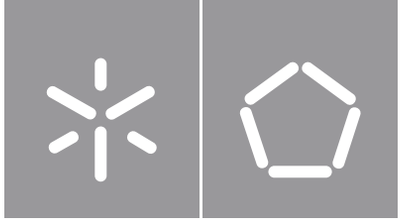
Universidade do Minho
Escola de Engenharia

André Filipe de Sousa Borges

Microservices Development and Deployment

Microservices Development and Deployment
André Filipe de Sousa Borges
UMinho | 2020

june 2020



Universidade do Minho
Escola de Engenharia

André Filipe de Sousa Borges

Microservices Development and Deployment

Master Dissertation
Integrated Master in Engineering and Management of Information Systems
Microservices

Work done on the orientation of
Professor Jorge de Oliveira e Sá
Engineer Pedro Miguel Gomes Mendes

june 2020

Direitos de autor e condições de utilização do trabalho por terceiros

Acknowledgements

This dissertation represents the culmination of 5 years of learning, continuous growth, experiences, and achievements that are widely reflected here. For this to be able to happen, I always had the support of several people to whom I address my special gratitude:

To Professor Jorge Sá for the support and advice during the development of both this master dissertation and article.

To Primavera Business Software Solutions, for the opportunity provided with this internship. A special thanks to Mr. Engineer Pedro Mendes for all the availability and time spent with me, and all the support, knowledge, advice, and assistance provided, without which the result obtained would not be possible.

To all my family, especially my parents and brother, who always supported me, believed in me, and were by my side during all the stages of my life. Also for all the fighting they had to make to allow this journey to be possible.

To all my friends, not only the ones that this journey allowed me to create but also to the ones that have been with me since my childhood. Thanks to them for always being so supportive, kind, motivational, and there for me.

To Soraia, not only for being the best girlfriend I could never ask for, but also for all the unconditional love, support, and motivation during all my achievements, including the development of this dissertation. Moreover, thanks for all the hard work put into being my personal mentor and to cross the finish line by my side. Summer has finally arrived.

Statement of integrity

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Abstract

Microservices is an architectural style inspired by service-oriented computing that has recently started gaining popularity and has been showing remarkable results in the development of large-scale systems. This new era of large-scale digitization has brought with it complex challenges regarding the scalability, efficiency, and effectiveness of the information systems and infrastructure that supports it, which the Monolithic Architecture (MA) has been unable to overcome, at least in scenarios where systems reach a considerable size. Microservices address how to build, manage, and evolve architectures out of small, self-contained services that perform single functions and collaborate with other services using a well-defined interface. This dissertation was developed in the context of an internship in a company that is giving its first steps in the Microservices Architecture (MSA) world. Not only a conceptual analysis of the state of the art in the field arises but also the idealization and improvement of the MSA currently existing in the company. Its goal is to emphasize the importance of analyzing the concepts, principles, and patterns behind the MSA to be able to make correct decisions, thus achieving success in the process of changing from MA to MSA. It also defines what software architecture is and explores other architectural approaches, like the MA and the service-oriented architecture, to compare the MSA with them. It tackles not only the advantages but also the challenges inherent to each architecture to set up a comparison between them, showing that, in certain situations, the monolithic, modular monolithic and service-oriented architecture remains preferable. Various topics, and technologies associated, that influence or are influenced by the MSA were taken into consideration and investigated, such as Continuous Integration, Domain Driven Design, Documentation, and other topics. Moreover, the company needed research in some specific topics related to microservices that were either not implemented or needed some further research and development. Some of them had different technologies with which they could be implemented. For those, a study of the plethora of technologies available to a certain theme, and a comparison to decide each of them better suit the company, was conducted.

Keywords: Microservices Architecture, Software Architecture, Software Deployment, Software Development, Software Engineering.

Index

Figure Index

Table Index

Acronyms

ACI Azure Container Instances.

AKS Azure Kubernetes Service.

API Application Programming Interface.

AR Action Research.

CCP Common Closure Principle.

CDEL Continuous Delivery.

CDEP Continuous Deployment.

CI Continuous Integration.

CLI Command-line Interface.

CM Configuration Management.

CPU Central Processing Unit.

CQRS Command Query Responsibility Segregation.

DDD Domain-Driven Design.

DevOps Development Operations.

DP Design Patterns.

DRY Don't Repeat Yourself.

DSR Design Science Research.

ESB Enterprise Service Bus.

HTTP Hypertext Transfer Protocol.

IaaS Infrastructure as a Service.

IDE Integrated Development Environment.

IS Information Systems.

IT Information Technology.

JSON JavaScript Object Notation.

JWT JSON Web Token.

K8S Kubernetes.

MA Monolithic Architecture.

MMA Modular Monolithic Architecture.

MSA Microservices Architecture.

MVC Model View Controller.

OAI OpenAPI Initiative.

OAS OpenAPI Specification.

OOD Object-Oriented Design.

PaaS Platform as a Service.

Primavera BSS Primavera Business Software Solutions.

RPC Remote Procedure Call.

RPI Remote Procedure Invocation.

RSS Really Simple Syndication.

SaaS Software as a Service.

SDK Software Development Kit.

SOA Service-Oriented Architecture.

SOAP Simple Object Access Protocol.

SOC Service-Oriented Computing.

SRP Single Responsibility Principle.

TCP Transmission Control Protocol.

TFS Team Foundation Server.

TLS Transport Layer Security.

TOGAF The Open Group Architecture Framework.

UI User Interface.

UML Unified Modeling Language.

URL Uniform Resource Locator.

UX User Experience.

VM Virtual Machine.

WSDL Web Service Definition Language.

Glossary

API First Design is a pattern that consists of designing a service, having as the main construction basis the interface and the contract made with the Service Consumer. It is used to emphasize that the most important is what the service requires.

Boilerplate Code is the term used to designate sections of code that have to be included in many places with little or no alteration.

Bounded context in software development represents the applicability of a model to a given context, defining its tangible boundaries of applicability.

Brown Field is a commonly used IT term, borrowed from the building industry, that refers to the implementation of new systems to resolve IT problem areas while accounting for existing systems.

Common Closure Principle defines that classes within a released component should share common closure. In other words, if one needs to be changed, they all are likely to need to be changed. What affects one, affects all.

Container in software deployment is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another.

Conway's Law is an aphorism in IT that posits the idea that organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.

Green Field is a commonly used IT term, borrowed from the building industry, which refers to the installation/implementation of an IT system where previously there was none.

Head-of-line blocking is a performance-limiting phenomenon that occurs when a line of packets is held up by the first packet.

Loosely coupled in software development represents a loose connection between software components, making them as independent as possible. They have very "separated" code, where every part of the code communicates with other parts through more-or-less standardized and neutral interfaces.

Multiplexing is a technique used to combine and send multiple data streams over a single medium.

SCRUM is an agile framework to address complex adaptive problems, while productively and creatively developing, delivering, and sustaining products of the highest possible value.

Separation of Concerns in software development is a design principle for separating a computer program into distinct sections such that each section addresses a separate concern to avoid co-locating different concerns within the design or code.

Service Consumer is the name given to the consumer of a service. In the MSA context, typically a microservice is both the consumer and provider.

Service Level Agreement in software development is a commitment made by an IT service provider to a customer to ensure the delivery of the agreed service quality. In the context of the MSA it is the contract established between the Service Consumer and the Service Provider.

Service Provider is the name given to the provider of a service. In the MSA context, every microservice is a service provider, otherwise, it would not be a microservice.

Single Point of Failure is a part of a system that, in case of failure, will stop the entire system from working.

Single Source of Truth in software development describes a unique state of being for company knowledge. All company knowledge is stored and managed centrally and is arranged and delineated in a way that avoids all overlaps and duplication.

Tightly coupled in software development represents a strong connection between software components, making them dependent on each other. It makes code highly cohesive, where every part of the code makes assumptions about the other parts.

1 Introduction

1.1 Contextualization

The technological advance that has been witnessed in the last decades has led to a great change in the way we think and develop systems and software. Nowadays, any organization, independent of its business area, needs to realize that to better serve and deliver more value to its customers in a competitive way, it must embrace the world of Information Technology (IT). Those that avoid this adoption tend to be unable to compete with the rest of the market and will eventually become irrelevant. However, this is not enough to keep up because if organizations want more growth, they have to invest in technologies that give them a market advantage.

It is well known that the digital world is increasingly present in our daily lives. We are facing a new era in the age of digitalization, and, consequently, if organizations want to explore this market and be competitive, they have to overcome some challenges. They are, for example, the additional complexity in software integration, delivery, and deployment, and they start suffering from dependency hell. One of the biggest challenges is scalability, which, motivated by this growth in the digital world and users, tends to be more common and needs improved solutions. One of the solutions is to adopt and take advantage of cloud computing.

Cloud computing is a technology that has been improving a lot and it is probably the one that had more impact in the organizations over the past decades. It offers numerous advantages for data and software sharing and thus making the management of complex IT systems and infrastructures much simpler. Organizations lose the need to have in-house servers and win the ability to, more easily, start to distribute their product, for example, as a Software as a Service (SaaS) or a Platform as a Service (PaaS) instead of on-premises.

It is from these needs and requirements that the Microservices Architecture (MSA) emerged. In software engineering, architecture is what allows systems to evolve and provide a certain level of service throughout their life cycle. It is concerned with providing a bridge between system functionality and requirements for quality attributes that the system has to meet (Richardson, 2018). Over the past decades, software architecture has been thoroughly studied. As a result, software engineers have come up with different ways to compose systems that provide broad functionality, satisfy a wide range of requirements, are fault-tolerant, highly available, scalable, and replicable systems that support this new digital world and the highest expectations of customers globally.

The MSA is an architecture that has been originated from the Service-Oriented Computing (SOC) and it

can be defined as a set of small, autonomous, and Loosely coupled services, each of which is independent and should implement only a single business capability or domain. It is important to note that the word micro in microservices does not concern the size or complexity of the service, but rather the scope of the domain to which it relates. It is expected that a microservice only concerns a specific functionality and performs it excellently, thus following the developmental ideology Single Responsibility Principle (SRP) (Martin, 2003).

To understand how the MSA was born, it is important to explore the Monolithic Architecture (MA) first. Classically, the type of software architecture that was, and in most cases still is, most commonly used in organizations' systems is the MA. It has only one multi-module codebase, that is composed of either technical or business features, which leads to one build of the system, which compiles the entire application, thus creating only one executable or binary.

These characteristics make initial development simpler. However, as the system increases in functionality and, consequently, in complexity, it becomes increasingly costly in terms of time and money to add new functionalities or changes. Even a small change on the codebase will force us to build and deploy the application again. This problem makes system changes much more complex and makes the product unable to keep up with the speed required in the IT market. The challenges and problems arising from the MA are medium/long term. They affect Continuous Delivery CDEL, limit the scalability of the solution as well as the change to it. Then, some difficulty arises to change the product in an agile manner, given the constant changes in market expectations.

Service-Oriented Architecture (SOA) is another key concept that is important to know since MSA is considered to be SOA done right and both of them are originated from the SOC. All key characteristics of SOA are transposable to MSA, and the two architectures are not necessarily competing with each other, but they differ in several ways. In short, MSA is a lightweight evolution of SOA.

This inefficiency in large systems, whose need for scalability is surgical, and the MA, created the need for an architecture like the MSA. Moreover, allied with new virtualization technologies, network communication, and cloud services, it is the perfect choice for the new era of digitalization. After all, this architecture has emerged in large companies, which are already implementing it, even before it was dubbed, like Google, Netflix, Amazon, Microsoft, among many others.

The development of microservices is based on a set of principles. Thus, the more aligned and refined the adoption of these principles, the more hypothetically successful the process of moving from an MA-based application to an MSA. It is very important to take these principles into account in the implementation process, as, according to many architecture experts, in the long run, ignoring some of these can have

major implications for managing the resulting solution. It may ultimately become more difficult to manage the microservice solution compared to its monolithic predecessor. According to Newman (2015), these principles are business domains-based modeling, automation culture, hide internal implementation details, decentralization of decision-making power and accountability, independent deployment, isolate fault, and highly observability.

Another important aspect of the architecture implementation is the correct choice of patterns to adopt. It is not always easy, as some patterns have major implications and the choice of one may lead to the exclusion of another. Implementing a pattern can also imply acceptance of the eventual inherent consistency of, for example, asynchronous communication. It is easy to conclude that the wrong choice of these standards can have negative consequences on the resulting application/system.

Lastly, regarding the contextualization of the MSA, it is only necessary to mention what are its main advantages and disadvantages/challenges.

In terms of advantages, the main feature is that each service has a different codebase. This will allow each service to use different languages, technologies, and frameworks while being able to communicate with each other through well-defined interfaces, thus creating an internal abstraction barrier for other services. Moreover, we can talk about the agility of software development, which in turn enhances Continuous Integration (CI), Continuous Deployment CDEP, and CDEL, refined and autonomous scalability. Implementing this architecture is the perfect preparation for migrating a system or application to the cloud.

The disadvantages and challenges are also important to take into consideration. Some stemming from the poor implementation of the architecture, while others are unavoidable. This architecture is unequivocally based on the context of distributed programming, so most of its complexities are attributed to it. Besides, there is additional work on fault tolerance and monitoring, especially if automation is not properly established. The tests are more complex, and there is a greater need for integration tests. They go beyond running them on a single machine, as it is necessary to test on the network and simulate the real environment as much as possible (this is also an advantage, as testing validates applications against a real context). Subsequently, we have an inherent organizational change in the sense that it implies a time and cost expense.

In the long term, it should be noted that the advantages are expected to outweigh the disadvantages, the larger and more complex the application concerned, and its need for future growth. However, in some cases, the adoption of this architecture may not be justified.

1.2 Background and motivation

Nowadays, there is a great tendency for organizations to move their solutions to the cloud. Most of these organizations still have MA-based solutions. Plenty of them are transitioning to the MSA because it enhances the power of the cloud. The MSA is a trend in the software development industry for building complex systems and applications (especially when involving the integration of multiple components or systems) that try to focus on building single-function modules with well-defined interface and operations. This trend is also due to the increased use of digital platforms in our daily lives, and these platforms now require high availability with a tendency to increase.

To be able to transition from the MA to the MSA, most of these organizations are going for the Brown Field approach, which is the process of transforming MA-based solutions in MSA. As Newman (2015) points out, this approach is more advantageous because there is more stability and knowledge of the business, and structure within the organization to split their MA into services. However, this transition is not always justified, as migrating from an MA to an MSA solution has an enormous cost and this change should only be made when the cost of managing the former is greater than the cost of planning, implementing, and managing the latter. This is one of the reasons why some organizations have not still adopted the MSA.

There is a growing need for software organizations to be more agile, faster, and dynamic in developing their products so that they can reach the market incrementally and continuously. In this regard, the MSA promotes a better division of responsibilities and autonomous work, resulting in smaller and self-sufficient teams responsible for ensuring the reliability of its services. This factor is perfectly aligned with SCRUM agile project management standards and Conway's Law (Conway, 1968).

Thereby, MSA is nothing less than a new way of building systems that better enables change, enhances team agility, and quickly adapts the solution to real-time consumer needs. It is also an architecture that comes along with the definition of cloud, which deals with a large part of the implementation of microservices, in a transparent way for the user. Hence, when we hear the term 'migrate to the cloud' these days, there may be inherent, a process of transforming today's applications into an MSA, which when in tune with the tools provided by the cloud, is streamlined and facilitated. This contribution is mutual, meaning the cloud facilitates the implementation of MSA and, in turn, an application that follows MSA will have low maintenance costs on the Cloud (using Cloud to publish MA solutions has both implications on costs as well as scalability). The cloud encourages the process of switching MAs to MSAs because the intuition that comes from using it points to service-based solutions.

The practical component of this dissertation took place in the organizational context, at Primavera

Business Software Solutions (Primavera BSS). It is a company that develops business management software and has been implementing the MSA to facilitate the integration of its various products with each other and external systems, with a particular focus on reusing similar features and components across different products.

The goal of this dissertation is to explore and implement technologies and algorithms that facilitate the construction of this MSA and its management on the Microsoft Azure platform for a potential universe of 30K+ customers. It uses Microsoft Azure platform as its cloud platform for numerous reasons, the main being that they have a partnership with Microsoft and mainly use their stack. So, the price plan is customized to be lower, and the support is always available. Additionally, they are one of the biggest and most reputable cloud platforms that offer an immense number of cloud-related services with an exceptionally capable cloud infrastructure. Moreover, they already have plenty of solutions to help companies develop and deploy their MSAs and related topics, like continuous deployment, databases, and monitoring tools.

Primavera BSS is a Microsoft Gold Partner, and its strategic requirement is to, preferably, adopt technology based on Microsoft's products, it was essential to align the research, development, and implementation of services associated with some of its technologies. However, they have also been investigating technologies from other sources, which proved relevant.

Academically, it intends to demonstrate what software architecture is, discuss the most used ones, their advantages, and disadvantages, and compare them. Moreover, the importance of principles, patterns, and concepts at an abstract and architectural level, as well as the materialization in technology, are going to be explored for the MSA. Concerning the organizational context in which this study is inserted, it is intended to contribute to the existing architecture in themes still under development or in need of study. These contributions will be supported by the literature review and best practices transmitted by it to create a high-quality output to the company. Therefore, in the light of the above, the interests of all parties involved are perfectly aligned and framed.

1.3 Purpose and objectives

Considering that Primavera BSS is giving its first steps in the MSA world, they need to explore new technologies and techniques for the first time. On the other hand, some were already implemented but need some improvements or restructuring. Thus, to help to build and to improve Primavera BSS's MSA from its current state, we are going to follow this plan:

- Understand the current reality about microservices and its state of the art;

- Explore the different techniques and best practices;
- Search different opportunities for improvement in the company and their MSA and associated concepts;
- Explore different algorithms that facilitate the construction of the MSA, its management on the cloud, and the implementation of some of them;
- Highlight the importance of architectural ideals and the decisions to be made in the process to achieve the expected benefits;
- Develop a functional prototype to exhibit the technology or concept, that has been investigated, to the team responsible for implementing new technology in the company;
- Produce a guideline to document the steps and best practices that the company needs to follow to successfully implement the presented technology.

More specifically, for each step proposed on the plan, the focus will be to do a study of the state of the art and to analyze and compare it with the current state of the company's MSA. The opportunities for improvement should be implemented with the point of defining and presenting a final approach for implementation on the Primavera Lithium Platform. Lithium is the Software Development Kit (SDK) for developing microservices by using a modeling framework that automatically generates code. By generating the microservice skeleton, it can speed up their development and create a standard among them.

Given the character of the project, this dissertation is composed of two complementary parts, the first is the literature review, and the other is the practical component. To underlie the background of the last component, it is necessary to assemble scientific knowledge and research about all the concepts and subjects related to the project. For this reason, the plan proposed will function to accomplish all the defined objectives.

For the literature review, two objectives are delineated:

1. Reflect and research not only MSA but also MA and SOA, and concepts surrounding it. Thereby, to better understand the purpose of MSA and the difference between all these architectures;
2. Study all the proposed topics by Primavera BSS as well as other relevant IT concepts linked to both topics and architectures.

On the other hand, the practical component has more specific objectives:

1. Investigate the current state of Primavera BSS's MSA. The goal is to get to know the internal structure of a microservice developed by the company and explore the SDK as a way of generating code and creating the company's microservices. It is also expected to identify MSA patterns and principles already implemented;

2. Explore the concept of API First Design and implement it using the OpenAPI Initiative (OAI) as well as other ways of documenting microservices, namely the microservice canvas. Primavera BSS was avoiding the production of this documentation to not promote the direct use of the Application Programming Interface (API) at the expense of the client lib. However, the company is needing this documentation and it became required in each microservice to help developers and users;

3. Research about the gRPC, its current state of the art, best practices, advantages and disadvantages, and determine if it is a good adoption for Primavera BSS's MSA. It is a framework that is growing a lot in the microservices' world, and the company wants to know if it is worth implementing it and if it is a good alternative to Hypertext Transfer Protocol (HTTP) APIs or even if both can be used in parallel;

4. Study the key concepts surrounding microservices deployment and analyze the main technologies to implement it, giving emphasis to containerization and pointing its advantages and disadvantages. It is a new technique that has been revolutionizing the deployment of applications and, especially, microservices because it already implements certain principles and patterns just by the way it works. It is gaining plenty of adoption in the past years, however, Primavera BSS still deploys its microservices as web apps and does yet take advantage of this technique;

5. Research the concept of configuration management and its current state of the art and best practices. This is a simple concept that Primavera BSS still has not looked into, which can improve a lot the settings management for their microservices;

6. Investigate the concept of webhooks, its advantages, disadvantages, and current state of the art as an efficient way to solve a limitation that APIs present. The company has postponed the implementation of this functionality, in their microservices, due to the development of an in-house alternative to it, but the situation became unsustainable, and now they need it. Primavera BSS needs a lightweight mechanism (in the SDK) to automatically add this functionality to a microservice and manage all the events.

There is the possibility that the topic or technology explored is not going to be implemented because, as it is going to be concluded, it might not fit the current state of the MSA or its future vision.

1.4 Methodological approach

The purpose of this section is to determine and characterize the methodology used in this dissertation. The two main approaches are the Design Science Research (DSR) for Information Systems (IS) IS (Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007) and the Action Research (AR). The first refers to the aggregation of knowledge acquired through a literature review and relevant theoretical concepts to construct a new reality instead of explaining an existing one or helping to make sense of it. While the latter seeks iterative changes through the simultaneous process of taking action and doing research, which is linked together by critical reflection. It can also be viewed as a type of research for practitioners to acquire and help in their field of work to solve a problem (Rowell, Polush, Riel, & Bruewer, 2015).

After an analysis of both research methodologies, and given the nature of this dissertation allied to the work in the organizational context, we can conclude that the best approach for this dissertation is to act in accordance to the AR.

One of the most widely adopted definitions of AR made by Rapoport (1970) says that it aims to contribute both to the practical concerns of people in an immediate problematic situation and the goals of social science by collaboration within a mutually acceptable ethical framework.

According to this definition, the AR has a dual goal of contributing to practice and research at the same time. It also states that there is a concrete client involved. As a consequence, the AR is highly context-dependent while attempting to address the specific client's concerns. It focuses on the fact that there is a client/organization involved, so in a dissertation that follows this methodology, it is expected that the focus will be on the definition and resolution of the problem within the organizational scope (Iivari & Venable, 2009).

After analyzing the several variations of AR, the one that suits better this dissertation is the Canonical Action Research (CAR) (Davison, Martinsons, & Kock, 2004). The CAR is known for being iterative, rigorous, and collaborative, with the basic objectives coming from the AR, which aims to address organizational problems while at the same time contributing to scholarly knowledge. Also, it is a growing prominence in the IS discipline. Its iterative feature has implied a cyclical intervention process, which can have one or more iterations, depending on the complexity of the problem at hand. Normally, it is expected to have more than one iteration, since it is from these cyclic executions that comes the most refined redefinition of the problem and, consequently, of the solution.

Thus, this methodology has two key components. One is the iteration of well-planned and executed task cycles so that researchers can both refine the problem and, as a result, the solution. Another one is the continuous prediction of the problem, which means that the tasks to be performed are always improved against the problem as it is currently defined.

