in the fact that while the synchronous version will fail due to request timeouts during heavy load or a service's failure, the asynchronous version in those situations will just take more time. The use of event-based communication also enables the prioritization of operations. Given that it does not require an immediate response, it is possible to associate each operation with a prioritization value and pop out of the queues the most relevant events first.

The two main benefits of using this approach essentially are that coupling between the services is mitigated and the existent fault isolation between the services. These benefits go hand-in-hand with the microservice's principles, which define the services as being independently developed and managed units.

With asynchronous communication, the producer services do not require any response from the consumer services, even if relying on others' business capabilities. Through the implementation of Saga Transactions, operations are handled as sets of independent events that are eventually consumed, contrasting with the request and response approach in which the user immediately gets status information of all their actions. Furthermore, making the inter-service communication completely event-based is a completely different paradigm that requires a new set of considerations and rethinking when compared with the development of the monolithic version. To provide to the users a notification whenever an operation has been completed, for instance, when an order has been successfully confirmed, it is done through the use of Web Sockets. To achieve that, a notification service is introduced, and it subscribes to specific events and notifies the users interested in those events.

Preferably, a given service is responsible for a restricted group of business operations that do not require interaction with other services. By modeling the microservices using a DDD approach that ideology is promoted as most of the operations did not require the span of other services, although some of them inevitably did. Moreover, some of those operations required a more sequential flow of events, in which having an eventual response would not be acceptable. As a concrete example, Figure 21 represents the flow of a user creating an order and all the subsequent events that are triggered.

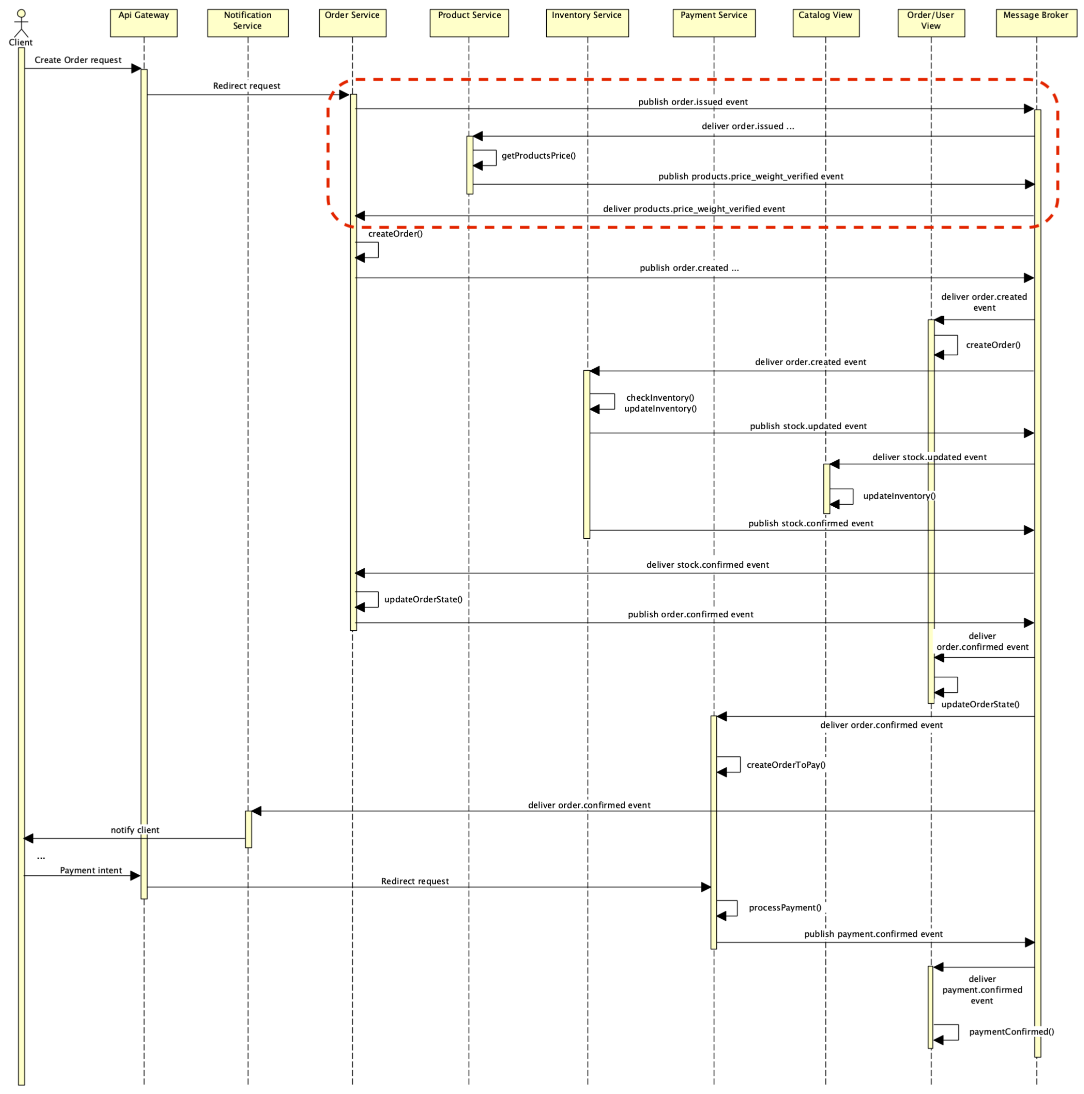

Client
Api Gateway
Notification Service
Order Service
Product Service
Inventory Service
Payment Service
Catalog View
Order/User View
Message Broker
Create Order request
Redirect request
publish order.issued event
deliver order.issued ...
getProductsPrice()
publish products.price_weight_verified event
deliver products.price_weight_verified event
createOrder()
publish order.created ...
deliver order.created event
createOrder()
deliver order.created event
checkInventory()
updateInventory()
publish stock.updated event
deliver stock.updated event
updateInventory()
publish stock.confirmed event
deliver stock.confirmed event
updateOrderState()
publish order.confirmed event
deliver order.confirmed event
updateOrderState()
deliver order.confirmed event
createOrderToPay()
deliver order.confirmed event
notify client
...
Payment intent
Redirect request
processPayment()
publish payment.confirmed event
deliver payment.confirmed event
paymentConfirmed()

Figure 21: Create order Sequence Diagram (before replicated entities)

This scenario exemplifies the complexity of an operation that in the microservices versions requires the span of several services, featuring multiple local transactions that are triggered through the consumption of events. The process of creating an order is a trivial operation in any e-commerce platform, and its efficiency is critical. In this case, between the user's click to create an order and for it to get confirmed there are numerous events consumed and published. Events typically correspond to a notification of something that has happened in the past. Using an exclusively event-driven approach, when data from other services is required, querying for data through the publication of events may not be the most efficient solution. Whenever a service needs data from other services, for instance, for the create order operations, the total products' price and shipping costs have to be calculated, and, for that, the products' price and weight are required. However, those fields exclusively resided on the Product Service at this point. Given that an event-driven approach is used, the Order Service would have to publish an event requesting the price and weight data of the products. Then, the Product Service would eventually produce another event, with the products' price and weight which would later get consumed by the Order Service. Since the price of the products should not be consumed directly from the customer's request, there is no apparent workaround besides having to query the Product Service through the use of events. The described flow is highlighted in Figure 21. All the other events from this operation are indeed notifications of something that has happened, which fit in the definition of an event.

This not ideal flow of events is generated because a lot of the data needed was present across several different services. Some authors sustain that any given service should own every data it needs to fulfill its business capabilities, so they become as autonomous as possible. Following this ideology, to achieve complete isolation from other services, the data required for creating an order was replicated inside the Order Service. An opportunity for the implementation of this approach was identified for the price and weight fetching process that initially happened through the publication of an event. This particular scenario could be improved by maintaining a replica of the product prices inside the Order Service (Figure 22) since it is data that is not frequently changed. Moreover, the price of the products is essentially only used by the Order Service, so, by aligning it with its bounded context, the price information should reside inside the Order Service.

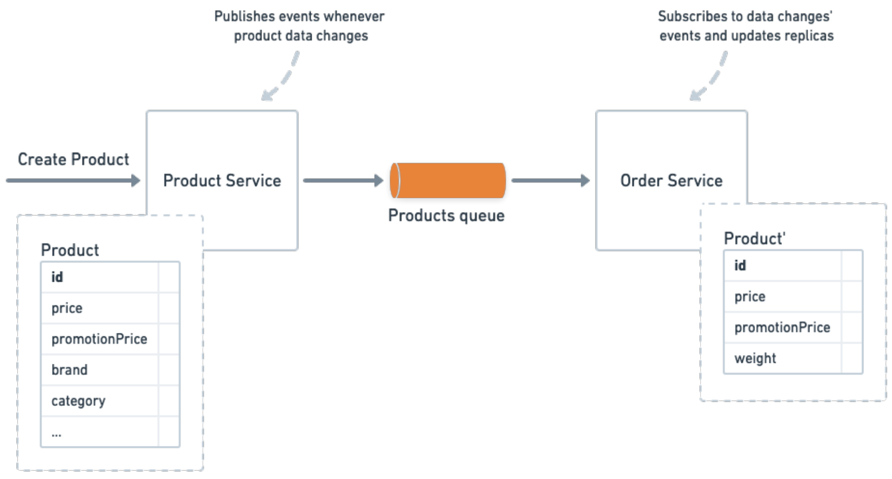


Figure 22: Example of data replication

Maintaining replicas of required data within a service is a frequently used approach when data from other services is often necessary, enabling services to handle requests while minimizing the interactions between each other. Although it may reduce the processing time of some operations, data integrity assurance becomes more a sensible topic, and mechanisms for fail tolerance, for example, if the message broker did not receive the message, need to be addressed.

Even though some cases can benefit from having replicated data, there are scenarios which deal with critical data that cannot bear the replication's synchronization overhead and eventual consistency, which is the case for the stock-related information. If the Order Service maintained replicas of inventory information it could lead to operations being made on top of yet-to-be synchronized data and, in this case, potentially confirm orders with out-of-stock products. In these sensitive situations, replication of data with eventual consistency is not an acceptable option. Additionally, the stock data is often changed, which would cause an intense flow of events being produced and consumed. In contrast, the products' information that the Order Service requires (the price, promotional price, and weight) is not as frequently updated, nor as sensitive, as stock information. Thus, having the needed product data stored inside the Order Service is a viable option that would avoid the publication and consumption of several events. Nonetheless, as a consequence of having replicas of data owned by other services, it is necessary to subscribe to a set of events that affect the replicated data in order to keep it eventually consistent. As an example, the Order Service stores basic product information such as product price, promotion price, weight, name, and id, which means it requires event handlers for products updated, deleted, and created.

The result of maintaining replicated product data is that whenever an order is created its total price can be calculated locally. The replicated product data could be out of sync, but since the price change operations are not that frequent, it can be considered an acceptable scenario with a low probability of being out of sync. Replicating large amounts of data could be inefficient, so the replicas maintained only hold the data they need, ignoring all other details. By replicating necessary data a service becomes more self-contained, avoiding the need to interact with other services.

Replication of necessary data makes each service even more independent as it aligns with DDD's bounded context principle. Some product attributes, such as the category, are part of the scope of managing products, Product Service's responsibility. While other attributes, for instance, weight, although is an attribute of a product, it is directly related to shipping and the calculation of shipping costs, which is a responsibility of the Order Service. If that information is exclusively inside the Order Service there is no inconsistency between the two services on that attribute. In this scenario, although both services have the same entity, it has different representations and purposes for each one of them, supporting a ubiquitous language.

This analysis process was conducted every time a given service needed data present in other services. For each case, it is analyzed whether it is more advantageous to maintain a replica's synchronization overhead, use a bounded context in shared entities or simply query for needed data through the publication of events on the message broker.

Either way, if data is requested through events or acquired by the use of local replicas, a flow of events takes place and it requires the maintenance of a complex communication infrastructure. Table 4 summarizes all the produced and consumed events by each service, all the interactions between them, and the maintained replicas.

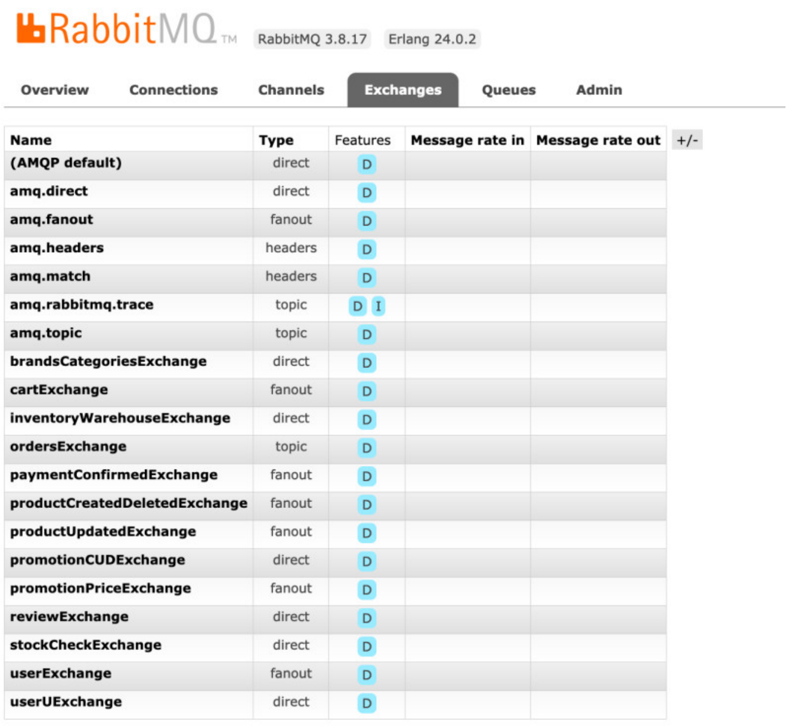
Table 4: Events produced/consumed by each service

Service	Events produced	Events consumed	Replicas/shared entities
Product Service	product.created product.deleted product.updated category.created category.deleted brand.created brand.deleted	promotion.started promotion.ended product.updated	None
Order Service	order.created order.confirmed order.rejected	product.created product.deleted product.updated payment.confirmed stock.checked	Products (required for prices information)
Inventory Service	stock.checked warehouse.created warehouse.deleted inventory.updated	product.created product.deleted order.created	Products (required to avoid maintaining inventories of nonexistent products)
Review Service	review.created review.deleted	product.deleted user.deleted	None
Shopping Cart Service	cart.itemAdded	product.deleted user.deleted	None
User Service	user.created user.deleted	None	None
Promotion Service	promotion.started promotion.ended promotion.created promotion.deleted promotion.updated	None	None
Payment Service	payment.confirmed payment.refused	order.confirmed	None
Products View	None	product.* category.* brand.* inventory.* review.* promotion.* cart.*	Products, Categories, Brands, Inventories, Reviews
Orders/Users View	None	order.* user.* payment.* address.*	Orders, Users, Payments, Addresses

The views that hold the read models mentioned in Section 5.2 are fully maintained through the subscription of events that reflect changes on the entities they hold. The asterisks present in some rows of the Events consumed column represent that every event related to the entity that precedes the asterisk is consumed by that service, which is the case of the services that hold the read models. As it is possible to observe in Table 4, there are multiple events that are of the interest of more than one service. With RabbitMQ as the message broker, messages are sent to queues and, once an event is consumed from a queue, it ceases to exist. This means that for each producer/consumer pair, a dedicated queue is required. Despite that, the messages are not sent directly to the queues, they first go through an exchange component, which makes the producer unaware of which queues the message produced is going to. This adds another level of abstraction between the consumer and producer services. These exchanges are message routing agents responsible for routing messages to the corresponding queues according to a defined routing key or, alternatively, broadcasts to every queue bound with the given exchange. There are three types of exchanges: direct, fanout, and topic, which are appropriate for different use cases, all of them used in the developed application. The selecting process of which messages are sent through which exchanges is a challenging process, but the criteria used was to create an exchange for a set of related messages that are all of the interest of certain subscriber services. Regarding the exchange types used, if only one service is interested in a set of messages, the exchange can be of type Direct. If multiple services are interested in a set of messages, with no exceptions, then the exchange can be of type Fanout, which suits the one-to-many use case. At last, if multiple services are interested in only a sub-group of related messages, a Topic exchange can be used, in which a routing key is associated with a given queue, and the messages are distributed across them, depending on the associated routing key. The more services and data replicas are present in the architecture, the more exchanges and queues it needs, and it can rapidly become complex to grasp.

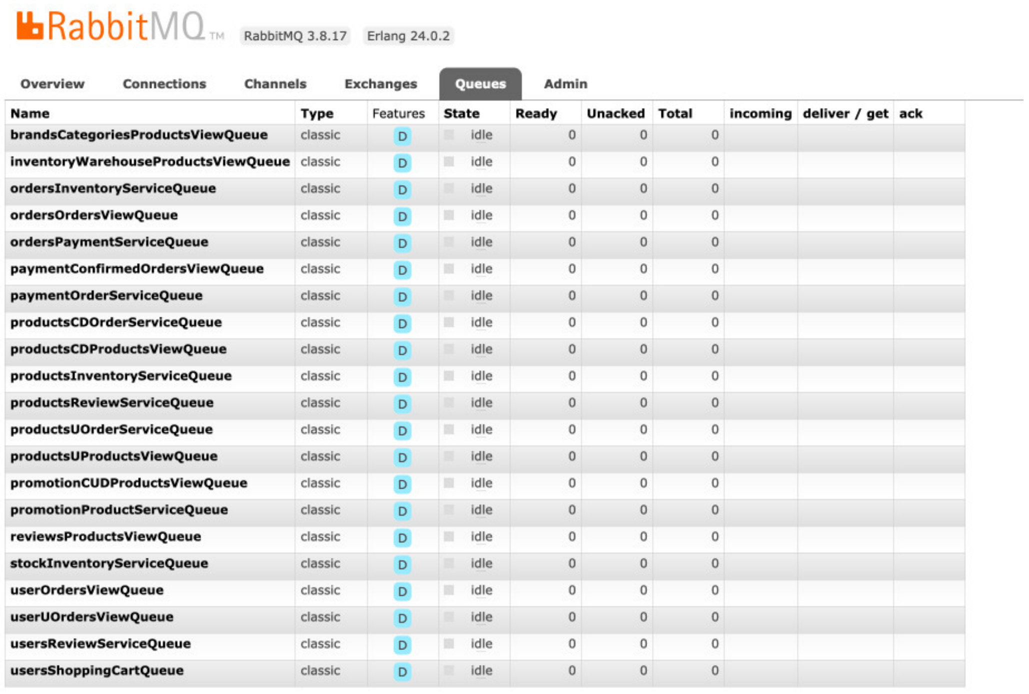
For the messages to get consumed the objects sent through the message broker had to be serialized by the producer and later deserialized by the consumer. Being Java object-oriented, the class with which the message was serialized has to be present in both services. The automatic mechanism of the RabbitMQ integration library for Java is to send the classpath as a header of the message so the consumer has all the information it needs to deserialize it. However, since both services have different project names and folder structure, the classpath is always invalid, resulting in *ClassNotFoundException* every time a message is received. To solve this, instead of sending the absolute classpath in the header, a custom token is sent. This token is associated with a given class, so when the consumer receives a message, the token is checked against its defined class and the messages get converted.

The defined exchanges and their type is represented in Figure 23, which is a screenshot from the RabbitMQ dashboard. The queues where the messages are sent over are represented in Figure 24, which is also a screenshot from the RabbitMQ dashboard.



Name	Type	Features	Message rate in	Message rate out	+/-
(AMQP default)	direct	D			
amq.direct	direct	D			
amq.fanout	fanout	D			
amq.headers	headers	D			
amq.match	headers	D			
amq.rabbitmq.trace	topic	D I			
amq.topic	topic	D			
brandsCategoriesExchange	direct	D			
cartExchange	fanout	D			
inventoryWarehouseExchange	direct	D			
ordersExchange	topic	D			
paymentConfirmedExchange	fanout	D			
productCreatedDeletedExchange	fanout	D			
productUpdatedExchange	fanout	D			
promotionCUDExchange	direct	D			
promotionPriceExchange	fanout	D			
reviewExchange	direct	D			
stockCheckExchange	direct	D			
userExchange	fanout	D			
userUExchange	direct	D			

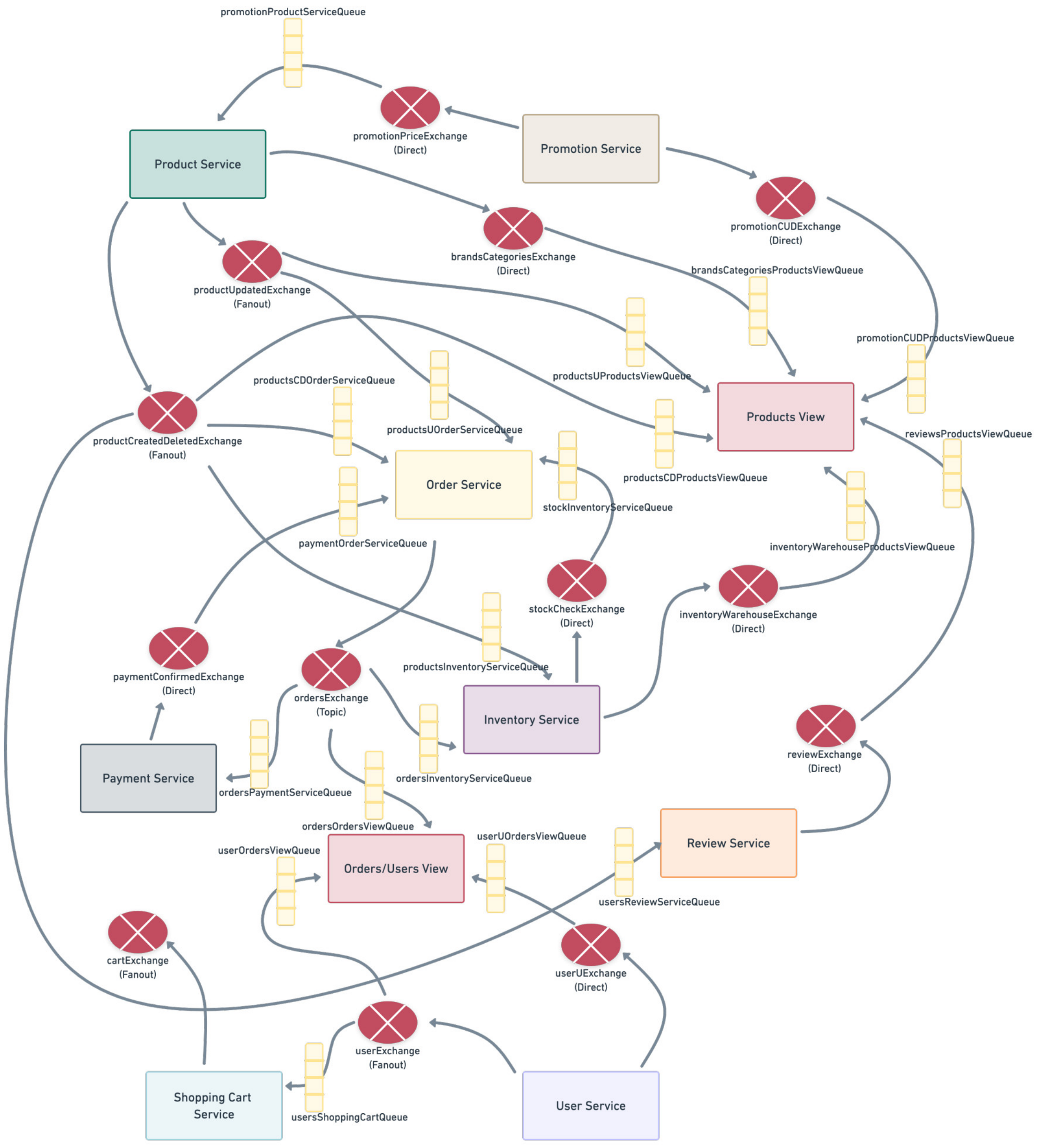
Figure 23: Defined Exchanges on RabbitMQ



Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
brandsCategoriesProductsViewQueue	classic	D	idle	0	0	0			
inventoryWarehouseProductsViewQueue	classic	D	idle	0	0	0			
ordersInventoryServiceQueue	classic	D	idle	0	0	0			
ordersOrdersViewQueue	classic	D	idle	0	0	0			
ordersPaymentServiceQueue	classic	D	idle	0	0	0			
paymentConfirmedOrdersViewQueue	classic	D	idle	0	0	0			
paymentOrderServiceQueue	classic	D	idle	0	0	0			
productsCDOrderServiceQueue	classic	D	idle	0	0	0			
productsCDProductsViewQueue	classic	D	idle	0	0	0			
productsInventoryServiceQueue	classic	D	idle	0	0	0			
productsReviewServiceQueue	classic	D	idle	0	0	0			
productsUOrderServiceQueue	classic	D	idle	0	0	0			
productsUProductsViewQueue	classic	D	idle	0	0	0			
promotionCUDProductsViewQueue	classic	D	idle	0	0	0			
promotionProductServiceQueue	classic	D	idle	0	0	0			
reviewsProductsViewQueue	classic	D	idle	0	0	0			
stockInventoryServiceQueue	classic	D	idle	0	0	0			
userOrdersViewQueue	classic	D	idle	0	0	0			
userUOrdersViewQueue	classic	D	idle	0	0	0			
usersReviewServiceQueue	classic	D	idle	0	0	0			
usersShoppingCartQueue	classic	D	idle	0	0	0			

Figure 24: Defined Queues on RabbitMQ

In order to provide a better perception of the dimension of the inter-service communication infrastructure existing within the event-driven microservices application, Figure 25 is a visual representation of all the integrating components. The horizontal rectangles correspond to the business services that consume and produce events, the red circles are the exchanges through which the services publish the events that are sent to the queues, which are represented as vertical yellow rectangles. The arrows represent the flow and direction that the events produced and consumed go through.


promotionProductServiceQueue
promotionPriceExchange (Direct)
Promotion Service
Product Service
promotionCUDExchange (Direct)
brandsCategoriesExchange (Direct)
productUpdatedExchange (Fanout)
brandsCategoriesProductsViewQueue
promotionCUDProductsViewQueue
productsCDOrderServiceQueue
productsUProductsViewQueue
productsUOrderServiceQueue
Products View
reviewsProductsViewQueue
productCreatedDeletedExchange (Fanout)
Order Service
stockInventoryServiceQueue
productsCDProductsViewQueue
paymentOrderServiceQueue
inventoryWarehouseProductsViewQueue
stockCheckExchange (Direct)
inventoryWarehouseExchange (Direct)
paymentConfirmedExchange (Direct)
ordersExchange (Topic)
productsInventoryServiceQueue
Inventory Service
reviewExchange (Direct)
Payment Service
ordersPaymentServiceQueue
ordersInventoryServiceQueue
ordersOrdersViewQueue
userUOrdersViewQueue
Review Service
userOrdersViewQueue
Orders/Users View
usersReviewServiceQueue
cartExchange (Fanout)
userUExchange (Direct)
userExchange (Fanout)
Shopping Cart Service
usersShoppingCartQueue
User Service
Figure 25: Microservices' RabbitMQ infrastructure

An overview of the several components, including the business services and read models, is represented in Figure 26. As previously stated, typical microservices components, such as a discovery service, are not represented since the used orchestration tool has an embedded DNS server and a built-in internal load balancing system. Although each service could have multiple running replicas, only a single instance of each is represented.

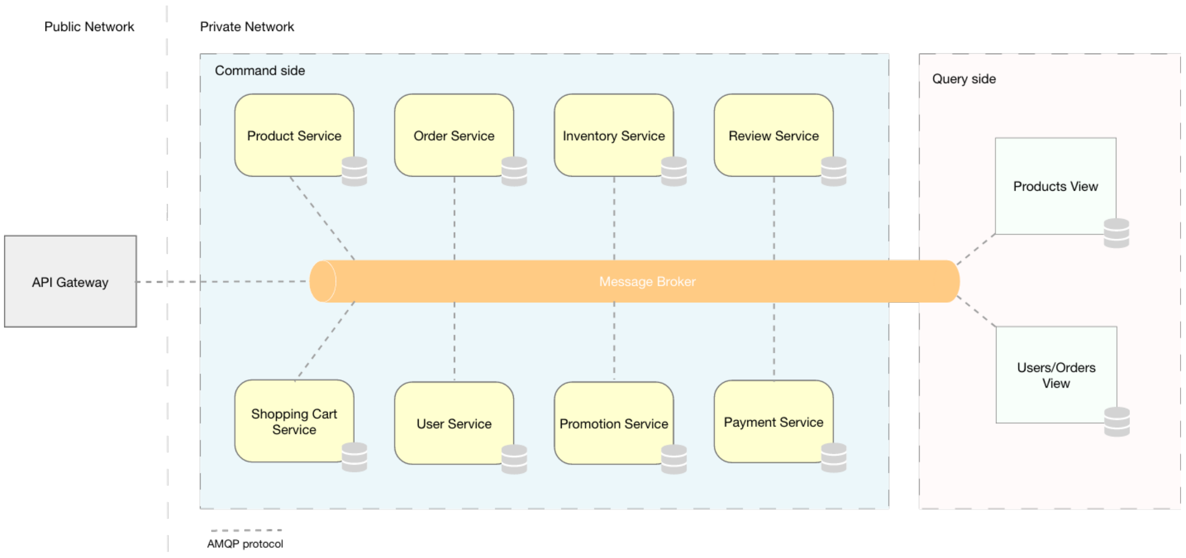


Figure 26: Overview of the components of the event-driven microservices solution

The end result of applying an event-driven approach is producer and consumer services that are completely unaware of each other, promoting loose coupling and fault isolation. The services subscribe to a specific set of events, get triggered by those events, and start processing them. All of that happens without even knowing the location of the producer services, enabling no coupling between them.

Regarding this event-driven approach strong points, if a consumer service fails, the producer service will continue operating as nothing has happened, providing resilience to the services. This degree of independence maximizes the availability of the application, which could represent a major challenge within a distributed system. When multiple components are part of the system, the higher are the chances of one of them failing, and if there is no fault isolation the system would be more likely to fail. With the use of a message broker, from the moment a message gets published to the corresponding queues, even if a consumer service is offline, there is the guarantee that when the service comes back online it will receive and process the message.

Another aspect is that if for some reason, the message broker is not able to receive a message, then the operations have to rollback in order to keep the integrity of the system. When possible, this behavior was programmed inside the Spring's ecosystem by annotating the business methods as *Transactional* with rollbacks being performed when RabbitMQ-related exceptions are thrown. If these situations are not handled it could result in a permanent state of inconsistency of the system, since the replicas have no business logic applied whatsoever. The operations performed on the replicas are just mimed from the events received by the subscribed queues.

In order to achieve independent deployability, one of the values of using microservices, the external interfaces of the services, in this case, the event handlers, should be treated as if they were public APIs. Whenever changes are made to one service it should not force changes on the other services. So, any necessary change to the format of the messages defined in the event handlers should be treated with cautiousness.

5.3.2 *Direct communication*

Another approach for inter-service communication can be implemented through the use of synchronous request-response-based inter-service communication, such as HTTP requests combined with a RESTful design. When using this communication approach a service makes requests to other services and expects a response within a defined, normally short, time period.

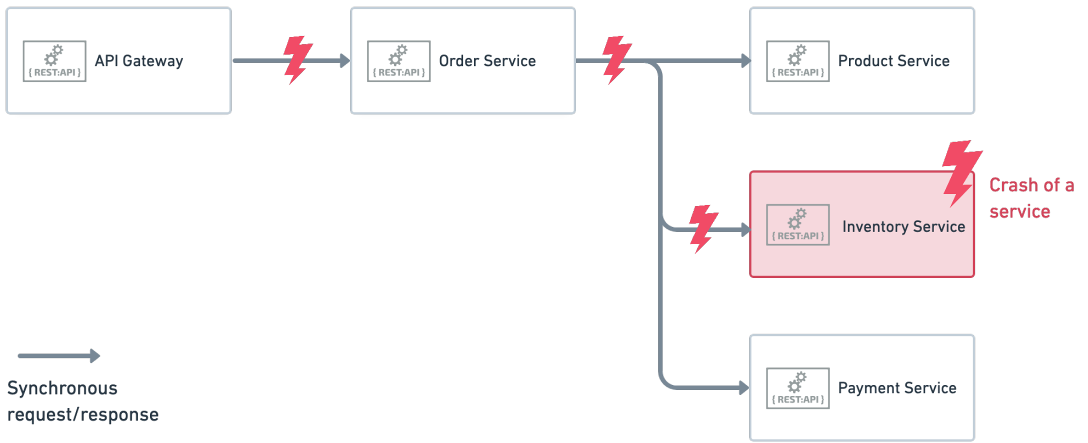
An alternative version of the same application was developed following this communication approach[5]. Each service has its REST API endpoints through which services communicate with each other synchronously, exchanging data in the JSON format.

Several existent microservices case studies make use of direct communication (Section 2.5), hinting that this approach may be used in the wild. That was the main reason behind the implementation of this version of the application, even though it does not comply with the microservices guidelines since it conflicts with the loose coupling principle.

Regarding implementation challenges, there is not much to report since it is a well-known paradigm of collaboration. All the asynchronous communication challenges mentioned in the previous section were not faced during the development of this version. The use of this inter-service communication methodology highly facilitated the development process since the infra-structure represented in Figure 25 is replaced by simple HTTP calls. Even though replicas of necessary data would also be possible with this paradigm, propagating those possibly frequent state changes using a request/response approach would not be reliable as more operations would constantly depend on the availability of other services.

The major drawback of using this approach is the tight coupling with the services' availability. Due to the request/response nature of the communication, if a requested service is unavailable or exhibits high latency, then the operation fails and potentially cascades the failure to other services. This chain of failures showcases the tight coupling between the services, which goes against the microservices principles. As an example (Figure 27), assuming the Order Service requests information from other services via HTTP requests when creating an Order, if one of the services is down or does not respond within a timely fashion, then the overall request will fail.

5 https://github.com/leandrocosta16/gama-microservices-direct



{ REST:API } API Gateway
{ REST:API } Order Service
{ REST:API } Product Service
{ REST:API } Inventory Service
{ REST:API } Payment Service
Crash of a service
Synchronous request/response

Figure 27: Example of a chain of failures using synchronous communication

With this approach replicas of frequently joined data are not maintained since forcing the data synchronization through synchronous requests could easily result in an inconsistent state of the system if a service went down. For those reasons, the CQRS pattern was not applied to this version. This implies that when, for instance, product data needs to be retrieved, the Product Service orchestrates a set of requests to all the services that own data related to a product so it can be displayed to the user. This approach is not ideal when complex querying that requires joining data from different services is needed. In the event-driven version, this would have been solved through the implementation of the CQRS, which is a non-viable solution with synchronous communication.

Although there are some workarounds to the eventual failure of a service during an operation, through the use of circuit breaker patterns, these solutions add extra effort and are not a reliable option in most cases. These patterns prevent the propagation of failures to other services by providing a default behavior when a failure is detected, enabling the requests to "fail fast". Netflix's Hystrix[6] is a library that implements this pattern by providing a fallback method that will get executed if the HTTP requests fail. The definition of a failed HTTP request can also be customized by defining custom properties, such as timeouts for calls to remote services. This pattern also prevents failed requests from accumulating and becoming a bottleneck.

Compared with the publication of events used in the asynchronous version, each event was mapped to the corresponding requests in the synchronous version, represented in Table 5. While the synchronous version directly requests a service in every distributed operation, the event-driven solution can either publish a message or use a maintained replica.

6 https://github.com/Netflix/Hystrix

Table 5: Events produced/consumed by each service

Operation	Flow of the request	Event-based approach
Inventory Check	Order Service → Inventory Service	Message publication
Get price for calculating order	Order Service → Product Service	Replica maintained
Set Promotion price	Promotion Service → Product Service	Message publication
Get order price	Payment Service → Order Service	Replica maintained
Verify if product exists	Inventory Service → Product Service	Replica maintained
Deleted product	Product Service → Inventory Service Product Service → Review Service	Message publication
Deleted User	User Service → Review Service	Message publication
Payment confirmed	Payment Service → Order Service	Message publication
Fetch a product	Product Service → Inventory Service Product Service → Review Service	Replica maintained
Fetch products of a promotion	Promotion Service → Product Service	Replica maintained

Given the nature of the requests, all of them require a real response from the upstream services since local cache or dummy values from a fallback method would not provide an adequate response. If, for instance, the inventory information was not directly requested to the inventory service and a local cache system was used instead, it would possibly cause stock problems if the stock information did not match the real values.

The rollback mechanism implemented in the event-driven implementation was also applied to this solution. The functionalities that require communication with other services will rollback all previous state changes if the HTTP request failed, so data inconsistencies are avoided. This behavior is triggered whenever a HTTP-related exception is thrown.

Although it is still considered a microservices architecture, by using a direct form of inter-service communication some of the promised benefits from adopting the architecture are invalidated. Even if the external APIs of the services are maintained as they were public, with versioning and technology agnostic, there is inevitable coupling between the services.

On the other hand, the utilization of a response/request-based communication turns the system more responsive as an operation features sequential steps with an immediate response instead of being triggered by an eventual notification. Latency is reduced due to the absence of middleware for all communications, achieving higher throughput values. Furthermore, there is less design complexity, similarly to the monolithic application, which also makes the debugging process easier. However, it often is not an ideal option and should be used only if strictly necessary due to its multiple downsides mostly related to non-functional matters, such as availability. Even if combined with "solutions" like a circuit breaker, which would prevent a cascade of failures, these workarounds do not solve much. Hence, this communication approach should be avoided, but, in fact, there are cases in which direct synchronous requests are the best option and its utilization is justifiable, even at the cost of some of the system's quality attributes. Being considered an anti-pattern highly depends on the context where it is applied, but for this e-commerce context, it does not add much value, quite the opposite.

This time without the message broker, Figure 28 represents an overview of the components and the interaction between the several services which integrate the synchronous microservices version.

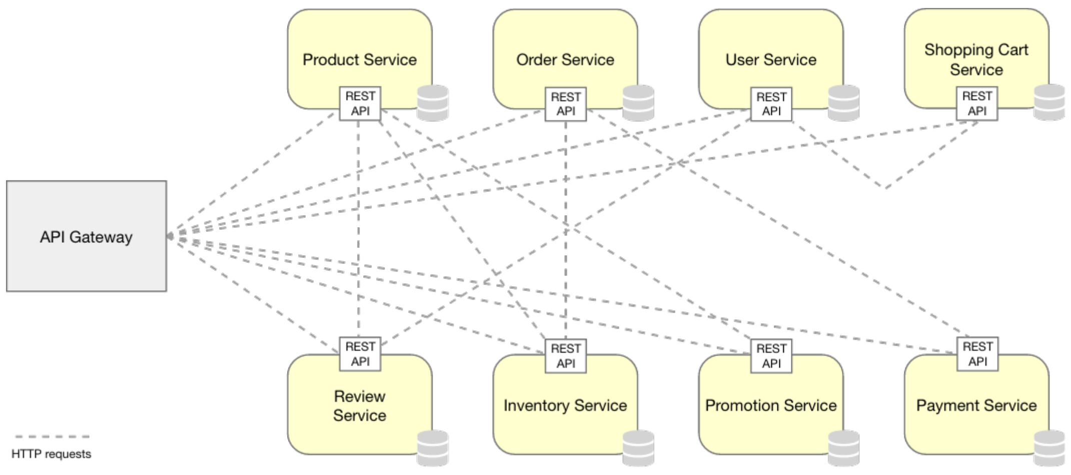


Figure 28: Overview of the components of the synchronous request/response microservices solution

6

ASSURANCE OF QUALITY ATTRIBUTES

At this point, the applications already feature all the planned functionalities, but for them to become ready to run in production there are still multiple relevant aspects to be taken care of, which are not related to functionalities, but quality attributes. The differences between monolithic applications and microservices extend to non-functional affairs, whose assurance requires a different set of methodologies. Throughout this chapter, the processes related to assuring quality attributes and robustness of the applications are evaluated, and its costs are estimated as an attempt to contribute to the decision of whether a migration to microservices is justifiable.

This chapter showcases the selected approaches and challenges associated with the assurance of the quality attributes specified in Section 4.2.

6.1 INTRODUCTION OF HIGH AVAILABILITY AND LOAD BALANCING

Microservices

Assuring high availability in the monolithic and microservices versions also required different approaches. In a microservices context, there are multiple units to assure high availability and the uptime of the system depends on the uptime of each of the individual services. Concerning the solution that uses a message broker for inter-service communication, the failure of a component does not compromise the overall system, which promotes availability. In the case of using direct communication between the services, they become more dependent on each other, since a failure of a service could result in a cascade of failures. So, resilience-wise, using a message broker as a communication intermediate is the option that offers more fault isolation. By keeping each service independent of each other availability is maximized. However, although a given service is not compromised by a failure of another, the system only provides full functionality when all services are operational.

In theory, assuring high availability within a distributed system is very challenging, but there are several tools that highly assist its introduction, namely, the used orchestration tool, Docker Swarm. If a physical host from the Swarm cluster fails, then all the services running on that machine will be restarted in another one. This self-sufficient behavior provided by Docker Swarm takes action without any human interaction, keeping the cluster constantly at a designated state. The automatic healing process highly decreases the services' downtime, since the restart process of a service should not take longer than a few seconds. The fact that a failure in one service is transparent to the others does not mean there will be no downtime. Fault isolation enables the continuity of an operation in the case of failure of an external service, but for that operation to take place, the service will have to be in an active state.

To further decrease downtime periods of a given service, replication of high demanded instances across multiple machines is introduced. Supposing one service fails and takes a few seconds to respawn, in the meantime the other replica is active and handling operations. Since Docker Swarm offers self-healing mechanisms, the addition of replicas is mainly for parallel processing purposes to increase the system's performance. Nonetheless, if only one instance of a service is running and it fails, that service will have a short downtime period, which corresponds to the time taken to restart the service in another machine. Wherefore, the introduction of replicas of the stateless services will also potentially decrease downtime periods.

The goal was to introduce high availability to all the components but to do it for the API Gateway the process is not as straightforward as the rest of the services. Docker Swarm's official documentation features an example of the usage of an API Gateway, and it is suggested that it is an external component to the Swarm cluster. By being external to the Docker network, every service would have to expose a port on the host machine so the external API Gateway could communicate, which is not a recommended approach security-wise.

Regardless, given that the API Gateway is the single entry point for the whole infrastructure, an effort was put into assuring it does not represent a Single Point Of Failure (SPOF). For these entry points, the typical approach to assure high availability would be to have two or more instances behind a virtual IP address, with only one working instance at a time. However, since it is the component that routes all the incoming requests, if only one instance is used, it could turn into a bottleneck. If more than one instance is active at a time a load balancer with an externally exposed IP address would have to be introduced.

However, Docker Swarm offers a routing mesh capability that declares that if a given service has an externally published port, that service can be reached from any of the machines composing the cluster. This implies that, even if the targeted service is not running on the requested machine (Figure 29), the request will automatically be redirected to a machine that holds the service. Consequently, this means that the API Gateway can be part of the cluster and can have multiple running instances since the routing mesh capability also load balances the requests across all the active API Gateway containers.

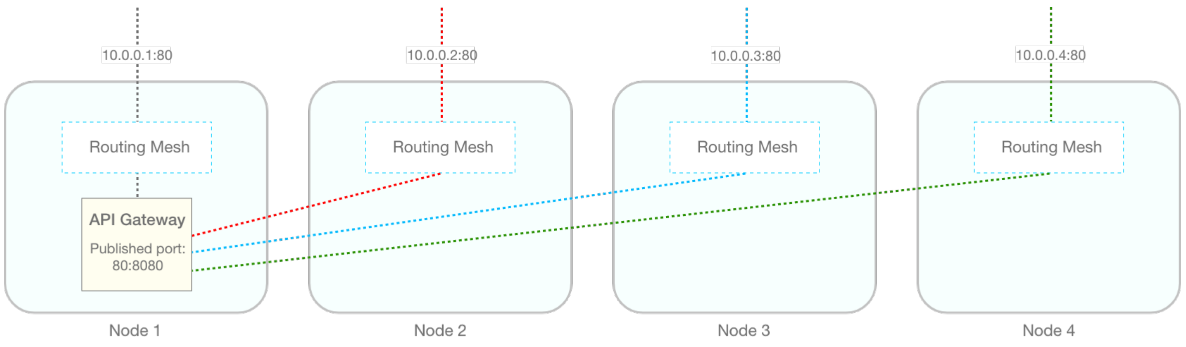


Figure 29: Routing Mesh capability of Docker Swarm

On top of that, the other services do not need to be externally exposed as the Docker Swarm environment features an internal DNS server that automatically assigns each service in the Swarm a DNS entry. As a result, the only exposed service is the API Gateway, which can have multiple running instances as easily as any of the other services.

Regarding the internal communication inside the Docker Swarm environment, all services are reachable by hostname thanks to the internal DNS server, which nullifies the need for a discovery service, as mentioned previously. Regardless of how many instances of the same service are running or in which machines they are running, reaching a service is as simple as using its defined hostname inside the Swarm environment. Given that there could be more than one instance of the same service, load balancing is necessary to distribute the load across multiple instances for better resource exploitation and performance boost. Having that said, if multiple tasks (running instances of the same service) are active, Docker Swarm's internal load balancing distributes requests among the services within the cluster based upon the DNS entry of the service. The use of this orchestration tool enables automatic internal and external load balancing, which otherwise would force the introduction of dedicated load balancers and a discovery service.

In a multi-machine cluster, although any node can be used to reach the API Gateway, some of them can be offline at a given time so, a specific machine's IP address should not be used. Instead, a virtual IP address that represents all of the machines that are part of the cluster was added. It guarantees that if a machine fails, the same IP address can be used interchangeably. This behavior is achieved by using the Keepalived[1] project, which is present in all the machines and constantly checks if any of them has failed, so the requests are not redirected to a faulty instance.

All these mechanisms work together to promote high availability on the whole system by adding redundancy, self-healing, load balancing, health checks, and the use of virtual IP addresses. This way, the downtime periods coming from failures are mitigated, the architecture is fault-tolerant, and automatically recovers from eventual failures.

Monolith

In order to promote high availability on the monolithic version, a different approach was used, since most of the high availability mechanisms in the microservices version were facilitated by the orchestration tool. Although it would be possible to use such a tool for the monolithic application, containers are often used for more granular components, not huge monolithic applications. Following a more traditional approach, virtual machines were used for each of the components of the monolithic version.

For the stateless components, the approach used was to add redundancy to the only existing instance and put them behind a load balancer, based on Linux Virtual Server (LVS). The load balancer itself also needed a backup instance so in the event of one of them failing the application still operates as expected. Health checks are continuously made between the load balancers so the backup one can take over when a failure is detected. The replication of instances prevents the existence of SPOFs. In the case of the LVS, being it the single entry point for the application, a single IP address that identifies both instances regardless of which is currently running is necessary. For that purpose, a virtual IP was introduced by using the Keepalived project.

1 https://github.com/acassen/keepalived

The ease of introducing redundancy on the application resides in the fact that it was developed targeting a stateless nature so it could facilitate the replication process. If the application had a state, for instance, by maintaining server-side user sessions, then only one instance could be running at a time, and the state would somehow have to be shared across the used virtual machines.

6.1.1 *Stateful Services*

Microservices

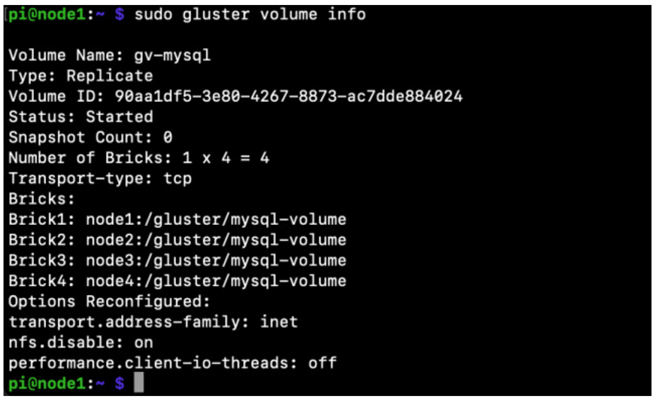
Adding redundancy to stateless services is relatively trivial as no previous state is required to resume operation, which is the case of the developed business services. When it comes to stateful services, such as databases, a previous state must be present during initialization. However, databases are not recommended to be set on containers for production. The typical approach is to use cloud-managed databases since it normally holds critical data which would be challenging to reliably operate using a container. Although, if for some reason the database must be part of the cluster, the approach would be the same as any other component with state.

The challenges associated with stateful services is that, for instance, in the event of a database being hosted in a machine that suddenly failed, if data is only stored locally on that machine then if the database service restarts on another one they would be empty. Stateful services require only one running instance at a time, but the architecture should be able to continue operation in the case of failure. The self-healing mechanism provided by Docker Swarm enables high availability of these components by immediately starting new instances on available machines. The only concern to be addressed to assure continuity of operations after a failure is the state that needs to be available across all machines.

Orchestration tools were originally designed to handle stateless containers that are meant to be terminated and initialized at any point. However, there are workarounds for enabling state persistence on the host machine's file system. The default location for every container's related data is completely managed by Docker, through the creation of Docker volumes, not accessible when the container is not at a running state. Despite that, there is a sub-type of Docker volumes that enables the specification of a custom path, anywhere on the host machine's file system for the container's storage, the bind mounts. When bind mounts are used, a file or directory on the host machine is mounted into a container, so even when it is not running, all its data is still accessible.