The collaborative nature of this variation of the AR methodology implies that researchers and clients / organizations work together, based on their competencies and due obligations with aligned objectives, which translate into solving the defined problem.

Given this nature, it is easy to see that the researcher does not have complete control over the entire process, being dependent on the employees of the organization, or the client. Hence, a meticulous definition of planning is not possible because it is constantly submitted to change and useful artifacts should be produced for the organization, as well as the knowledge generated for the scientific area in question.

The CAR methodology is based on five principles, namely the principle of:

1. The Researcher–Client Agreement (RCA);
2. The Cyclical Process Model (CPM);
3. Theory;
4. Change through Action;
5. Learning through Reflection.

The definition of AR by Rapoport (1970) emphasizes the first principle.

The second principle, the (Susman & Evered, 1978) CPM, consists of the diagnosis, planning, implementation, and evaluation of the action and specification of learning that emerged from the whole process. (McKay & Marshall, 2001) summarize other cyclical models and then propose the dual model, where the research and problem-solving cycles run in parallel and interact with each other. That is, concerning this dissertation, the most conceptual investigation of MSA and all the concepts surrounding it will be applied to the concrete problem to be solved in the organization.

Concerning the third principle, as understood by Davison et al. (2004), it suggests that researchers using AR need theory to align and focus their activities. This is the main of the literature review.

The principle of change through action attempts to ensure that diagnosing the problem, action planning, action-taking, and evaluating is appropriately done so that one can expect an improvement in the client's problematic situation (Davison et al., 2004).

Lastly, the principle of learning through reflection, attempts to ensure that both researchers and the client examine what they have learned in an explicit, systematic, and critical manner (Davison et al., 2004).

According to Davison et al. (2004), the cyclic process model of the AR methodology consists of five stages:

1. Diagnosis - Consists of researching the scientific area, the client, and their expectations and the problem to be solved;
2. Planning - Actions are defined based on current circumstances. This step is formulated based on the researchers' assumptions on what may be the possible ways to solve the problem;
3. Intervention - This is the stage where the planned actions are carried out;
4. Evaluation - Lessons are drawn about the interventions performed, in the light of the theoretical hypotheses formulated;
5. Reflection - Implies the dissemination of the knowledge acquired to all participants in the process, and then the next iteration will be considered.

This cyclic process model can be seen in Figure 1.

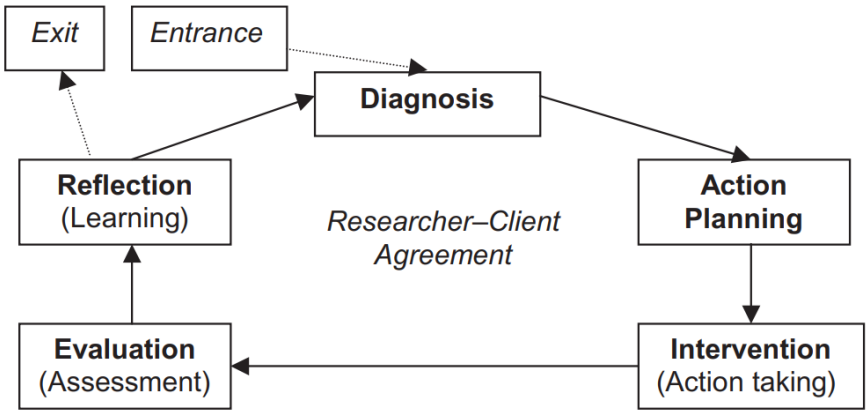


Figure 1: CAR cyclic process model (Davison et al., 2004)

In the context of this dissertation, the diagnosis was carried out at the beginning of the internship at Primavera BSS, where the main problems to be solved were identified. Afterward, the functional requirements were specified, and the academic and organizational interests were aligned to produce a project plan for the whole research and development. Once the plan was developed, the intervention was started by developing the literature review to create scientific knowledge for a practical component basis. After

there was a good comprehension of a certain topic, it was developed, taking into account its concepts, state of the art, and best practices. The evaluation was then made by the company to assess if the deliverable fulfilled its delineated objectives. Lastly, this dissertation, all the prototypes, guidelines, and presentations developed for the company were the reflection. There was a new iteration of some of the steps, in some circumstances, for example, when the prototype developed and the functional requirements were not fully aligned.

1.5 Document structure

The present dissertation is structured in four chapters, as follows:

Chapter 1 - Introduction: Presents the theme of the dissertation and contextualizes it. It gives a background and motivation and describes its main purposes and objectives, as well as the alignment with the work done at Primavera BSS. Moreover, it specifies the methodology of research used and the structure of the document.

Chapter 2 - Literature Review: Focuses on the literature review of MSA and all related concepts, technologies, principles, and design patterns. It also defines what software architecture is and explores other architectural approaches, like the MA and the SOA, to compare the MSA with them. Subsequently, the advantages and disadvantages of each were pointed out and a comparison between them was established.

Chapter 3 - Literature Review: Studies all the proposed topics by Primavera BSS as well as other relevant IT concepts linked to both topics and architectures.

Chapter 4 - Primavera's microservices architecture: Firstly describes the current Primavera BSS MSA and then explores each of the proposed themes by the company. In the latter, for each theme, we investigate how to implement them and give some best practices to apply in the company. For the ones that have more than one technology available for it, a comparison, at both technological and conceptual levels, was made between the most relevant ones in the studied field. Furthermore, for the most complex themes that require an architecture, one was idealized, developed, and tested in the company to propose a mature architecture to solve the problem presented.

Chapter 5 - Conclusion: Exposes our last considerations regarding this dissertation and the Microservices topic in general. Moreover, it contemplates the conclusions obtained with this study, difficulties felt and future expectations.

2 Literature Review

The literature review represents one of the most important stages in scientific research and work foundation for research in IS. As such, review articles are critical to strengthening IS as a field of study and, a review of prior, relevant literature is an essential feature of any academic project. An effective review creates a solid foundation for advancing knowledge. It facilitates theory development, closes areas where a plethora of research exists, and uncovers areas where research is needed. A high-quality review is complete and focuses on concepts. Additionally, it covers relevant literature on the topic and is not confined to one research methodology or one geographic region. Not to mention, one should not limit the study of the literature by drawing from a small sample of journals, even if they are reputable top journals, and should, instead, search by topic across all the relevant journals (Webster & Watson, 2002).

According to Webster and Watson (2002), a systematic search should ensure that we accumulate a relatively complete census of relevant literature. We can gauge that the review is nearing completion when we are not finding new concepts in your article set. Taking that into account, they recommend the following structured approach to determine the source material for the review:

1. Keywords search - The major contributions are likely to be in the leading journals. It makes sense, therefore, to start with them. While journal databases accelerate the identification of relevant articles, scanning a journal's table of contents is a useful way to pinpoint others not caught by your keyword sieve. Selected conference proceedings should also be examined, especially those with a reputation for quality. Because IS is an interdisciplinary field that extends to other disciplines, when reviewing and developing theory, we must look not only within the IS discipline but also outside the field to create a multidisciplinary approach.

2. Backward search - The goal is to go backward by reviewing the citations for the articles identified in step 1 to determine prior articles that should be considered;

3. Forward search - We should search forward by using the Web of Science (the electronic version of the Social Sciences Citation Index) to identify articles citing the key articles identified in the previous steps, again, to determine prior articles that should be considered.

A literature review is concept-centric. Therefore, concepts determine the organizing framework of a review. When the reading is complete, synthesizing the literature by discussing each identified concept in a Concept Matrix table helps to identify the best concepts in each article. Essentially, the presented

literature review is based upon the fulfillment of these practices, so, in sum, we are going to determine the state of art and the theoretical review of the main subjects related to this research work.

2.1 Software Architecture

To start analyzing the architectures, first, we are going to explore what software architecture is.

The architecture of a software application is its high-level structure, which consists of distinctive parts and their dependencies (Richardson, 2018). It is multidimensional, so there are various perspectives to view and describe it. One of the most widely known approaches to describe software architecture is the 4+1 view model created by Kruchten (1995), as presented in Figure 2.

The 4+1 view model defines four different views of software architecture. Each view describes a specific aspect of the architecture and consists of a particular set of software elements and relationships between them.

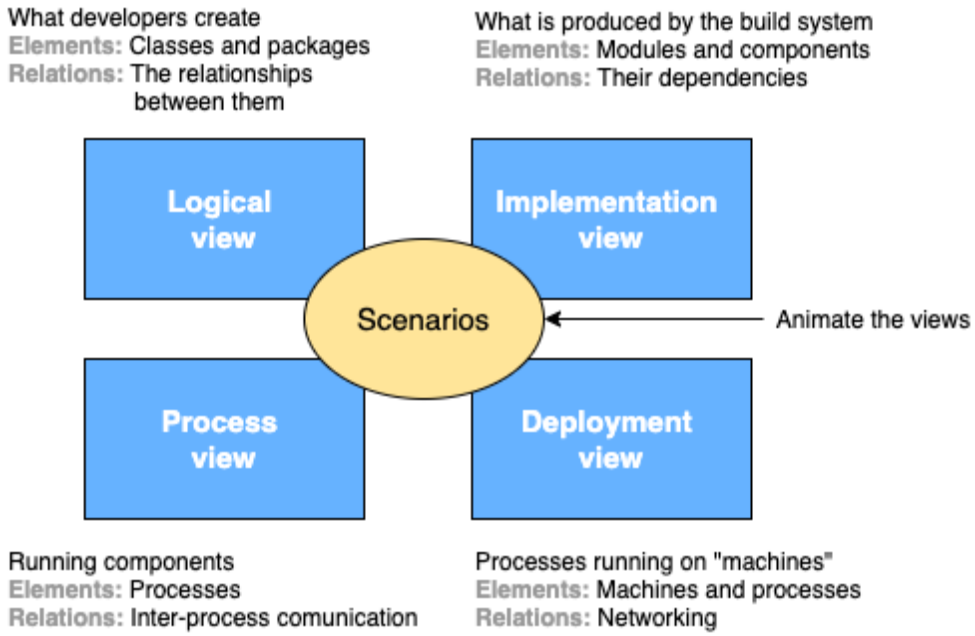


Figure 2: 4+1 view model of software architecture (Adapted (Kruchten, 1995))

- Logical view - Represents the software elements that are created by developers, for example, in object-oriented languages these elements are classes and packages. It describes the relations between them, including inheritance, associations, and dependency;
- Implementation view - This view depicts the output of the build system. It consists of modules, which represent packaged code, such as libraries and packages, and components, which are executable or

deployable units consisting of one or more modules. It describes the relations between modules and components, including composition relationships between both and dependency among modules;

- Process view - It is the view that deals with the components at runtime. Each element is a process, and the relations between processes represent interprocess communication;
- Deployment view - Concerns about how the processes are mapped to machines. The element in this view consists of physical and/or virtual machines and the processes. The relations between machines represent the network. Besides, this view also describes the relationship between the process and the machine.

In addition to these four views, we have the scenarios, which are the +1 in the 4+1 model, that animate views. They describe how the various architectural components, within a particular view, collaborate to handle a request. So, each view describes an important aspect of the architecture and the scenarios illustrate how the elements of a view collaborate to answer that request.

2.1.1 The importance of architecture

An application has two categories of requirements. The first category includes the functional requirements, which define what the application must do, usually in the form of use cases or user stories. However, it has very little to do with the functional requirements because we can implement them with almost any architecture, even a big ball of mud.

The second category explains that an architecture is important because it enables an application to satisfy its quality of service requirements, also known as quality attributes and are the so-called "-ilities". These requirements define the runtime qualities such as scalability and reliability. They also define development time qualities including maintainability, testability, and deployability, which an MSA specifically implements efficiently (Richardson, 2018).

The architecture we choose for our application determines how well it meets these quality requirements. Therefore, we are going to explore in-depth the most used architectures nowadays and determine which is better for each use case.

2.2 Monolithic Architecture

To start explaining the MSA, it is useful to first explore the MA since it has been the main software architecture used worldwide. The MA represents the typical enterprise application, meaning it has a

single codebase that is compiled together and produces a single artifact. More specifically, it is a software architecture whose components cannot be executed independently, since it is a single artifact, and dictates that an application consists of components that are all Tightly coupled together. These components have to be developed, deployed, and managed as one entity since they all run as a single operating system process and scale by replication of all functions on multiple servers.

Figure 3 demonstrates the typical structure of an MA. This design was widely used before SOA became popular. It is composed of four main layers. On the presentation layer, we are concerned with how we present something to the user, in other words, the User Interface (UI) and User Experience (UX) component of our application. The application logic defines the actual features of our app, and the business logic implements the business rules to it. Lastly, the Data Access Object is an object that provides an abstract interface to some type of database or other persistence mechanisms.

On an MSA approach, any unit in the figure would be scaled independently, including the data.

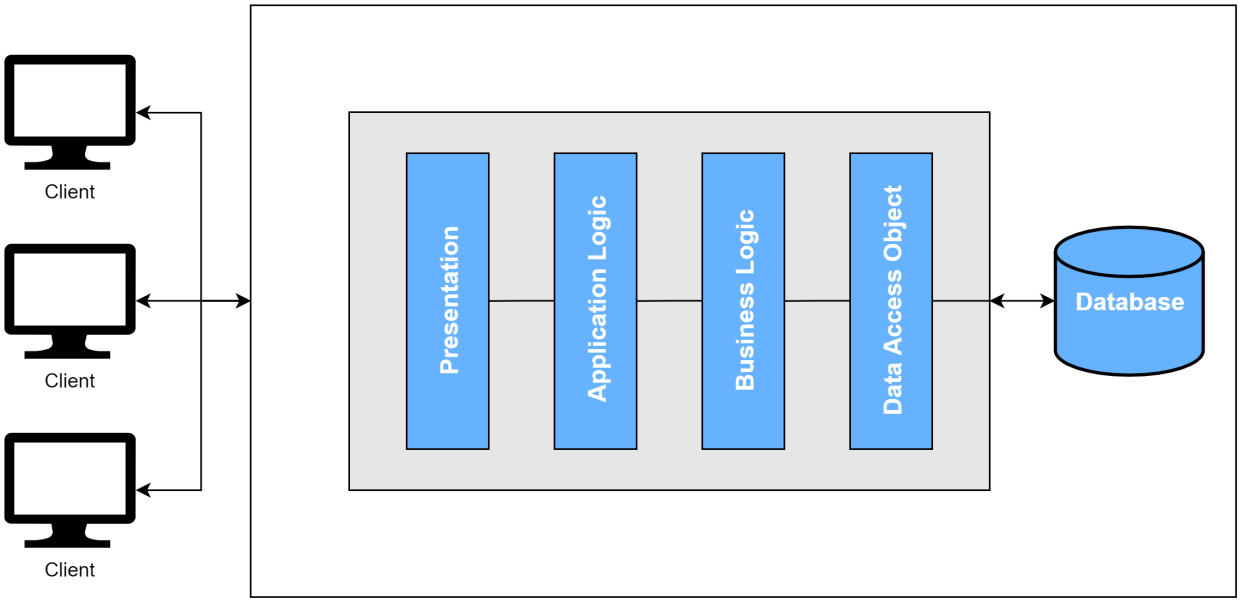


Figure 3: Example of an MA

2.2.1 Advantages and disadvantages

The characteristics of an MA bring some benefits with it, such as:

- It is faster and easier to develop since the goal of current development tools and Integrated Development Environments (IDE) is to support the development of monolithic applications;
- In case our system is not yet well defined it is easier to deploy because we simply need to deploy a single executable file (or directory hierarchy) on the appropriate runtime;

- We can scale the application only by running multiple copies of the application behind a load balancer.

The main problem with the MA is that, once the application's codebase becomes large and the team grows in size, we start entering the monolithic hell. This approach has several drawbacks that become increasingly significant:

- It becomes too difficult to understand and make any functional changes to the codebase to a point where we can not maintain and evolve it anymore due to its complexity. This will ultimately slow productivity and cause a decline in the quality of the code;
- Tracking down bugs requires long perusals through the codebase. Moreover, because all the modules are running within the same process, it is also hard to implement fault isolation and, every so often, a bug in one module, for example, a memory leak, crashes all instances of the application, one by one;
- With the growing application, it is difficult to add new developers or replace leaving team members. Plus, they can get intimidated by the codebase size and complexity. It also prevents developers from working independently since modularity breaks down over time;
- It makes continuous development very difficult, or even impossible. Any change, even in a small component, requires rebooting the whole application. For large-sized projects, restarting usually entails considerable downtimes, hindering the development, testing, and maintenance of the project;
- Scaling becomes difficult;
- They suffer from dependency hell (Merkel, 2014), in which adding or updating libraries results in inconsistent systems that either do not compile/run or, worse, misbehave;
- When choosing a deployment environment, the developer must compromise with a one-size-fits-all configuration, which is either expensive or suboptimal concerning the individual modules;
- It also represents a technology lock-in for developers, which are bound to use the same language and frameworks of the original application as it might be really hard, or impossible, to change those. In the words of Richardson (2014c), an MA forces us to be married to the technology stack.

2.2.2 Scalability

One of the biggest challenges of an Monolithic Architecture is when it is getting plenty of traffic, and we need to improve our application availability by scaling it. It can only scale in one dimension and, since it can not scale each component independently, it can only increase transaction volume by running more copies of the application. Moreover, this architecture can not scale with an increasing data volume, because each copy of the application instance will access all of the same data, making caching less effective, increasing memory consumption and input/output traffic.

Different application components may have different resource requirements, as one might be Central Processing Unit (CPU) intensive, while another might be memory intensive, and others might even require ad hoc components (for example, SQL-based instead of graph-based graphs) (Namiot & Sneps-Sneppe, 2014). Another problem with this scalability strategy is that the increased traffic could stress only a subset of the modules, making the new resources for the other components inconvenient.

Figure 4 represents the scale cube. It defines three ways to scale an application represented by three axes, the X, Y, and Z.

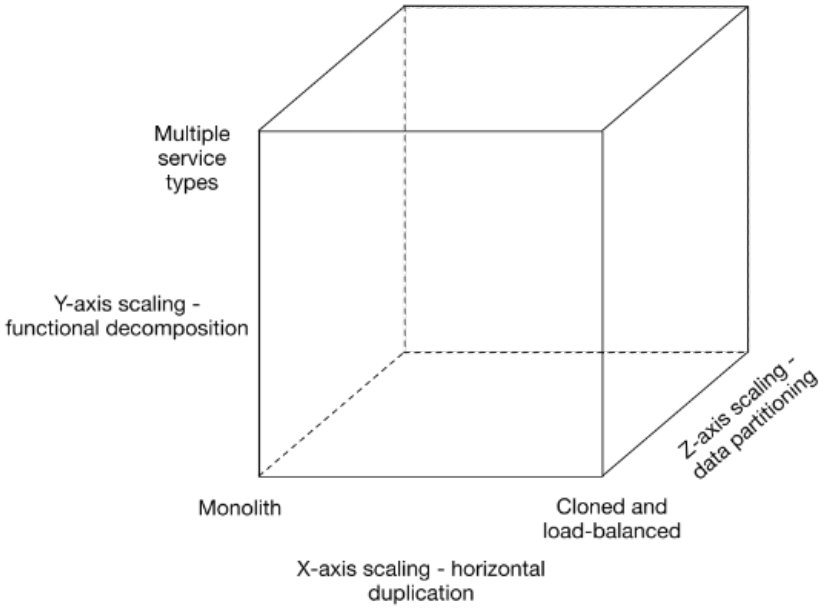


Figure 4: AKF scale cube (Abbott & Fisher, 2009)

As we can see, the X-axis scaling is denominated by horizontal scaling as we scale our application by running multiple identical copies of the application behind a load balancer. That way, if there are N copies then, each copy handles 1/N of the load. In other words, this is simply scaling by cloning, which is a common way to scale a monolithic application.

For Z-axis scaling, so-called data partitioning, each server runs an identical copy of the code, similar

to X-axis scaling. The main difference in this approach is that each server is responsible for only a subset of the data. A special component of the system, for instance, a proxy, is responsible for routing each request to the appropriate server. Commonly used routing criteria are, for example, the customer type or an attribute of the request, such as the primary key of the entity being accessed. This method is called sharding (Stonebraker, 2010). One practical case for this is a service division for free/payable users. The service's codebase is the same, but servers may have different capacities (depends on the bill). Those characteristics make this axis splits ideal to scale databases. Both Z-axis and X-axis scaling improve the application's capacity and availability.

Lastly, neither of the last approaches solves the problem of increasing development and application complexity. For this, we can follow the Y-axis scaling, also known as the functional decomposition axis. Unlike the X-axis and Z-axis, which consist of running multiple, identical copies of the same application, this axis scaling splits the application into multiple different services. Regarding the application tier, Y-axis scaling splits a monolithic application into a set of services. Each service is a mini-application that implements narrowly focused functions that have related functionality. A service is scaled using X-axis scaling, though, some services may also use Z-axis scaling (Abbott & Fisher, 2009).

2.2.3 Modular Monolithic Architecture

Over the years plenty of organizations have built up technical debt. They had a big monolith that had messy code and was difficult to maintain and scale, a big ball of mud as Foote and Yoder (1999) popularly nicknamed this phenomenon. As a result of that, and to try to solve this problem, we started to separate the codebase into smaller modules. Using well-encapsulated components, a monolith can be written (or easily refactored) to provide a set of Loosely coupled, modular components with well-defined interfaces, and encapsulated data access enabling easier maintenance down the line. We can also easily substitute one implementation for another if our interfaces remain consistent.

Figure 5 shows an example of an Modular Monolithic Architecture (MMA). The business logic consists of modules, each of which is a collection of domain objects. The modules represented here are Products, Customers, and Orders.

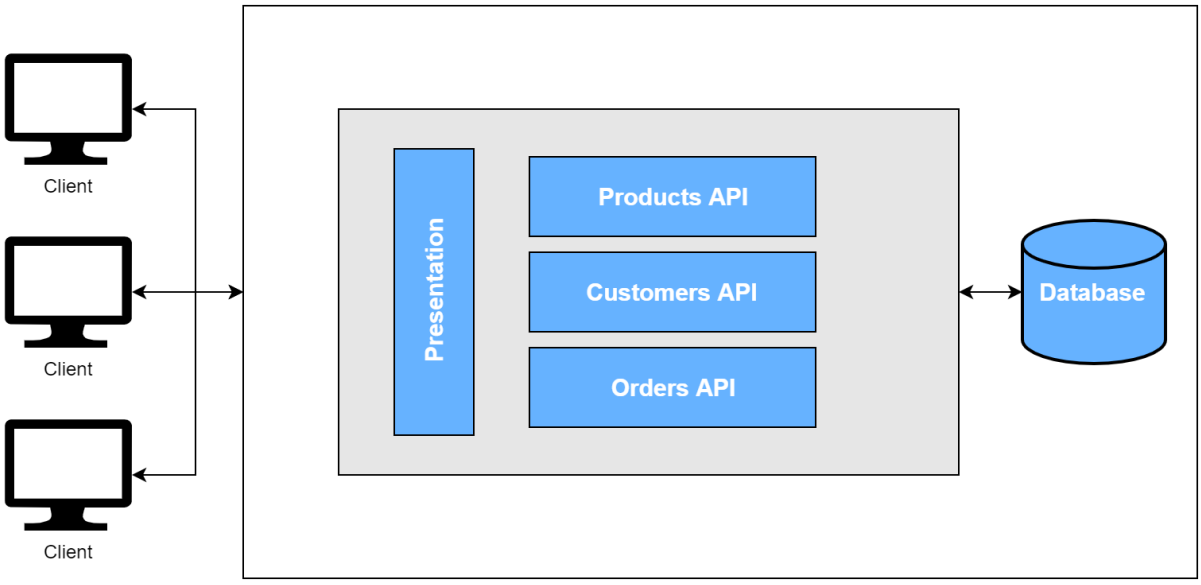


Figure 5: Example of an MMA

2.3 Service Oriented Architecture

SOA is yet another key concept that is important to explore since MSA is considered to be SOA done right. It is a style of architecting applications in such a way that they are composed of discrete services that have simple, well-defined interfaces, and are orchestrated to perform a required function. In this type of architecture, a service generally means an operating system process, and communication between services occurs through invocations on the network, rather than invocations to local process methods (Newman, 2015).

Figure 6 is an example of a SOA and represents its typical structure with a high-level graphic that describes The Open Group Architecture Framework (TOGAF) SOA Reference Architecture.

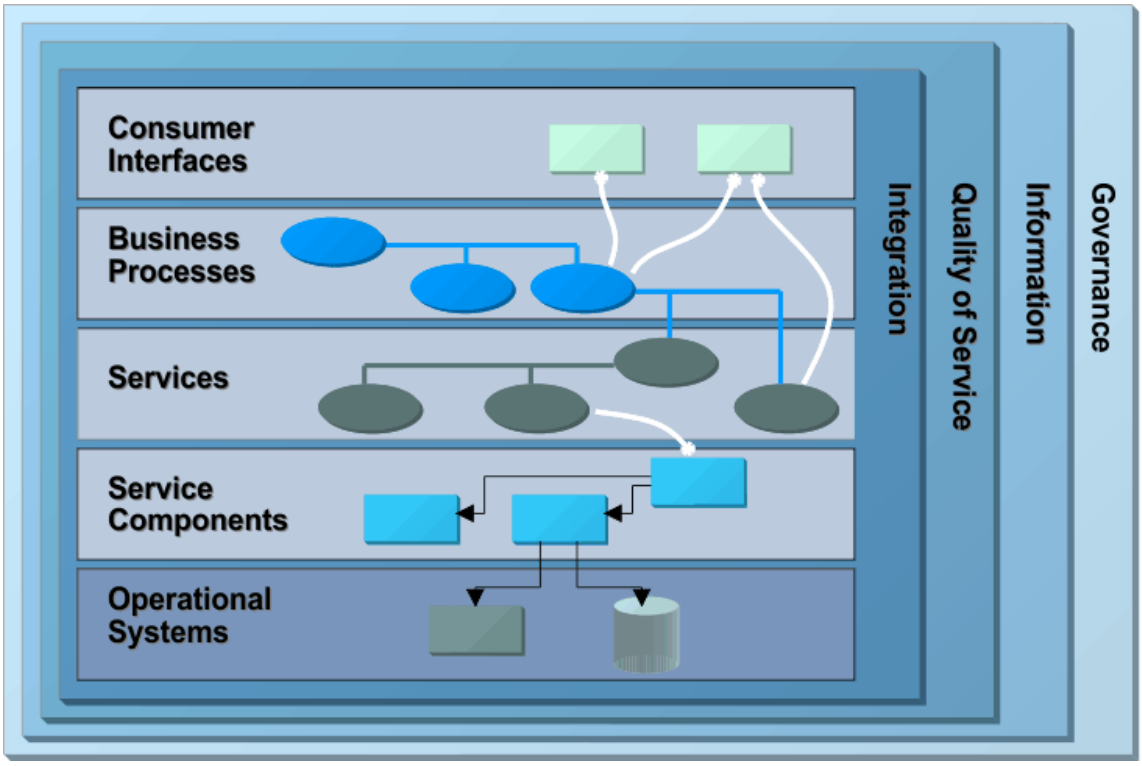


Figure 6: Example of a SOA (TOGAF, 2014)

2.3.1 Service Oriented Computing

The idea of service-orientation started when attention to the Separation of Concerns led to the emergence of the so-called component-based software engineering (Szyperski, 1998), which has given better control over design, implementation, and evolution of software systems (Dragoni et al., 2017). That created a further shift towards the concept of service first (W3C, 2014) and the natural evolution to microservices afterwards.

Service-Oriented Computing (SOC) is a distributed computing and e-business processing paradigm that is emerging and finds its foundation in object-oriented and component computing. It has been introduced to tackle the complexity of distributed systems and to integrate different software applications (MacKenzie et al., 2006). In SOC, we have a program called service that provides functionalities to other components, which are accessible via message passing. Services decouple their interfaces (that is, how other services access their functionalities) from their implementation. On top of that, specific workflow languages are defined to orchestrate the complex actions of services (for example, WS-BPEL (OASIS, 2007)).

The benefits of service-orientation are as follows:

- Dynamism - New instances of the same service can be launched to split the load on the system, Y-axis of the scale cube. This makes it easier for companies to manage their IT infrastructure;

- Modularity and reuse - Complex services are composed of simpler ones. The same services can be used by different systems making faster deliveries of new implementations;
- Distributed development - By agreeing on the distributed system's interfaces, distinct development teams can develop parts of it in parallel;
- Integration of heterogeneous and legacy systems - Services merely have to implement standard protocols to communicate, simplifying the transition to cloud-based solutions.

That's why service-orientation has steadily increased in popularity. Even though plenty of project managers resisted the approach at first, the advantages of adopting it became inescapable. It continues to grow and drive the market. According to TechNavio (2017) latest report on the global system infrastructure market, the report shows the segment's revenue will grow at an average annual rate of 5.57% till 2021, and the "increasing shift toward service-oriented architectures" is mentioned as a key element of that growth.

SOA is considered to be the first "generation" of SOC. It defined daunting and nebulous requirements for services (for example, discoverability and service contracts), and this hindered the adoption of the SOA model. Furthermore, SOA means too many different things, and most of the time that we come across something called "SOA" it is significantly different from the style we are describing here, usually due to a focus on the Enterprise Service Bus (ESB) used to integrate monolithic applications. Consequently, this arises the problem that the implementation of business concepts over time is spread across multiple service layers, creating a huge tightly Coupled, and making change unfeasible. Another characteristic of SOA is that, in most situations, databases are not exclusive or owned by specific services, but shared among them.

Microservices are the second iteration of the concept of SOA and SOC. Many of the techniques in use in the microservice's community have grown from the experiences of developers integrating services in large organizations. All of the previously mentioned key characteristics are transposable from SOA to MSA, and the two architectures are not necessarily competing, but they differ in several ways. The aim is to strip away unnecessary levels of complexity to focus on the programming of simple services that effectively implement a single functionality, making them a lightweight evolution of SOA. Like in the object-oriented, the microservice paradigm needs ad hoc tools to support developers and naturally leads to the emergence of specific design patterns.

2.3.2 Enterprise Service Bus

ESB is an architectural pattern that tells us how multiple applications, components, and systems can communicate/interact with each other following the SOA principles, though here there is no direct producer and/or consumer. It's about recognizing the centralization of the model that requires us to put all the logic in one place, and it provides all of the routing and data transformation required to get our applications talking to each other, behaving like a middleware.

The ESB is not the panacea for all our ills, and relying on it to solve all problems is not the right approach. The problem comes when we have botched implementations of service orientation, from the tendency to hide complexity away in ESBs, creating the famous "spaghetti box". Another problem with the ESB is that, by connecting our services through it, it creates a Single Point of Failure, meaning that if it is down, no communication between clients and services can take place. This does not happen in an MSA, as congestion in a service impacts only that service, and all others work and have their channels of communication. Moreover, the extra level of indirection created with the ESB may result in decreased performance of client-service communication.

2.4 Microservices Architecture

There are plenty of good definitions made by different authors about MSA. According to Richardson (2014b), an MSA is an architectural style that structures an application as a set of services that are easily manageable and testable, Loosely coupled, independently deployed and organized by Business Capabilities or Business Domains.

One of the most complete and well-accepted definitions is the one written by Fowler and Lewis (2014). In their view, the MSA style is an approach for developing a single application as a suite of small services, each running independently in their processes and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable, scalable, and tested by fully automated deployment machinery. There is a bare minimum of centralized management (completely separate service) of these services, which may be written in different programming languages and use different data storage technologies.

It is important to note that the word micro in microservices does not concern the size or complexity of the service, but rather the scope of the domain to which it relates. It is expected that a microservice only concerns a very specific functionality, and performs it excellently, following, thus, the developmental ideology Single Responsibility Principle (Martin, 2003). Additionally, it is probably more important to think

about how many services we are capable of supporting operationally, than it is to think about how small they are because it is better to have slightly bigger ones and fewer of them if we do not have fully automated deployment into production (CDEP/CDEL).

Figure 7 is an example of a simple food delivery application that implements the MSA. As we can observe, the application is composed of six services that interact with each other, and their relationships. They are Loosely coupled, and each one of them has its database. There is a front-end microservice that consumes and orchestrates the way the services are consumed, depending on the request.

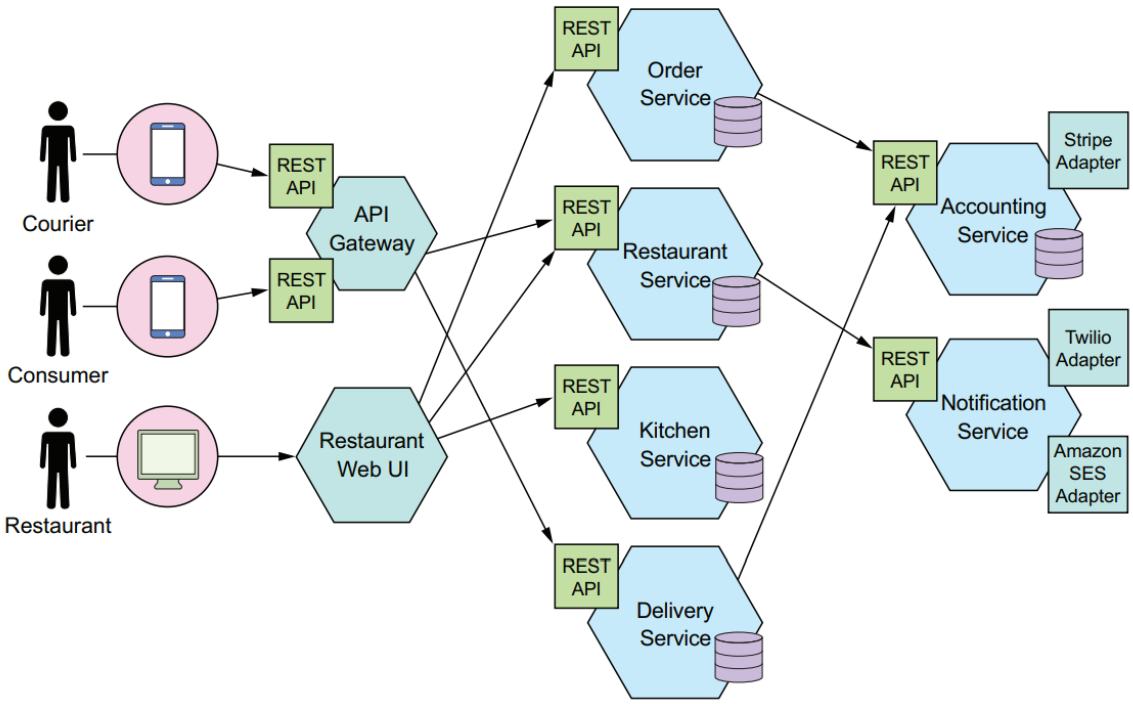


Figure 7: Example of an MSA (Richardson, 2018)

The whole application is created from the sum of its parts, which are different microservices. Each one of them, when alone, does only one thing, but when we start to connect them, an application starts to arise. It can be perceived as a Lego construction where the construction is the MSA, and the single pieces of Lego are the microservices. Every time we add a Lego to the construction, it gains a different shape and becomes more complete. We do that until we reach the final form.

In the future, if we want to change the shape or add more details (more features in the MSA), we can just add new Lego because the construction is modular. This is just an analogy to simplify the construction of an MSA with Legos. However, there is one feature on the MSA that is not possible to represent on this

analogy that is the fact that microservices can serve countless applications at the same time, which is not physically possible to do with Legos.

2.4.1 Microservices and the Cloud

The nature and characteristics of the MSA make it able to take full advantage of the cloud services, because, in terms of scalability, the MSA is an application of Y-axis scaling, according to the scale cube. The services provided by cloud providers are microservices themselves, hence their great scalability, and high reliability. The service models that they sell are classified by Platform as a Service (PaaS), Infrastructure as a Service (IaaS) and Software as a Service (SaaS).

Currently, the best end-user software model is the SaaS, where the platform to maintain it is in the cloud, and the architecture that should be followed is the MSA. This allows consumer organizations to focus on the business and try to remove technology from the list of concerns or limitations.

Naturally, associated with this decrease in responsibility, there is a cost, and the greater the responsibility to transfer to the cloud provider, the greater the cost. It is important to analyze in detail which organizational processes are crucial to digitalization and whether they should be in the cloud or in the form of SaaS. This process is the most expensive, and provides less freedom of extensibility as many technology decisions are made by cloud providers.

As for companies that produce IT, they can, through IaaS or SaaS models, develop the product more quickly. Thus following the practice of Continuous Delivery (CDEL), as this is the only way to keep up with the IT industry market. Moreover, they can also efficiently manage product monitoring, automation, and scalability to meet the needs of different customers. These companies provide their customers with the product as SaaS.

Note that the implementation of MSA is not cloud-dependent and can be adopted under the On-Premises model. In some particular cases, because of the freedom in the configuration and extensibility of services/technologies, it may be preferable to have a particular On-Premises technology to exploit its full potential without the implied restrictions for the same technology provided by a cloud provider.

However, all complexity of resource and infrastructure management is the responsibility of the organization. When choosing to migrate an application to the cloud, the choice of service model defines which responsibilities the company wants to hold and which ones it wants to transfer to the cloud provider.

In Figure 8, the distributions of responsibility for the various cloud services models are explicit. It is very important to reflect on what level of responsibility the companies want to hold for cloud solutions to better manage, not only the costs but also the limitations of cloud services.

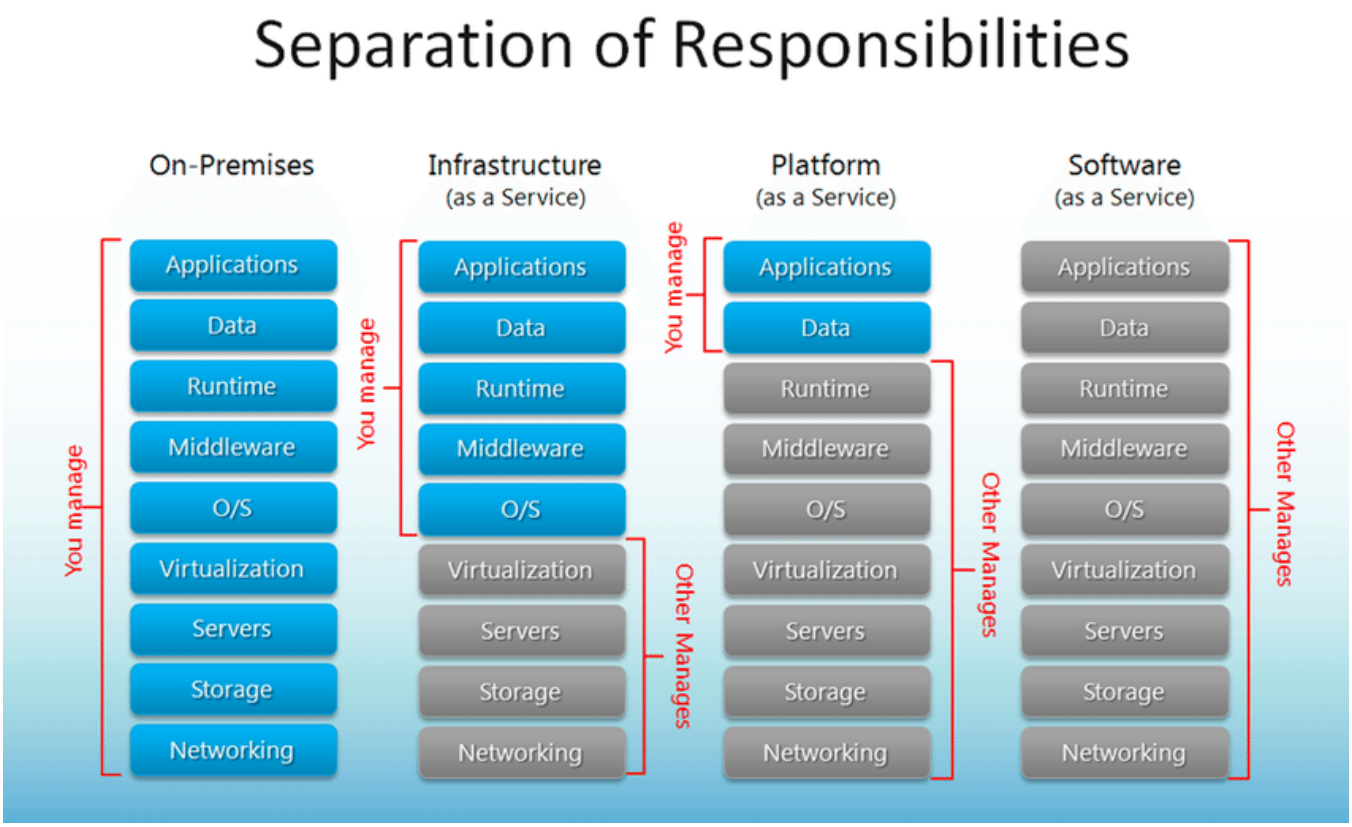


Figure 8: Separation of Responsibilities (Kissami, 2017)

- On-Premises - This is the software that is installed and executed directly on the infrastructure at the company's facilities. In this case, the software provider only distributes the software itself to its customers, and everything else is managed by them;
- IaaS - It is the provision of resources included in the scope of the infrastructure, in a scalable, automated way and with monitoring and access tools. It allows organizations to use infrastructure managed by third parties in the form of contracting service. They are mainly cloud-based services with high-level APIs used to dereference various low-level details of the underlying network infrastructure;
- PaaS - It provides customers, typically programmers, with a Framework on which they can develop application components without having to worry about managing the infrastructure;
- SaaS - Provides customers with applications over the Internet, without the need for installation. These applications are managed by third parties, and the customer only has to access the internet and use them, without having to manage anything.

2.4.2 Principles

According to Newman (2015), for a service to be a microservice, and for our MSA to be successful, we have to correctly adopt and implement the MSA principles presented in Figure 9. These principles help us frame the various decisions we have to make when building our systems. They can be adopted entirely like that or tweaked to make sense, depending on the company's needs. However, the whole should be greater than the sum of the parts, and if we decide to drop one of them, we should, firstly, understand the impact it will have.

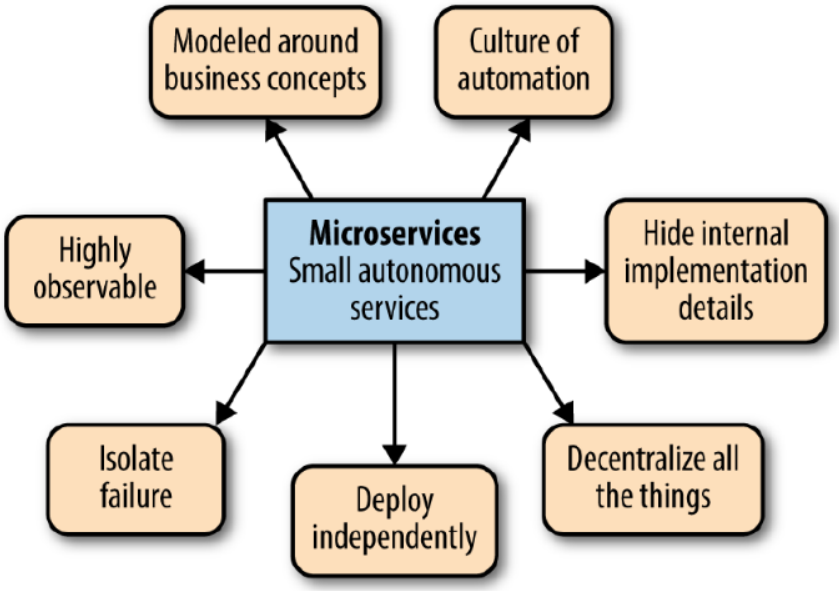


Figure 9: Microservices principles (Newman, 2015)

a) Modeled around business concepts

The modeling around business concepts describes that we should model our services around business-bounded contexts instead of technical concepts, because of their stability. Furthermore, a service should represent a Business Domain and should be structured with Bounded Contexts to not only attempt to form more stable interfaces but also to ensure that we are better able to reflect changes in business processes easily.

This principle focuses on the design of software architecture in function of the Business Domains, Business Models, and Bounded Contexts. All these concepts derive from the Domain-Driven Design (DDD), which is an approach to developing software for complex needs by deeply connecting the implementation, to an evolving model of the core business concepts (Evans, 2003). We have to understand what the business problems are, what the business landscape looks like, and what the business processes are,

and then drive a software product underneath that. Additionally, this concept helps to eliminate some model-code gaps because, when we look at our business, we should see our IT systems and, when we look at our architecture, we should see our business. Moreover, there will not be any issues between departments, if one department decides to change their software because the software architecture matches the organization structure.

Figure 10 presents an example of the context of Bounded Context. As Fowler (2014) pointed, the same entity, in this case, Customer or Product, in a different Bounded Context, has different meanings. This happens frequently with polysemes within large organizations. A microservice should be based on a Bounded context, as the Sales Context in the figure, where modeling Bounded Context entities will only present the relevant information for each entity in that Bounded Context, creating a unified model for that entity.