However, the most desired outcome of bind mounts is that since the state of the containers is on the host machine's file system, it enables the introduction of a distributed file system, for example, GlusterFS, NFS, or DRBD. GlusterFS[2] was the one used and it was installed on each machine that composes the cluster. This tool enables the synchronization of data between the nodes by defining a disk partition on each of them that is constantly in sync. As a result, the Gluster volume is mounted as a Docker volume, using bind mounts. This assures that, at a given time, all disks should contain the same data. The used volumes are represented in Figure 30.

2 https://www.gluster.org/



```
pi@node1:~ $ sudo gluster volume info

Volume Name: gv-mysql
Type: Replicate
Volume ID: 90aa1df5-3e80-4267-8873-ac7dde884024
Status: Started
Snapshot Count: 0
Number of Bricks: 1 x 4 = 4
Transport-type: tcp
Bricks:
Brick1: node1:/gluster/mysql-volume
Brick2: node2:/gluster/mysql-volume
Brick3: node3:/gluster/mysql-volume
Brick4: node4:/gluster/mysql-volume
Options Reconfigured:
transport.address-family: inet
nfs.disable: on
performance.client-io-threads: off
pi@node1:~ $
```

Figure 30: Gluster volume

If the state from the stateful services is stored on the partition that is being synchronized by GlusterFS, then any service can start in any of the machines because the state is persisted across all of them (Figure 31). When a stateful service starts, it binds with the shared partition and any change of state will be done directly on that partition, instead of the default Docker-managed filesystem. Then, GlusterFS propagates the changes to the other partitions in the other nodes and assures that, whenever and wherever a stateful container initializes, it will have access to the up-to-date data.

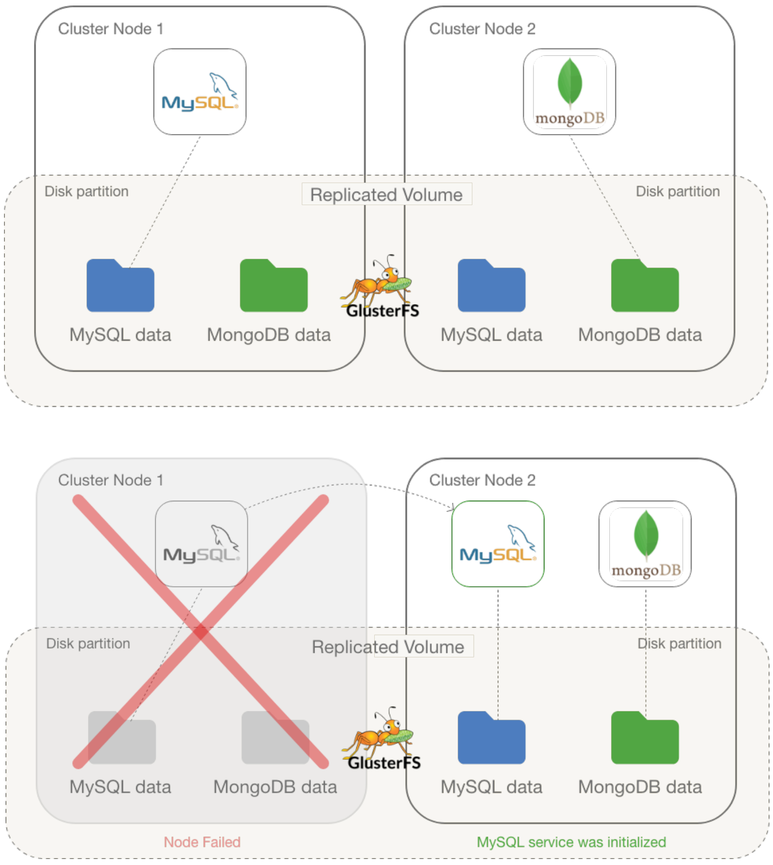


Figure 31: Failure of a statefull service

This technique was used to keep the database data available across all machines, so even if the ones responsible for hosting the stateful services fail, they will be restarted seamlessly.

The constant synchronization of the data causes an increase in the I/O operations and CPU usage between all machines. Nonetheless, the goal was to keep all the services present in the Docker Swarm cluster, including the database, even though in a real scenario it should be cloud-based.

Monolith

The typical techniques to implement high availability to a traditional system are mostly based on hardware redundancy, clustering, and replication of data physically or at a software level. For the monolithic solution, in order to introduce high availability to the database, the only stateful, a high availability cluster was created, whose data is shared using a DFS.

Similar to the microservices version, a DFS is required in order to make the state available across the instances part of the architecture. Since there is hardware redundancy to assure high availability in both versions, a DFS is not something that only applies to microservices architectures, as it was also necessary for the monolithic version.

Two virtual machines were used to form the cluster that will introduce redundancy to both the database system and its data, so it does not become a SPOF. Corosync[3] and Pacemaker[4] were used in order to manage the cluster and provide the messaging capabilities. For the cluster three resources were defined (Figure 32): a floating IP, which provides a virtual IP that always identifies the running instance of the service, the MySQL service, and the data, which is shared across the machines using GlusterFS, similarly to what was implemented in the microservices solution.

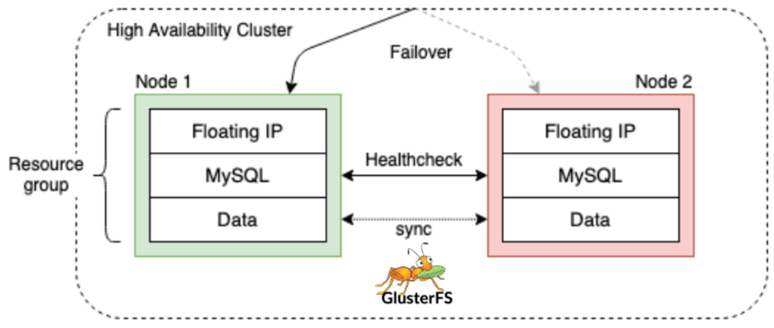


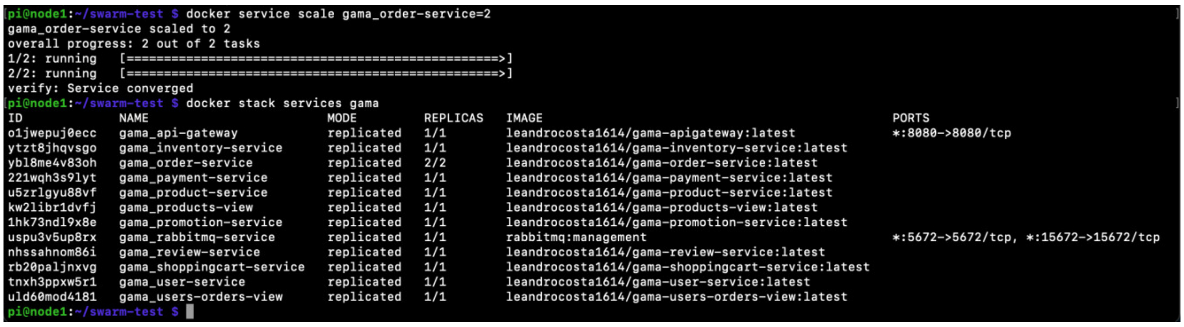
Figure 32: High availability cluster for the database for the monolithic version

Similar to the self-healing mechanism provided by Docker Swarm, this cluster introduces failover which means that if the resource group fails, there is a second node that takes over. Once the backup resource group initializes it becomes associated with the floating IP and MySQL is initialized over the synchronized data. With the high availability cluster, high availability is assured for the database of the application.

3 http://corosync.github.io/corosync/
4 https://github.com/ClusterLabs/pacemaker

6.2 SCALABILITY

Although the replication of the stateless components furthers high availability, it also potentially improves performance and the number of simultaneous clients supported. One of the most desired outcomes of adopting microservices is the flexibility when scaling the application, as mentioned in several surveys (see Section 2.5). Scaling the number of instances within a Docker Swarm cluster is as simple as typing one command (see Figure 33). Docker Swarm handles the creation and initialization of new instances of existing services, which are placed in the machine with the lowest load. Similarly as simple, it is possible to remove the replicated instances from the cluster. In this e-commerce context, when promotions are created, the Products View service will likely get saturated, and having the ability to specifically scale that service without having to scale the entire application provides a great degree of flexibility that is non-existent in traditional applications.

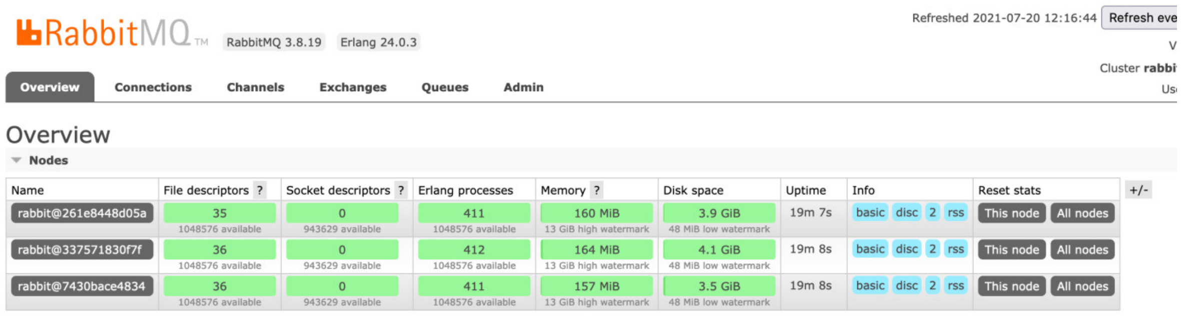


```
pi@node1:~/swarm-test $ docker service scale gama_order-service=2
gama_order-service scaled to 2
overall progress: 2 out of 2 tasks
1/2: running   [==================================================>]
2/2: running   [==================================================>]
verify: Service converged
pi@node1:~/swarm-test $ docker stack services gama
ID             NAME                        MODE         REPLICAS   IMAGE                                               PORTS
o1jwepuj0ecc   gama_api-gateway            replicated   1/1        leandrocosta1614/gama-apigateway:latest             *:8080->8080/tcp
ytzt8jhqvsgo   gama_inventory-service      replicated   1/1        leandrocosta1614/gama-inventory-service:latest
ybl8me4v83oh   gama_order-service          replicated   2/2        leandrocosta1614/gama-order-service:latest
221wqh3s9lyt   gama_payment-service        replicated   1/1        leandrocosta1614/gama-payment-service:latest
u5zrlgyu88vf   gama_product-service        replicated   1/1        leandrocosta1614/gama-product-service:latest
kw2libr1dvfj   gama_products-view          replicated   1/1        leandrocosta1614/gama-products-view:latest
1hk73ndl9x8e   gama_promotion-service      replicated   1/1        leandrocosta1614/gama-promotion-service:latest
uspu3v5up8rx   gama_rabbitmq-service       replicated   1/1        rabbitmq:management                                 *:5672->5672/tcp, *:15672->15672/tcp
nhssahnom86i   gama_review-service         replicated   1/1        leandrocosta1614/gama-review-service:latest
rb20paljnxvg   gama_shoppingcart-service   replicated   1/1        leandrocosta1614/gama-shoppingcart-service:latest
tnxh3ppxw5r1   gama_user-service           replicated   1/1        leandrocosta1614/gama-user-service:latest
uld60mod4181   gama_users-orders-view      replicated   1/1        leandrocosta1614/gama-users-orders-view:latest
pi@node1:~/swarm-test $
```

Figure 33: Process of scaling one service

The stateful services, in this case, the databases and the message broker, already offer a cluster mode that provides the ability to have more than one running instance at a time and also deals with the replication, at the application level. By using a DFS instead, it only assures that if a stateful service fails in one machine, its data will be present wherever it initializes next. On the other hand, with a cluster mode, while it also deals with data synchronization, it provides the ability to have multiple running instances at a time, improving performance. For instance, a RabbitMQ cluster manages the data mirroring between the instances of the cluster and distributes the queries across them. If a message is sent to one of the instances of the cluster, even if the specific queue is not present in that instance, RabbitMQ handles the redirection to the one that owns the queue. A RabbitMQ cluster composed of 3 instances was created and not only does it assure high availability but also allows parallel processing from having multiple running instances, each responsible for a set of queues.

The GlusterFS approach described in the previous subsection is an alternative and would be necessary if the stateful services did not offer a cluster mode. By using a DFS high availability would still be assured to the system, but it does not enable the existence of multiple stateful instances running at a time, limiting scalability and performance.



Overview

Nodes

Name	File descriptors ?	Socket descriptors ?	Erlang processes	Memory ?	Disk space	Uptime	Info	Reset stats
rabbit@261e8448d05a	35 1048576 available	0 943629 available	411 1048576 available	160 MiB 13 GiB high watermark	3.9 GiB 48 MiB low watermark	19m 7s	basic disc 2 rss	This node All nodes
rabbit@337571830f7f	36 1048576 available	0 943629 available	412 1048576 available	164 MiB 13 GiB high watermark	4.1 GiB 48 MiB low watermark	19m 8s	basic disc 2 rss	This node All nodes
rabbit@7430bace4834	36 1048576 available	0 943629 available	411 1048576 available	157 MiB 13 GiB high watermark	3.5 GiB 48 MiB low watermark	19m 8s	basic disc 2 rss	This node All nodes

Figure 34: RabbitMQ Cluster

Regarding automatic scaling, if it is detected that a given service is under considerable load, the automatic creation of new instances is easily integrated with Docker Swarm but external plugins are required. However, with other orchestration tools, that is possible out of the box and is very easily implemented.

Scalability is the perfect candidate to showcase the capabilities offered by an orchestration tool. If such a tool was not used, manually replicating the multiple services and adding load balancers to each one of them would be very inefficient and not practical. Arguably, some of the potentially most challenging processes of maintaining an architecture with the characteristic of microservices are almost fully delegated to orchestration tools. Without them, microservices would be a much less viable option due to the arduous path of assuring high availability and service-specific scalability.

Regarding scalability in the monolithic application, in Section 6.1 it was mentioned that the application was replicated for high availability purposes. Nonetheless, it is possible to have as many instances as desired, but there is no flexibility in scaling only parts of the application. The only approach available is to scale the entire application, which can be an inefficient way of scalability.

If it is intended to replicate the monolithic application even further, a discovery service could be added so the load balancer would be able to dynamically distribute the requests to new instances without needing to manually add or remove new addresses. Despite efforts, there are limited options to scale monolithic solutions, although scaling the entirety of the application can in some cases be enough for smaller applications.

6.3 OBSERVABILITY

Monitoring applied to single process monolithic applications fundamentally resides in observing the state of a single instance, which essentially can be either up or down. Combined with observability techniques, which provide insights from the behavior of the system, it is possible to identify the cause of an issue and even make a correspondence with the monitoring metrics. Even though after applying high availability methodologies more components were introduced to the monolithic solution, there are still a small number of instances to monitor. In some cases, manually keeping track of the health of a monolithic infrastructure can be reasonable, since tracing errors is a straightforward process due to everything happening within one same process. The log files of a monolithic application are the single source of truth, and the only place needed to check what was the cause of an issue. In these scenarios, the approach is to collect logging data and performance metrics so it is possible to trace what happened when something goes wrong and why it happened.

In contrast, observability in a microservices context is a crucial concern that may have a great impact on the availability of the system. Its distributed nature constrains the ability to have a universal view of the overall infrastructure. Each participant service has its own state and a set of logs and metrics to observe, and its collection is fundamental to infer the state of the system as a whole. In the event of something failing, in order to track what happened, the interactions between the services must be monitored. Since microservices architectures are composed of constantly moving parts due to the volatile nature of the containers in a cluster environment, keeping track of the multiple services can be very challenging. Moreover, it is hard to debug a distributed system since operations propagate over many different services, which is not traceable by reading through a single service's log file.

Observability was promoted through the implementation of dedicated components for distributed tracing, centralized logging, and monitoring, which are described in this section.

6.3.1 *Distributed Tracing*

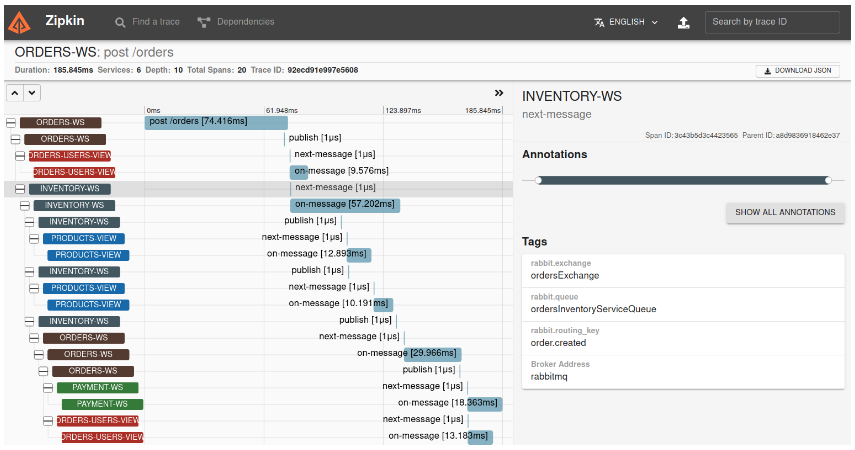
In the context of an operation that spans multiple services, troubleshooting problems is highly assisted by a distributed tracing system. It provides insights into the path taken by the operations part of business use cases that invoke multiple services, which can help identify what is the cause of problems.

A distributed tracing system is something that needs to be defined at an application level. The services implemented were developed using the Java Spring ecosystem, which can be integrated with Spring Cloud Sleuth[5], an auto-configuration library for distributed tracing. The tool adds span IDs and trace IDs to the service calls logs so it is possible to reconstruct the path taken by a given operation.

In order to aggregate the tracing data from all services in one same place for an overall visualization, Zipkin[6] was used, which manages the collected data and provides a search interface. The outputs from service calls, after Sleuth adds the necessary metadata, are sent over to the Zipkin server via HTTP. Inside Zipkin's UI it is possible to visualize the several spanned components and extra metrics such as the time spent in each service, which could be useful to identify potential performance bottlenecks (see Figure 35).

When integrated with Zipkin, Sleuth sends over tracing information of service calls made using RestTemplate, requests received by Spring MVC controllers, and even messages sent to message brokers, such as RabbitMQ. Both the event-driven and direct communication versions of the microservices applications are fully traceable using these tools.

5 https://spring.io/projects/spring-cloud-sleuth
6 https://zipkin.io/



ORDERS-WS: post /orders

Duration: 185.845ms Services: 6 Depth: 10 Total Spans: 20 Trace ID: 92ecd91e997e5608

0ms | 61.948ms | 123.897ms | 185.845ms

ORDERS-WS	post /orders [74.416ms]
ORDERS-WS	publish [1μs]
ORDERS-USERS-VIEW	next-message [1μs]
ORDERS-USERS-VIEW	on-message [9.576ms]
INVENTORY-WS	next-message [1μs]
INVENTORY-WS	on-message [57.202ms]
INVENTORY-WS	publish [1μs]
PRODUCTS-VIEW	next-message [1μs]
PRODUCTS-VIEW	on-message [12.893ms]
INVENTORY-WS	publish [1μs]
PRODUCTS-VIEW	next-message [1μs]
PRODUCTS-VIEW	on-message [10.191ms]
INVENTORY-WS	publish [1μs]
ORDERS-WS	next-message [1μs]
ORDERS-WS	on-message [29.966ms]
ORDERS-WS	publish [1μs]
PAYMENT-WS	next-message [1μs]
PAYMENT-WS	on-message [18.363ms]
ORDERS-USERS-VIEW	next-message [1μs]
ORDERS-USERS-VIEW	on-message [13.183ms]

INVENTORY-WS
next-message

Span ID: 3c43b5d3c4423565 Parent ID: a8d9836918462e37

Annotations

Tags

rabbit.exchange	ordersExchange
rabbit.queue	ordersInventoryServiceQueue
rabbit.routing_key	order.created
Broker Address	rabbitmq

Figure 35: Trace of a request viewed on Zipkin's dashboard

Traceability is something fundamental for microservices and these tools highly assist its assurance. The use of a distributed tracing tool is not only useful for troubleshooting when something goes wrong, but it can also suggest what needs to be improved in an application. For instance, the analysis of which operations are taking the longest time to process can help to identify performance bottlenecks that may have room for improvement.

6.3.2 *Centralized Logging*

Whereas the tracing of the operations provides insights into which service or function something has gone wrong, the logging enables the identification of the exact reason for the problem. The tracing was automatically generated by Sleuth but the logging is defined by the developer. All the exceptions in the application are handled and logs are issued so it is possible to track what went wrong on each service. Essentially, the intention was to provide logs only with direct and useful information that could assist the debugging process.

To monitor all the logs from the different services in a centralized way, the ELK stack was used, composed of Elastic Search, Logstash, and Kibana. To integrate the services with the ELK stack Logback was used as the logging framework for each Java-based service, which formats the logs using JSON so they can be easily stored and retrieved from the Elastic Search data-store. The Logstash service is required in each node of the cluster because it is responsible for collecting the container's logs, aggregating them, and sending them to Elastic Search. For that purpose, it was necessary to change Docker's default logging driver inside configuration files so logs are directly pushed to Logstash. Inside Kibana, which exposes the data present in Elastic Search, it is possible to visualize the logs from all the services and apply multiple filters to the queried data.

6.3.3 *Monitoring*

In order to monitor both the containers and physical machines, a monitoring stack was introduced. Monitoring hardware metrics can provide information about the usage of the system, namely, the CPU and memory usage, I/O operations, network latency, and more.

Docker Swarm does not offer built-in monitoring tools so open-source solutions were used. Each node composing the cluster has a node exporter[7] service which is a metrics collector from the host machine, and cAdvisor[8], a service responsible for exporting the containers metrics. Both those services gather and send the metrics collected to Prometheus[9], the elected database for storing the metrics. The UI layer is provided by Grafana[10], which was integrated with Prometheus, being able to present all the metrics collected.

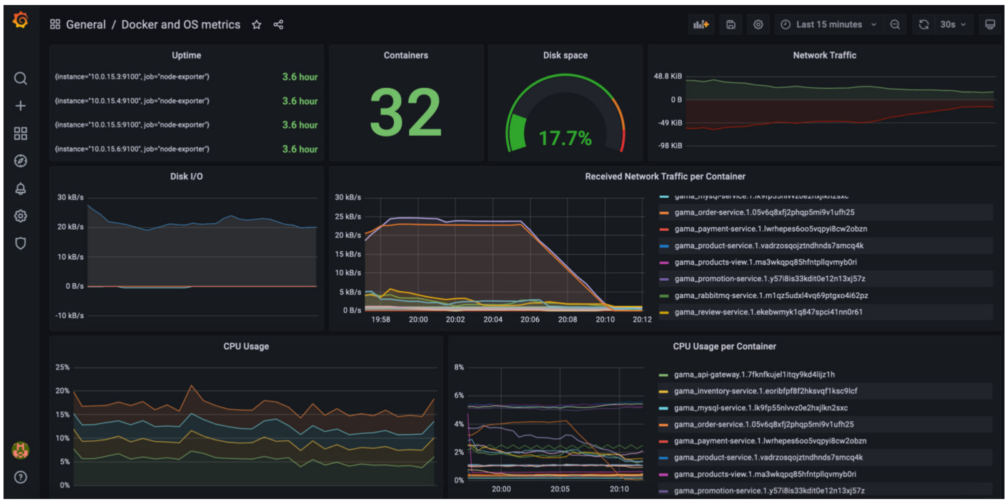

General / Docker and OS metrics
Last 15 minutes
30s
Uptime
{instance="10.0.15.3:9100", job="node-exporter"} 3.6 hour
{instance="10.0.15.4:9100", job="node-exporter"} 3.6 hour
{instance="10.0.15.5:9100", job="node-exporter"} 3.6 hour
{instance="10.0.15.6:9100", job="node-exporter"} 3.6 hour
Containers
32
Disk space
17.7%
Network Traffic
Disk I/O
Received Network Traffic per Container
gama_mysql-service.1.lk9fp55nlvvz0e2hxjlkn2sxc
gama_order-service.1.05v6q8xfj2phqp5mi9v1ufh25
gama_payment-service.1.lwrhepes6oo5vqpyi8cw2obzn
gama_product-service.1.vadrzosqojztndhnds7smcq4k
gama_products-view.1.ma3wkqpq85hfntpllqvmyb0ri
gama_promotion-service.1.y57i8is33kdit0e12n13xj57z
gama_rabbitmq-service.1.m1qz5udxl4vq69ptgxo4i62pz
gama_review-service.1.ekebwmyk1q847spci41nn0r61
CPU Usage
CPU Usage per Container
gama_api-gateway.1.7fknfkujel1itqy9kd4lijz1h
gama_inventory-service.1.eoribfpf8f2hksvqf1ksc9lcf
gama_mysql-service.1.lk9fp55nlvvz0e2hxjlkn2sxc
gama_order-service.1.05v6q8xfj2phqp5mi9v1ufh25
gama_payment-service.1.lwrhepes6oo5vqpyi8cw2obzn
gama_product-service.1.vadrzosqojztndhnds7smcq4k
gama_products-view.1.ma3wkqpq85hfntpllqvmyb0ri
gama_promotion-service.1.y57i8is33kdit0e12n13xj57z

Figure 36: Grafana dashboard with metrics from the deployed system

Additionally, it is possible to link the monitoring stack to platforms such as Slack, in which alerts are sent whenever metrics reach critical values so that engineers can take actions early.