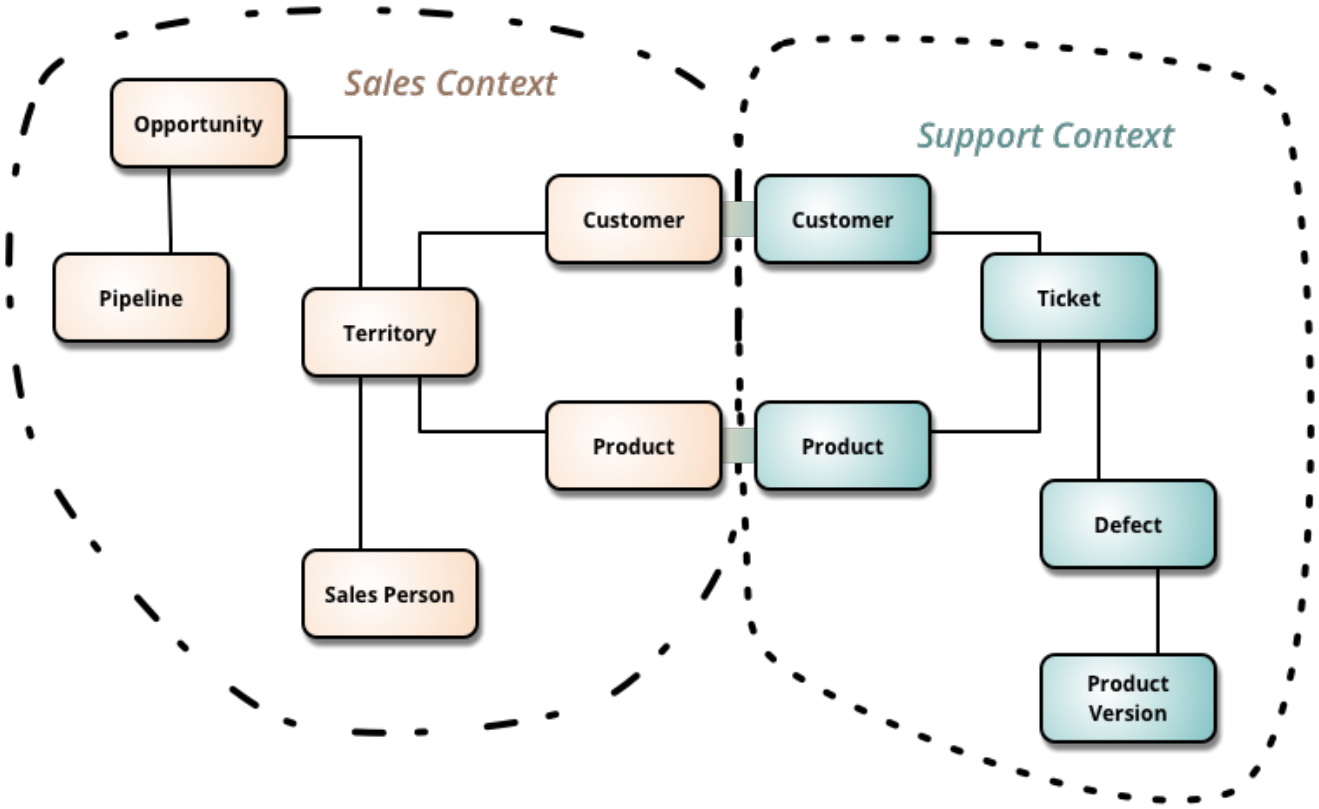


Figure 10: Example of the concept of Bounded Context (Fowler, 2014)

This concept is particularly important in the MSA, as it is based on the need to split a monolithic application into small services. This partitioning process should, in some microservices, be done in light of DDD principles, so that the business logic of a given context is in one microservice. If not, when we want to make changes in business logic, it is difficult to locate, in the code, where to make that change. Also, this can also lead to more serious implications, where the change has to be done at the level of multiple

microservices, which shows that responsibility for a particular business domain is spread across multiple work teams.

b) Culture of automation

The culture of automation is one of the most important principles for the MSA since they have an additional high complexity, which mostly comes from the sheer number of moving parts. The front-loading effort to create the tooling necessary to support our MSA can make a big difference because the speed at which new services can be commissioned will increase over time. Moreover, without those tools, it becomes difficult, or impossible, depending on the size of the organization, to manually manage our infrastructure.

The concepts of CI, CDEL, CDEP, and automatic generation of Boilerplate Code, when well implemented on an organization, means that it adopted, deeply, the culture of automation (Newman, 2015). The ideal point of automation would be when, with a click of a button, all the necessary operations are performed automatically from the development environment to the production environment, creating the application's deployment, or returning an error in case of failure at any stage of this process.

Automated testing is also essential because testing in an MSA can be quite complex when compared to an MA. Tools to reduce testing will shorten the amount of time required for manual regression testing, the time taken to test integration between services and clients, and also the time taken to set up test environments.

c) Hide internal implementation details

This principle tells that one should hide the internal implementation details of the service for it to evolve independently of any others. It allows our service to be Loosely coupled from his Service Consumers. Consequently, we can change the implementation of a functionality without impacting our Service Consumers, since they are abstracted from the implementation details, as long as the input and output of the functionality remain the same.

Modeling bounded contexts helps us focus on those models that should be shared, and those that should be hidden. Services should also hide their databases to avoid falling into some of the most commons sorts of coupling, which can appear in traditional SOA, and use data pumps to consolidate data across multiple services for reporting purposes. We should use APIs to share data and not a database. Where possible, APIs should be technology-agnostic to gives us the freedom to use different technology

stacks. Moreover, we can use REST and/or Remote Procedure Call (RPC) to formalize the separation of internal and external implementation details.

d) Decentralize all the things

Decentralize all the things is the principle in which we need to constantly be looking for the chance to delegate decision making and control to the teams that own the services themselves, that way, maximizing the autonomy that microservices make possible. This may cause some organizational changes, but it is a fundamental architectural principle. This process starts with embracing self-service wherever possible, allowing people to deploy software on-demand, and acquire infrastructure resources without having to request them from a third party. It makes development and testing as easy as possible and avoiding the need for separate teams to perform these activities, making the CDEL faster.

The organization should align their teams to ensure that Conway's Law works for them and should help them to become domain experts in the business-focused services they are creating (Conway, 1968). When some overarching guidance is needed, we should try to embrace a shared governance model where people from each team collectively share responsibility for evolving the technical vision of the system.

According to Newman (2015), this principle can apply to architecture too. We should avoid approaches that can lead to the centralization of business logic and dumb services, such as EBSs or orchestration systems. Instead, we should prefer choreography over orchestration and dumb middleware, with smart endpoints to ensure that we keep associated logic and data within service boundaries, helping keep things cohesive.

e) Deploy independently

The independent deploy principle means that each service must be capable of being deployed by themselves, without having to make a change to or affecting, any other service. This allows the team to efficiently provide CDEP and feature CDEL to customers. The service-owning team should have the power to release a new version for production without any approval from others. However, the team should put in place mechanisms to enable its consumers to adapt to the new version at their own pace, for example by providing endpoints to several versions of the service, allowing the user to upgrade overtime when he feels prepared and using consumer-driven contracts to catch breaking changes before they happen.

By adopting a one-service-per-host model, we reduce the side effects that could cause deploying one service to impact another unrelated service. By using blue/green or canary release techniques to separate

deployment from release, we reduce the risk of a release going wrong.

f) Isolate failure

An MSA increases the risk of failure since it is a distributed system. According to this principle, the approach to be taken is that the system can and will fail, and one should plan accordingly. If we do not take this into account, these systems are subject to cascading failures which can be catastrophic, and make the system more fragile than its monolithic version.

In the view of Newman (2015), if we hold the tenets of antifragility in mind, and expect failure will occur anywhere and everywhere, we are on the right track. Moreover, we should also make sure that our timeouts are set appropriately. Understand when and how to use bulkheads and circuit breakers to limit the fallout of a failing component and what the customer-facing impact will be if only one part of the system is misbehaving. We should also know what the implications of a network partition might be, and whether sacrificing availability or consistency is a given situation is the right thing to do.

g) Highly observable

Another key design principle, coupled with fault isolation, is that a micro-service should be highly observable. We cannot simply rely on observing the behavior of a single service instance or the status of a single machine to determine if the system is functioning correctly and, instead, we should do a joined-up view of the system to see what is happening. We need a way to be able to observe our system's health in terms of system status and logs. This type of monitoring and logging needs to be centralized so that there is one place where we need to go to view this information regarding the system's health.

We need this level of monitoring and logging in a centralized place because we now have distributed transactions. A transaction must go across the network and use several services; therefore, knowing the health of the system is vital, and also useful, for quick problem solving because the whole system is distributed and a lot is going on. That way, we can quickly work out where a potential problem possibly lies. Moreover, and because we are also using automated tools for deployment, it will be very quick, and we also need a quick way of getting feedback in response to deployment, so if there are any issues, we can see from a centralized place.

2.4.3 Patterns

The patterns address issues encountered when applying the MSA. Their choice is what defines how the principles above described will be followed. According to Richardson (2014b), the MSA pattern language consists of numerous groups of patterns, such as core, decomposition, data management, deployment, cross-cutting concerns, communication which also includes discovery and style, security, testing, observability, and UI. The three most critical decisions we have to make when applying the MSA pattern language to architect our application are on the core, decomposition, and data management groups. Additionally, we can have three types of relationships between patterns, predecession, succession, and alternative between patterns, respectively. A predecessor pattern is a pattern that motivates the need for a solution pattern. A successor pattern solves an issue that is introduced by another pattern. An alternative pattern provides an alternative solution to another pattern.

Figure 11 gives an overview of some of the most important patterns grouped by categories, and the relationships between each other.

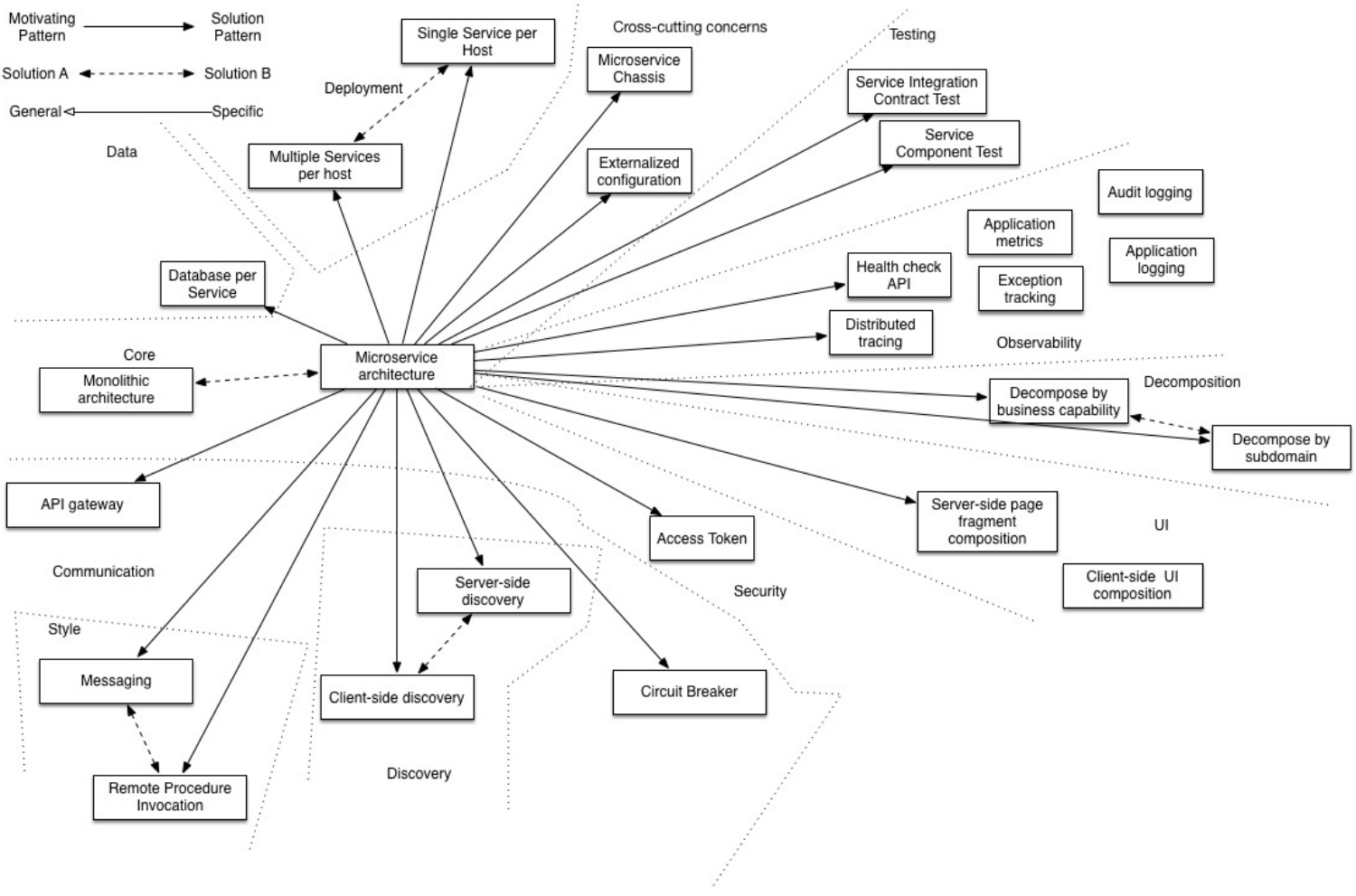


Figure 11: MSA patterns (Richardson, 2014b)

a) Core

The MA is an alternative to the MSA and, more often than not, it is the core of it. Empirically, most of the organizations that are building an MSA-style implementation have started with something big and have split that big thing up. This phenomenon is denominated by Brown Field and is the most common way of creating an MSA. Asides, we also have the Green Field. However, this technique is less used because, when we are creating something new, it is better to start with an MA-style implementation. As we going to be explored further into this dissertation, the choice between either of them depends on the situation but, if we pick the MSA pattern, we must choose numerous other patterns to deal with the consequences of our decision.

b) Decomposition

Another critical decision we have to make is how we decompose our application. Decomposition is the way we define our services. Ideally, a service must be small enough to be developed by a small team and to be easily tested. A useful guideline from Object-Oriented Design (OOD) is the SRP (Martin, 2003). The SRP states that a service has a single reason to change and/or to be replaced, it does only one thing, and one thing alone, and can be easily understood.

Applications should also be decomposed in a way so that most new and changed requirements only affect a single service because changes that affect multiple services requires coordination across multiple teams, which slows down development. To help ensure that each change should impact only one service, we can use another principle from the OOD, the Common Closure Principle (CCP) (Martin, 2003). The goal is that when business rules change, developers only need to change code in a small number of packages, ideally one.

Several strategies can help with the service decomposition. The two main options are: decompose by business capability, that defines services corresponding to business capabilities, and decompose by business domains, that defines services corresponding to DDD (Evans, 2003) subdomains. These patterns yield equivalent results: a set of services organized around business concepts rather than technical concepts. However, they differ in a fundamental matter. The first decomposes an application based on its technology functionality. The second focus his attention on information flows, being decomposed at a more conceptual level, at the business process level.

We can also decompose by verb or use case and define services that are responsible for particular actions, for example, a checkout service. Additionally, we can decompose by nouns or resources by

defining a service that is responsible for all operations on entities/resources of a given type, for example, customer manager service. However, and supporting the opinion expressed by Richardson (2018), we should use the decomposition by business domains.

A good analogy that helps with service design is the design of Unix utilities. Unix provides a large number of utilities such as grep, cat, and find. Each utility does exactly one thing, often exceptionally well, and can be combined with other utilities using a shell script to perform complex tasks.

c) Data management

The last critical decision is on the data management of our MSA. The main question here is how to maintain data consistency and perform queries. Since microservices are loosely coupled, it enables us to implement the Database per Service pattern. It states that each microservice has its persistent data private and is accessible only via its API. Furthermore, microservices' transactions only involve their database. However, some business transactions span multiple services, so we need a mechanism to ensure data consistency across services. To solve this problem we implement each business transaction that spans multiple services using the Saga pattern. A Saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule then the saga executes a series of compensating transactions, that undo the changes that were made by the preceding local transactions.

Alternatively, we have the Shared Database pattern. It is an anti-pattern for MSA, because, here, multiple services share a single database which makes them tightly coupled. In this case, the team that owns a service can modify neither the database nor their service, without, first, guarantee that the other services that use the same database will not be harmed. This has a big impact on the CDEP of a service.

When we use the Database per Service pattern, it gets harder to implement queries that join data from multiple services. To solve this, we can use the Command Query Responsibility Segregation (CQRS) pattern. It picks the CRUD (Create, Read, Update and Delete) operations and separates the Read operation from the others. We must define a view database, which is a read-only replica that is designed to support that query. The application keeps the replica up to date by subscribing to Domain events published by the service that owns the data. This pattern is useful, especially when we have a big disparity between the need to perform the various operations so that they can be scaled independently. Another pattern that we can use to solve this challenge is the API composition pattern.

d) Deployment

The deployment deals with the way we package and deploy our services. We can achieve that with a Single or Multiple Service Instance(s) per Host. Taking into account the MSA principles, the best option is the Single Service Instance per Host, because by having only one service instance per host, we get a bigger level of isolation between them. This eliminates the possibility of conflicting resource requirements or dependency versions, and it becomes easier to monitor, manage, and redeploy each service instance. Furthermore, by having one service instance per host, we take advantage of the benefits offered by emerging technologies, such as containers and serverless deployment.

Alternatively, we have Multiple Service Instances per Host. Naturally, this offers a more efficient resource utilization than the Service Instance per host pattern. However, there are MSA principles that can not be implemented when adopting this pattern.

e) Cross-cutting concerns

Cross-cutting concerns are technical requirements that are commonly applicable in all the microservices, essentially for management reasons. Even though in MA we mostly imagine autonomous teams working on independent services, sometimes, despite all the independence, some components benefit from system-level thinking. This is the case, for example, of the authentication, authorization, external configuration components, communication channels, monitoring and performance, service health verification, and metrics that provide information about the state of the service. Moreover, some patterns are considered cross-cutting concerns because their implementation is indispensable.

By creating a kind of rule book explaining how the cross-cutting concerns should be addressed uniformly across all microservices, we guarantee that, not only, the developer can focus on the business functionality of the service, but also that, regardless of who is developing them, they will always be implemented the same way. This guarantees compliance on important issues such as safety.

Another useful approach we can adopt is by using the microservice chassis framework, which handles cross-cutting concerns. There are tens or hundreds of services in an application, and we will frequently create new ones, each of which will only take days or weeks to develop. We cannot afford to spend a few days configuring the mechanisms to handle cross-cutting concerns. By using this framework we can create a new microservice faster and easier. However, we need a microservice chassis for each programming language/framework that we want to use, which can be a big obstacle.

Here, the MA has an advantage of code reusability. Cross-cutting concerns can be shared centrally

through single instances of libraries or frameworks. However, on the MSA, they are either applied or copied across to every microservice or called externally over the network via an API call. This introduces new forms of challenges and results in latency overhead. By adopting a platform such as Kubernetes (K8S), we can remove some cross-cutting concerns from the internal architecture of microservices, which can be implemented by service mesh mechanisms that mediate all communication in and out of each service.

f) Communication

As soon as we talk about distributed systems and MSA, the network part of the system becomes crucial. Communication can either be synchronous or asynchronous, depending on the message style. In the MSA, to be able to guarantee Loose coupling between the services, the use of asynchronous messaging mechanisms is required.

In addition to this, we also need to choose a message architecture, where there is the possibility of Message Driven Architecture (MDA) or Event-Driven Architecture (EDA). According to Chris Richardson, typically, EDA is chosen in MSA implementations as it provides more powerful failover mechanisms than MDA and implements the concept of Event Sourcing, which is the Single Source of Truth. Those fail mechanisms, in an MSA, are important, because the best way to ensure data consistency when a microservice fails is by reconstructing that data by consuming events, from the moment of failure to the present, thus returning to the current consistency of the data. Still, regarding fail mechanisms, we can use a Circuit Breaker to prevent a network or service failure from cascading to other services.

Communication mechanisms allow services to handle clients' requests and to communicate with each other to handle those requests. Here we have patterns such as the Remote Procedure Invocation (RPI), Messaging, and Domain-specific protocol. Additionally, for an external client of a Microservices-based application to access the individual services, we can use patterns such as the API gateway or the back-end for front-end pattern.

Services discovery allows the client of a service (that can be the API gateway or another service) to discover the location of a service instance. This can be done by several standards. The most common one is the Client-side Discovery or Server-side Discovery. As the names indicate, in the first, it is assumed that the customer will have to find out the means to communicate with the service concerned. In the second, it is the service's responsibility to ensure that all customers have access to it. However, and as stated by Richardson (2018), in an MSA, the second option is preferable since it is important to abstract Service Consumer from as many implementation details as possible.

Some of these concerns related to networks and access to services are currently handled by tools such as K8S.

g) Security

Security is one of the most important matters in any software type. On the MSA, the security pattern deals with how to communicate the identity of the requestor to the services that handle the request. Generally, the access to microservices, as with any standard application, follows security standards, such as the access token to verify that a user is authorized to perform an operation. This Token is generated at the API Gateway level since it is the single entry point for client requests. It authenticates requests and will give access to a user, verified by the system, to the microservices it is allowed to invoke. It is common for each microservice to have access only to the services on which it depends, and this detail is implemented in MSA's network configuration at the virtual network level.

h) Testing

Services often invoke other services, and testing deals with the creation of automated tests that verify that a service behaves correctly. The most widely used pattern for testing is the Service Integration Contract Test which consists of creating automated tests for the microservice-based on the Service Level Agreement(s), which was defined for that consumer. According to Richardson (2018), there are two different contract tests: the Consumer-driven that is a test suite for a service that is written by the developers of another service that consumes it, and the Consumer-side that is a test suite for a service client (for example, another service) that verifies that it can communicate with the service.

i) Observation

Regards observation, we need system monitoring mechanisms to be able to observe our system's health in terms of system status, logs, and traces. This includes activities and errors currently happening in the system, and regular checks to see if the service remains available and with the desired performance. Since we have distributed transactions, this type of monitoring and logging needs to be centralized, so that there is one place where we need to go to view this information regarding the system's health. It is also important to create well-defined metrics that can later be used to automate security mechanisms and fault tolerance.

j) UI

The last pattern category is concerned with the way we implement a UI screen or page that displays data from multiple services since they are developed by business capability/subdomain-specific teams that are also responsible for the user experience. By using the Server-side page fragment composition, we build a webpage on the server by composing HTML fragments generated by multiple, business capability/subdomain-specific web applications. Another option is to use the client-side UI composition where we build a UI on the client by composing UI fragments rendered by multiple, business capability/subdomain-specific UI components.

2.4.4 Advantages and disadvantages

Successful software development requires the right organizational structure, processes, and software architecture, as represented by Figure 12 (Richardson, 2014a). To fully benefit from the MSA, the organization must, as described by the success triangle, change its process and structure. Consequently, a software development organization should either be a small team, or a collection of small, autonomous teams because they can move fast (without breaking things) and keep up with the needs of the business. Moreover, it makes sense for these teams to use an agile development process. Teams should do continuous delivery or, ideally, continuous deployment. The MSA enables teams to be agile and autonomous. Together, the team of teams and the MSA enable continuous delivery/deployment. The ultimate goal is to deliver better software faster.