The monitoring stack introduced the ability to constantly track how the system is behaving, what are the possible performance bottlenecks, and which services are struggling at a given time. By analyzing the collected metrics several conclusions can be taken and it is possible to improve the system based on that analysis. The percentage of CPU usage of a node and its containers can suggest that a given service may benefit from being scaled to other nodes to distribute the load.

6.4 FINAL ARCHITECTURE

Microservices

The end result of the application of all discussed components and patterns is represented in Figure 37, which aims to provide a high-level overview of the integrating components that compose the infrastructure.

7 https://github.com/prometheus/node_exporter
8 https://github.com/google/cadvisor
9 https://prometheus.io/
10 https://grafana.com/

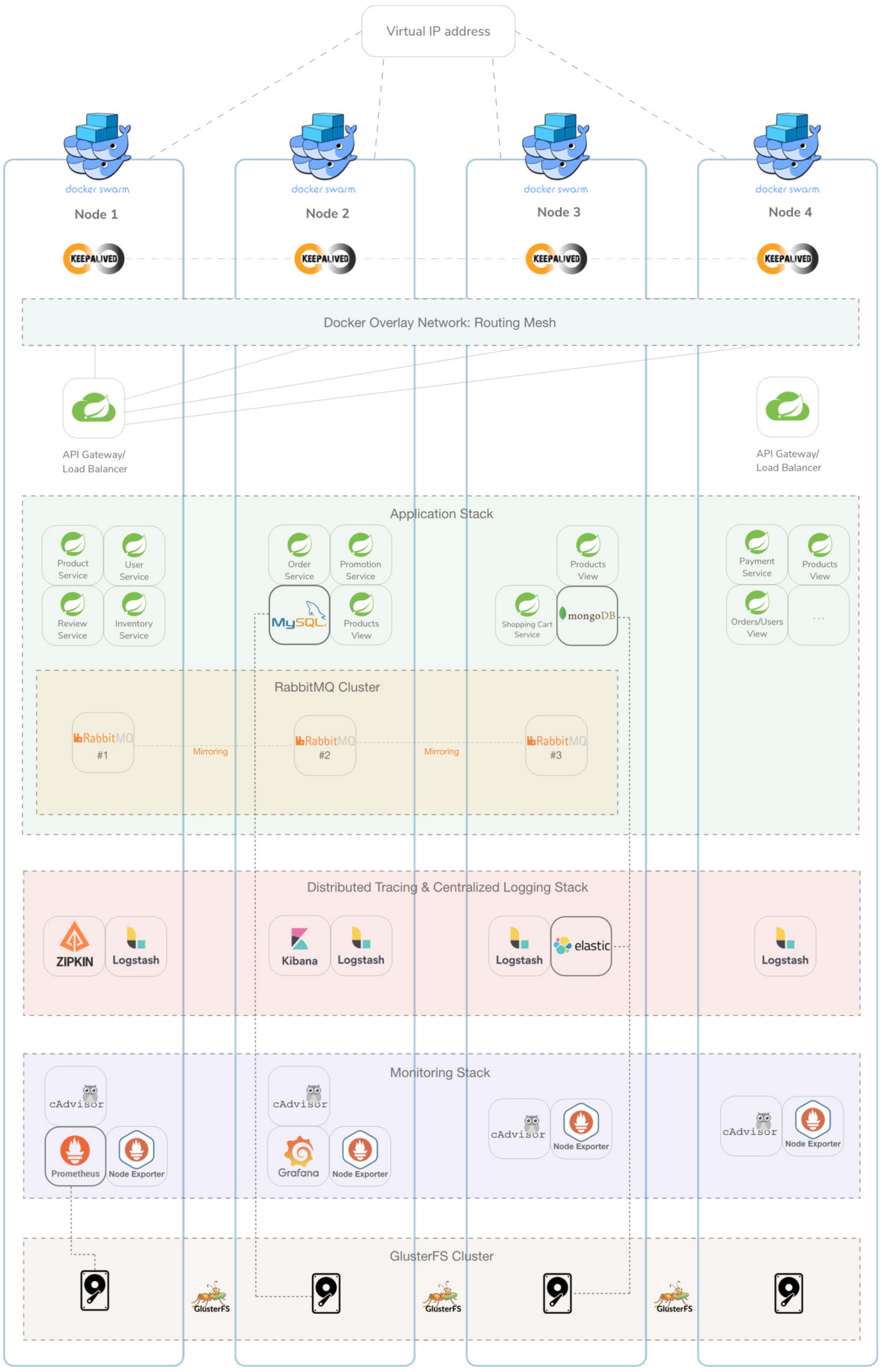


Figure 37: Final architecture of the event-based microservices solution

A four-node Docker Swarm cluster was used to integrate the several components which are grouped in stacks, a concept maintained by Docker Swarm. In Figure 37 each node is represented as a vertical rectangle and the Docker stacks as horizontal rectangles. Each stack consists of multiple related services that are initialized together with a single command. From the use of a virtual IP to identify the cluster as if it was a single instance, to the implementation of a DFS, all development processes came down to this final architecture composed of multiple services that cooperate as a unified system.

Apart from some services that require one running instance per node, all the others are distributed across the nodes by Docker Swarm, so the placement of the services illustrated in Figure 37 is just a possible distribution. As stated previously, Docker Swarm provides the agility to easily scale each one of the services, so all the business services from the main application stack can have multiple replicas, according to the needs.

The stateful components are connected to the GlusterFS managed volume, which illustrates that those components require the sharing of data across the nodes, as described in Section 6.1.1. The GlusterFS volume is mounted upon initialization of the physical machine, so the stateful services have access to the data whenever and in whichever machine they are initialized on.

Two out of the three defined stacks are exclusively related to non-functional matters and were introduced in order to promote the quality attributes of the system. The used tools for observability are containerized versions of the ones described in Section 6.3 and are part of the *Distributed Tracing & Centralized Logging* and *Monitoring Stack*. Since leveraging the costs of developing a robust fail tolerant application using microservices is part of the case study, the process of implementing it was conducted.

The *Application Stack* is composed of the developed services for the e-commerce platform, which are responsible for handling all the business operations. Still concerning this stack, there are also the database containers, which are stateful components, so they were defined with bind mount volumes mounted by GlusterFS. For the message broker, a cluster of three RabbitMQ instances was implemented and their data is synchronized at an application level, without needing to use a GlusterFS volume.

The *Monitoring Stack* possesses a global service, node exporter, which has to be part of each node of the cluster so it is able to collect the hardware and container metrics. Similarly, the Logstash service is also a global service that needs to integrate all of the nodes as it collects the logs from all the services.

For the direct communication version, the overview of the architecture is exactly the same as what is represented in Figure 37, except that RabbitMQ was not present. All the observability tools used work the same way for both versions of the microservices applications.

Monolith

In regards to the monolithic version, Figure 38 illustrates a high-level overview of the architecture after introducing the high availability related components, namely, the load balancers, replica of the main application, and a high availability cluster. The high availability methodologies described in previous sections were introduced for both stateful and stateless components, as represented in Figure 38.

Both instances of the application server run simultaneously, which allows parallel processing and offers redundancy. The load balancers distribute the load across the existent applicational servers, whose addresses are manually inserted in configuration files. If the plan is to have more instances added and removed dynamically, a discovery service would be useful to enable the load balancer to distribute the load to the active instances of the application at a given time.

Targeting a production-ready infrastructure, this monolithic application continues fully operational even if one of each component fails, since all possible SPOFs are eliminated through the introduction of redundancy.

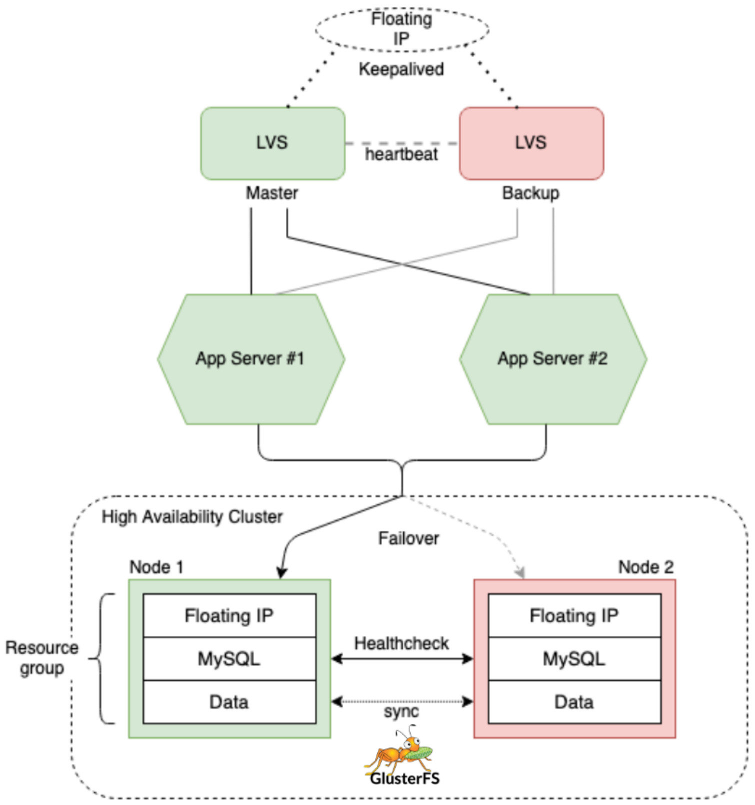


Figure 38: Infrastructure of the monolithic version with high availability assured

7 SYSTEM DEPLOYMENT

Posteriorly to the development of all the planned applications, the deployment process takes place. Once every business component is developed and high availability-related components are defined, the deployment takes place.

Different deployment strategies are used for the different applications since both use distinct virtualization mechanisms and feature different components. The used approaches and strategies are described throughout this chapter.

7.1 CLUSTER ARCHITECTURE DESIGN

Four physical machines were used to integrate the cluster, in this case, 4 Raspberry Pi 4 SBC modules, whose specifications are represented in Table 6. The official OS for the boards is Raspberry Pi OS, based on Debian Linux distribution, which comes with a graphical interface and several libraries and applications pre-installed. Since in this context the boards are used as servers and managed via SSH, the minimal version of the OS was installed, the Raspberry OS Lite, which does not have a graphical interface or desktop environment. Additionally, Docker engine was installed on each node, as well as some minor utils used for metrics monitoring.

Table 6: Technical specifications of the used Raspberry Pi 4

Processor	Broadcom BCM2711, quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz
Memory	4GB LPDDR4
Storage	SanDisk Extreme microSD Card 32GB

All the machines are connected to a private network through a physical switch (model D-LINK DGS-105), so the several Docker services can reach each other regardless of in which physical node they are hosted. The switch assigns to each physical machine connected a fixed IP address with an infinite lease time. Nonetheless, for each node, a hostname was defined and is reachable by it. Inside the Swarm cluster environment, the several services communicate through an overlay network, which enables the communication between containers present in different physical machines.

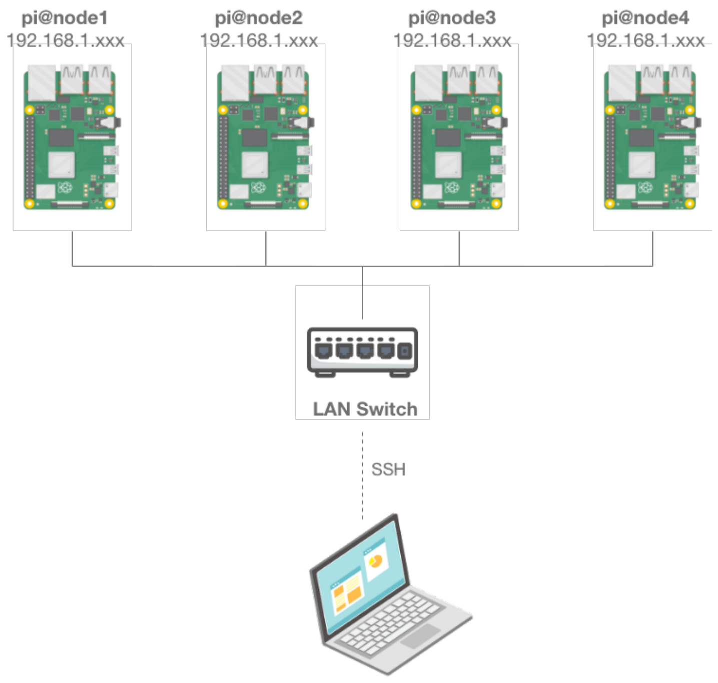


Figure 39: Topology of the physical setup

One key characteristic about using Raspberry Pis as Docker Swarm cluster nodes is that they have an ARMv8 CPU architecture and a 32 bits OS. This implied that the used Docker images had to be compatible with its ARM processor, otherwise it is unsupported. It happens that a considerable amount of the official Docker images are not built for the ARM processor architecture and the 32 bits OS of the Raspberry Pis. When a wanted image is not available for the targeted architecture it can be possible to build them using a selection of packages that are compatible with its architecture. Hence, there are alternatives publicly available from the community, which built ARM-compatible images that are not made available by the official developers. Some popular images, such as Java run time environments and RabbitMQ have official support for ARM architectures. However, some of the observability-related images were not as universal, but alternatives developed by the community were available. The "write once, run everywhere" motto from Docker essentially only applies to AMD64 based CPU architectures.

7.2 MICROSERVICES DEPLOYMENT

In Figure 40 all the components that integrate the final deployed architecture are illustrated in a traditional deployment diagram. The physical machines in which the containers are deployed are not represented in the diagram since there is not a fixed number of instances nor a defined distribution of the services across the nodes of a cluster.

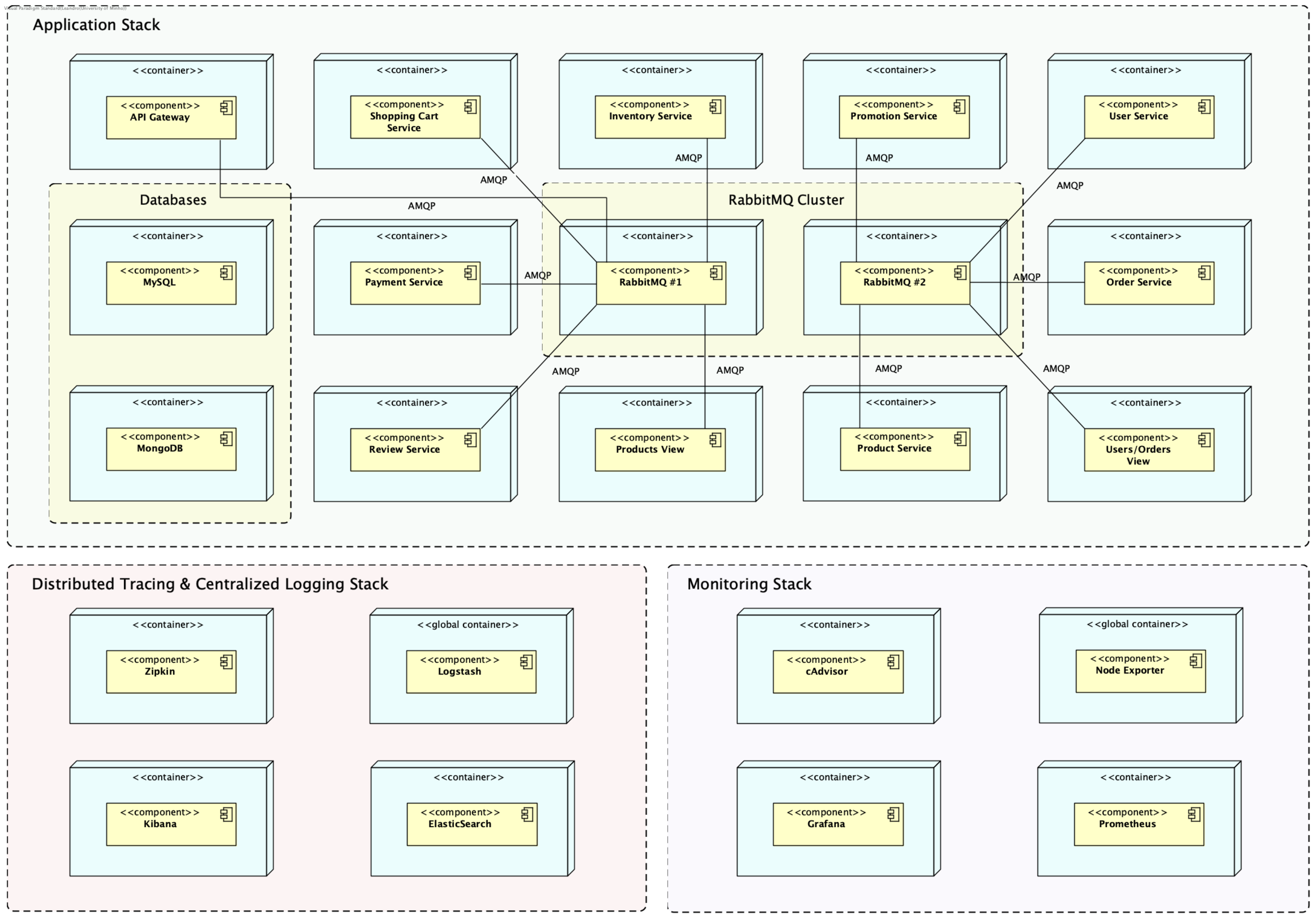


Figure 40: Deployment Diagram of the microservices version

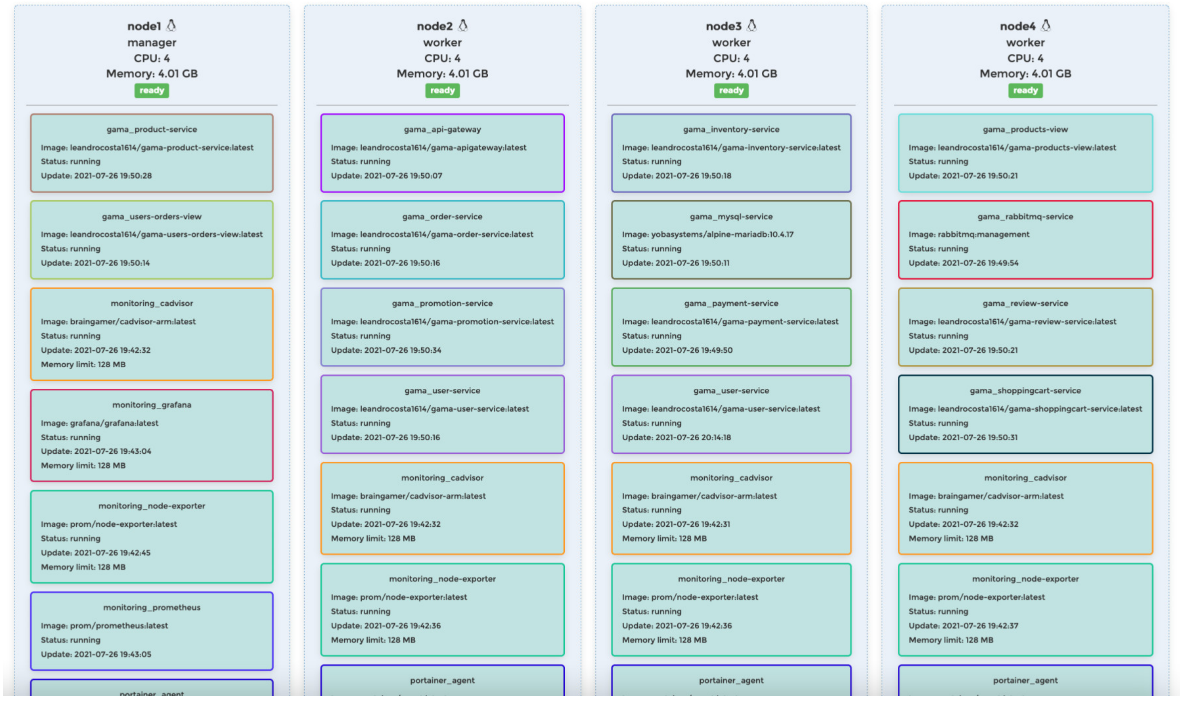
The deployment of the microservices application proceeded using a Docker Swarm cluster, in which three Docker stacks were defined: Main application stack, monitoring stack, and logging/tracing stack (as illustrated in Figure 37). The main application stack is composed of services that were built into a Docker image through the use of Docker Files and were afterward pushed to Docker Hub, the used container registry platform since each node of the cluster has its own Docker engine. The used orchestration tool is responsible for distributing the several containers across the available machines.

To deploy the stacks into the Swarm cluster it first needs to be initialized with the master and worker nodes joined as members of the cluster. Upon initialization of a stack, the specified images are pulled from Docker Hub and the containers are created and started. A stack is defined using a *yml* file, in which the services and their images are specified. These configuration files also hold information about the several constraints related to deployment, such as the initial number of replicas, how many replicas per node, and more. It is also possible to define resources constraints, such as the amount of CPU and memory reserved and the maximum usage limits for each running task. Then the stack initializes according to the defined constraints and the composing services are distributed across the nodes. Some services required to have a replica on each physical node, which is the case for the metrics collectors services. Others are required to be initialized exclusively in a manager node, which is all settable inside the stack configuration file. Furthermore, for the services that maintain a state, which is shared across the instances through the GlusterFS replicated volume, the bind mounts definition is also specified inside the stack configuration. Once the stacks are initialized everything is managed by Docker Swarm, if a failure on a container is detected, it is scheduled to be initialized in another instance and if a physical node fails, then all the services that were present on it are migrated to other instances.

Within a Docker Swarm cluster, there has to be at least one manager node that is responsible for maintaining the cluster state and schedule services. According to the official documentation, to make use of Swarm's fail tolerance features it is recommended to use an odd number of manager nodes in order to be able to recover from a failure of a node with no downtime. An N number of manager nodes tolerates the loss of at most (N-1)/2 managers, so, with four physical machines, the best option was to define three manager nodes so one of them could fail transparently. In the event of a node requiring maintenance, Swarm offers the possibility of setting its state to *DRAIN* and its containers will be scheduled to other available nodes.

An additional stack, the Portainer stack, was introduced to provide a graphical overview of the running tasks and how they are distributed across the physical instances at a given time. Portainer[1] is a container management tool that enables the management of the cluster through a graphical interface, in which it is possible to have a general overview of the running cluster, start and stop containers, check the running instances and its distribution (Figure 41) and even scale them (Figure 42). The portainer stack defines a globally deployed agent that enables the management of the whole Swarm cluster, which directly through Docker's CLI it is limited to a target node.