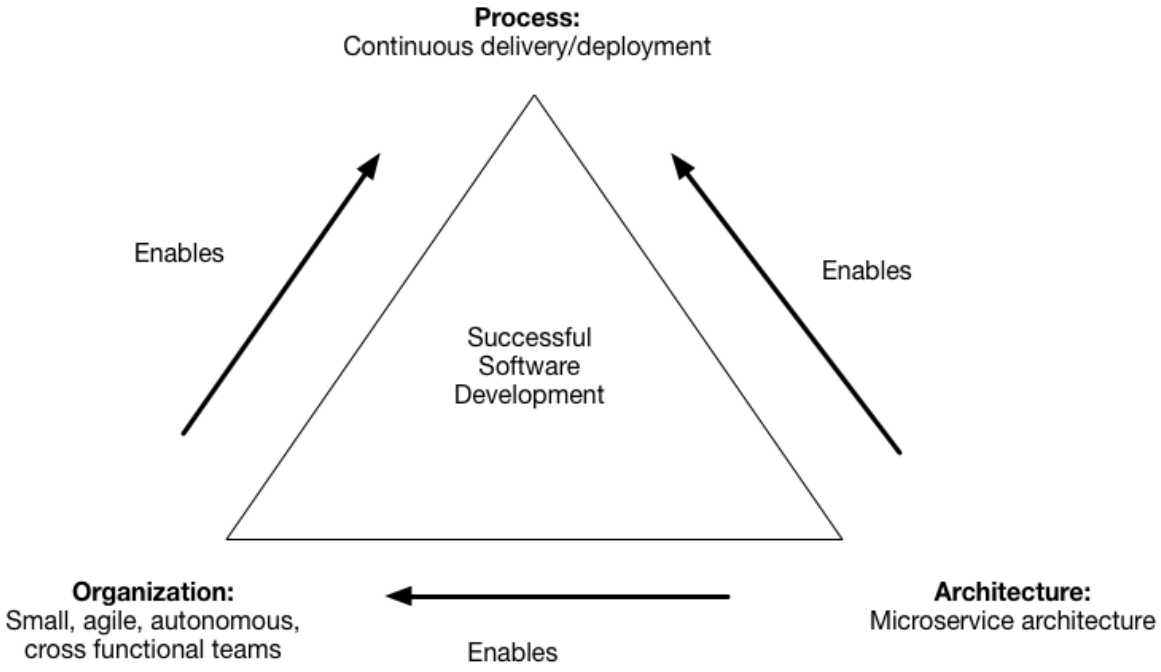


Figure 12: Success triangle for the MSA (Richardson, 2014a)

The MSA copes with plenty of the mentioned issues of MA. The correct implementation of the MSA, allied with the success triangle, brings plenty of improvements and advantages to the organization, such as:

- They implement a limited amount of functionalities, which makes their codebase small. This limits the scope of a bug, makes it easier for a developer to understand it and also makes the IDE and application start faster, making developers more productive and speeding up the deployment;
- They are independent, so a developer can directly test and investigate their functionalities in isolation concerning the rest of the system;
- It is possible to plan gradual transitions to new versions of a microservice. The new version can be deployed "next" to the old one, and the services that depend on the latter can be gradually modified to interact with the former. This fosters CI and greatly eases software maintenance;
- Changing a module of an MSA does not require a complete reboot of the whole system. The reboot regards only the microservice that needed to do it;
- Enables the CDEL and CDEP of large, complex applications. This will improve the maintainability, testability, and deployability of a software;

- Microservices naturally lend themselves to containerization, which has a high degree of freedom in the configuration of the deployment environment that best suits their needs (both in terms of costs and quality of service);
- Scaling an MSA does not imply duplication of all its components, and developers can conveniently deploy/dispose instances of services concerning their load (Gabbrielli, Giallorenzo, Guidi, Mauro, & Montesi, 2016);
- Improved fault isolation. In case of a problem and timeout on a service, the others will continue to handle requests. Contrarily, one misbehaving component of an MA can bring down the entire system. Moreover, at the infrastructure level, with containerization, the virtual environment of the service is delimited, so the resources that it may erroneously allocate are limited to the level of his virtual environment;
- The freedom to choose plenty of the tooling and resources on a case-by-case basis gives the flexibility to make informed decisions based on the right tooling for the case in question. The only constraint is on the technology used to make the network of interoperating microservices communicate.

The MSA is by no means a Silver Bullet that will solve all the organization's problems. So, it also presents some disadvantages, like:

- Developers must deal with the additional complexity of creating a distributed system;
- As Fowler and Lewis (2014) said, we are shifting the accidental complexity from inside our application in glue code in our components and modules within our application out into the infrastructure. However, new technologies are being constantly released to help us fight this problem;
- The communication protocols and communication channels to be used, as well as asynchronous communication, also represent a source of additional complexity;
- Testing is more complex as it has to be performed at the microservice and consumer level. When launching a new version of a service, it should ensure, that none of the consumers get their needs unmet;
- It leads to increased memory consumption since it allocates memory for each service, which, in sum, occupies more space;

- In addition to all the inherent technological complexity, there is also a process of organizational change, in which service-holding teams should be able to manage their entire life cycle independently. They should be multidisciplinary, which implies working together and bypasses the typical department concept.

2.5 Architectural choice

After an extensive analysis of the MSA and an overview of the MA and SOA, we can now take our conclusions to see in which case we should use which. As pointed previously, the MSA is by no means a silver bullet that will solve all the company's problems. The design thinking required to create a good MSA is the same as that needed to create a well structured MA and SOA. So, if we cannot build a well-structured MA, we probably cannot build a well-structured MSA as well.

2.5.1 Monolithic and Microservices

Taking the success triangle, previously presented, into account, and regarding the architecture, for small and simple applications, the MA is often the best choice to start with. Development is more simple, testing is easier and the application is easier to deploy and manage. Moreover, when developing the first version of an application, we often do not have the problems that the MSA solves and, using an elaborate, distributed architecture will slow down development. This can be a major problem for startups whose biggest challenge is often how to rapidly evolve the business model and the corresponding application. Additionally, using Y-axis splits might make it much more difficult to iterate rapidly.

However, successful applications have a habit of growing, and the MA will, eventually, become large and complex. It will become extremely difficult to develop and deploy in an agile way. Teams will no longer be autonomous. Delivering software will require lengthy merges and excessive amounts of communication and coordination, and we will likely end up in monolithic hell. The best approach would be to migrate to an MSA when we reach a certain point of maturity and before we end up in monolithic hell.

Today, the growing consensus is that if we are building a large, and complex application, we should consider using the MSA. However, if our components and their interfaces are not well-defined, then microservices certainly will not help. We will just end up with distributed balls of mud, and worst, with added latency. Like Newman (2015) pointed out, starting with an MA is, most of the times, more advantageous because there is more stability and knowledge of the business, and structure within the organization to refract it into services. However, it is important to note that there are some use cases where starting with

an MSA might be the best option for the organization.

Specialists warn that there is a risk of going too far on tech and language diversity and, maintaining such diversity, might become a headache in the future. The point is to choose an MSA because these additional benefits, compared against the additional complexity, are needed and warranted in the company and not to make the code better or to adopt a market trend.

2.5.2 Modular Monolithic and Microservices

An MMA is a big architectural improvement from the MA because it fixes some of the MA problems and implements some architectural good practices. It will take the company a long way toward the more agile architecture promised by the MSA with a lot less cost than their operational overhead. However, an MSA does offer additional benefits that may be important to the team. Most importantly, they allow it to version, release, and scale individual services separately, and allows for polyglot programming. As either the scale of the system or the size of the team grows significantly, these benefits will become more important and will eventually outweigh the complexity and inherent latency that microservices necessarily introduce.

Since it is recommended starting with an MA, for speed and simplicity, if the company's system is not yet well defined, starting with an MMA might be the best option for most companies because it gets the best of both the MA and MSA worlds. Additionally, by implementing an MMA, there might be some use cases where there will be no need for a migration to the MSA, whereas if they used the MA, there might be. So, the MMA enhances the capabilities of the MA, and it decreases the need for adopting an MSA.

2.5.3 Microservices and Service Oriented

Moving from an MA to a SOA or MA would have enormous implications and complexity. However, the case is different when it comes down to SOA and MSA. They have plenty of transposable characteristics between them, and the two architectures are not necessarily competing, but they differ in several ways. Their service definition is different, as the SOA defines four different service types while an MSA divides simply between two. Moreover, the SOA looks to share as much as possible, the MSA does the opposite. The first goes for multi-threaded and heavily resourced, the latter goes for single-threaded. The SOA services are coarse-grained, the MSA's are fine-grained. Additionally, speed, agility, lower costs, and little impact in the legacy infrastructure are some of the most important characteristics of an MSA, which further distinguishes it from SOA. It allows services to evolve independently, works with different technologies and

frameworks and makes testing and debugging much easier.

Companies who have been successful with a SOA implementation do not have to choose between what they have been doing and the new way of thinking about their software design. They are not mutually exclusive, and they are not competing against each other. One does not even replace the other, contrary to what is commonly said. However, the MSA is an evolution over the SOA.

While it is hard to say which one is the best, it is clear that the MSA has gained momentum, and not only because it is a trend or because SOAs have failed in the past. However, the MSA has a handful of key advantages over the SOA, and some of the biggest companies, like Netflix and Spotify, two giants with massive requirements and resources, are already using it. Moreover, plenty of new technologies and frameworks, standards, conferences, and communities are growing around it. So, currently, the MSA might be the best option for companies that want to start their service journey as it is the one that has the biggest support and is going to be more future proof.

3 Literature Review for the proposed Primavera BSS's topics

3.1 Development Operations

The concept of Development Operations (DevOps) is the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity: evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes. This speed enables organizations to better serve their customers and compete more effectively in the market Freeman (2019).

As seen previously, on the success triangle, DevOps is an approach of Agile Software Delivery, which promotes closer collaboration between business, development, and IT operations. The main goal is to bring them together to improve agility and reduce the time needed to get customer feedback.

The concept of DevOps involves the entire Continuous Model. It is a very important component of the MSA, as the entire deployment is managed by development and operations engineers. They use tools, such as Azure DevOps or Microsoft Team Foundation Services, to create microservices Deployment Pipelines and, thus, also implement the entire CI, CDEP, and CDEL process.

DevOps brings numerous benefits to a company, for instance:

- Improving the speed and agility of software delivery. This way we can innovate for customers faster by creating a more engaging user experience, adapt to changing markets better, more quickly release new features and fix bugs, and grow more efficient at driving business results to build a competitive advantage;
- Scaling successfully without interrupting the business;
- Building a Startup culture that adds business, development, and operations, thus emphasizing values such as ownership and accountability. Developers and operations teams collaborate closely, share many responsibilities, and combine their workflows, reducing inefficiencies and saving time;
- We can adopt DevOps without sacrificing security and retaining control by using automated compliance policies, fine-grained controls, and configuration management techniques.

3.1.1 Continuous Integration

According to Fowler (2006), CI is a software development practice where members of a team integrate their work frequently into a central repository, like GitHub. Usually, each member integrates at least daily, leading to multiple integrations per day. Each integration is verified by an automated build and has to pass through a load of tests, to detect integration errors as quickly as possible.

Many teams find that this approach leads to significantly reduced integration problems because the reduced time between integrations ensures that non-conformities will be identified more quickly, thus making it a less complex problem. Moreover, it allows a team to develop cohesive software more rapidly, improves software quality, and reduces the time it takes to validate and release new software updates.

Contrarily, in the past, developers on a team used to work in isolation for an extended period and only merged their changes to the master branch once their work was completed. This made code changes difficult and time-consuming and also resulted in bugs accumulating for a long time without correction, which made them harder to fix.

3.1.2 Continuous Delivery

CDEL is an extension of CI because it deploys all code changes to a testing environment and/or a production environment after the build stage. It is a software development discipline where we build software in such a way that it can be released to production at any time, allowing new changes to reach the customer quickly. This means that the release process has also been automated, and the deployment is only pending manual action.

To achieve CDEL, we need to use a Deployment Pipeline to be continuously integrating the software done by the development team, building executables, and running automated tests on those builds to detect problems. Furthermore, we must push the executables into increasingly production-like environments to ensure the software will work in production.

3.1.3 Continuous Deployment

CDEP goes one step ahead of CDEL because the process, in which the new release moves to the production environment, is also automated. This means the implementation of the new functionality increment will be automatically published for the production environment, except in cases where any of the unit tests fail. Therefore, the difference between CDEP and CDEL is the presence of a manual approval to update to production. With CDEP, production happens automatically without explicit approval.

This is the state-of-the-art for SaaS applications: deploying changes to production many times a day during business hours. It is especially useful for big companies. According to Richardson (2018), as of 2011, Amazon.com deployed a change into production every 11.6 seconds without ever impacting the user.

3.1.4 Configuration Management

Microservices are cloud-based applications and generally have many components that are distributed in nature, often running on multiple virtual machines or containers in multiple regions and using multiple external services (Crane et al., 1995). The diffusion of configuration settings in these components can lead to errors that are difficult to solve when deploying an application.

This increasing complexity of building applications is described in various programming methodologies to help developers deal with it. For example, the Twelve-Factor App (Wiggins, 2017) describes many well-tested architectural patterns and best practices for use with cloud-based applications. One of the main recommendations from this guide is to separate configuration from code. Therefore, an application's configuration settings should be kept external to its executable and read in from its runtime environment or an external source.

Configuration Management (CM) consists of configuring the infrastructure and microservices that are within it, centrally and externally. The fact that it is centralized allows DevOps teams to manage at a higher level without having, for example, to access the source code to do it. Moreover, a configuration platform is used to automate, monitor, design, and manage otherwise manual configuration processes.

For the implementation to be successful, it is necessary to create an intuitive and general namespace division for the configurations that is easy to apply to the microservices, in the short and long term, and that reflects the reality of the MSA. The objective is that it is not necessary to have a future refactoring of the configurations' namespaces, as it would be a very hard task, and that the same standard can always be maintained. A good practice is to design the key names in hierarchies instead of using flat key names because they present several benefits like they are easier to read, manage, and use. Furthermore, we can also use delimiters to separate the levels of the hierarchy, whether based on microservice components or deployment regions, such as "AppName:Service1:ApiEndpoint" or "AppName:Region1:DbEndpoint". The best approach can depend on a variety of factors like the programming language or framework needs (Microsoft, 2019a).

Additionally to the keys, we can also have labels to help to provide better namespace management. They are used to differentiate keys with the same name. We can take advantage of them to instantiate

different environments, geographic regions, and other useful parameters.

The main benefits are the centralized management and distribution of hierarchical configuration data for different environments and geographies, the fact that allows us to control resource availability in real-time and allows us to dynamically change application settings for one or more microservices without having to redeploy or restart. If we restart or redeploy our microservice, we have to test it again to check if it is still working correctly. Also, by storing all the configurations of our microservices in one place, we prevent errors in changing a cross-cut setting to several microservices, which can lead to their malfunction, thus creating a Single Source of Truth to centralize configurations, making a more robust and scalable application. This way, the configurations are applied in real-time and the microservices never get unavailable.

3.2 Microservices Documentation

3.2.1 OpenAPI Initiative

When developers consume a Web API, understanding its various methods can be challenging. The proper documentation of a REST API is important for its successful adoption. It should be designed with an interface that the consumer can understand to help developers learn about the API functionality and enable them to start using it easily. The OpenAPI Initiative (OAI), formerly known as Swagger, solves the problem of generating useful documentation and help pages for Web APIs. It provides benefits such as the automatic generation of interactive and useful documentation, API discoverability, and client SDK generation (De, 2017).

The OAI was created by a consortium of forward-looking industry experts and is backed by some large companies such as SAP, Oracle, IBM, MuleSoft, and SmartBear, among many others. They recognize the immense value of standardizing on how APIs are described and the importance of API First Design.

The OpenAPI Specification (OAS) is an OAI specification that defines a standard, programming language-agnostic interface description for designing and documenting REST APIs. It allows both humans and computers to discover and understand the capabilities of a service without requiring any direct implementation, such as access to source code, additional documentation, or inspection of network traffic. One of the goals is to make a consumer understand and interact with the remote service with a minimal amount of implementation logic and work needed to connect disassociated services. Moreover, it reduces the amount of time needed to accurately document a service (Specification, 2020).

The API definition can be done manually in a simple text editor, but there are OAI tools that can simplify the definition and visualization of the generated documentation. The format used for this definition is

JavaScript Object Notation (JSON) or (YAML), and all typical components of a REST API can be defined. Also, components can be defined as they represent API requirements to be implemented, such as infrastructure requirements for data persistence or cache, external services, among many others. After creating the API design, the generated document can be used for automatic code creation, where resources can be created in the cloud to allocate to the resulting application.

3.2.2 Microservices Canvas

Even though the development of microservices canvas was not a requirement for this project, it is a useful way to document microservices and needed to be mentioned. It is a more complete and multidimensional way of documenting microservices. The purpose is to expose to both the service consumers and developers, explicitly, the reason for the existence of a microservice.

According to Richardson (2019), the microservice canvas can be divided into different topics:

- Name - Name of the microservice;
- Description - A brief description of the microservice;
- Capabilities - The business capabilities implemented by the service;
- Service API - The operations (typically REST) implemented by the service and the domain events published by the service;
- Quality attributes - The microservices's quality attributes, which are also known as non-functional attributes;
- Observability - Includes health check endpoint, key metrics exposed on the microservice, and other observability functionalities;
- Dependencies - Lists the operations invoked, which are operations implemented by other microservices that the microservice being documented invokes. These operations can be synchronous, asynchronous, HTTP, or Event/Message-Driven. It also lists the operations subscribed, which are the messages and events that this microservice subscribes to;
- Implementation - It is where the domain model implemented in the service is defined.

It is important to notice that the OAI represents only a small part of the canvas, and all the documentation generated under the OAI specification would be located in the Service API topic. That said, this is

a more in-depth way of describing microservices that presents some useful information that simple API documentation does not.

The attached Figure A.1 presents an example of a microservice canvas for an order service, and documents all the previous topics.

3.3 Remote Procedure Calls

Remote Procedure Call (RPC) is a popular distributed systems paradigm used for inter-process communication between processes in different computers across the network. It is used for point-to-point communications between software applications where client and server applications communicate during this process. An RPC can also be denominated as a function call, method call, or a subroutine call (Tay & Ananda, 1990).

The way RPCs work is that a client, also denominated as a sender, creates requests in the form a procedure to the remote server, and the RPC translates and sends. When the remote server receives the request, it processes it and sends a response back to the client, and the application continues its process.

RPC applications use software modules called proxies and stubs, which make the process look like a normal local procedure call. It does not require the programmer to explicitly code the details for the remote interaction, calling it as if it was running in the same process/application. However, remote calls are usually slower and less reliable than local calls, so it is important to distinguish them (Tay & Ananda, 1990).

Due to its characteristics, the RPC model can be a good implementation to an MSA since it has some advantages over using APIs for communication, namely, its level of location transparency, allowing microservices to call methods from others more easily.

3.3.1 gRPC

RPCs have been around for decades as a mechanism for inter-process communication. Recently, a tool called gRPC, based on the RPC concept has attracted plenty of attention and is being adopted in numerous MSAs. It is a language-agnostic, high-performance, lightweight, and open-source universal RPC framework, initially pioneered by a team at Google and, nowadays, is backed by the Cloud Native Computing Foundation project (gRPC, 2020b). Due to its recent creation, there is not an extensive literature review for this topic.

It is built with modern technologies like HTTP/2, which is used for transport. This new version of the

HTTP network protocol is a major revision and brings a handful of new updates that mainly revolve around performance, efficient use of the network resources, and speed with reduced perception of latency. It leaves all the high-level semantics, like headers, fields, URLs, methods, and status codes, the same as the previous version. What is new is how the data is framed and transported between the client and the server that are allowed to have multiple concurrent exchanges on the same connection (Belshe, Peon, Thomson, & Ed., 2015).

When it comes to transport, the HTTP/2 breaks down the HTTP protocol communication into an exchange of binary-encoded frames, which are then mapped to messages that belong to a particular stream, all of which are Multiplexing into a single Transmission Control Protocol (TCP) connection. This is the foundation that enables all the features and performance optimizations provided by the HTTP/2 protocol (Google, 2019). In the case of gRPC, it allows it to support different streaming calls (gRPC, 2020a):

- Unary - The normal calls where the client sends a single request and gets back a single response;
- Client - It is similar to a unary call, except that the client sends a stream of messages to the server instead of a single message. The server then responds with a single message after it has received all the client's messages;
- Server - It is also similar to a unary call, except that here the server returns a stream of messages in response to a client's request. After sending all its messages, and to complete processing on the server-side, the server's status details and optional trailing metadata are sent to the client. The client completes his processing once it has all the server's messages;
- Bi-directional - The call is initiated by the client invoking the method and the server receiving the client metadata, method name, and deadline, which he can choose to send back its initial metadata or wait for the client to start streaming messages. Since the two streams are independent, the client and server can read and write messages in any order. This lasts until one of them ends the process.

Although the design of HTTP/2 effectively addresses the HTTP-transaction-level Head-of-line blocking problem by allowing multiple concurrent HTTP transactions, all those transactions are multiplexed over a single TCP connection. It means that any packet-level head-of-line blocking of the TCP stream simultaneously blocks all transactions being accessed by that connection (Google, 2019).

At the core of all performance enhancements of HTTP/2 is the new binary framing layer, as presented in Figure 13. It dictates how the HTTP messages are encapsulated and transferred between the client and the server.

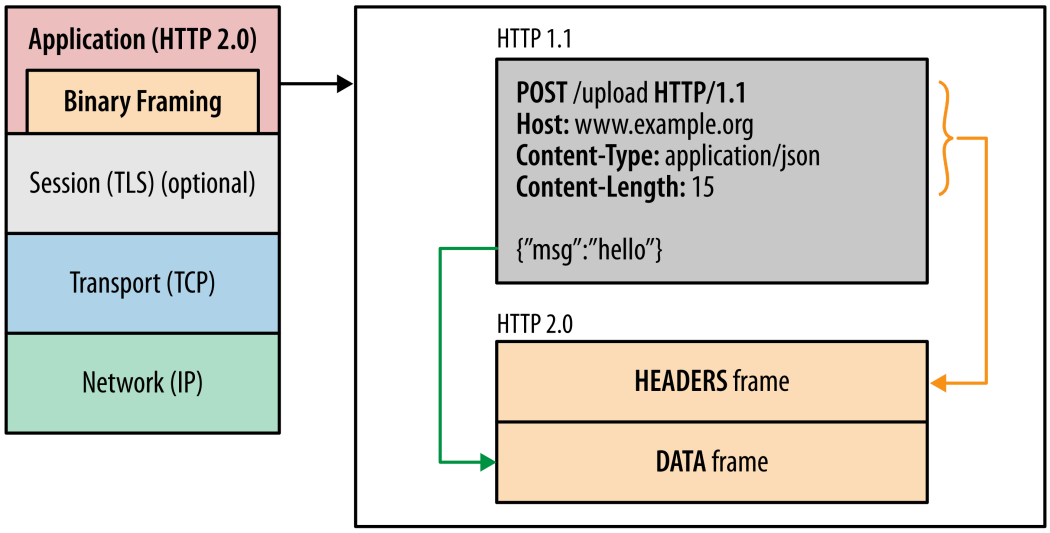


Figure 13: HTTP/2 binary framing (Google, 2019)

Another modern technology used is the Protocol Buffer that represents the interface description language. These protocols are a flexible, automated mechanism for serializing structured data, that encourages a Contract-First API development, allowing for language-agnostic implementations. Using Protocol Buffers enables applications to create a Single Source of Truth for their interface because this is all they have to share with others for them to know exactly how to call their services, and what to expect back. Moreover, it generates a strongly-typed cross-platform client and server bindings for many languages. A Protocol Buffer file contains the definition of the gRPC services and the messages sent between clients and servers.