1 https://www.portainer.io/



node1
manager
CPU: 4
Memory: 4.01 GB
ready

gama_product-service
Image: leandrocosta1614/gama-product-service:latest
Status: running
Update: 2021-07-26 19:50:28

gama_users-orders-view
Image: leandrocosta1614/gama-users-orders-view:latest
Status: running
Update: 2021-07-26 19:50:14

monitoring_cadvisor
Image: braingamer/cadvisor-arm:latest
Status: running
Update: 2021-07-26 19:42:32
Memory limit: 128 MB

monitoring_grafana
Image: grafana/grafana:latest
Status: running
Update: 2021-07-26 19:43:04
Memory limit: 128 MB

monitoring_node-exporter
Image: prom/node-exporter:latest
Status: running
Update: 2021-07-26 19:42:45
Memory limit: 128 MB

monitoring_prometheus
Image: prom/prometheus:latest
Status: running
Update: 2021-07-26 19:43:05

portainer_agent

node2
worker
CPU: 4
Memory: 4.01 GB
ready

gama_api-gateway
Image: leandrocosta1614/gama-apigateway:latest
Status: running
Update: 2021-07-26 19:50:07

gama_order-service
Image: leandrocosta1614/gama-order-service:latest
Status: running
Update: 2021-07-26 19:50:16

gama_promotion-service
Image: leandrocosta1614/gama-promotion-service:latest
Status: running
Update: 2021-07-26 19:50:34

gama_user-service
Image: leandrocosta1614/gama-user-service:latest
Status: running
Update: 2021-07-26 19:50:16

monitoring_cadvisor
Image: braingamer/cadvisor-arm:latest
Status: running
Update: 2021-07-26 19:42:32
Memory limit: 128 MB

monitoring_node-exporter
Image: prom/node-exporter:latest
Status: running
Update: 2021-07-26 19:42:36
Memory limit: 128 MB

portainer_agent

node3
worker
CPU: 4
Memory: 4.01 GB
ready

gama_inventory-service
Image: leandrocosta1614/gama-inventory-service:latest
Status: running
Update: 2021-07-26 19:50:18

gama_mysql-service
Image: yobasystems/alpine-mariadb:10.4.17
Status: running
Update: 2021-07-26 19:50:11

gama_payment-service
Image: leandrocosta1614/gama-payment-service:latest
Status: running
Update: 2021-07-26 19:49:50

gama_user-service
Image: leandrocosta1614/gama-user-service:latest
Status: running
Update: 2021-07-26 20:14:18

monitoring_cadvisor
Image: braingamer/cadvisor-arm:latest
Status: running
Update: 2021-07-26 19:42:31
Memory limit: 128 MB

monitoring_node-exporter
Image: prom/node-exporter:latest
Status: running
Update: 2021-07-26 19:42:36
Memory limit: 128 MB

portainer_agent

node4
worker
CPU: 4
Memory: 4.01 GB
ready

gama_products-view
Image: leandrocosta1614/gama-products-view:latest
Status: running
Update: 2021-07-26 19:50:21

gama_rabbitmq-service
Image: rabbitmq:management
Status: running
Update: 2021-07-26 19:49:54

gama_review-service
Image: leandrocosta1614/gama-review-service:latest
Status: running
Update: 2021-07-26 19:50:21

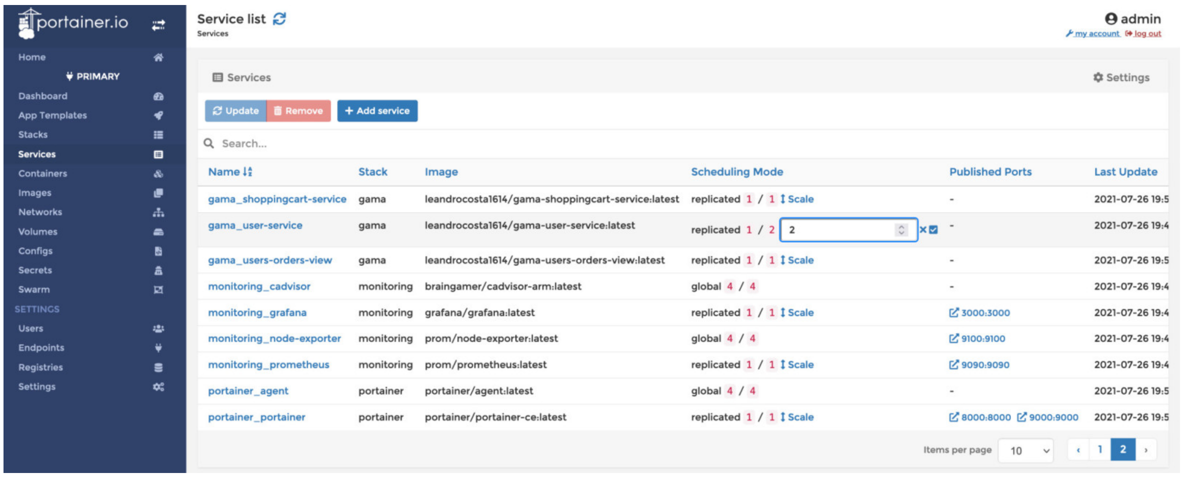
gama_shoppingcart-service
Image: leandrocosta1614/gama-shoppingcart-service:latest
Status: running
Update: 2021-07-26 19:50:31

monitoring_cadvisor
Image: braingamer/cadvisor-arm:latest
Status: running
Update: 2021-07-26 19:42:32
Memory limit: 128 MB

monitoring_node-exporter
Image: prom/node-exporter:latest
Status: running
Update: 2021-07-26 19:42:37
Memory limit: 128 MB

portainer_agent

Figure 41: Services distribution across the nodes from the Portainer dashboard



Service list
Services

Services

Name	Stack	Image	Scheduling Mode	Published Ports	Last Update
gama_shoppingcart-service	gama	leandrocosta1614/gama-shoppingcart-service:latest	replicated 1 / 1 Scale	-	2021-07-26 19:5
gama_user-service	gama	leandrocosta1614/gama-user-service:latest	replicated 1 / 2 2	-	2021-07-26 19:4
gama_users-orders-view	gama	leandrocosta1614/gama-users-orders-view:latest	replicated 1 / 1 Scale	-	2021-07-26 19:5
monitoring_cadvisor	monitoring	braingamer/cadvisor-arm:latest	global 4 / 4	-	2021-07-26 19:4
monitoring_grafana	monitoring	grafana/grafana:latest	replicated 1 / 1 Scale	3000:3000	2021-07-26 19:4
monitoring_node-exporter	monitoring	prom/node-exporter:latest	global 4 / 4	9100:9100	2021-07-26 19:4
monitoring_prometheus	monitoring	prom/prometheus:latest	replicated 1 / 1 Scale	9090:9090	2021-07-26 19:4
portainer_agent	portainer	portainer/agent:latest	global 4 / 4	-	2021-07-26 19:5
portainer_portainer	portainer	portainer/portainer-ce:latest	replicated 1 / 1 Scale	8000:8000 9000:9000	2021-07-26 19:5

Items per page 10 ‹ 1 2 ›

Figure 42: Scaling services using the Portainer dashboard

7.3 MONOLITH DEPLOYMENT

For the monolithic version, deployment proceeded in a different way since no orchestration tool was used. Although possible, containers were not used for the deployment of the monolithic version. Targeting a more traditional environment for a legacy application, virtual machines were used instead. Each of the virtual machines used, represented in Figure 43, consists of a machine with 3Gb of memory and a CPU with 2 cores.

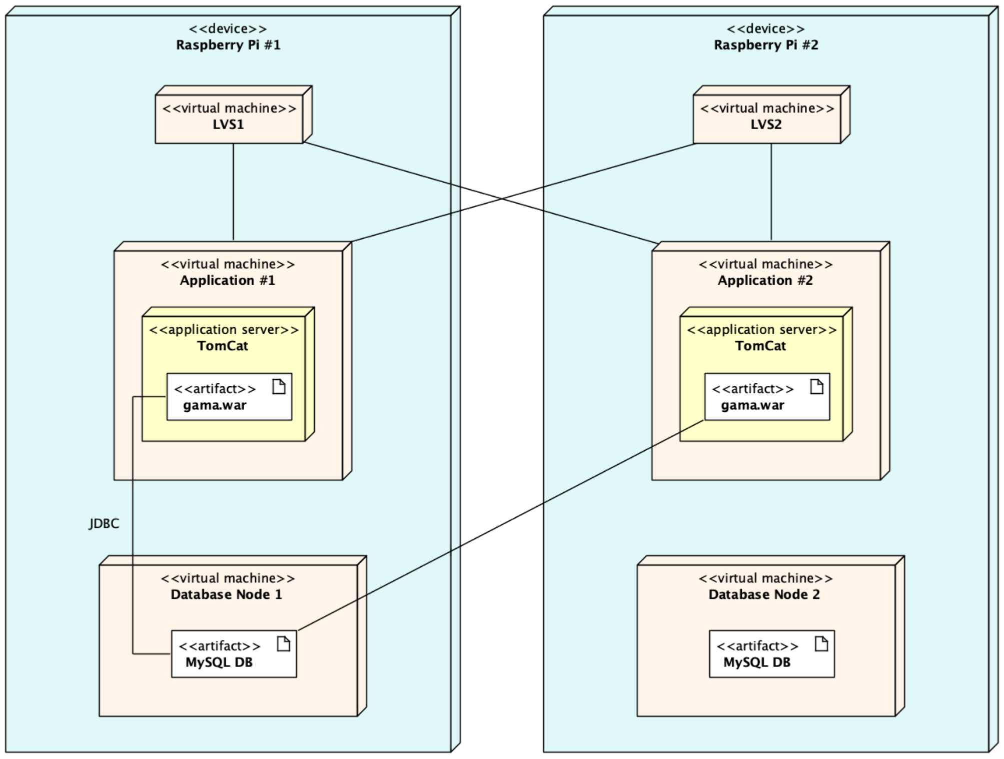


Figure 43: Deployment Diagram of the monolithic version

Regarding the deployment process for monolithic applications, there is only one deployment pipeline, and teams must depend on the others to be able to deploy its new features. DevOps activities are very limited in the context of a monolithic application. Efforts can be put into automating the deployment process, but the drawbacks of having a single deployment pipeline cannot be simply solved. Although it is relatively easy to deploy, any change made on the application, small or big, inevitably requires the entire application server to be redeployed. This fact can deeply reduce the frequency of deploys, which is something valuable in most modern applications.

Concerning downtime periods, with more than one running instance it is possible to implement updates with no planned downtime. This is done by redirecting all the requests to the active instance while the other is being updated.

7.4 CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY

Microservices potentially have a great affinity DevOps activities, highly promoted by the possibility of having a separate deployment pipeline for each service. Operational efficiency and shorter release cycles are some of the promoted values of using microservices However, similarly to all its other promised benefits, it strongly depends on the way it is implemented, which in this case, depends on how good are the CI/CD processes. Managing microservices is a more complex process compared with traditional applications due to a higher number of instances. Hence, automation is key to overcoming the existent complexity and achieving operational efficiency.

The way that the event-driven microservices version was developed made each service independently deployable. If a given service is modified it can be deployed without notifying or coordinating with any other service or even development teams, enabling more flexibility in the deploys. The small degree of coupling extends from code-level dependencies of the services to the development teams, which can work on new features without even knowing the other services exist. This behavior is one of the theoretical real values of using microservices, which would not be achieved if its implementation did not comply with the best practices that promote loose coupling.

As a result, each service that integrates the architecture has its independent deployment pipeline. Being the modern development characterized by the frequent need for change, having the ability to deploy new version while being isolated from the other services, will highly decrease the time to market, one of the most desired outcomes as stated in Section 2.5.

Regarding the automation of processes, there are several different ways of enabling automation of testing, building new images, and deploying them to production. The point of automating these processes is to decrease the time taken between deploys of new features or changes. Automation reduces the time to market by decreasing the human interaction in the deployment process.

In a microservices context, simply having a separate deployment pipeline for each service is already an improvement compared with traditional applications. Moreover, this architectural style frequently makes use of containers, which highly facilitates the introduction of CI/CD processes by using containers registry platforms with versioning. For the microservices repositories, GitLab was used due to its built-in CI/CD mechanisms, avoiding the need for extra components as automation servers. With GitLab's CI tools, when new code is committed, in this case, maven tests are covered and if they pass, then a new Docker image is built from the Dockerfile. The built images are pushed to Docker Hub, the service registry used for the Docker Swarm cluster implemented. Inside Docker Hub it is possible to use Webhooks, which in response to an image push to a repository, trigger some defined action. The action defined, in this context, was to communicate with the Portainer service inside the Docker Swarm. Portainer has the privileges to make updates to the Docker images used by the services, and when it receives the information that there is a new version of a used Docker image, it performs rolling updates, enabling continuous deployment. The described flow of events is represented in Figure 44.

If it is needed to update a running service, the manual approach would be to stop the running instance, pull the changes, and serve a static maintenance page while it is being deployed. However, by automating the process, downtime can be highly reduced. Since in a microservices and containers context starting a new instance of a service is a rather fast process, there is no reason to have downtime when a service is being updated, which highly relates to the CD concept. Docker Swarm enables the configuration of multiple parameters, including the update delay for the rolling updates, which defines the time delay between updates to a service. Therefore, when there are multiple replicas of the same task, each is updated individually within a timed interval so, there is always an active instance, enabling zero downtime.

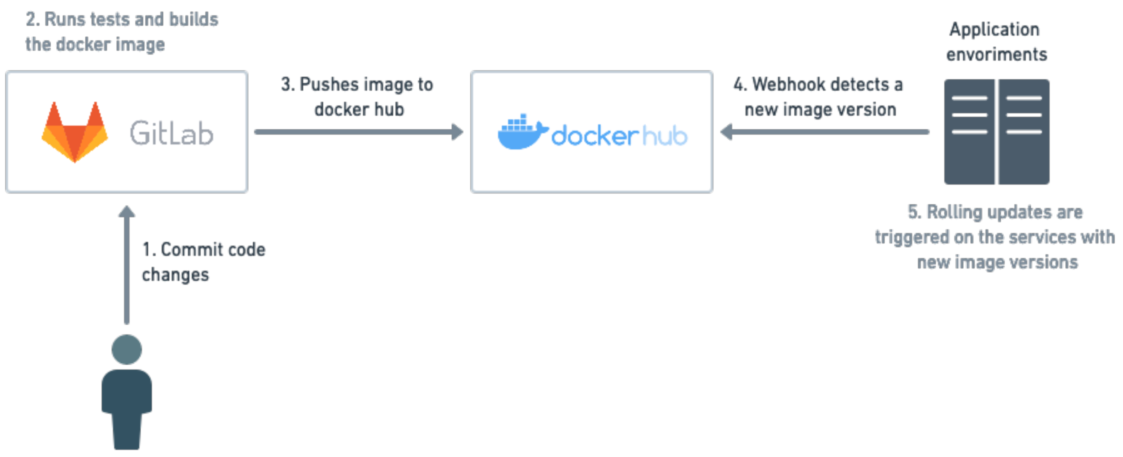

2. Runs tests and builds the docker image
GitLab
3. Pushes image to docker hub
docker hub
4. Webhook detects a new image version
Application enviroments
5. Rolling updates are triggered on the services with new image versions
1. Commit code changes

Figure 44: CI/CD pipeline of the microservices solution

Regarding the monolithic version, assuming no automation of processes is introduced, any change made requires that the whole application has to be redeployed. For every new version, it will have to be placed on the only deployment pipeline available, shared with several other contexts, potentially unrelated. What often happens is that deployments take so long that while a new version is ready, an older one is still waiting to be deployed. Furthermore, if a new version is deployed and ends up featuring bugs, then the pending updates will be further delayed. Moreover, if there is no redundancy, a redeploy of the application, which requires it to be restarted, inevitably means that there will be planned downtime.

Nonetheless, automation of processes is also achievable in traditional applications and can also benefit from it. A rapid deployment process is key due to the existence of a single deployment pipeline, which forces new updates to wait for each other. However, even an automated and optimized flow of CI/CD does not change the most critical factor, a single shared deployment pipeline that is shared with all development teams, which is an obstacle in having frequent deployments.

8

PERFORMANCE EVALUATION

Several different aspects can have an impact on the performance benchmarks of an application, namely, the architecture chosen. Regarding microservices, to achieve the promised benefits there are a set of principles to follow, mostly related to maintaining the services decoupled. To assure that, the introduction of specific patterns is required, from the communication approach used to the data management methodologies applied, which impact fields, such as performance, differently.

The existent academic research on how microservices impact performance is limited and still has not reached unanimity (see Section 2.5). Moreover, the existent research typically does not feature microservices implementations in accordance with its core principles, failing to achieve loose coupling between the services, and, consequently, the majority of their promised benefits. The performance impact under different circumstances, namely implementation patterns, is not a particularly sustained topic in research work.

From the impact of different inter-service communication approaches, the overhead of the maintenance of replicas, Saga Transactions, and the use of a DFS, every potential factor that can impact performance is evaluated through the experiments conducted.

Through this chapter, the experiments conducted are described and the impact on the performance of different architectural styles and patterns is evaluated.

8.1 EXPERIMENTS SETTING

Most of the microservices applications are hosted and managed in cloud infrastructures, often making use of PaaS tools, which assist and automate several aspects and enable the configuration of numerous parameters through some dashboard. There are clear benefits in using cloud-based solutions, but it results in some control and freedom being rescinded from the developers. Usually, that automation of complex processes is actually desired, however, for these experiments, full control was intended. Nonetheless, cloud hosting options also offer Infrastructure-as-a-Service (IaaS) functionalities, which provide almost full control of the instances, but the machines are still owned, managed, and monitored by the service provider. Moreover, the instances provided by cloud services are based on virtualization, not assuring that the CPU is entirely reserved for a given user, possibly being shared with other "noisy neighbors", the called *CPU Steal* phenomenon. Additionally, due to the considerable amount of machinery accountable in microservices applications, it is a costly service. For those reasons and given that the material was already in possession, it was decided to conduct the experiment in a cluster of four Raspberry Pi 4 SBC, a low cost and energy-efficient solution. Furthermore, being local machines there are no external factors that could interfere with the benchmarks so the results can be consistent throughout the experiments and all parameters are fully customizable and can be tuned.

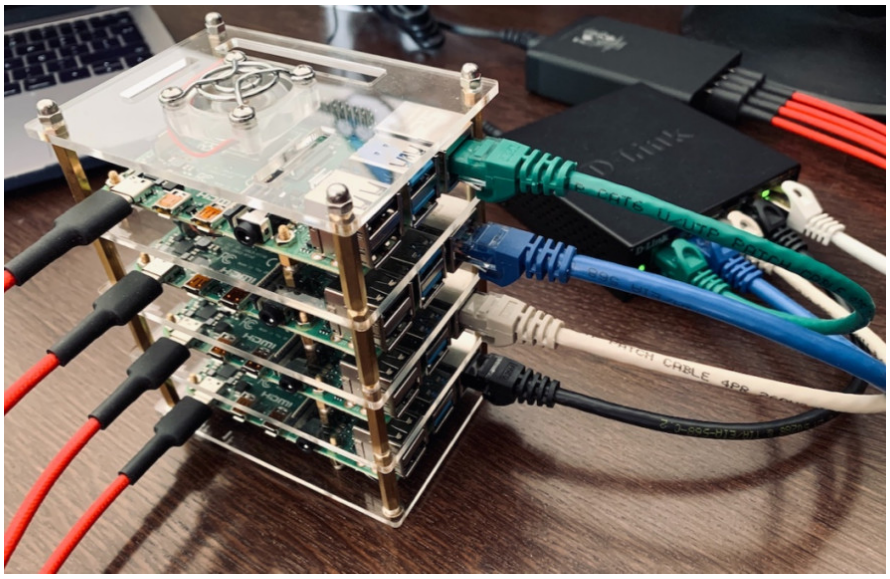


Figure 45: Raspberry Pi cluster used in the experiments

Hence, the microservices implementations of the application were deployed on a cluster of Raspberry pi 4 SBC, represented in Figure 45. Furthering a more realistic setting, for the microservices applications, the multiple services are distributed across different physical hosts, which is something rarely seen in performance testing research works, as mentioned in Section 2.5.

8.2 MEASUREMENT METHODOLOGY

In order to evaluate the performance and capacity of the developed applications load testing was conducted, which simulates user load in a realistic scenario. The methodology used was to apply a workload of multiple requests at a specific rate per time unit and assess how the system behaves under the load. Several test scenarios are conducted, and for each iteration, the number of requests per minute (the ramp-up period selected) is progressively increased, and metrics, such as response time and throughput are analyzed. All test scenarios start with a load of 500 requests/minute and, in each iteration, the number of requests is increased.

The analysis of the response time, which is the time taken to fully complete the requests, may indicate what is the saturation point of an application. This point corresponds to the stage in which the load is so high that the application cannot respond in a timely fashion, drastically increasing the response time. When this point is reached it means that the application or service has met its load limit, which is often accompanied by a high error rate.

Another relevant metric is the throughput, which is the measure of the number of successful requests per unit of time, usually, per second. This metric helps to determine the performance capacity of a system and indicates how much load it can handle. Along with the hardware metrics collected, throughput and response times are part of the dependent variables of the experiments.

To measure the performance of the systems two workloads were defined, providing different scenarios that may benefit from the characteristics of each version. Essentially, the two workloads are composed of different operations with distinct complexity in terms of computational resources usage and the number of spanned services. Thus, the benchmarks feature a simple workload and a compound workload.

The simple workload comprises the Edit Stock operation followed by the fetching of a single product's information. The two actions are essentially an HTTP PUT request, that in the microservices version, implies propagation of an event and at least two database accesses, and then a simple GET operation. Nonetheless, in terms of resources usage, it is one of the simplest operations with fewer services interactions.

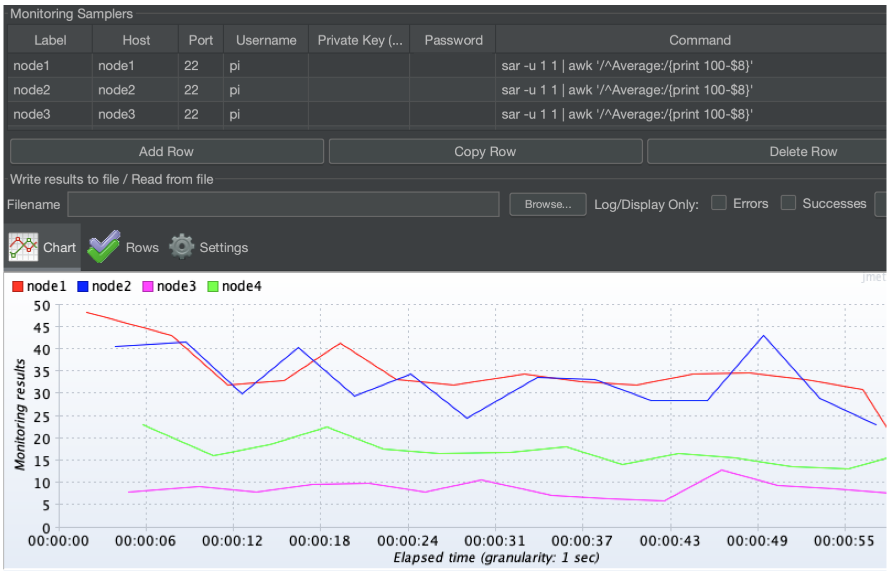
The compound workload consists of the creation of an order, which is one of the most resource-intensive operations of the system. Regarding the microservices version, this request, represented in Figure 21, spans over five different services and the publication of at least four events. Along with the different implementation patterns used and the number of requests per unit of time, the workloads constitute the independent variables of the experiments.

All the performance tests were conducted using Apache Jmeter[1], which simulates traffic of requests that would take place in a real scenario. It also provides the performance metrics necessary to evaluate the performance of the systems. JMeter was installed on a local laptop connected to the applications' network.

In order to monitor the hardware metrics, the monitoring stack represented in Figure 37 could have been used as it enables the tracking of all the relevant metrics for the benchmarks. However, it implies the running of the cAdvisor agent on each node, Grafana, and Prometheus, which is inconvenient as they may affect the performance of the system, which is not present in the monolithic counterpart. Likewise, the logging and tracing stacks were also removed for the performance tests to provide a more similar scenario for the monolith and microservices, which would be compared.

The approach used to measure the hardware metrics during the load testing was to use the SSHMon[2] plugin for JMeter. This agentless solution connects to the nodes via SSH and periodically executes a defined command. The used commands come from the sysstat utilities and it provides the desired metrics: CPU and memory usage and I/O wait. The output from the execution of the commands is displayed inside JMeter's interface in the form of a graph (Figure 46), enabling the tracking of the metrics during load testing in real-time.

1 https://jmeter.apache.org/
2 https://github.com/tilln/jmeter-sshmon


Monitoring Samplers

Label	Host	Port	Username	Private Key (...	Password	Command
node1	node1	22	pi			sar -u 1 1 | awk '/^Average:/{print 100-$8}'
node2	node2	22	pi			sar -u 1 1 | awk '/^Average:/{print 100-$8}'
node3	node3	22	pi			sar -u 1 1 | awk '/^Average:/{print 100-$8}'

Figure 46: CPU metrics collected by SSHMon

The analysis of these metrics can help to identify what are the performance bottlenecks of the system. For instance, a high percentage of CPU usage may indicate that the instance is facing computational issues, in the sense that it may not have room for maneuver. A high I/O wait value may hint that the database operations represent the bottleneck, among other possible conclusions.

Regarding the databases, throughout the dissertation, it has been mentioned that they should not be part of the cluster and be cloud-based. Due to the sensitive nature of the database data and also easy access to scalability, sharding mechanisms, and migration options, a cloud solution would be more suitable. Nonetheless, a cluster of the databases used could be integrated into the Swarm cluster.

In order to avoid misleading results, the possibly out-of-scope I/O overhead of synchronizing database data across several instances of the cluster was removed. Having the databases deployed together with the business services could turn them into a substantial performance bottleneck candidate since the databases alone could integrate a dedicated cluster

If the whole system had been hosted on the cloud, a cloud-based database would be the best option. However, since the applications are hosted on local machines, connecting to the database over the internet would add a considerable amount of latency that would impact the performance tests. Targeting an as realistic as possible scenario for the performance tests and making it comparable with the monolithic counterpart, the considered solution was to use another local machine, a Linux desktop as the host for database servers. This component is also part of the private network in which the nodes are integrated, so communication happens as if the applications were hosted on the cloud and connected to a database within a local network.

This solution was used for all experiments except one, in which databases' data is kept in sync across all nodes using a DFS. This specific experiment aims to analyze what is the overhead in terms of performance of keeping the state of the stateful containers synchronized across multiple instances.

.

8.3 TEST CASES

The multiple test case scenarios were defined in order to evaluate the impact of using microservices and their many implementation patterns. Throughout the following subsections, each test case is further described and the results from the performance benchmarks are discussed.