The gRPC also provides features such as authentication, supporting the usage of Transport Layer Security (TLS) and token-based authentication, flow control, blocking or nonblocking bindings, and timeouts and cancellation.

3.3.2 gRPC vs HTTP APIs

There are some considerable differences between gRPC and HTTP APIs. Table 1 presents a high-level comparison between them. One of the main differences between both approaches is that gRPC is Contract First and uses Protocol Buffers to reduce the network usage with its binary serialization, and HTTP APIs are Content First and uses JSON, which takes a lot longer to serialize/deserialize the content.

Feature	gRPC	HTTP APIs with JSON
Type	Contract First (designed for humans)	Content First (designed for humans)
Contract	Required (.proto)	Optional (OpenAPI)
Protocol	HTTP/2	HTTP (any version)
Payload	Protocol Buffers (small, binary, machine readable)	JSON (large, human readable)
Prescriptiveness	Strict specification	Loose. Any HTTP is valid
Streaming	Unary, client, server, bi-directional	Client, server
Browser support	No (requires grpc-web)	Yes
Security	Transport (TLS)	Transport (TLS)
Client code-generation	Yes	OpenAPI + third-party tooling

Table 1: High-level comparison between gRPC and HTTP APIs (Adapted from (Newton-King, 2019))

Contrary to the HTTP APIs with JSON, gRPC has a more strict specification and requires more attention but eliminates debate and saves developing time, creating consistency across platforms and implementations. This consistency is even bigger when we take into account that it generates client code natively, while HTTP APIs need third party tooling to do that. One of the features where gRPC has a drawback over HTTP APIs is on the browser support where it requires the JavaScript gRPC-web library to work properly.

Depending on the requirements, a solution can benefit more from one or the other. All these characteristics make gRPC ideal for low latency, highly scalable, distributed systems like microservices where efficiency is critical, point-to-point real-time services that need to handle streaming requests or responses, and polyglot systems where multiple languages are required for the development.

3.3.3 gRPC vs SOAP/WSDL

gRPC is based on RPCs, we can call methods over the Internet, almost without worrying about what platform the service or the client is running on, like the original Web Service technologies Simple Object Access Protocol (SOAP) and Web Service Definition Language (WSDL). Another concept they have in common is that both have machine-readable API contracts but, gRPC implements them with Protocol Buffers and SOAP with XML. For that reason, plenty of people believe that gRPC is just a throwback to them and will fail for the same reasons they did. However, even though they have some similar concepts, gRPC avoids major SOAP/WSDL failures by learning with their mistakes and presents numerous differences, such as:

- SOAP and WSDL are inextricably tied to XML and there is no way of switching to a new serialization

format when, or if, a new one is released. Whereas Protocol Buffers are pluggable and can be switched;

- XML and XSD have a very heavyweight service definition format which makes it more complex and time-consuming to understand, develop, and maintain;
- SOAP and WSDL have some unnecessarily complex features;
- WSDL is inflexible and intolerant of forward-compatibility, unlike Protocol Buffers;
- With SOAP, clients were responsible for generating libraries, instead of vendors.

3.3.4 Advantages and disadvantages of gRPC

Taking into account everything learned about gRPC, we can conclude that its main advantages are:

- Low internet usage and better performance due to the HTTP/2 binary framing and header compression (HPACK) and Protocol Buffers Message Serialization (JSON takes a lot longer to serialize/deserialize);
- Ability to break free from the call-and-response architecture due to HTTP/2, which supports traditional request/response model and streams;
- Multiplexing, also due to HTTP/2, which allows for multiple calls via a TCP connection and avoids head-of-line blocking;
- All gRPC libraries have first-class code generation support, so it is possible to use multiple languages;
- Loose coupling between clients/server makes changes easy
- It is less complex when compared to other event-driven technologies;
- The lower processing time for requests reduces the costs;
- The improved development efficiency reduces the costs of development for companies, allowing them to achieve more new features developed.

Nonetheless, it is a new tool that is using recent technology, so it has some disadvantages, in particular:

- Limited browser support because browsers have great HTTP/2 support, but JavaScript APIs have not caught up, and gRPC-web provides limited support for calling gRPC services;
- There is still no standardization across languages and streaming is difficult, if not impossible, in some languages;
- HTTP/2 and Protocol Buffers are binary protocols, meaning they are not human-readable;
- Sometimes additional tools, like BloomRPC and Wireshark, are required to debug calls.

3.4 Deployment Platforms

Deployment platforms are useful technologies since they provide us the tools to more easily implement CDEP in our software life cycle. Once implemented and well configured, they allow us to automate the whole pipeline process to deploy new changes into production.

3.4.1 Docker

Docker is an open-source platform for developing that automates the deployment of applications into Containers. It enables us to separate applications from our infrastructure, and consequently developers from IT DevOps so we can deliver software faster and easier. With it, we can manage our infrastructure in the same way we manage our applications (Docker, 2017a).

The Docker platform brought greater agility in the creation of Virtual Machines (VM) in the form of Containers. This allows saving plenty of computational resources when compared to a traditional VM, significantly reducing the delay between writing code and running it in production, and also allows to automate our infrastructures.

Creating a container with Docker is as simple as making a definition of a YAML or JSON document, and the interpretation of that file generates the entire container's infrastructure and configurations automatically.

Figure 14 presents the differences between Docker Containers and VMs. They have similar resource isolation and allocation benefits on the infrastructure but function differently. While containers virtualize the operating system, VMs virtualize the hardware. Docker allows running multiple containers on the same machine, sharing the OS kernel with other containers, each running as isolated processes in userspace. VMs are an abstraction of physical hardware that convert one server into many. The hypervisor allows multiple VMs to run on a single hardware machine. Moreover, each VM includes a full copy of an operating

system, the applications, necessary binaries, and libraries, while Docker containers only need one full copy per infrastructure (Merkel, 2014).

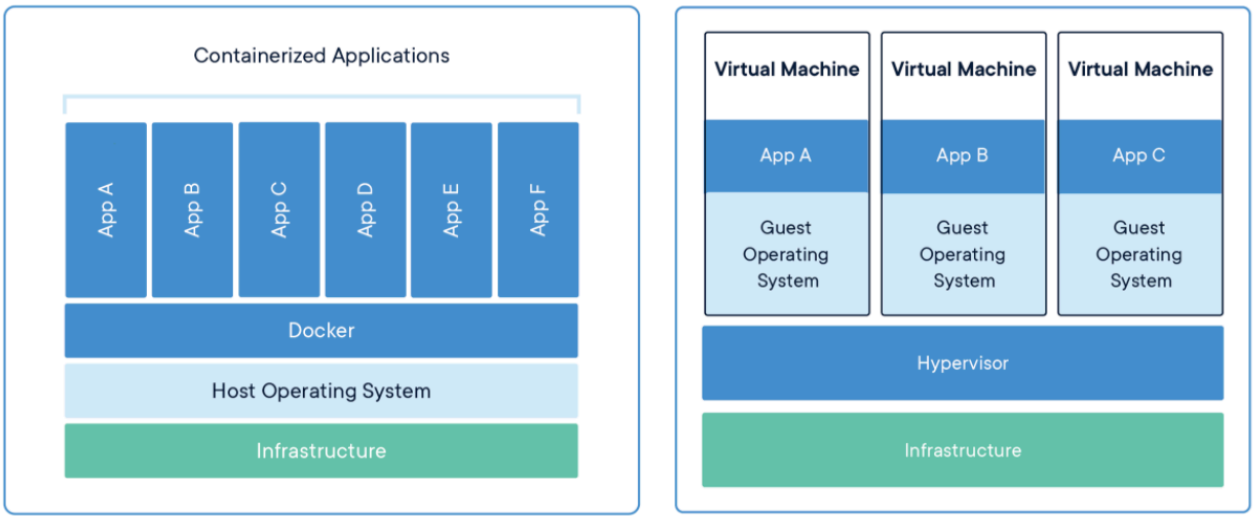


Figure 14: Differences between Docker Containers and Virtual Machines (Docker, 2017b)

Notoriously, VMs have a huge waste in terms of resources. Besides, containers are more portable and take up less space than VMs since container images are typically tens of MBs in size, while VMs are typically tens of GBs. They are also more efficient because they can handle more applications and require fewer VMs and Operating systems, which represents smaller costs and are less prone to fail.

3.4.2 Docker Swarm

Docker Swarm is an open-source container orchestration platform and scheduling tool, being the native Docker Clustering engine. One of its key advantages over standalone containers is that we can modify a service's configuration, including the networks and volumes it is connected to, without the need to manually restart the service. Docker will update the service's configuration, stop its tasks with the outdated configuration, and create new ones matching the new configuration (Docker, 2020c). The configuration definition in Docker Swarm is similar to the creation of Containers in Docker. We only need to define a YAML or JSON document with the desired structure and the rest is taken care of by Docker.

This particular orchestration technology has an advantage over the others, which is the alignment and familiarity with the Docker Container system. A Docker Swarm is nothing more than a collection of Docker Engines, called nodes, joined into a cluster. This alignment allows people already familiar with Docker to adapt easier to the Docker Swarm.

A node is an instance of the Docker Engine participating in the cluster. There are two different types of

nodes, manager and worker nodes. Manager nodes perform the orchestration, electing a single leader to conduct this task, and cluster management functions needed to maintain the desired state of the swarm. When we want to deploy our application to a swarm, we submit a service definition to a manager node which dispatches units of work called tasks to worker nodes. Worker nodes receive tasks transmitted from manager nodes and execute them. They notify the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker (Docker, 2020c).

It builds a cooperative group of systems that can provide redundancy if one or more nodes fail and also provide workload balancing for containers. It assigns containers to underlying nodes and optimizes resources by automatically scheduling container workloads to run on the most appropriate host with adequate resources while maintaining necessary performance levels. An IT administrator or developer controls the swarm using a swarm manager, which organizes and schedules containers (Naik, 2016). That said, it provides automation mechanisms such as service discovery, high availability, load balancing, health checks, real-time scalability, storage, networks, among other mechanisms.

Figure 15 portrays the structure of a Docker Swarm cluster. The document created for this service cluster states that the desired structure is to create 3 nginx replicas. Once we submit this service definition to the manager node it accepts our service definition as the desired state for the service. This means that we are always going to have 3 worker nodes running that latest version of nginx as the image of the container. If one of them fails, the manager node will automatically replace it with another node to respect the service definition, thus always having 3 nodes up and running.

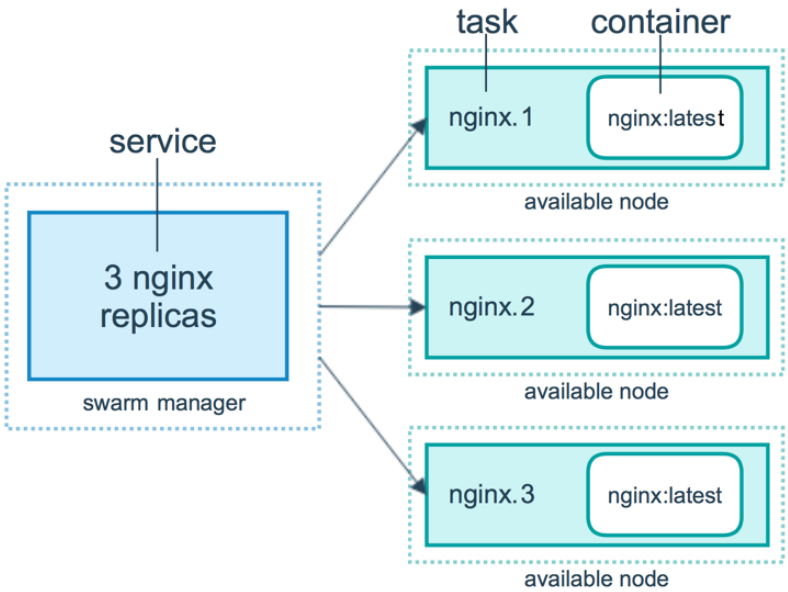


Figure 15: Example of a Docker Swarm cluster (Docker, 2020b)

3.4.3 Kubernetes

Kubernetes (K8S) is an open-source orchestration platform for managing containerized workloads, scalability, and services, that facilitates both declarative configuration and automation. It groups the containers that constitute an application in logical units for easier management and discovery of these services in the network. K8S is based on the Google Borg system and, by a decade of experience deploying scalable, reliable systems in containers via application-oriented APIs (Bernstein, 2014).

K8S is a very influential platform in the context of successfully build and deploy reliable, scalable distributed systems with virtualization by containers. It provides tools for the entire technological stack associated with container management, such as service discovery, load balancing, storage orchestration, self-healing, automated rollouts and rollbacks, automatic bin packing, horizontal scalability, secret, configuration management, among others (Burns, Beda, & Hightower, 2018).

Similarly to Docker Swarm, K8S has nodes. They are clusters consisting of a set of pods that run containerized applications and, every cluster has at least one pod. Pods are a new K8S component. They are the smallest deployable unit of computing that can be created and managed in this technology. It is a group of one or more containers with shared storage/network, and a specification for how to run the containers (Kubernetes, 2019).

Figure 16 exhibits K8S's architecture.

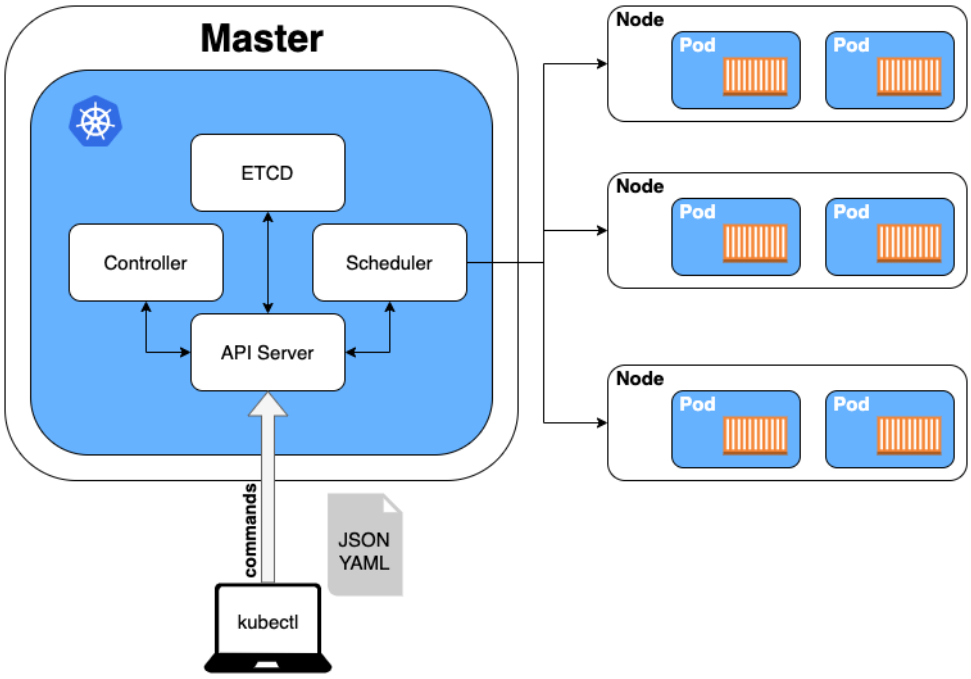


Figure 16: K8S's architecture (Adapted from (Kubernetes, 2019))

The master node is a collection of components that make up the control panel of K8S. It is the first and most vital component and manages the worker nodes and the Pods in the cluster. In production environments, it usually runs across multiple computers and a cluster usually runs multiple master nodes, providing fault-tolerance and high availability. It is composed of various components, specifically (Kubernetes, 2019):

- API Server - Is the component that exposes the K8S API. It acts as an entry point for all the REST commands used for controlling the cluster. The most common way of passing the commands is through the kubectl command-line tool. Apart from commands, it also receives configurations in the form of declarative manifests defined either in JSON or YAML. These manifests are the files we use to configure everything inside our cluster, for example, the deployment of 10 pods running the nginx image;
- ETCD - Is a consistent and highly-available key-value store used as K8S' backing store for all cluster data, like configuration details. It communicates with most components to receive commands and work. It also manages network rules and port forwarding activity;
- Controller - Manages all the infrastructure acting according to the configuration logic. It controls the nodes, replication, endpoints, and service account and tokens;
- Scheduler - Schedules the tasks to the slave node and is responsible for distributing the workload. It also watches for newly created Pods with no assigned node and selects one for them to run on.

Currently, community support on the K8S, as well as its characteristics, makes it the clear predominant solution for the best microservice deployment platform. The K8S, by itself, implements a considerable part of the MSA standards, allowing to follow a good part of the previously mentioned principles.

3.4.4 Advantages and disadvantages of containerization

The containerization growth, and adoption by the biggest companies in the world, is not unreasonable. It represents a big evolution from previous technologies and has plenty of key advantages that make it appealing to plenty of applications, such as:

- Greater efficiency and speed in the whole Continuous Model, namely CI, CDEL, and CDEP. Thereby, the deployment is reduced to seconds;
- The amount of time it takes from build to production is substantially faster;

- It abstracts the infrastructure, allowing developers to focus on development, improving their productivity;
- Return on investment and cost savings by dramatically reducing infrastructure resources;
- Compatibility, maintainability, and portability since a container works consistently in every machine and environment without additional configuration. This reduces the time spent setting up environments, debugging environment-specific issues, and a more portable and easy-to-set-up codebase (Bernstein, 2014);
- They represent a big evolution in infrastructures, software delivery, and are a big jump from previous VMs technologies, which open new opportunities to companies;
- The applications and resources are isolated and segregated, allowing various containers for separate applications running completely different stacks;
- Since applications are isolated and segregated, it is easier to have complete control over traffic flow and management;
- Containers are lightweight by design, they can be created within seconds enabling us to scale instantly and helping, for example, to react to unexpected website traffic load seamlessly (Docker, 2017a);
- The requirements of the infrastructure are no longer linked with the environment of the application.

Albeit, like any other technologies, they have their downsides, and it is important to understand them if we are considering migrating to these technologies. Some of their disadvantages are:

- Containers consume resources more efficiently than VMs, nevertheless, they are still subject to performance overhead due to, for example, overlay networking, interfacing between containers and the host system, and so on. If we want full performance, we need to use the infrastructures directly in a bare-metal way (Kubernetes, 2020);
- UIs and dashboards for those technologies are not as useful as they should be;
- They are complex and unnecessary in environments where all development is done locally, which can lead to reduced productivity;

- They can be an overkill for simple applications that are not going to have a large or distributed audience or high computing resource needs, which additionally increases the costs;
- Not all applications benefit from containers because only the ones that are designed like microservices stand to gain the most from them. Thus, if we have a big MA it will not get plenty of benefits from this technology;
- If we are not starting from a Green Field, it can be cumbersome to transition to these technologies.

3.5 Webhooks

3.5.1 APIs' limitations

Over the years, the Web evolved considerably, becoming increasingly interconnected due to components like APIs. They allow us to use features and data from one site in a completely different one. Another aspect that evolved a lot in the past few years was data, being nowadays considered the new gold. More than ever, we live in a world full of information where having the right data at the right time is the key to achieve a market advantage. With this increase in data complexity, it is increasingly necessary to transmit data in real-time. Besides, with the amount of data flowing on the internet every minute, it becomes difficult to keep an application up to date with the best and latest data. This is something that the APIs do not allow doing because they only provide an answer when the programmer makes a request, meaning that, to obtain data in real-time, we need to be constantly making requests to an API (SendGrid, 2013).

There are some approaches applied in the APIs, like the short and long polling, that try to accomplish it more efficiently Biehl (2017). In the short polling, the client asks at regular intervals for a new status from the server. However, if it does not have new data available, it just keeps sending an empty response. Some events may only happen once in a while, so we have to figure out how to make the calls, or we might miss them. In the case of the long polling, the client sends an initial polling request, but the established Hypertext Transfer Protocol (HTTP) communication channel is kept open, and the server does not send any response back until there is data available. The problem with the latter approach is that HTTP calls cannot be blocked for an arbitrary period and, eventually, they will timeout, and the client still needs to send a request to be ready for receiving the next event.

Apart from that, they both have disadvantages in common, namely, they still overload the server even when there are no changes in the data, open plenty of HTTP connections, and they do not scale well. Besides, these approaches still have some latency, and we are unable to find out, in real-time, when an

action was performed, for example, a user was created or when an error was launched in a solution in our monitoring service. Thus, this model does not meet all the needs of today's solutions and the evolving expectations of API consumers.

3.5.2 Solving APIs' limitations with Webhooks

To solve the previously mentioned limitations, we can use webhooks. They are the most widely used technique for realizing events, and a more proactive and flexible approach, for both the provider and the consumer, when compared to the APIs. It is an HTTP POST that occurs when something happens, sending an event notification to the HTTP callback that was listening to that event. They are like a Really Simple Syndication (RSS) feed for events. When an action happens, an event is triggered in real-time, and a notification is sent to all the subscribers of that event, without requiring user interaction.

An application can subscribe to an event by providing a callback Uniform Resource Locator (URL), to which the application generating the events will send a notification. Hence, webhooks do not come to replace APIs, but to complete them with new concepts like events, subscriptions, triggers, and notifications. Providing events that are well-coordinated with the available APIs is a best practice that enriches the provided service. According to Biehl (2017), this coordination allows API providers to create a competitive advantage, enabling them to differentiate in the market and providing optimal possibilities for integration. It can be applied by:

- Recognizing that there is a relation between the actions that can be performed via APIs and the state transitions that can be observed via events. So, for each method, endpoint, and action, an API provider exposes, it should strive to provide a matching event. In general, one can say, the better the coverage, the more opportunities for integration exists, and the more attractive the API becomes as a product;
- The APIs for managing webhooks need to blend in naturally with the overall API portfolio. They should follow REST constraint as far as possible, and be protected with the same security mechanisms as other APIs in the portfolio.

Webhooks can also benefit from implementations already made for APIs, namely the OAuth internet protocol, allowing platforms built on the APIs to tie into the activity of their users (Leach, 2017). This makes the service provided even more complete and efficient since these concepts are not natively supported by the REST architecture, allowing it to respond to more use cases.

That said, webhooks are a loosely coupled architecture that allows creating new innovative integrations and powerful workflows between different services that know little or nothing about each other (Stamat, 2019). This definition is perfectly aligned with the MSA's one and could even be an alternative definition of it. Most MSAs implement APIs, however, only a few of them use the power of webhooks as a lightweight mechanism to respond to the events triggered on the microservices. These, when applied to the MSA, reduce interactions that are sometimes unnecessary between the different microservices that make up the architecture, and also allow interacting with microservices outside the architecture.

The most basic patterns for realizing events are the interrupt and polling pattern. In the interrupt pattern, the client is notified by an external event source when something interesting happens and, the latter is responsible for the event execution. In the polling pattern, the client needs to figure out when something interesting happens, so the complete responsibility for event execution is with the client (Gao, Hum, Theobald, Xin-Min Tian, & Maquelin, 1996). Therefore, webhooks implement the interrupt pattern because it is the external event source that notifies the consumer when something that they subscribed to happened. As seen previously, APIs can implement the polling pattern but still have some limitations.

There is not an extensive literature review about this topic, consequently, the scientific research developed presents a more practical basis.

3.5.3 Events and subscriptions

The two main elements of webhooks are the events and subscriptions. The events are the actions produced by the application, and the subscriptions are all the events that the consumers subscribed to. Through this subscription, the client indicates its interest in receiving these events. For creating new webhooks events and subscriptions, we should treat the subscription like any other resource in an HTTP API.

Regarding the events, a good webhook service should provide as much information as possible about the notified event, as well as additional information for the client to act upon that event more easily. In the case of an MSA, it should also identify which microservice is producing it, because different microservices can produce similar events. Moreover, a type attribute should be included to help consumers manipulate the event, even if they are not being sent to a single endpoint (Iacobelli, 2016).

Regarding subscriptions, the consumer has to set up an appropriate event receiver endpoint that can process events, according to the specifications of the API provider. When webhooks are built, we should think about the consumer that will receive the data. Giving them the chance to subscribe to different events under one single URL is not the best approach because it limits them and can create problems.

3.5.4 Security

Some questions regarding security must be addressed. The first one is when an attacker tries to impersonate the sender and transmits fake events to the receiver. If the consumer's application exposes sensitive data, it can verify that requests are generated by the true sender and not a third-party pretending to be him. There are numerous ways to solves this problem. If we want to put the work on the consumer side, we could opt to request from a whitelisted IP address, but an easier method is to set up a secret token and validate the information.

Another security concern is that a faked event can be injected, or the data of an existing event can be manipulated. One way to mitigate this risk is to make the sender sign the event payload with a shared secret. This guarantees the legitimacy of both the event payload and the sender. Normally, this kind of validation is passed through the HTTP header and, with the use of timestamps, this can also solve the risk of an attacker recording the traffic and playing it back later.

The last risk is when an attacker uses a fake receiver to collect events. This risk can be solved by requiring the consumer an HTTPS endpoint with an SSL certificate, allowing the sender to verify its identity. Sometimes, self-signed certificates are used for receiver endpoints but they are not signed by a trusted authority and are thus now trustworthy (Biehl, 2017).

3.5.5 Advantages and disadvantages

Webhooks cope with numerous limitations that the APIs present and also has some advantages on its own, such as:

- It removes pressure from both the consumer and provider because the consumer does not need to make constant calls to the API to know when there is a new change available, which overload both parties;
- They provide a simple and lightweight, yet powerful, way of implementing in real-time streaming of events with low latency;
- They allow to create more powerful integrations and workflows between different two identities that know little or nothing about each other;
- Webhooks are ubiquitous across every programming language and framework, which means that everyone can receive a webhook without having to use dependencies (Leach, 2017);

- When embedded with the already implemented APIs, they create a better service and experience for both the consumer and provider;
- They can use and take advantage of some components already implemented for the APIs, like the OAuth protocol and documentation, getting more out of the already implemented functionalities.

Due to its reverse flow, when compared to the APIs, webhooks present some obvious disadvantages, like:

- For bureaucratic reasons, it can be hard to create an endpoint to receive a webhook, especially in big companies where this needs to be negotiated between infrastructure and security teams;
- Webhooks might be wholly incompatible with the organization's security model because they can be receiving data from a third-party application (Leach, 2017);
- We rely on the consumer to develop a fluid and error-free endpoint for the whole action to be performed as intended. However, there can be transmission failures, variations in latency, and quirks in the provider's implementation, meaning that even if webhooks are sent to an endpoint ordered, there are no guarantees that they will be received in that same order;
- Version upgrades are already a problem in APIs because, when providers upgrade them, they can be incompatible with the consumer's integration. However, in this case, the consumer can explicitly request a new version and verify if their integration works before upgrading. Otherwise, they can stick to the older version while they upgrade their application to integrate with the new one. In the case of webhooks, this is not that simple because the provider has to decide in advance what version to send to the consumer, and this may lead to problems;
- It puts pressure on the provider's infrastructure, and it can increase if it has millions of outgoing webhooks and some of the consumers do not set up their endpoints correctly, leading to retries and latency and, consequently, to a degraded system for everyone;
- In terms of communication, it is inefficient because it represents one HTTP request per event, which, compared to APIs, is still better but can be improved with some tricks and techniques.

4 Primavera's microservices architecture

4.1 Lithium Framework architecture

The Lithium Framework is a Primavera BSS SDK that defines a very simple, yet standard, architecture that all microservices must adhere to to be correctly generated, developed, and managed. It provides a set of components to assist and accelerate the development of microservices, according to a base architecture. Moreover, it provides runtime components, like Hydrogen (a centralized Primavera BSS's internal package for microservices with plenty of abstractions for different Microsoft and third party libraries) that provide standard and cross-cutting features, which can be reused in any microservice, creating a development standard and source code.

Some of its cross-cutting features are logging, application signature, Primavera BSS's copyright, and a test environment for the ClientLib, Web Restful API, and model projects.

To connect different microservices, and use the Don't Repeat Yourself (DRY) principle, some of the microservices developed with Lithium are also part of the framework. So, they provide common features that can be referenced and reused by other microservices.

It uses the API First Design paradigm, which is a relevant part of the entire MSA. Among other things, it allows the design/development of independent APIs per client for the same data set, independently if they are mobile, web, or another client. This allows, for example, to evolve clients and APIs at different speeds, enabling teams that develop back-end and front-end, to work independently, after the API is defined.

Figure 17 represents the Lithium Framework architecture. It has 2 different stages, the design-time and runtime.

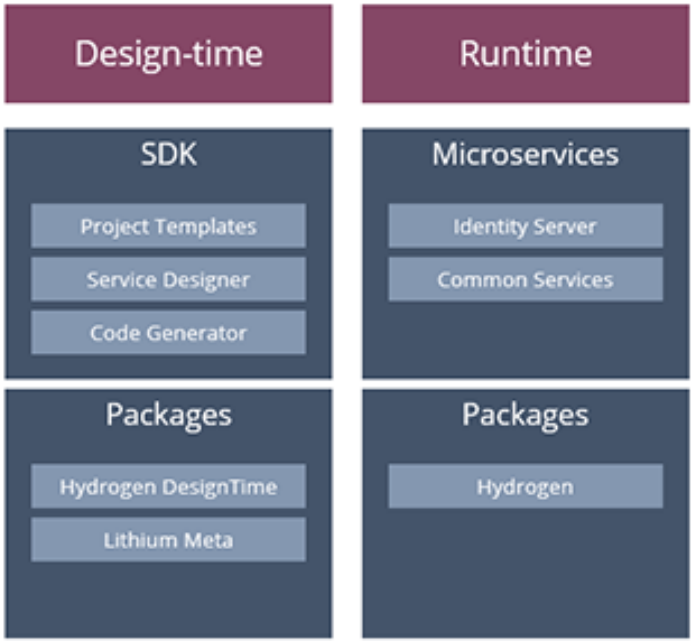


Figure 17: Lithium Framework architecture (PrimaveraBSS, 2019a)

As seen above, on the design-time, we have a drag and drop environment where we can model our microservice with a Unified Modeling Language (UML) class diagram like logic. Here we can create our entities, API actions, background services, workers, add common services, like databases, and many other functionalities. When we finish modeling our microservice, Lithium will generate all the source code from the model. Figure 18 represents an example of a service design model. As we can verify, we have those rectangular elements that are the boxes that we can drag and drop to create a model. Each color represents a different function, for example, the pink is the API controller. It is connected to other elements that constitute and extend it, in this case, one model and two actions.

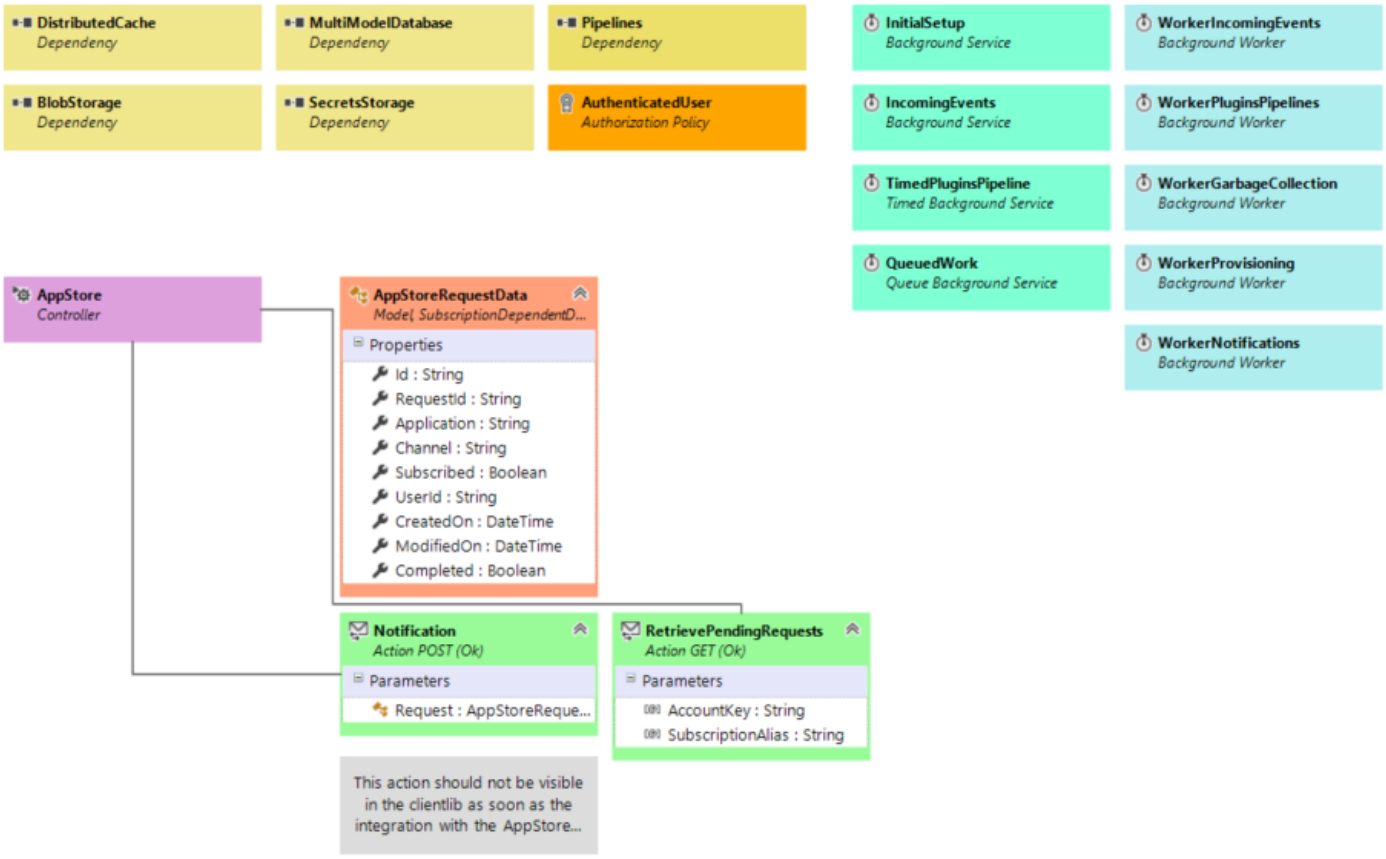


Figure 18: Example of a Lithium service design model

The code generators will produce the skeleton for all operations and leave up to the developers to implement the actual business logic. This skeleton includes all the code required to configure the ASP.NET Core and the respective infrastructure, so the developer needs to focus on the business-logic only. Figure 19 reflects the Lithium project structure generated from the model. It has two different master folders, Runtime, and Tests. Inside the Tests folder, we have the projects for testing the clientlib, the models, and the Web Api.

As for the Runtime, we have the projects for the client console application, clientlib, models, and Web API. Within the solutions, we have a GeneratedCode folder that is the folder that contains all the generated assets and code. Additionally, some of them also have a CustomCode folder so, we can extend the generated classes by creating custom code to apply our business logic or modify the default one. That said, we can develop any functionality we want in our custom code. We are not dependent on the service designer to model all our logic because it can only do so much, and it has its functional and technical restrictions.

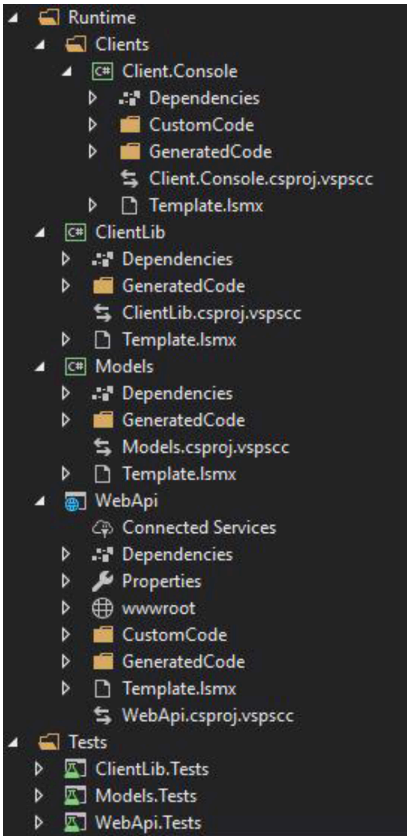


Figure 19: Generated Lithium project structure

Using the Lithium SDK and its service design model and code generation feature has plenty of advantages, such as:

- Accelerates the development of a microservice by producing its skeleton;
- Eliminates plenty of unnecessary time doing repetitive and menial coding tasks, which can be automated;
- We only need to express any action once, implementing the DRY principle;
- Since there is less handwritten code there will be fewer errors;
- It is really useful, especially for small microservices, where little business logic is needed and all the time spent developing it would be allocated creating the repetitive code;
- Already set up the basis for some of the services that a microservice can use, like the documentation, monitoring, and other services, which saves plenty of time repeating code between microservices;

- When there is an error, for example on the routing service due to a bug from a Microsoft release, a Lithium update of its framework can fix all the microservices at once without having to write a single line of code and just by doing a build of the microservice;
- Creates a development standard to which microservices adhere, making it easier for developers to understand and manage their codebase;
- Ensures that any service can be easily upgraded and adapted as the Lithium Framework evolves either in the design-time components as in the runtime components with little to no effort;
- Since this framework was created and is maintained by the company, they can customize it the way they want by, for example, adding new elements to model with, generate code differently, and implement new design patterns.

However, there are some disadvantages linked to it, for example:

- Whenever there is a bug on Lithium, developers have to wait for the responsible developer to correct the error. It can affect several microservices that will not be allowed to deploy a new version until the problem is fixed;
- Changes to Lithium will be rolled out to all generated files, so changes must be highly compatible and tested thoroughly to work in all microservices;
- There can be custom code that requires to modify some generated code parts that can lead to incompatibilities;
- The generated code is complex, which makes it difficult to understand it in the beginning. Plus, it is useful to understand what is being generated and why to understand the whole process, which makes it even more complex for starters.

4.2 Current architecture

The current Primavera BSS's MSA already includes, through the implementation of Design Patterns (DP), some MSA principles and patterns. An MSA is composed of microservices and, here, they can be divided into two types Cloud Services and InProc Services.

Cloud Services are microservices that execute in the cloud and provide a REST Web API for consumers. They implement a given set of related features, or they can act simply as proxies for other (third-party) services.

The InProc Services are provided to consumers as binary client libraries that execute the whole or part of the microservice business logic in the same process as the consumer. Typically, this happens due to performance restrictions, or because internet access is not always guaranteed. That business logic can be implemented by invoking cloud services or not.

The microservices' consumers can also be of two kinds, they can be other microservices or the end-user itself.

It is important to design a macro architecture that represents the infrastructure that will allow the management of Primavera BSS's MSA, the internal base structure of a microservice, and the technology that will compose it to satisfy the needs of microservices.

Figure 20 represents the current Primavera BSS's MSA structure. It is the representation of the current structure of the company's microservices developed using the Lithium Framework. It defines the pieces that compose the microservice and, each piece, in reality, is a .NET Core or .NET Standard project.

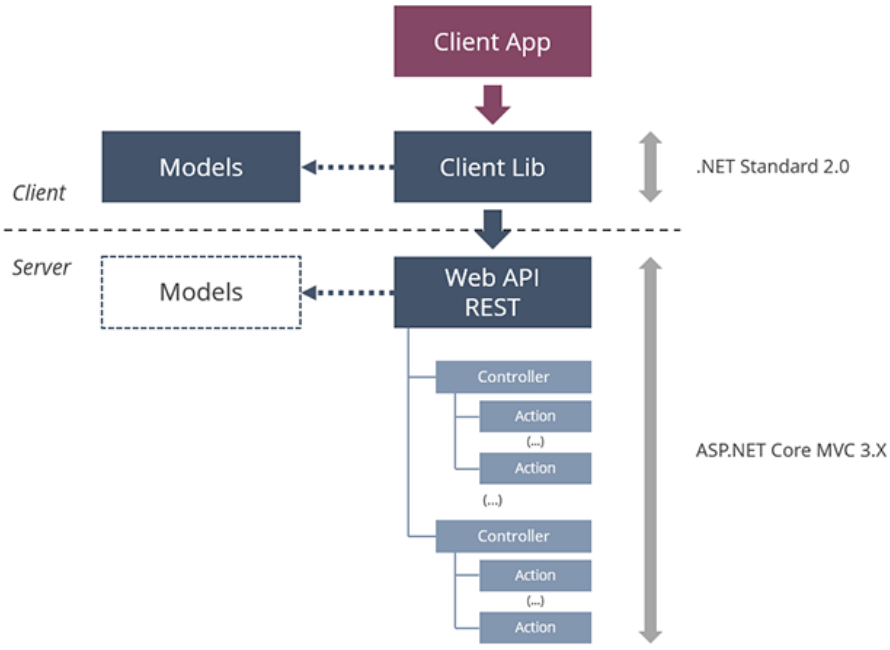


Figure 20: Current Primavera BSS's MSA structure (PrimaveraBSS, 2019b)

- Client App - Represents the application that consumes the microservice using the client library, or, in some special cases, the Web API directly;
- Client Lib - Provides an assembly that represents the Web API and allows calling it correctly and simplifies the usage of most cross-cutting features like authorization, versioning, localization, and other features. It can also be designated by Shared Libraries (Newman, 2015);

- Project Models - Are an assembly that is shared between the client library and the Web API, which defines the data structures (the models) used in the Web API actions parameters and results;
- Web API REST - Is the project where are defined the business functionalities using APIs. The microservice Web API is an ASP.NET Core Web API and it implements the RESTful style guidelines. Here is where we implement the Service Level Agreement(s) between microservice and consumer.

Being an ASP.NET Core Web API, all the microservices operations are provided in controllers and by actions inside those controllers.

All the modeling and code generation for the ClientLib and Models is handled by the Lithium Framework.

Now that we know how the Primavera BSS's MSA and SDK works, the following sections are the topics of interest identified and their research and implementation in the company. Moreover, along the way, some bugs were found and fixed in both the Lithium Framework and Hydrogen, and some suggestions were given to improve them.

4.3 Microservices Documentation

The documentation is a very important aspect of a microservice. It allows the entire team responsible for the service to have a high-level perception of what the microservices' objectives are, what makes it up internally, what is its environment, and what are the possible interactions with it. Plus, since an MSA is composed of different microservices, it is really useful if they are all well documented.

Primavera BSS did not have API documentation for its microservices. The only one they had was the one generated for the client lib. They have avoided the production of this documentation in order not to promote the direct use of the API at the expense of the use of the client lib. However, they already followed the approach of API First Design, and the base of the service was already automatically generated from the service modeling. They intended to go a step further in this regard, aiming at the adoption of the OAI specification. Therefore, since the API is generated using the Lithium Framework and it already implemented the API First Design, to adopt the OAI and create the documentation, we only needed to use tools to perform the reverse process and produce it from the generated code.

Since they work with the Microsoft stack, the most used and know tools are the SwashBuckle and NSwag. By creating a .NET Core API using the Lithium Framework, these tools, with little configuration, automatically detect the API and generate the JSON or YAML document implementing the OAS. Using this

document we can generate the SwaggerUI for the API documentation of our microservice. It is the user interface of the documentation generated by the tool.

Figure 21 shows an example of the SwaggerUI applied in one of Primavera BSS's microservices. It presents us information about the microservice itself (name and description), the schemas (the models or entities of the microservice), and the API, where we have the different controllers as titles and, under them, different actions, with their endpoint and description. Moreover, we can select different definitions of the API, that are different versions of it, and have authentication to only allow testing to users with the right permissions. For this, we used a JSON Web Token JWT with the Identity Server.

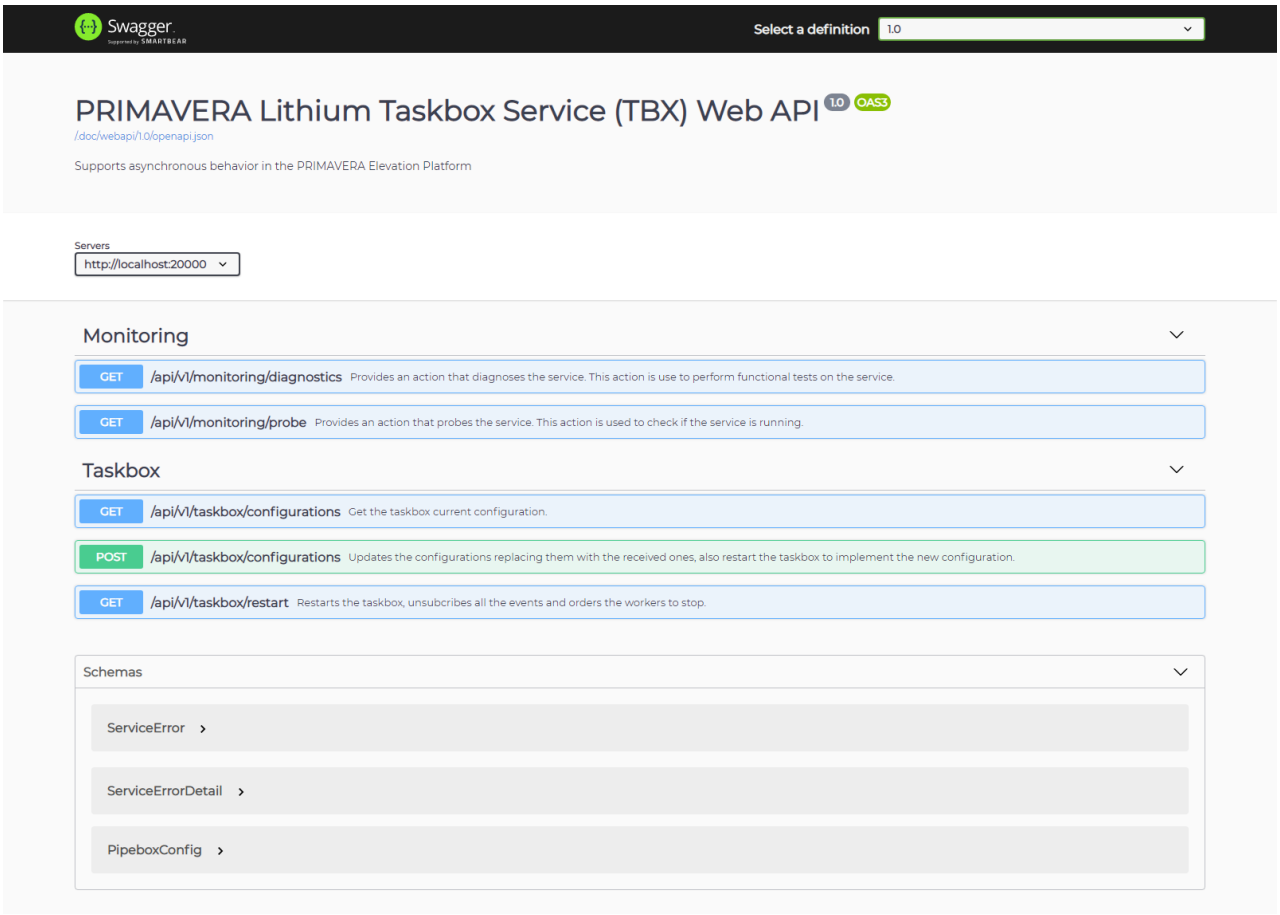


Figure 21: Example of a microservice SwaggerUI applied to Primavera BSS

When we select a specific action, a dropdown opens. Figure 22 presents an example of the specific information displayed for an action. It gives information about the parameters, request body, and responses. Parameters are the values that we can pass in our endpoint URL and shows information like the parameters available and values accepted. The request body has information, such as content type, an example of a default request body, accepted values, and other optional information. The responses expose the codes and descriptions of the possible responses, the media type, and the example of a response. All

of this is automatically generated by the SDK and then by the documentation tools.