8.3.1 *Monolith*

The monolithic version of the reference application packs all the business operations into one single codebase and runs within a single process. Through the comparison with its microservices counterpart, it is evaluated how its characteristics impact the system's performance. This version aims to portrait a traditional legacy application scenario with high availability assured. Independently from the development process, with the conduction of the performance benchmarks and subsequent comparison of results it is targeted to evaluate how traditional applications perform in comparison to the modern distributed solutions.

While the microservices infrastructure is composed of four nodes which host multiple services that run on its own process, it would be unfair, performance-wise, to have monolithic instances distributed across four physical nodes. The infrastructure used for the monolithic version consists of two physical machines that host a virtual machine for each component. The used virtual machines are the same described in Section 7.3, which have 3Gb of memory and a CPU with 2 cores. For each component, high availability was assured through the redundancy of the applicational server, the introduction of LVS load balancers, and a virtual IP address. The MySQL database across all experiments is, as previously described, hosted on a dedicated Linux server.

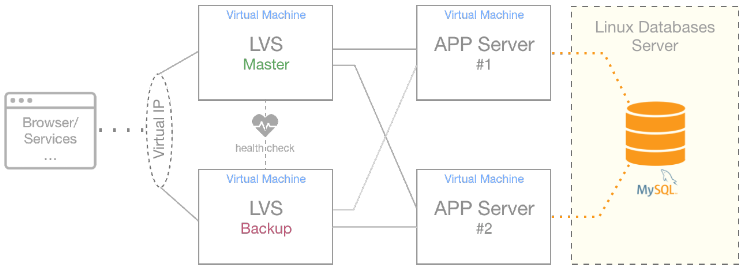


Figure 47: Test case - Monolith

8.3.2 *Synchronous request/response microservices*

This microservices version is the one often featured in performance benchmarks research, which makes use of a request/response communication methodology. As already stated, this approach should preferably be avoided since it adds dependencies between the services, which highly contrasts with the microservices principles of having decoupled services, as discussed in Section 5.3.

Through the comparison of this version's benchmark results and the monolithic version, it is possible to directly evaluate the impact on the latency that comes from a distributed system. Given that this version does not feature data synchronization mechanisms, the only distinction between the monolithic version is the use of Saga Transactions and different database schemas for each service.

Additionally, by comparing the benchmarks from this version and the asynchronous one, it is possible to evaluate the impact on benchmarks from the use of an event-driven approach with data synchronization.

Containers were used for this version and it is deployed on a Docker Swarm cluster composed of four Raspberry Pi SBC and for the database, the external Linux server was used.

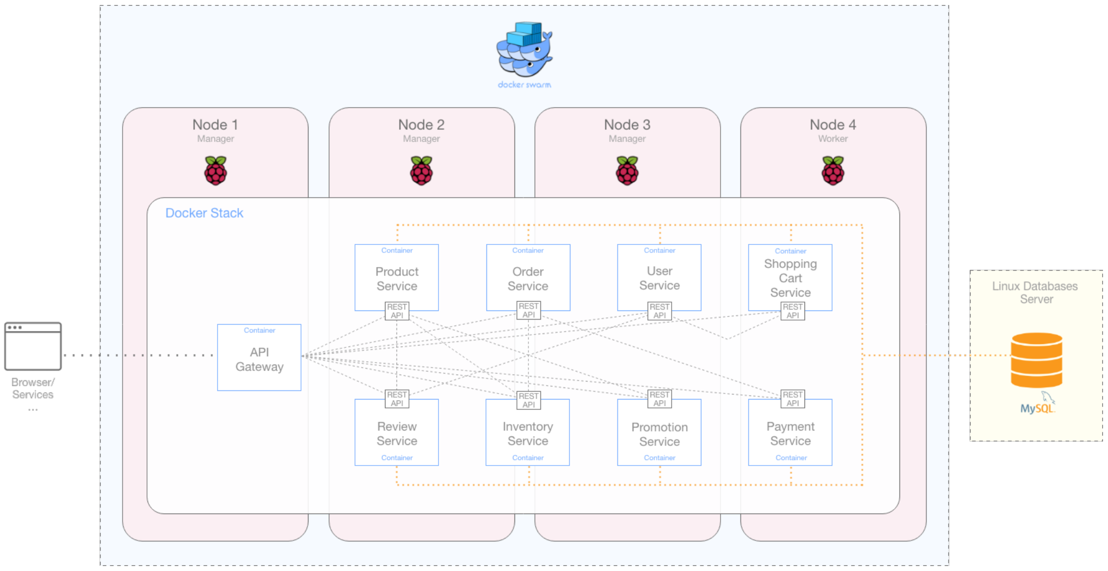


Figure 48: Test case - Synchronous request/response microservices

8.3.3 *Asynchronous event-driven microservices*

The asynchronous version of the microservices solution consists of the same business services as the request/response microservices version, but with the addition of a message broker and two additional services, the read models/views (Figure 49). As mentioned in Section 5.3, in order to avoid too much asynchronous communication between the services to fulfill an operation, data was replicated across some services. Although it makes the services more independent from each other, there is an overhead associated with the data synchronization for the maintenance of replicas, which was predicted beforehand that its impact may be patent in the benchmarks. The eventual consistency outcome of using event-driven microservices is also something that was expected to have an impact on performance benchmarks.

The conduction of benchmarks to this version aims to contribute to the lack of performance analysis of a microservices architecture that complies with the best practices, which promotes decoupling between the services. Furthermore, the comparison with the other versions enables the evaluation of the impact of several factors, such as the use of a message broker and the overhead of having replicated data being synchronized.

This version was also deployed across the same four Raspberry Pi SBC which compose the used Docker Swarm cluster. The database systems are also hosted on an external Linux server. Given the identical conditions between the experiments, by comparing the benchmark results from the synchronous and asynchronous microservices versions it is possible to evaluate the impact of using a message broker and maintaining replicated data.

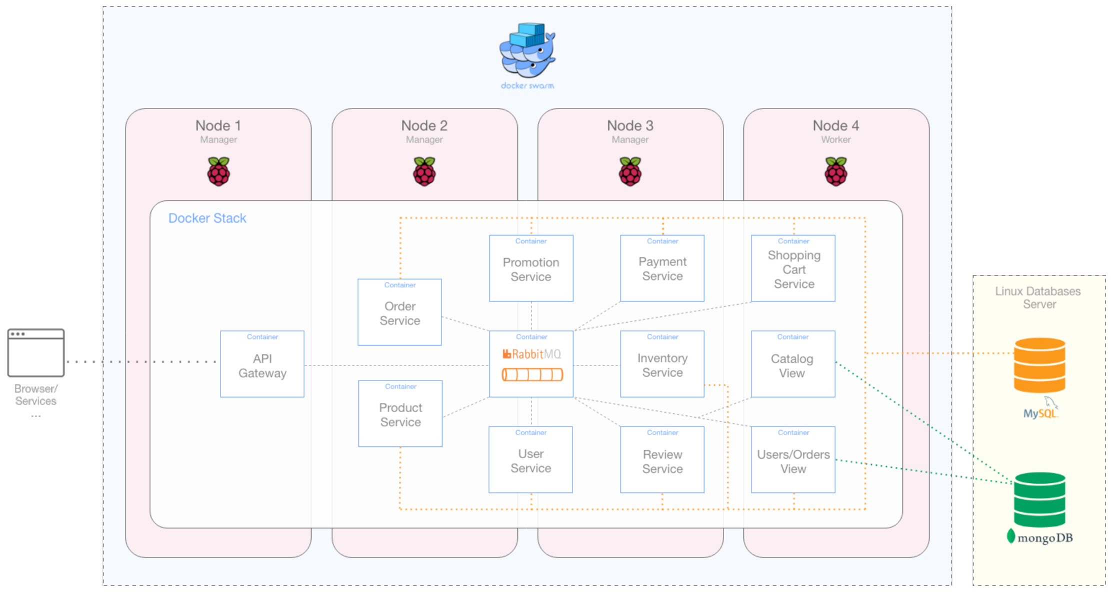


Figure 49: Test case - Asynchronous event-driven microservices without a DFS

As mentioned in Section 8.2 and illustrated in Figure 49, the databases were kept outside the cluster since it is not recommended to keep such sensitive data inside a volatile cluster of containers. Regardless, there are scenarios that demand the synchronization of the data from the stateful components. These components, when initialized in any of the nodes of the cluster should have access to a previously maintained state in order to continue operating. To assure the state is available across all nodes, a DFS is commonly used.

To analyze the impact of using a DFS, this version was further split into two test cases, one with the databases hosted on a separate server (Figure 49), and one featuring database data synchronization across the nodes of the cluster (Figure 50). Given that the databases are stateful components, as explained in Section 6.1.1, a DFS that keeps the service's data available across all nodes was implemented using GlusterFS.

Both used databases offer cluster capabilities that could have been used instead, which deals with the data synchronization at an application level. However, the use of a DFS is something commonly necessary in a microservices system, so evaluating the impact of its introduction provides valuable data. GlusterFS was used to synchronize the MySQL and MongoDB data, making it available across all nodes, enabling its initialization in any physical instance. Although it was predicted beforehand that there would be a lot of I/O load due to the intensive data synchronization mechanisms, how it impacts the overall performance composes one of the endorsed test cases. Hence, benchmarks were conducted to both an asynchronous event-driven architecture with data synchronization between the nodes and without data synchronization. The comparison between the results from the two asynchronous versions provides information on how the introduction of a DFS for state synchronization within a cluster affects the performance of a system.

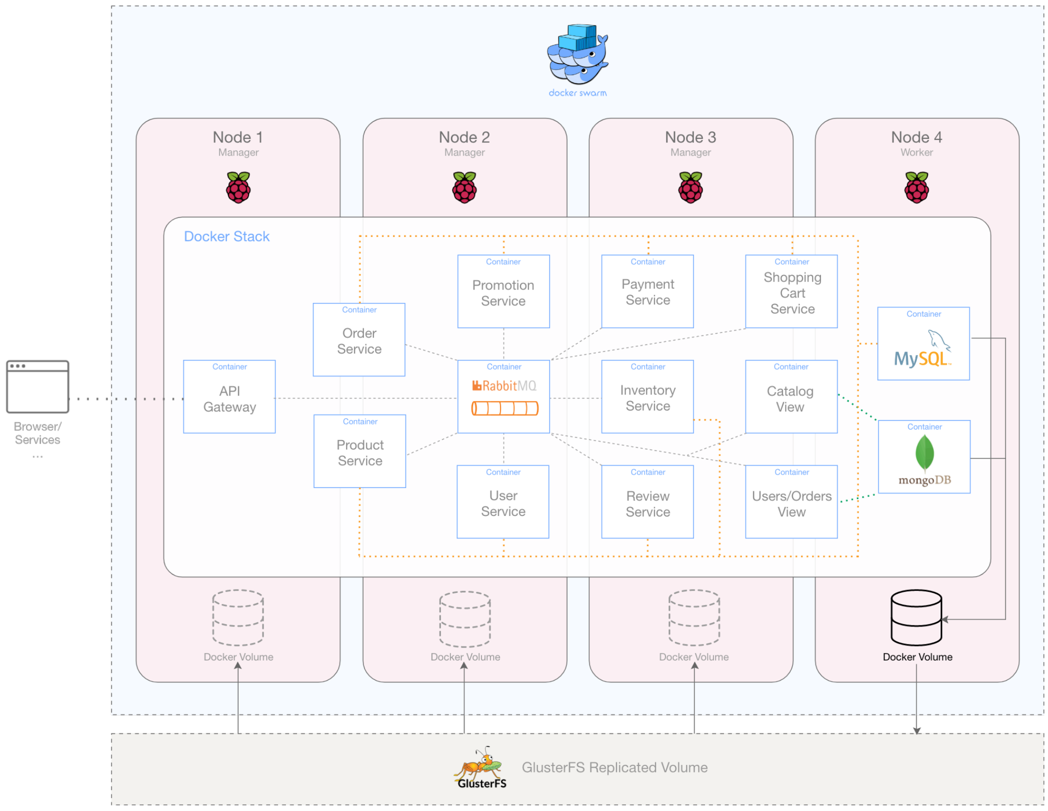


Figure 50: Test case - Asynchronous event-driven microservices with a DFS

Benchmarking the asynchronous microservices versions was not as straightforward as the rest of the test cases. The performance data provided by JMeter is the measure between the request from the load testing tool and the API Gateway since the operation is classified as completed after the event gets published to the message broker, which does not represent the complete processing time. The approach of measuring the response times from the asynchronous versions was to use a tracing tool that provides information about the time spent in each service (see Figure 35).

8.4 VALIDITY THREATS

Throughout the whole development of different versions of the same application, the number of variables was tried to be minimized by using the same programming language and frameworks in each version.

All the applications from the test cases ran in complete isolation with no background tasks running, except for the external database server, which is a general use desktop with background tasks constantly running. The several physical hosts were connected using cat6 Ethernet cables and a gigabit switch that connects all the hosts to enable high-speed connections between the services.

The asynchronous microservices versions used MongoDB as the database system for the read models introduced from the application of the CQRS pattern. MongoDB is a database system that has different performance characteristics which are not present in the synchronous microservices version nor the monolithic. Nonetheless, maintaining read models is an overhead consciously introduced on the asynchronous version, which may have an impact on the performance results.

Since there are no official images of the used database systems for the Raspberry Pi's CPU architecture, the database images used in the microservices version were developed from the community. MongoDB does not even natively support a 32-bit operating system and MySQL does not officially support ARM32 architectures, which can impact the performance results of the test case that keeps the database systems inside the Swarm cluster. Regardless, all the other implementations used a dedicated database server.

8.5 BENCHMARK RESULTS

The experiments over the test cases defined were conducted according to the methodology specified in Section 8.2 and the results are showcased and analyzed through the following subsections.

8.5.1 *Compound Workload*

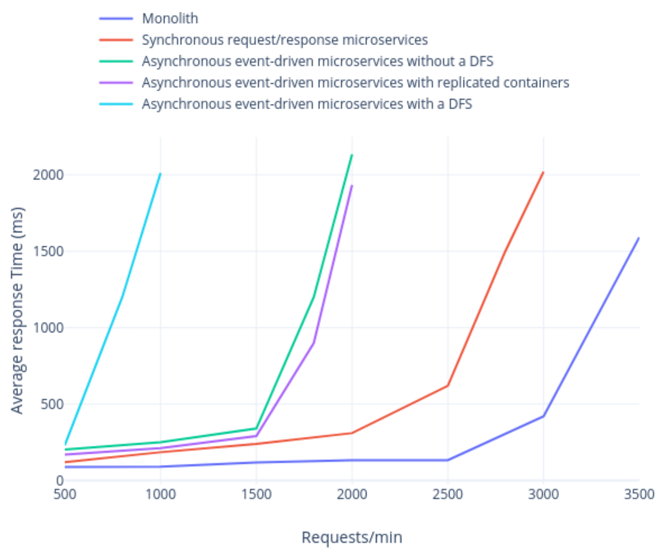


Figure 51: Benchmarks of the compound workload

Regarding the compound workload, the results (see Figure 51) have revealed that the monolithic version is the one with better performance results and the asynchronous versions the worst. That is partially explained by the fact that within a monolithic application all operations take place locally at language-level calls. In contrast, microservices have to communicate with different services to perform the same operations, asynchronously. Regardless, simply by being a distributed system, there is latency in the communication between the services in the microservices versions, which may reside in different physical hosts. The existence of latency also partially explains why the synchronous microservices version has also performed worse than the monolithic version.

Furthermore, in the monolithic application, database operations are executed using local transactions, which aggregates the multiple necessary operations all at once, while in the microservices versions the operations are proceeded through the use of Saga Transactions (see Section 5.2). This approach often requires multiple database accesses, which is the case of the compound workload, contributing to higher response times for the asynchronous versions.

The steps of an operation within the asynchronous versions take place eventually, in which notifications are placed in queues to be later processed. This methodology, as previously stated, provides a great degree of resilience since services do not directly depend on each other. Additionally, during the compound workload, the asynchronous versions maintain replicated data that is constantly being synchronized, adding additional load that is not present in the synchronous version nor the monolith. The differences in performance between the asynchronous and synchronous microservices versions are also explained by the fact that the first one communicates through a middle-ware, a message broker, adding even more latency to the communication process. At the cost of not having a sequential flow of events and the overhead of synchronizing replicated data, as expected, the performance benchmarks from the asynchronous versions are not the best for complex workloads. There is a clear trade-off between having loosely coupled services and performance, in which it may be acceptable to take seconds to complete an operation, which obviously may not suit every scenario.

For these reasons, the worst performance comes from the asynchronous versions, which reach the saturation point earlier compared with the synchronous version and monolith, which although was expected, the collected metrics were analyzed to identify possible bottlenecks and potentially mitigate them.

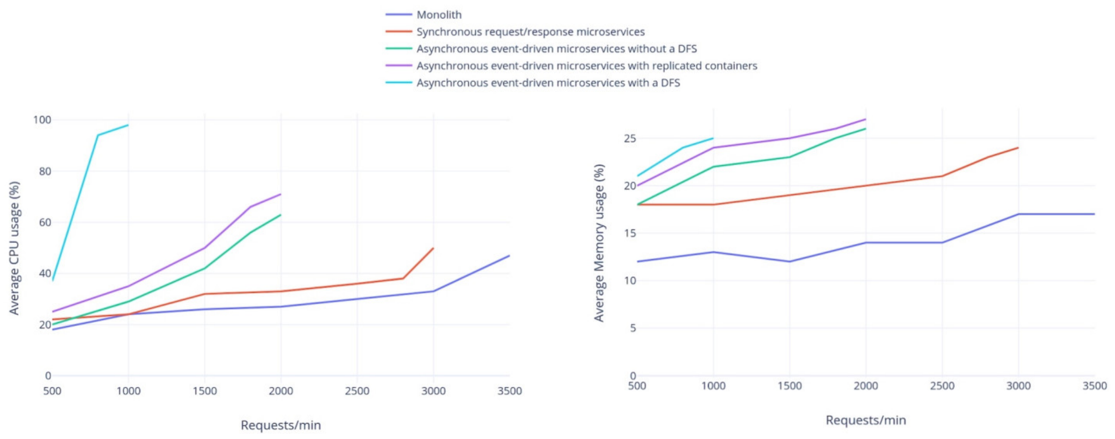


Figure 52: Hardware metrics collected during the load tests of the compound workload

As an attempt of improving the results and delay the saturation point of the asynchronous versions, the main participant services were scaled. However, replicating the number of running instances of some containers did not considerably improve the response time results nor delay the saturation point. This is explained by the fact that the resource capacity of the services has not reached its limits, which is showcased by the acceptable CPU usage percentage during the entirety of the load tests (see Figure 52). A high CPU utilization often means that a lot of context switching is happening, but during the saturation point, it was not verified. The effect of scaling the services resulted in the CPU utilization being more uniformly distributed across the nodes, which did not even reach full capacity after hitting the saturation point. The asynchronous versions have a slightly higher amount of CPU usage in general due to extra steps taken for each operation and also due to having more running containers which are not present in the other versions. Regarding the memory usage during the load tests, the values did not feature substantial oscillations, but it is possible to analyze the impact of the different types of virtualization, containers (used for microservices versions), and virtual machines (used for the monolithic version).

Hence, the performance bottleneck did not reside in the CPU nor memory usage, but in the database systems. The asynchronous versions perform up to five times more database operations than the other versions due to it maintaining replicated data that needs to be synchronized. Consequently, scaling the containers and improving the processing capacity of the business services did not delay the saturation point. As mentioned previously, inside the configuration file of a Docker Swarm stack it is possible to set resource constraints. Several combinations of CPU and memory limits were tested but since the bottleneck was not due to limited capacity it did not improve the results. Essentially, at the cost of having more isolated services with all the data needed to fulfill its operations, databases are more often accessed, which nonetheless, are typically the bottleneck of web applications. Therefore, since the database systems can be described as being the bottleneck and given that the asynchronous version features more database operations than the other versions, it means its performance is the most affected by this bottleneck, which explains its eager saturation point.

However, when the database is still capable of handling the load, while the number of requests per minute is still low, the response times were better when replicated instances were introduced, which enabled the possibility of parallel processing. If the bottleneck was due to capacity limitation from the containers, showcased by a high CPU usage, having replicated instances would not only improve the response times but also delay the saturation point. In order to delay the saturation point across all versions whose bottleneck is the database systems, a more performant database infrastructure would have to be introduced, for instance, a dedicated cluster with sharding mechanisms and parallel processing. Regardless, with the exact same database system, the benchmark results highly differ across the different implementations. Due to the higher load of databases accesses per operation on the asynchronous version, in order to obtain similar results to the other versions, it would require a more performant database solution.

Through the comparison of the benchmarks from the monolithic and synchronous microservices version, it is possible to inspect what is the direct impact of having a legacy application made distributed. The synchronous microservices version does not maintain replicas of data so the only difference between the monolithic version is that operations are executed through Saga Transactions, in which the different services spawn each other in a request/response cycle. Although the monolithic version featured better response times and a later saturation point, the benchmarks between these two versions are similar. However, in most cases using synchronous communication is an anti-pattern that introduces tight coupling between the services.

In terms of throughput values, they were also analyzed as they provide insights of what is the performance capacity of the system (Figure 53). Regarding the asynchronous versions, since the inter-service communication does not occur via a synchronous request/response style, such as through the use of HTTP, the timeout periods that are associated with it do not apply to the asynchronous version, making the error rate zero due to its eventual consistency nature. The throughput values of this approach are affected by that factor, while in the synchronous versions requests that take too long or are discarded simply cause an increase in the error rate.

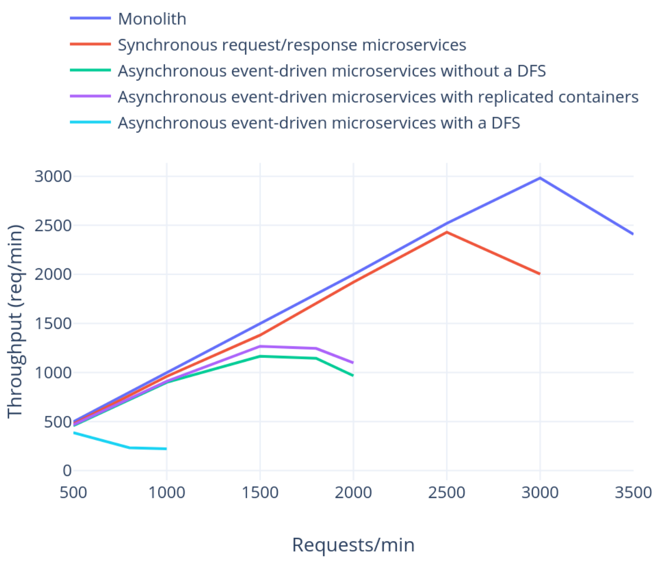


Figure 53: Throughputs of the compound workload

The monolithic version was capable of successfully completing up to 3000 requests per minute for the compound workload, which means any load bigger than that has a substantially higher response time or will increase the error rate. The maximum capacity of the synchronous microservices version was slightly lower, capable of completing up to 2500 requests per minute. The throughput starts to decrease once the saturation point is reached in both versions, which showcases what is the maximum capacity of the system.

Distributed file system

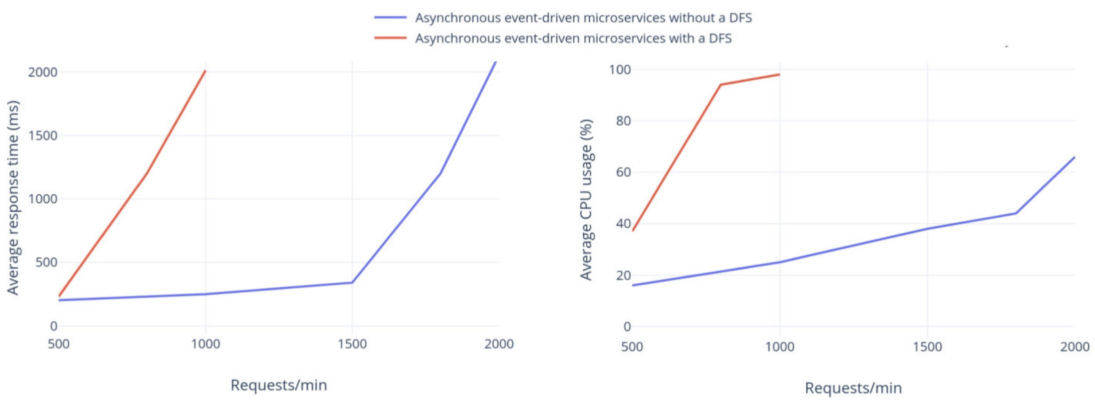


Figure 54: Asynchronous event-driven microservices with and without a DFS

Regarding the two asynchronous versions, it was evaluated how the introduction of a DFS impacts the benchmark results. The database systems are the bottleneck across all versions, except for the version that uses a DFS, whose bottleneck resides in resource capacity limitations, highlighted by the high CPU percentage across all nodes (see Figure 52). This version requires the synchronization of the database data every time a write operation takes place, which represents a huge limitation with significant overheads in terms of resources utilization. The CPU usage percentage when using a DFS has increased significantly which ends up becoming the performance bottleneck in this version, reaching the saturation point considerably earlier.

Since GlusterFS is mainly a file-system storage, replication does not happen at a block level, which causes a great impact on the CPU utilization and also networking, since during load testing there is constantly synchronization happening.

As previously mentioned, the asynchronous version is composed of services that maintain shared replicated entities, which demands the publication of events whenever data changes so the consumer services can keep all the replicas consistent. The maintenance of shared entities is already a considerable overhead, increasing the number of database operations per request when compared with the synchronous versions. On top of that, having database data files being constantly synchronized across multiple instances requires a substantial amount of resources. The combination of all these mechanisms demands a very performant infrastructure to have the same performance results as its synchronous counterparts. The results have clearly demonstrated that when data is synchronized across the nodes of the cluster, the overall performance substantially decreases.

8.5.2 *Simple Workload*

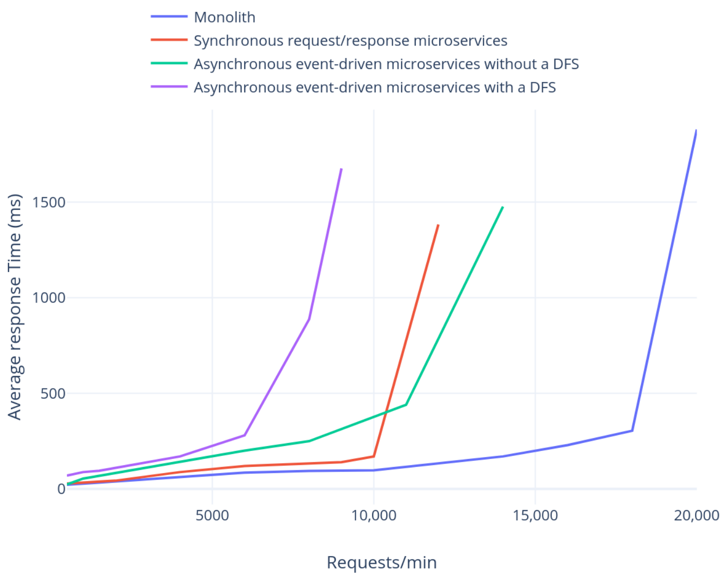


Figure 55: Benchmarks of the simple workload

The results from the benchmark of the simple workload are represented in Figure 55, and it showcases once again that the monolith version is the one with the best performance values. The saturation points are naturally reached at a higher number of requests per minute and the response time values are lower since the simple workload is composed of less intensive tasks. Latency is more emphasized in the compound workload since it requires more interactions between the services, so, overall, compared with the compound workload, there are fewer differences between the different versions.

The analysis of the results of the simple workload from the asynchronous and synchronous microservices versions enables the evaluation of the impact of different types of inter-process communication in microservices architectures, in this context, RabbitMQ and REST API's. Although the asynchronous version has the overhead of maintaining replicated data across different services, the simple workload features basic operations making the communication mechanisms used between the services the major influential factor.

The performance results have shown that even though the synchronous microservices version exhibited better response times during lower load and reaches a higher maximum throughput value, it hits the saturation point earlier than the asynchronous version. The same was not verified during the compound workload because it featured more database operations on the asynchronous version, which was the bottleneck in that scenario. In this test case, the database operations were not as limiting as they were during the compound workload, allowing the CPU to reach higher utilization and throughput capacity.

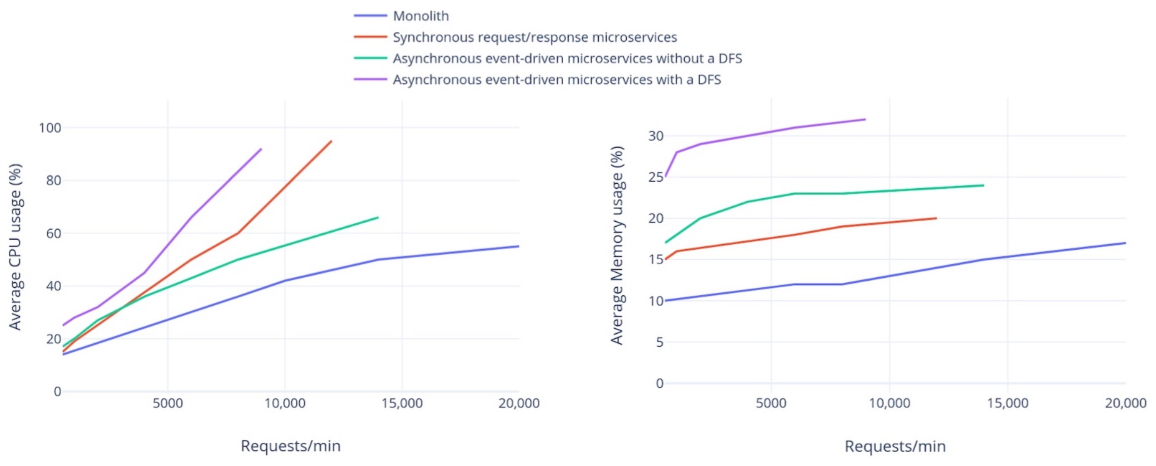


Figure 56: Hardware metrics collected during the load tests of the simple workload

The blocking behavior from the use of a synchronous approach exhibited limitations as the load increased. While the asynchronous version has the messages sent to a broker with no wait time for a response from other services when using a synchronous methodology, each thread is blocked until a response comes back from the requested service. This approach results in exponentially increased response times when saturated. Thus, for the synchronous version, the saturation point was reached earlier and the bottleneck resided in CPU limitations, showcased by a very high CPU usage percentage during the saturation point (see Figure 56). On the other hand, the asynchronous version's bottleneck was the database systems, since during the load tests the CPU utilization never reached critical values, similar to what happened in the compound workload. Using an event-based approach with RabbitMQ as the message broker proved to be more resource-efficient when the load is higher, reaching the saturation point later than the synchronous microservices version. In terms of response time, the synchronous version can provide higher throughput values since there is no middle-ware component, so latency is reduced, but only until it reaches the saturation point.

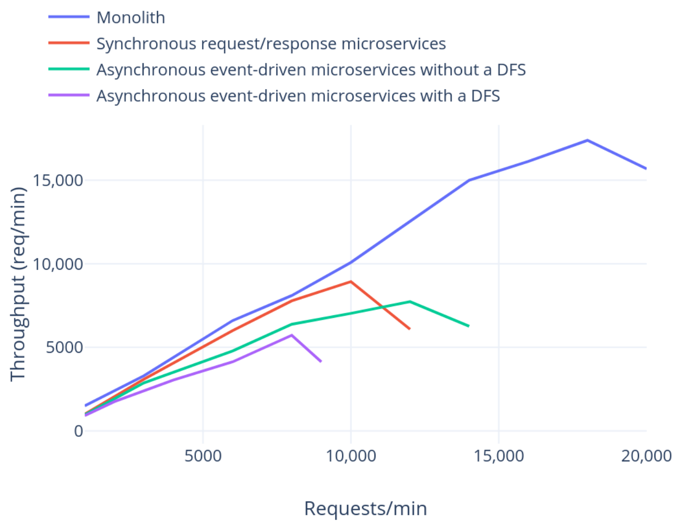


Figure 57: Throughputs of the simple workload

In regards to the throughput, the monolith once more presents the best performance results, reaching throughput values up to 18.000 requests per minute. The differences in throughput between the monolith and microservices are mainly justified by the fact that the microservices version spans multiple services in order to complete the requested operations, which adds further latency and more database accesses, while the monolith has access to every data locally.

The synchronous microservices version features the second-highest throughput value with approximately 8000 requests per minute. The asynchronous version maintains a relatively constant throughput value from 8000 up to 12.0000 requests per minute, which although is lower than the synchronous version, it supports higher load before reaching the saturation point. Similar to what happened with the compound workload, the throughput values are inversely proportional to the response time values.

Even though it is not patent during simple workloads, the several patterns introduced to improve the resilience of microservices described throughout Section 6, involve overheads that stand out on more complex workloads. If those patterns were not implemented in the asynchronous version the number of database operations per request would have been the same for both synchronous and asynchronous versions. The simple workload represents a benchmark with the inter-service communication mechanisms used as the main variable. Hence, its results suggest that for complex workloads the asynchronous version will likely reach the saturation point after the synchronous version if, for instance, the CQRS pattern was not implemented. Regardless, the patterns were applied to the asynchronous version to an, as robust as possible, microservices application following the industry best practices. All the results are also presented in the form of tables in Chapter A.

9

DISCUSSION OF THE IMPACT OF MICROSERVICES

Once the development phases from both monolithic and microservices applications are reported, it is possible to discuss and analyze the cost and value of microservices and the design patterns in the whole development process. This chapter is divided into the main influenced aspects experienced throughout the development of the applications and the practical challenges involved in assuring both functional and non-functional requirements are addressed.

The development process and methodologies used to build microservices applications that comply with its guidelines showcased the existent architectural trade-offs, which are addressed in this chapter.

9.1 OPERATIONAL COMPLEXITY

The main challenges and most impact perceived from the adoption of microservices unquestionably reside in the initial development phases. Overall, an increase in the operational complexity was perceived, showcased by the more complex programming model and the introduction of the data consistency complexity, which are unfamiliar topics coming from a traditional application development background.

With a distributed system arises the need for managing a communication infrastructure which is a complex process not featured in traditional applications. The scenario of when a service requires data from other services becomes a paradigm that requires analysis of the possible best approaches, which depends on the context it is applied. Moreover, transactions across microservices frequently require the definition of the several steps distributed across multiple services, which demands the implementation of the compensating operations that take place if any of the steps fail. Besides, due to its distributed nature, several additional components are added, for instance, an API Gateway, a Discovery Service, and observability-related components, which require further configurations and management. The costs associated with achieving loose coupling and, consequently, independent development and deployment environments have an impact on the complexity and effort to develop applications, forcing changes in the way operations are performed, how data is queried, and much more.

It is agreed that microservices feature an increase of complexity, which is stated in the existent literature (see Section 2.3.2). The unawareness resides on to what extent the increase of complexity is perceived and how the costs and values are deliberated.

One of the initial goals was to build a microservices application that achieves the promised values of such an architecture, requiring it to comply with some key fundamentals. For instance, maintaining the services decoupled, which due to the distributed nature of microservices requires the introduction of patterns (section 2.3.3) that help to assure that the services are not coupled to one another. However, the implementation of those patterns comes with additional challenges and overheads that extend over several topics, further detailed in the following sections.

9.1.1 *Data management*

With traditional applications data is typically centralized in a single database that features strong data integrity constraints, which highly facilitates the data management its integrity. With microservices, assuring that data is consistent across the services requires different ways of dealing with business operations that affect the state of the application. Given that the application context, and consequently, the data, is split across multiple services, it is usual that they may need each other to complete its operations. As stated in Section 5.2, the use of a separate database for each service is a major influential factor to further decoupling between the services, a key fundamental of microservices. However, the added overhead when data from different services is required constitutes one of the most significant drawbacks of implementing microservices. It highly contrasts with the ease of accessing any piece of data within the monolithic solution, in which everything is locally available. While querying for a given product's information required only one database call on the monolithic version, the microservices solution requires the span of at least three different services to get the exact same information. Since each service is the single owner of its data, it has to be requested from different sources and then aggregated.

On the other hand, despite the efforts of maintaining multiple databases, the end result was that changes in a service's database schema are performed transparently to the other services/development teams, which does not happen in the context of the monolithic application. This accomplishment is one step towards achieving complete isolation between the services, allowing completely separated development and deployment environments.

As a consequence of having the data and business functionalities split across different services, not all operations can be performed entirely by a requested service. Contrasting with the simplicity of implementing business operations within traditional applications, that are proceeded at a method-invocation level with database operations' integrity assured through the use of ACID transactions, microservices requires a different approach. Operations were reconstructed to fit a distributed environment, which was implemented through the use of the Saga Transactions pattern (see Section 5.2). This different paradigm of implementing business logic adds further complexity since its steps are only eventually consistent and could fail, which requires the implementation of compensating behavior.

A big part of the business operations became more complex and needed more steps compared with the monolithic version. This is more accentuated when different services are required, demanding the coordination of the steps and error handling mechanisms. Even the creation of an entity, which within a traditional application is as simple as persisting it on the only existent database, in an event-based distributed system, the same scenario generates an entity-created event which could span dozens of other services that are interested in that information. Maintaining replicas of denormalized data is a common practice within microservices, but it can easily threaten the system's data integrity if not done properly. The lack of ACID transactions can be astonishing coming from a traditional applications development background since it changes the business logic implementation paradigm and makes the programming model more complex. Dealing with data in a distributed way is another major challenge of implementing microservices.

The added complexity present in simple operations is noticeable and is an undebatable cost of implementing microservices. Performing operations using the Saga Transactions pattern almost demands the use of a message broker to avoid tight coupling, which in its turn, adds even more complexity, requiring the definition of the events, queues, and exchanges, which is further detailed in Section 5.3. Nonetheless, a message broker should be used in a microservices context, since direct communication is, in most cases, considered an anti-pattern.

As previously mentioned, it is common that services require data that is present in other services, being each database exclusively accessed by the owner service. The event-driven approach is the one that complies with the best practices, but querying for data through the publication of events is not the most efficient approach and may not suit every scenario. A commonly used alternative is for services to maintain replicas of all the data needed to fulfill their operations instead of requesting it every time it is needed. However, by replicating data owned by other services, it is necessary to subscribe to events that notify that the data has changed. To achieve that, the events, the event listeners, and handlers have to be defined, which is not only an overhead in terms of performance but also for the development of functionalities.

When a client application needs to fetch data that is distributed across multiple services, it needs to be aggregated. To avoid the need for distributed joins CQRS is implemented and introduces additional services that hold frequently joined data. These components are exclusively responsible for handling read operations and the maintenance of the data they hold, typically through the subscription of events. This pattern constitutes an approach for fetching data that resides in multiple services. However, since the additional services are composed of replicated data, it needs to be constantly synchronized whenever some entity changes its state. Assuring that every change applied to a given entity will be properly reflected in the read models is definitely an overhead that did not exist while developing the monolithic version, requiring the definition of events for every single state change of an entity. There was a significant effort put into maintaining the read models but there was not really much of an option since, as stated, other patterns feature significant issues.

The data management topic within traditional applications can be almost fully delegated to the database systems and ORM frameworks, which assure all the necessary integrity mechanisms and the concept of ACID transactions. With microservices, the data becomes distributed across different services, and the assurance of consistency is something that needs to be programmed and adds a substantial amount of effort, not experienced during the development of the monolithic solution. The simplicity in developing the same functionalities using a traditional architectural style highly contrasts with the difficult path of implementing similar behavior with microservices.

9.1.2 *Inter-service communication*

To promote the targeted isolation between the services, an event-based approach for inter-service communication is used, introducing a message broker to the infrastructure. As highlighted in Section 5.3, this communication approach requires the creation and maintenance of a complex infrastructure. By only communicating via queues and exchanges, the several subdomains of the system are less coupled when compared with the monolithic version, which favors development teams' independence in terms of both development and deployment.

However, it is crucial to emphasize the amount of added complexity to the data management when comparing with the monolithic version of the same application. Apart from being a complete change of paradigm that contrasts with the traditional way of developing applications, communicating through the use of events requires much more planning and analysis of what is the best approach in each scenario.

Defining and managing the groups of services that consume the events is a challenging process when there are several services involved. As a single developer, it is difficult to constantly keep track of all the communication that happens between the services. For instance, knowing which services are interested in which events, or which exchanges are associated with which queues. Multiple considerations require awareness because implementing such an infrastructure only for communication constitutes a huge overhead. It is also worth mentioning that Figure 25 is the communication infrastructure from a microservices-based architecture featuring only 10 services, while a typical microservices architecture can have hundreds of services.

At the cost of having the services unaware of each other, since they communicate through a middleman, eventual consistency is an outcome that requires attention. Programming-wise, the effort necessary for implementing simple tasks exponentially increased when developing for an event-driven scenario, but that only happened in the initial phases, because the application was developed from scratch and the exchanges, queues, and event listeners had to be defined all at once. That fact relates to the productivity chart made by Fowler (see Figure 5), which describes that in initial phases with less complexity the productivity when using a microservices approach is actually lower compared with its monolithic counterpart, which was indeed verified. However, as complexity increases, productivity benefits from the low coupling between services, enabling isolated development environments.

As described previously, not all cases suit an eventual response scenario. When a more sequential flow of events is necessary asynchronous communication may not be ideal. In this e-commerce context, events are used to proceed the order, which means the user could be left waiting for an eventual notification for the order to be confirmed, and only then proceed to payment. From a user experience perspective, waiting indefinitely to pay for the purchase could be something critical and ultimately cause the loss of potential clients. If a direct communication approach was used instead, it would either fail or respond within a timely fashion, which sometimes can be preferred. It immensely depends on the scenario and, as already mentioned, migrating to microservices is essentially a set of trade-offs and the outcomes in perspective have to be clear beforehand.

The use of an asynchronous approach, although favors the availability of the system, can be considered a cost necessary to achieve loose coupling, required to validate the benefits from a microservices architecture. Most of the described effort could be avoided if a synchronous inter-service communication mechanism was used instead. Hence, a microservices version featuring a direct communication approach was also implemented in which whenever a service needed data from other services it would directly request it. If the services simply interact with each other in a direct way then the development is not much more complex than the monolithic version. The effort in maintaining an event-based infrastructure and replicated data is not present in this version.

However, almost every promised benefit from the adoption of a microservices architecture is invalidated as the services are not decoupled, highly depending on the availability of each other.

All these processes described for implementing operations and accessing data are just not present within the development of a traditional monolithic application. The effort put into maintaining the communication infrastructure, defining distributed operations, and implementing the mechanisms for synchronizing replicated data constitute major overheads that should be taken into consideration.

The drawbacks described during the literature review are indeed very vague and do not convey the message of how much operational complexity is added when implementing this architectural pattern. Although the potential drawbacks are moderately known, it is extremely relevant to analyze what the associated challenges are on a deeper level, which is not possible to be described in the form of a bullet point list.

9.2 IMPACT ON QUALITY ATTRIBUTES

Allegedly microservices promote a set of quality attributes that may offer a competitive advantage, from a business perspective. The quality attributes of a system highly depend on how it is designed, and microservices characteristics provide the agility to achieve a wide catalog of them if implemented according to certain guidelines. The multiple used microservices patterns were implemented to enable it to be in accordance with the guidelines.

The different processes to achieve the quality attributes in microservices are analyzed and described in the following subsections.

9.2.1 *Maintainability and Extensibility*

One of the promised benefits of implementing microservices is the ability to maintain multiple segregated development workloads. The characteristics of a microservices architecture favor decoupling, which, consequently, promotes the maintainability of the system. A small number of links between the services enables that changes and error fixings made to one service are completely transparent to the others. To achieve isolation, the services that compose the developed applications followed a DDD approach and defined bounded contexts, which had an important role in promoting the maintainability of the system.

Since within a microservices context the business capabilities are divided into multiple granular instances, presumably independent from each other, it is possible to have separate development teams responsible for maintaining specific domains of a system. For instance, the Review Service's team can make any change to its business capabilities without even notifying the other teams. That degree of independence highly promotes productivity and flexibility within each bounded context. Given that services are smaller, its readability is improved, which makes the whole application more easily maintainable. Furthermore, due to the splitting into multiple clear domains, it is easier to keep track of the existent business capabilities. Making changes to a service knowing that it will not break anything on other services also provides confidence to the developers.

However, all of those benefits highly depend on how well delimited are the domain contexts of the services. If the services' domains overlay, then it is not possible to evolve them independently, approximating to the scenario of a monolithic application. The decomposition of the domain requires a high degree of understanding of the context in question. Once again, to reach the values of the microservices architecture, how it is implemented is a critical factor. The use of an asynchronous inter-service communication mechanism is fundamental to achieving loose coupling and maintainability, but, as stated in the previous section, there is significant complexity that comes with it.

Nonetheless, even though assuming there is isolation in the development of the services, it requires cautiousness not to overdevelop a given service. Nothing prevents it from becoming too large and complex to grasp, falling into a distributed monolith scenario, which features the same problems as a monolithic application. It would take more time to evolve a service into a complex to manage a monolithic-like application, but it is definitely a possible scenario in a long term. It is generally accepted that monolithic applications can rapidly become complex to a degree that it is not possible to be entirely understood by a single developer, which can inhibit productivity. Regardless, there are patterns to compensate for the lack of modularity and tight coupling in monolithic applications, which mainly reside in promoting separation of concerns and internal organization of the codebase.

Despite efforts being put into favoring maintainability and extensibility on the monolithic application, nothing can change the fact that it is a single deployment unit with a single deployment pipeline. Any introduced change forces the redeployment of the whole application, which is impractical. Those factors can be highly restrictive when it is necessary to have a high rate of updates that need to be frequently delivered. In that sense, the maintainability and extensibility of monolithic applications are limited. However, while maintainability is indeed more easily promoted within a microservices context due to its characteristics, it is also something entirely achievable using a traditional architecture, although it requires more organizational efforts.

Another worth mentioning limiting factor on the monolithic solution is that there is only one single database. Every write or read operation is concentrated in the single existent database. Although having a unified shared database schema highly facilitates the data management process, it introduces coupling. Within the database, there are several constraints, such as the multiple foreign keys that result in coupling between the entities, which means that any changes on the schema could impact the overall application since the whole system shares the same database layer. Performing queries to the catalog of products, for example, an SQL schema forces the joining of data in order to fetch all the information about a given product. Some scenarios, such as the one described, would benefit from more appropriate database systems, but monolithic applications are fixed to one technology stack.

9.2.2 *Availability and Scalability*

Monolithic applications typically feature only one instance whose failure compromises the entirety of the application if high availability strategies are not introduced. A crash on a monolithic application could mean that the whole application gets offline. Regarding microservices, the availability of the system is influenced by the degree of isolation between the services and, therefore, directly affected by the availability of each individual service. If the services are decoupled a failure in one of them can occur transparently. However, if services are coupled to one another, a failure of a service can compromise multiple others, resulting in a chain of failures. The less coupled are the services, the more availability is promoted in the system, enabling the continuity of operations through fault isolation. Even though asynchronous communication prevents the cascade effect, with which there are no SPOF in the system, high availability can be further improved by introducing redundancy of the running instances and automating processes.

With any architectural pattern self-healing mechanisms and high availability can be achieved, however, the strategies to implement them are distinct. Theoretically, assuring high availability in a microservices context is a more arduous process since it is composed of more components and moving parts, making it more fault-prone. However, orchestration tools proved to highly assist the process of eliminating SPOF's through the easy access to scalability, automation of self-healing mechanisms, and load balancing. Thanks to orchestration tools microservices are a more viable option given that they take care of major challenges existing in managing a distributed system.

The promised flexibility provided in terms of scalability by implementing microservices was validated, it is possible to resize the number of running instances of a service at will. By using an orchestration tool it is only needed to define the services once and then it is possible to scale each service individually in a very simple way and anytime needed, even through a dashboard. With traditional applications, the approach used is to add replicated instances of the entire application and put them behind a load balancer. Given that the entirety of a monolithic application runs within one same process, it is not possible to scale specific parts of it. This lack of flexibility oftentimes results in scaling parts of the system that did not need to be scaled.

The process of assuring high availability and automatic recovering from failures in the microservices version was fully handled by the used orchestration tool. In order to implement high availability to the monolithic application, redundancy was introduced manually by deploying two instances of the applicational server, which required the introduction of a load balancer to distribute the load, which, in its turn, also had to be replicated. Something similar would be needed for the microservices version, but the orchestration tool offers built-in features, such as load balancing, routing mesh, and an embedded DNS server.

Stateful components require more complex mechanisms to assure high availability, in which a DFS can be used to maintain the state available across multiple instances. Its introduction is not exclusive to the microservices solutions, being also implemented in the monolithic application. In order to introduce redundancy to components with state on the monolithic version a high availability cluster was introduced, while the microservices version makes use of the self-sufficient mechanisms provided by the orchestration tool.

Assuring high availability would constitute a much bigger concern and challenge within a microservices context if it was not for the orchestration tool. But in the end, the process of introducing high availability mechanisms to the entire infrastructure required less effort in the microservices version and proved to offer more scalability options and also simpler ways to achieve it.

Although the microservices version has more instances to introduce redundancy, Docker Swarm highly facilitates the process by automating a set of mechanisms that provide self-sufficient behavior right out of the box.

The biggest challenge in both approaches was finding an acceptable way of providing the resources required by the stateful services across all physical instances.

9.2.3 *Observability*

Whereas tracing errors and monitoring the state of the system is a simple process within a traditional application, microservices require an entire dedicated observability infrastructure in order to watch over the system's state. If observability techniques are not introduced to a distributed system, tracing errors and fixing them becomes a more arduous task, which is a concern not existent on traditional applications, in which log files are often enough.

To assure the system is easy to debug and maintained healthily, distributed tracing, centralized logging, and monitoring stacks are introduced. All those mechanisms require effort to implement and maintain, which represents additional overheads not strictly necessary with traditional applications.

Observability in a distributed system is one of the costs of adopting a microservices architecture. When compared with the monolithic counterpart, the importance of having proper observability techniques is distinctly greater and constitutes a way more critical topic. A microservices-based application without a robust distributed observability stack can be extremely complex to fix since it is very difficult to track the path taken by an anomalous situation and quickly perform the proper fixes.

9.3 DEPLOYMENT AND DEVOPS

By achieving independent deployability, through the compliance of best practices and assurance of low coupling between the services, the internal implementation of a service can be changed without coordinating with the others. The absence of synchronizing deliveries enables complete isolation between development teams, which are free to work on changes independently and therefore, faster. Combining with DevOps activities, which introduce automation of processes, deployment can be much more frequent and decrease the time to market, something very valuable in modern applications. This affinity with the DevOps methodologies was validated through the implementation of CI/CD processes that automate the deployment process applied to containers. The end result is that changes can be deployed into production simply through git commits, which trigger a set of operations that test, build and deploy the new Docker images, in a short period of time. Longer deployment periods come with a higher risk of issues, and the ease of deployment with this architectural style makes the deployment process less risky. Microservices characteristics make it the perfect candidate when there is demand for frequent deploys, which is something very difficult to achieve with legacy applications.

Although well-structured monolith applications can be simple to develop and maintain, the fact that there is only one shared deployment pipeline makes continuous deployment very challenging to achieve. Regardless of efforts, which can accelerate the deployment process, it cannot change the fact that any modifications to deploy in the application are scheduled within the same deployment pipeline on which everyone relies on. Releasing new versions can easily become a major productivity bottleneck in these types of applications.

9.4 PERFORMANCE

Regarding the impact on performance, microservices presented worse throughput and response time values than its monolithic counterpart across all tested workloads. The latency associated with distributed systems, the eventual response nature of the operations, and the overhead of synchronizing data showcased its impact on the benchmark results.

Besides the comparison between monoliths and microservices, multiple variations of microservices scenarios were tested. The impact of having distinct inter-service communication mechanisms, data management methodologies, and the utilization of a DFS that assures state sharing between the physical instances were evaluated. The microservices version that features all the best practices in terms of having loosely coupled services, the event-based version, was the one with worse response times overall.

Nonetheless, the performance results are in accordance with the related work analyzed described in Section 2.5, which concludes that microservices, using similar hardware, will perform worse than a monolith. Hence, performance improvement should not be a reason to migrate to microservices. However, through the easy access to scalability provided by the orchestration tools used for managing microservices, more resources can be reserved for specific services, increasing its performance on demand, which is not possible with traditional applications. The full analysis of performance results is discussed in Section 8.

9.5 SUMMARY

This chapter highlighted the most impactful costs and values experienced throughout the development of the microservices applications and how it compares with a traditional development paradigm. The majority of the theoretical values of microservices were experienced and validated, but the chapter mostly focused on the costs associated with achieving the values, which compose the typically less disclosed topic.

The additional effort present in almost every aspect of the implementation process of microservices was emphasized and showcases the necessary paradigm shift that introduces several concerns. With the introduction of patterns that grant the promised benefits comes the costs associated with its implementation. These extend to the maintenance of a whole dedicated communication infrastructure, the introduction of observability stacks, the constant synchronization of data that is eventually consistent, and much more. The challenges related to the coordination of the multiple instances without introducing coupling and the implementation of compensating behavior to every operation that involves more than one are some of the concerns exclusively present in a distributed system. Operational complexity and a far more complex programming model proved to be inevitable costs of the implementation process of microservices.

The different necessary approaches to assure the presumably promoted quality attributes require knowledge that was specifically acquired for a microservices context. Achieving the microservices benefits constitutes a complex process with considerable costs and requirements associated, but with significant values that deeply relate to the current market needs.

10

CONCLUSIONS AND FUTURE WORK

10.1 CONCLUSIONS

This dissertation emphasizes the overheads and challenges experienced throughout the development of microservices based applications. By covering the whole development process, it was possible to contribute to the lack of empirical reports that focuses on the impact associated with the implementation of microservices. The analysis of the architectural trade-offs aims to contribute to the delimitation in which circumstances a microservices architecture should be considered. By doing so, it was possible to demonstrate how microservices and the different implementation patterns impact several topics, namely quality attributes. Targeting a robust web application with the same functional and non-functional requirements for each version developed, the reporting of the development process emphasized the overheads and complexity associated with microservices.

The insights provided showcased that, in order to properly implement a microservices-based architecture that achieves its claimed benefits, several principles need to be honored, mostly related to maintaining the services decoupled and isolated. To achieve that however, specific patterns had to be implemented, so the distributed system can offer the same behavior as a monolithic counterpart. The patterns associated with maintaining both functionalities and quality attributes in a microservices context added a significant amount of complexity in multiple subjects, as repeatedly emphasized throughout this document. Furthermore, a rather different approach is required for developing microservices-based applications, which contrasts with a traditional development background. The differences mainly reside in a more complex programming model and an overall increase of complexity that demands a specific set of skills.

Not having the required knowledge leads to implementations that do not achieve the promised benefits of a microservices architecture. The analysis of the impact in quality attributes of the different implementations strategies demonstrated that solutions that do not achieve loose coupling between the services will not benefit from microservices' potential gains.

It is not straightforward to provide a concrete delimitation in which specific circumstances a microservices architecture would offer a positive ROI balance, as it deeply depends on the context applied. Nonetheless, as an attempt to contribute to the decision, the empirical data supplied throughout the dissertation provides insights and awareness of what is necessary to achieve the promised benefits of microservices, how they relate with the literature review, and the costs associated with it.

The development of a microservices application in conformity with the best practices proved to validate the claimed benefits described in the literature review conducted. It is a fact that microservices enable technology heterogeneity, offer more flexible scalability, independent deployment pipelines, availability, fault isolation and, bring easy access to DevOps methodologies. The verdict is that microservices effectively solve the challenges presented by traditional applications. Regardless, most of the described benefits highly depend on how the architecture is implemented, and that is the real cost of microservices. When a system with the characteristics of microservices is adopted, effort is required to deal with eventual consistency, observability requirements, distributed data management, data synchronization, and much more, which are concerns not addressed with traditional applications.

Furthermore, as a result of the performance benchmarks conducted, it was possible to analyze the impact on performance from using microservices and its multiple implementation patterns. The impact of the patterns extends from operational complexity to throughput values, and the solution which follows the microservices' guidelines proved to have the most negative impact on the performance results. However, the performed experiments were conducted using local machines with very specific characteristics, which obviously does not portray a typical production environment for a microservices application. The performance benchmarks could have been conducted using cloud-based solutions, featuring the latency associated with connecting to a remote server.

In summary, the decision of building a microservices-based application should preferably be highly sustained by concrete reasons. Either by challenges exhibited from an existent legacy application or when it is predicted beforehand that complexity will considerably increase over time and frequent delivery of software iterations is a certainty. Nonetheless, the reasons behind several authors' controversial statements warning about the necessary cautiousness before deciding to implement microservices have been fully comprehended. The decision of adopting microservices should always be taken with a deep understanding and awareness of what to expect, and hopefully, the insights reported through this dissertation may provide exactly that.

Additionally, the dissertation enabled the production of supplementary research work. A scientific paper focusing on the impact of different microservices implementation patterns in performance, which integrates this dissertation, has been accepted and will be presented in the International Conference on Intelligent Systems Design and Applications (ISDA) (see Chapter B).

10.2 PROSPECT FOR FUTURE WORK

Given the current lack of empirical studies on the impact of adopting microservices and its patterns, one possible plan would be to extend the developed work into more research papers. For instance, focusing on the processes and techniques used for assuring quality attributes in a microservices context, or the impact of microservices and its patterns in quality attributes. The existence of improper microservices implementations in research works showcases that the architectural style lacks established principles, which would benefit from more research on the subject. Alternatively, contributions focusing on the construction of a microservices cluster using devices typically associated with IoT, which is the case for the Raspberry Pi SBC, a cost and energy-efficient solution that proved to portray a decent scenario for the experiments.

Regarding possible extensions of the developed work, there are some topics only partially explored in this document. For instance, since the present work was conducted within a limited amount of time, the evaluation of the maintenance and extension of the applications developed in long term was not possible. This may constitute a topic of interest since the natural flow of any application is for it to grow over time, whose study would enable the evaluation of how the architectural style can impact that topic.

BIBLIOGRAPHY

[1] S. Newman, *Monolith to Microservices*. O'Reilly Media, Inc., 1st ed., 2019.

[2] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, *Microservices: Yesterday, Today, and Tomorrow*, pp. 195–216. Springer International Publishing, 2017.

[3] D. Martin, D. Kshirsagar, T. Sherer, A. Boeglin, A. Buck, N. Schonning, M. Wilson, and B. Tam, "Ci/cd for microservices architectures." https://docs.microsoft.com/en-us/azure/architecture/microservices/ci-cd, Microsoft Docs, 2019. (Visited on 08/12/2020).

[4] M. Fowler, "Microservicespremium." https://martinfowler.com/bliki/MicroservicePremium.html, 2015. (Visited on 02/01/2021).

[5] C. Richardson, "Pattern: Microservice architecture." https://microservices.io/patterns/microservices.html. (Visited on 13/11/2020).

[6] C. Richardson, "Pattern: Saga." https://microservices.io/patterns/data/saga.html. (Visited on 13/11/2020).

[7] A. Nish, D. Coulter, M. Veloso, J. Parente, M. Wenzel, N. Schonning, S. Gosh, G. Warren, and V. Youssef, "The api gateway pattern versus the direct client-to-microservice communication." https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-application, Microsoft Docs, 2021. (Visited on 08/12/2020).

[8] K. Kumar, "Microservices with cqrs and event sourcing." https://dzone.com/articles/microservices-with-cqrs-and-event-sourcing. (Visited on 15/11/2020).

[9] C. Richardson, *Microservices Patterns*. Manning Publications Co., 2018.

[10] H. Binani, "Kubernetes vs docker swarm — a comprehensive comparison." https://hackernoon.com/kubernetes-vs-docker-swarm-a-comprehensive-comparison-73058543771e, 2018. (Visited on 12/01/2021).

[11] "Digital 2020: October global statshot report," 2020. We Are Social; DataReportal; Hootsuite.

[12] "Internet growth statistics," 2000. Internet World Stats.

[13] N. Kratzke and P.-C. Quint, "Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study," *Journal of Systems and Software*, vol. 126, pp. 1–16, 01 2017.

[14] D. Taibi, V. Lenarduzzi, and C. Pahl, "Processes, motivations and issues for migrating to microservices architectures: An empirical investigation," *IEEE Cloud Computing*, vol. 4, 2017.

[15] J. Lewis and M. Fowler, "Microservices." `https://martinfowler.com/articles/microservices.html`, 2014. (Visited on 05/01/2021).

[16] E. Kappelman, "Why microservices and devops are a match made in heaven." `https://hub.packtpub.com/why-microservices-and-devops-are-match-made-heaven/`, 2017. (Visited on 05/11/2020).

[17] P. D. Francesco, I. Malavolta, and P. Lago, "Research on architecting microservices: Trends, focus, and potential for industrial adoption," in *2017 IEEE International Conference on Software Architecture (ICSA)*, pp. 21–30, 2017.

[18] S. Newman and M. Fowler, "When to use microservices (and when not to!)." `https://gotopia.tech/bookclub/episodes/moving-to-microservices-with-sam-newman-and-martin-fowler`, 2020. GOTO.

[19] A. Carrasco, B. van Bladel, and S. Demeyer, "Migrating towards microservices: migration and architecture smells," in *IWoR 2018: Proceedings of the 2nd International Workshop on Refactoring*, pp. 1–6, 2018.

[20] S. Baskarada, V. Nguyen, and A. Koronios, "Architecting microservices: Practical opportunities and challenges," *Journal of Computer Information Systems*, vol. 60, pp. 1–9, 2018.

[21] D. Perry and A. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, p. 40–52, 1992.

[22] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*. Addison-Wesley Professional, 3rd ed., 2012.

[23] M. Fowler, *Patterns of Enterprise Application Architecture*. USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[24] C. MacKenzie, K. Laskey, F. Mccabe, P. Brown, and R. Metz, "Reference model for service oriented architecture 1.0," *Public Rev. Draft*, vol. 2, 2006.

[25] J. Fritzsch, J. Bogner, A. Zimmermann, and S. Wagner, "From monolith to microservices: A classification of refactoring approaches," in *DEVOPS*, 2018.

[26] Z. Ren, W. Wang, G. Wu, C. Gao, W. Chen, J. Wei, and T. Huang, "Migrating web applications from monolithic structure to microservices architecture," *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*, pp. 1–10, 2018.

[27] P. Rodgers, “Service-oriented development on netkernel- patterns, processes & products to reduce system complexity web services edge 2005 east: Cs-3.” https://web.archive.org/web/20180520124343/http://www.cloudcomputingexpo.com/node/80883, 2005. CloudComputingExpo 2005. SYS-CON TV (Visited on 08/12/2020).

[28] N. Josuttis, *Soa in Practice: The Art of Distributed System Design*. O’Reilly Media, Inc., 2007.

[29] M. Richards, *Microservices vs. service-oriented architecture*. O’Reilly Media, Inc., 2016.

[30] T. Cerny, M. J. Donahoo, and J. Pechanec, “Disambiguation and comparison of soa, microservices and self-contained systems,” RACS ’17, p. 228–235, Association for Computing Machinery, 2017.

[31] P. Jamshidi, C. Pahl, N. Mendonça, J. Lewis, and S. Tilkov, “Microservices: The journey so far and challenges ahead,” *IEEE Software*, vol. 35, pp. 24–35, 2018.

[32] M. Fowler, “Microservice trade-offs.” https://martinfowler.com/articles/microservice-trade-offs.html, 2015. (Visited on 11/11/2020).

[33] G. Pallis, D. Trihinas, A. Tryfonos, and M. Dikaiakos, “Devops as a service: Pushing the boundaries of microservice adoption,” *IEEE Internet Computing*, vol. 22, pp. 65–71, 2018.

[34] L. Chen, “Microservices: Architecting for continuous delivery and devops,” in *2018 IEEE International Conference on Software Architecture (ICSA)*, pp. 39–397, 2018.

[35] C. Wu and R. Buyya, eds., *Cloud Data Centers and Cost Modeling*. Morgan Kaufmann, 2015.

[36] A. Amanse, “Why should you use microservices and containers?.” https://developer.ibm.com/depmodels/microservices/articles/why-should-we-use-microservices-and-containers/, IBM, 2018. (Visited on 29/12/2020).

[37] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud,” in *2015 10th Computing Colombian Conference (10CCC)*, pp. 583–590, 2015.

[38] A. Rotem-Gal-Oz, “Fallacies of distributed computing explained,” *Doctor Dobbs Journal*, 2008.

[39] P. Todkar, “How to extract a data-rich service from a monolith.” https://martinfowler.com/articles/extract-data-rich-service.html, 2015. (Visited on 09/12/2020).

[40] M. Fowler, “Microservicesprerequisites.” https://martinfowler.com/bliki/MicroservicePrerequisites.html, 2014. (Visited on 08/12/2020).

[41] C. Peltz, “Web services orchestration and choreography,” *Computer*, vol. 36, no. 10, pp. 46–52, 2003.

[42] J. Schabowsky, “Microservices choreography vs orchestration: The benefits of choreography.” https://solace.com/blog/microservices-choreography-vs-orchestration/, Solace, 2021. (Visited on 13/11/2020).

[43] G. Schmutz, “Building event-driven (micro)services with apache kafka.” Voxxed Days Zurich, 2018.

[44] C. Richardson, “Not just events: Developing asynchronous microservices.” GOTO, 2019.

[45] C. Richardson, “Pattern: Api composition.” https://microservices.io/patterns/data/api-composition.html. (Visited on 13/11/2020).

[46] C. Richardson, “Pattern: Api gateway / backends for frontends.” https://microservices.io/patterns/apigateway.html. (Visited on 13/11/2020).

[47] C. Richardson, “Pattern: Command query responsibility segregation (cqrs).” https://microservices.io/patterns/data/cqrs.html. (Visited on 14/11/2020).

[48] V. F. Pacheco, *Microservice Patterns and Best Practices: Explore Patterns like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices*. Packt Publishing, 2018.

[49] V. Marmol, R. Jnagal, and T. Hockin, “Networking in containers and container clusters,” 2015. in Proceedings NetDev.

[50] IBM Cloud Team, “Containers vs. vms: What’s the difference?.” https://www.ibm.com/cloud/blog/containers-vs-vms, IBM, 2020. (Visited on 08/01/2021).

[51] A. M. Joy, “Performance comparison between linux containers and virtual machines,” in *2015 International Conference on Advances in Computer Engineering and Applications*, pp. 342–346, 2015.

[52] R. K. Barik, R. K. Lenka, K. R. Rao, and D. Ghose, “Performance analysis of virtual machines and containers in cloud computing,” in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 1204–1210, 2016.

[53] C. Richardson and F. Smith, *Microservices From Design to Deployment*. NGINX Inc., 2016.

[54] P. Dobbelaere and K. S. Esmaili, “Kafka versus rabbitmq: A comparative study of two industry reference publish/subscribe implementations: Industry paper,” in *Proceedings of the 11th ACM International Conference on Distributed and Event-Based Systems*, p. 227–238, Association for Computing Machinery, 2017.

[55] Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, and R. Sinnott, “A performance comparison of cloud-based container orchestration tools,” in *2019 IEEE International Conference on Big Knowledge (ICBK)*, pp. 191–198, 2019.

[56] C. Sridharan, *Distributed Systems Observability: A Guide to Building Robust Systems*. O’Reilly Media, 2018.

[57] S. Li, “Understanding quality attributes in microservice architecture,” in *2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW)*, pp. 9–10, 2017.

[58] J. Ghofrani and D. Lübke, “Challenges of microservices architecture: A survey on the state of the practice,” in *ZEUS*, 2018.

[59] V. Singh and S. K. Peddoju, “Container-based microservice architecture for cloud applications,” in *2017 International Conference on Computing, Communication and Automation (ICCCA)*, pp. 847–852, 2017.

[60] O. Al-Debagy and P. Martinek, “A comparative review of microservices and monolithic architectures,” in *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 149–154, 2018.

[61] R. Flygare and A. Holmqvist, “Performance characteristics between monolithic and microservice-based systems,” bachelor’s thesis, Faculty of Computing Blekinge Institute of Technology Karlskrona, Sweden, 2017.

[62] N. Bjørndal, A. Bucchiarone, M. Mazzara, N. Dragoni, and S. Dustdar, “Migration from monolith to microservices : Benchmarking a case study,” 2020.

[63] A. Akbulut and H. G. Perros, “Performance analysis of microservice design patterns,” *IEEE Internet Computing*, vol. 23, no. 6, pp. 19–27, 2019.

[64] C. Wohlin, M. Höst, and K. Henningsson, *Empirical Research Methods in Software Engineering*, pp. 7–23. Springer Berlin Heidelberg, 2003.

[65] B. Kitchenham, L. Pickard, and S. Pfleeger, “Case studies for method and tool evaluation,” *IEEE Softw.*, vol. 12, pp. 52–62, 1995.

[66] R. E. Stake, *The art of case study research / Robert E. Stake*. Sage Publications Thousand Oaks, 1995.

[67] P. Eeles, “Capturing architectural requirements.” `http://www.ibm.com/developerworks/rational/library/4706.html`, IBM, 2001. (Visited on 02/11/2020).

[68] N. A. N. Alshuqayran and R. Evans, “A systematic mapping study in microservice architecture,” *Proceedings of the 2016 IEEE 9th Inter-national Conference on Service-Oriented Computing and Applications(SOCA)*, vol. 22, p. 44–51, 2016.

[69] M. Fowler, “Monolithfirst.” `https://martinfowler.com/bliki/MonolithFirst.html`, 2015. (Visited on 02/02/2021).

[70] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[71] R. C. Martin, “Design principles and design patterns,” *Object Mentor*, vol. 1, no. 34, p. 597, 2000.

[72] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

Part III

APPENDICES

A

DETAILS OF RESULTS

A.1 BENCHMARK RESULTS FROM THE COMPOUND WORKLOAD

Table 7: Benchmark results obtained from the Monolith during the compound workload

Requests/ min	**Response Time (ms)**	**Throughput (req/min)**	**CPU Usage (%)**	**Memory Usage (%)**
500	88	498	18	12
1000	91	996	24	13
1500	118	1501	26	12
2000	132	1998	27	14
2500	133	2520	30	14
3000	420	2992	33	17
3500	1592	2478	47	17

Table 8: Benchmark results obtained from the Synchronous request/response microservices during the compound workload

Requests/ min	**Response Time (ms)**	**Throughput (req/min)**	**CPU Usage (%)**	**Memory Usage (%)**
500	120	474	22	18
1000	185	960	24	18
1500	240	1380	32	19
2000	310	1920	36	20
2500	620	2430	38	21
3000	1500	2001	50	24

Table 9: Benchmark results obtained from the Asynchronous event-driven microservices without a DFS during the compound workload

Requests/ min	**Response Time (ms)**	**Throughput (req/min)**	**CPU Usage (%)**	**Memory Usage (%)**
500	202	424	20	18
1000	250	902	29	22
1500	340	1201	42	23
1800	1200	1157	56	25
2000	2134	998	63	26

Table 10: Benchmark results obtained from the Asynchronous event-driven microservices with replicated services during the compound workload

Requests/ min	**Response Time (ms)**	**Throughput (req/min)**	**CPU Usage (%)**	**Memory Usage (%)**
500	170	444	25	20
1000	211	910	35	24
1500	290	1321	50	25
1800	900	1297	66	26
2000	1934	1182	71	27

Table 11: Benchmark results obtained from the Asynchronous event-driven microservices with a DFS during the compound workload

Requests/ min	**Response Time (ms)**	**Throughput (req/min)**	**CPU Usage (%)**	**Memory Usage (%)**
500	231	419	37	21
800	1200	324	94	24
1000	2002	292	98	25

A.2 BENCHMARK RESULTS FROM THE SIMPLE WORKLOAD

Table 12: Benchmark results obtained from the Monolith during the simple workload

Requests/ min	Response Time (ms)	Throughput (req/min)	CPU Usage (%)	Memory Usage (%)
1000	56	997	17	11
6000	81	6010	30	13
8000	96	7926	38	12
10000	110	10021	42	13
14000	280	15001	51	15
18000	321	18101	54	16
20000	1962	16212	57	17

Table 13: Benchmark results obtained from the Synchronous request/response microservices during the simple workload

Requests/ min	Response Time (ms)	Throughput (req/min)	CPU Usage (%)	Memory Usage (%)
1000	60	1010	19	13
6000	143	6011	49	17
8000	176	7927	70	19
10000	202	8434	81	19
12000	1411	5958	93	20

Table 14: Benchmark results obtained from the Asynchronous event-driven microservices without a DFS during the simple workload

Requests/ min	Response Time (ms)	Throughput (req/min)	CPU Usage (%)	Memory Usage (%)
1000	79	999	20	20
6000	211	5118	42	23
8000	288	5828	50	22
10000	382	7621	55	23
12000	455	8026	61	23
14000	1490	5998	66	24

Table 15: Benchmark results obtained from the Asynchronous event-driven microservices with a DFS during the simple workload

Requests/ min	**Response Time (ms)**	**Throughput (req/min)**	**CPU Usage (%)**	**Memory Usage (%)**
1000	101	888	28	28
2000	130	1882	31	29
4000	182	3828	44	30
6000	261	4855	67	31
8000	812	5127	81	32
9000	1712	4911	94	33

B

PUBLICATIONS

B.1 PERFORMANCE EVALUATION OF MICROSERVICES FEATURING DIFFERENT IMPLEMENTATION PATTERNS

AUTHORS: Leandro Costa and António Nestor Ribeiro

TITLE: Performance evaluation of microservices featuring different implementation patterns

CONFERENCE: 21st International Conference on Intelligent Systems Design and Applications (ISDA)

ABSTRACT: The process of migrating from a monolithic to a microservices-based architecture is currently described as a form of modernizing applications. The core principles of microservices, which mostly reside in achieving loose coupling between the services, highly depend on the implementation approaches used. Being microservices a complete change of paradigm that contrasts with the traditional way of developing software, the current lack of established principles often results in implementations that conflict with its alleged benefits. Given its distributed nature, performance is affected, but specific implementation patterns can further impact it. This paper aims to address the impact that microservices-based solutions, featuring different implementation patterns, have on performance and how it compares with monolithic applications. To do so, benchmarks are conducted over one application developed following a traditional monolithic approach, and two equivalent microservices-based implementations featuring distinct inter-service communication mechanisms and data management methodologies.

KEYWORDS: Microservices, Monolithic, Inter Process Communication, Distributed Systems, Performance evaluation.

STATE OF PUBLICATION: Accepted