Here, we also have a test environment for that endpoint created by the tool. That way, we can quickly test and explore any API action without having to waste time setting up our test environment.

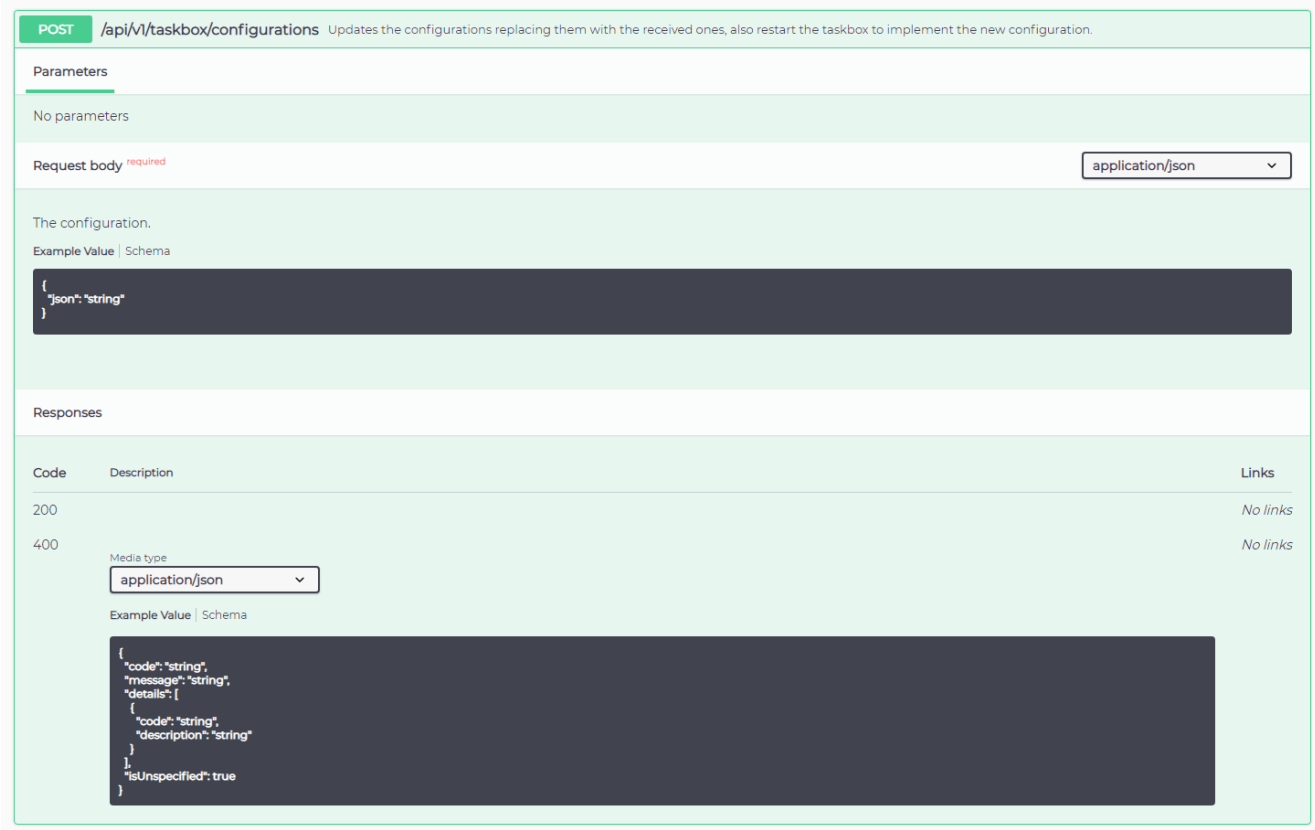


Figure 22: Example of an action and its details

4.3.1 Decision about the tool to be used

Choosing the best tool for a job is essential and makes our work easier. After extensive research and test of both SwashBuckle and NSwag to find the one that fits best the requirements of the company, we concluded that the NSwag is the best choice.

Overall, they are similar, but all in all, the NSwag workflow offers more than SwashBuckle. It not only provides the functionality of SwashBuckle (OAS generation and SwaggerUI) but also generates complete, robust, and efficient API client code for C# and TypeScript. We also have the choice to use it directly in code and, this way, we can avoid incompatibilities, and offer more features and a more streamlined toolchain.

When applying both tools in the Lithium Framework, it was found that NSwag automatically fills the documentation more precisely, robustly, and with fewer errors. SwashBuckle needs a separate library to

read the inherited commentaries from extended classes, otherwise, it does not recognize them, and some fields, such as the API controller description, appear blank.

The basic configuration needed to set up the NSwag, allowing it to generate the documentation, is composed of the configuration of the tool itself and its addition to a microservice's middleware pipeline. Figure A.2 represents the method to register the Swagger services to the microservice itself. Here, we can configure every information that is going to be displayed in the documentation page and the authentication. The authentication can be of several types. In this case, it is being implemented to authenticate through Primavera BSS's Identity Server, but it could also be done through a JWT token directly. As all the code presented for this topic, it was developed using C# and ASP.NET Core.

Figure A.3 exhibits the method created to add the documentation to the microservice's middleware pipeline. It can be placed anywhere in the middleware, the only requirement is that it is after the version declaration. The middleware will be used to generate the Swagger specification (UseOpenApi()) and SwaggerUI (UseSwaggerUi3). In these methods, it is possible to pass additional parameters to configure them. If the microservice has more than one version, and we want to generate the documentation to support all of them, we can use a method like the one shown in Figure A.2. More features and extensibility were developed for the company prototype. However, for privacy reasons, these are the only parts we are allowed to analyze.

That said, a final prototype using the NSwag was developed and presented to the team responsible for implementing new technologies in the company. The project was approved and, then, a guideline document with the steps to implement successfully this tool and its best practices was develop. It was already implemented successfully in Primavera BSS's microservices and is now being used in production. To note that, during this process, some bugs were found and fixed, namely a versioning problem, which also helped to make the SDK more stable.

4.4 gRPC

gRPC is a tool that has been gaining plenty of traction, and Primavera BSS was interested in researching if it was a good adoption to their MSA.

To test the Protocol Buffers and the way they work, a simple example, like the one presented in Figure 23, was developed.

```
// "message" keyword for defining data structure
// "service" keyword for defining service
// "rpc" keyword for defining function of a service
syntax = "proto3";

import "google/protobuf/empty.proto";

option csharp_namespace = "Primavera.Lithium.Sample.WebApi.CustomCode";

package alive;

service Alive {
    rpc Sum (SumRequest) returns (SumReply);
}

message SumRequest {
    int32 number1 = 1;
    int32 number2 = 2;
}

message SumReply {
    int32 message = 1;
}
```

Figure 23: Example of a Protocol Buffer

In this example, we are defining a service called "Alive" that contains a method called "Sum". This method sends a request of the type "SumRequest" that defines two numbers as parameters and receives a reply of the type "SumReply" that returns a message with the total of the sum.

When the desired structure for the Procotol Buffer has been created, we can generate the gRPC service's code from it, for both client and server applications. To do that, we need to reference the files that we want to generate from, and use a gRPC library, developed for the programming language we are using, to execute that task. The Protocol Buffer file should be the same on both sides, otherwise, for example, if the server creates new functions, but it does not expose the new file to the client, it will not be able to execute them because we will not know they these functionalities exist.

The assets generated by the gRPC library are different for the server and client, and they are generated on an "as-needed" basis every time we build our project. On the server-side, it generates an abstract service from the base type, and it contains the definition for all the gRPC calls defined on the Protocol Buffer file. Once it is generated, we can extend it, creating our custom and concrete gRPC service and implementing our business logic there. Furthermore, we can use popular features in our services, such as logging, dependency injection, authentication, authorization, and other features. Although, on the client-side, it generates an asset concrete to the client type that translates the file into concrete type methods that the client can invoke to request a function from the server.

Figure A.5 is a simple example of a gRPC service called Alive. It is a concrete service that derives from

its abstract service base class that is the one that was generated by the framework and implements the business logic for the Sum method that simply sums two numbers. Here, we also use the ILogger feature to log all the operations executed by this class to our console. To be able to use the gRPC, we need to add it to our microservices' configuration, as seen in Figure A.6. It also configures the microservice to support authentication using a JWT Token. Figure A.7 represents the method to configure and generate a JWT Token to our gRPC application.

After having the gRPC service enabled, we need to add it to the middleware pipeline to be able to use it when a request comes through. Additionally, we need to map the gRPC services that we created, to be able to access them, and create a route to allow the client application to generate tokens, as observed in Figure A.8.

For the gRPC endpoints to work, they require an HTTP/2 connection and must be secure with TLS. Primavera BSS's middleware and features share the routing pipeline. Hence, additional request handlers, like MVC controllers, work in parallel with the configured gRPC services. This way, it allows a microservice to have a REST API and gRPC services working at the same time, which is good to migrate from one to another, or if we want to offer both options. For this to be possible, we also need to configure our application to use both HTTP/1 and HTTP/2 as the default protocols for our endpoints. Furthermore, we have to verify if our endpoint has the prefix "https", and if it does, the endpoint is using TLS; otherwise, it is not, and we need to set it up because gRPC also needs it to work.

A practical benchmark was conducted to test the time that it takes for REST API and gRPC service to respond to a request. The REST API was developed using the Primavera BSS's Lithium Framework, and a tool called Postman was used to measure its response time. As for the gRPC, the code was generated from the Protocol Buffer, and we had to capture the response time through the client application terminal since no tool is still available to ease this benchmarking process. For that, a simple gRPC client application, like the one presented in Figure A.9, which makes the sum of two numbers and returned the total, was developed to test this approach.

Figure 24 compares the speed of response to a request between a REST API and gRPC.

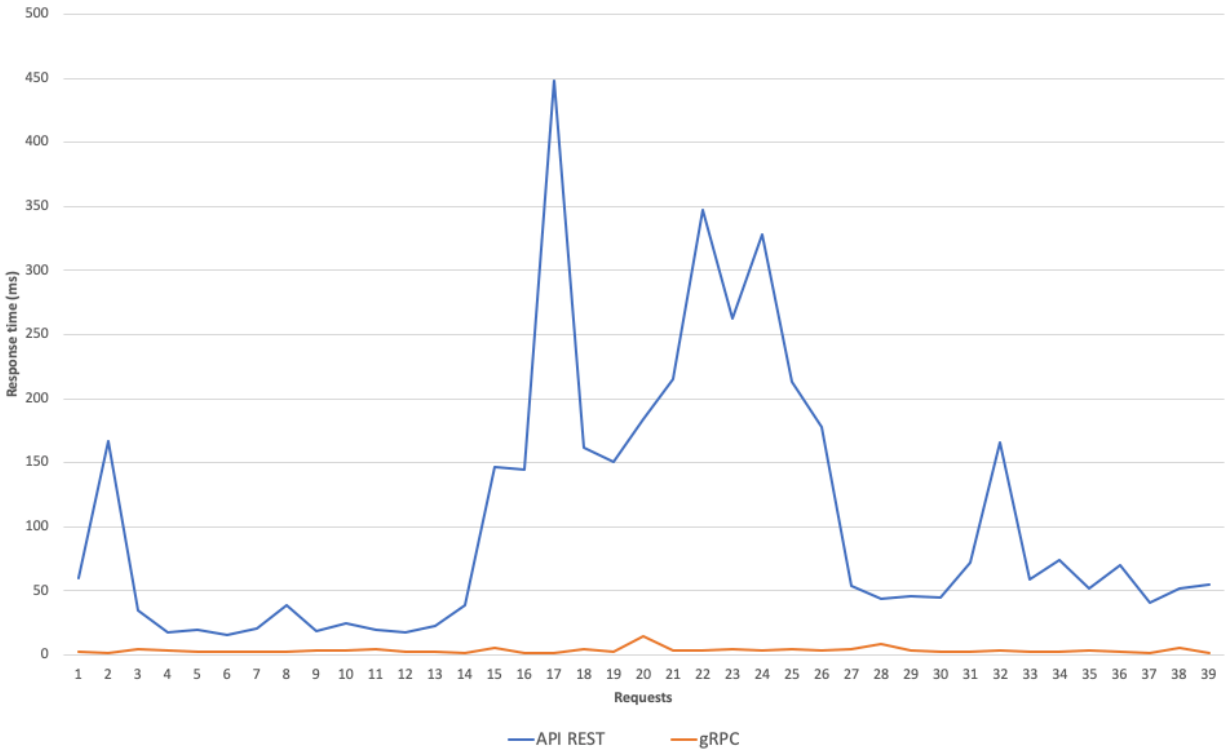


Figure 24: Speed of response to a request between a REST API and gRPC

A total of 40 tests to both the REST API and gRPC programs were executed. The average time that the REST API took to respond to a request was 97.43ms, while gRPC only took 3.66ms. The gRPC not only took more than 30 times less time to respond to requests, but it also was more consistent in general having a low variation between the slowest and fastest responses. On the other hand, the REST API is significantly more unstable, having times as low as 27ms and as high as 449ms.

All the gRPC theory and technical details were demonstrated through the developed prototype that was presented to the team responsible for implementing new technologies in the company. The prototype not only implemented all the gRPC related questions, but also the repository pattern, authentication through JWT, a logging system, and usage of Dependency Injection to inject the context of the repository. However, the pieces of code presented here are only a small portion of the entire prototype that was developed to the company, which can not be entirely shown for privacy reasons.

4.4.1 Obstacles

Although gRPC is a technology with plenty of competences, it has some drawbacks and obstacles because of its characteristics or because of Primavera BSS's current reality, such as:

- Migration is hard and requires time, not only for the implementation but also for developers to get

used to it, which can take a long time, depending on the learning curve;

- It has costs inherent to the migration to new technology, infrastructures to support it, and other indirect costs;
- It requires an entire restructuring of Primavera BSS's Lithium Framework to start generating code using this technology, either exclusively or in parallel with the REST APIs;
- The additional complexity of gRPC over APIs can create difficulties when adopting and using it;
- There are currently some issues with gRPC on macOS. It does not support ASP.NET Core with TLS and additional configuration is required to successfully run gRPC services;
- At the moment, it is hard to implement the HTTP/2 gRPC on the browser, as there is simply no browser API with enough fine-grained control over the request.

After a debate with the responsible team, it was concluded that the gRPC is not going to be implemented in the company, for now. At the moment, it is still giving its first steps and has some obstacles that make it a deal-breaker. However, the company showed interest in its potential and is going to revisit this technology in the future.

4.5 Microservices Deployment

The deployment affects almost the entire software development process at Primavera BSS, as it impacts the implementation of the Continuous Model, and also implies the work of multiple departments, namely the CMS, SWE, and INT. The goal, in the context of this dissertation, is to start exploring the deployment platforms, namely Docker, Docker Swarm, and K8S, and propose a deployment pipeline architecture that implements them and the whole Continuous Model.

Currently, Primavera BSS's microservices run on the Azure cloud using Web apps. The tools used to implement the deployment pipeline and Continuous Model, specifically the build, release, and deployment processes, are the Azure DevOps Server (formerly Team Team Foundation Server (TFS)) and Release Manager. Both tools are compatible with deployment platforms, like K8S and Docker.

The Azure DevOps Server and Release Manage allow the creation of orchestration workflows for different stages and respective transformations to be made in the artifact produced by the programmer, from testing, creation of environments, among many other operations. Each microservice can be deployed to the following environments:

- Development (dv) - Used by the development teams to create and test versions under development;
- Staging (st) - Used by the development teams for regression tests;
- Preview (pr) - Used for testing to check if it is ready to be distributed to the consumers and for fast swapping to production;
- Production (pd) - Final and stable version used by the consumers.

There is a release configured on the Development branch and another for the Mainline branch. These branches are the ones where Primavera BSS stores the source code for each microservice. The Development release can only be deployed to dv and st, and the Mainline release is allowed to be deployed to all the environments.

4.5.1 Docker

After analyzing the current deployment practices at Primavera BSS, we need to adapt them to apply the containerization. To test these technologies, in the company's context, a small MSA was developed and deployed using them. Figure 25 presents the MSA developed to test the deployment tools.

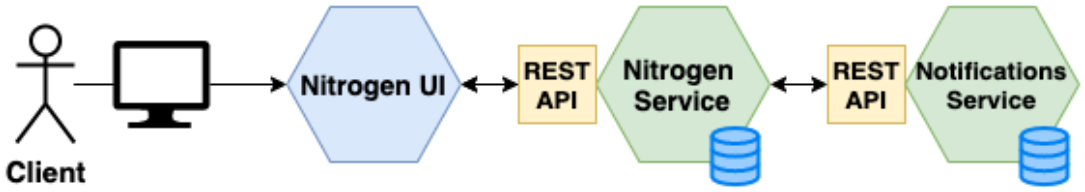


Figure 25: MSA developed to test the deployment tools

It is composed of two Primavera BSS's microservices that interact with each other via a REST API, and each one of them has its database. Moreover, it also has a UI that is the entry point with which the clients interact with the application, and it is connected to the Nitrogen microservice. Creating this way the front-end and back-end, respectively.

The first step is to create the Docker images for the microservices containers as seen in Figure 26 and, to achieve this, we have to produce a DockerFile per microservice.

```
#Nitrogen Microservice Back-end
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2
WORKDIR /NTR
COPY /NTR /NTR
ENTRYPOINT ["dotnet", "Primavera.Lithium.Nitrogen.WebApi.dll", "--environment=Production"]

#Nitrogen Microservice Front-end
FROM nginx:alpine
COPY /wwwroot /usr/share/nginx/html
```

Figure 26: Example of a DockerFile

A DockerFile is a text document that contains all the commands users can call on the command-line to compile a Docker Image. It executes these commands instructions in succession to build images automatically. In this example, we can see how the images for both the Nitrogen microservice and UI were created. To start, a DockerFile must begin with a "FROM" instruction specifying the parent or base image from which we are building. Another instruction we have to use to create a DockerFile is the "COPY" instruction. It copies new files or directories from a source and adds them to the filesystem of the container at the destination path (Docker, 2020a). There are more instructions we can use to define our DockerFile, but these are the two most basic ones needed to deploy an image, as seen by the Nitrogen Front-end example. To note that, in this deployment, we configured the environment of the Nitrogen microservice as Production to further test the Docker capabilities and see if it responds to the company's needs.

Once all the DockerFiles are defined, we have to build the images from them, in this case, using the Docker Command-line Interface (CLI) and each DockerFile will result in a different image. When the images are built, we can run them locally to test if they are working as expected. To test them, we have to create a container for each and assign an URL to access them.

When our containers images are created and working the desired way, they can be stored in a container registry service to be easier to manage, version, and deliver them to other developers and teams. The most known container registry service is the Docker Hub. However, this is not the one that we are going to use because the Primavera BSS's container images, and related artifacts, have to be private since they are microservices that constitute their MSA and can not be shared with the public. Docker Hub has a paid plan with private repositories but, since the company has a partnership with Microsoft, it is already familiarised with the whole Azure stack and mainly uses it to all their cloud concerns and has a custom price plan for it. That said, we are going to use the Azure Container Registry as a private repository to store the container images. Since both options offer similar solutions and features, there is not a big difference between them, so the price and familiarity with the solution were the main decision factors here.

4.5.2 Decision about the tool to be used

After the container images were created and stored in the Azure Container Registry, we started connecting them using orchestration tools to compose a functional application.

The main orchestration tools, as seen previously, are the Docker Swarm and K8S. Both are open-source systems that provide similar functions such as automating deployment, scaling, and the management of containerized applications. One drawback of K8S when compared to Docker Swarm that is the installation process is more complex and time-consuming. However, this technology has plenty of advantages over Docker Swarm, like:

- It has broader adoption, growth, support, and popularity among all the container orchestration solutions. It gained a very large active user and developer open-source community, as well as the support from the biggest companies in the world and major cloud providers;
- Due to the large and diverse user community and the team behind K8S, new features are constantly being released, with a bigger pace than Docker Swarm;
- They offer more automation and configuration of its infrastructure;
- It implements some features that Docker Swarm does not have, namely, the auto-scaling, manually configured settings for load balancing, and GUI. Moreover, it supports a wide spectrum of workloads, programming languages, and frameworks, enabling stateless, stateful, and data-processing workloads. These features make K8S flexible enough to meet the needs of a wide range of users and use cases (Burns et al., 2018);
- It has an in-built library and process for monitoring and logging, while Docker Swarm has to use third-party applications, like ELK, to implement these features (Modak, Chaudhary, Paygude, & Ldate, 2018);
- Docker Swarm is limited to the Docker API's capabilities, while K8S can overcome the constraints of both Docker and Docker API.

After analyzing the main advantages and disadvantage that K8S have over Docker Swarm for container orchestration, it is safe to say that this is going to be the technology used to orchestrate the containers. K8S excel in both power as well as performance and enables greater flexibility in container management. This is why, not only in Primavera BSS's case but in most of the use cases, K8S is the best orchestration

technology to adopt. Even in small use cases, the initial burden to set up K8S over Docker Swarm is worth it because its features and deeper configuration can be beneficial.

4.5.3 Kubernetes

K8S is a good implementation as a microservice deployment platform since they allow, in an easier way, the implementation of many architecture standards, as well as better management of the deployments. It takes full advantage of applications built out of loosely coupled services that can communicate through APIs, which is the case of microservices. Therefore, Primavera BSS's model can be improved by using K8S due to its features and more efficient behavior in containerization and deployment.

The creation of deployments and respective configuration is done in a declarative way, through YAML manifest files, which facilitates the work of the DevOps team. The first step to implement a K8S cluster is to define a manifests to configure the object we want to use to deploy.

There are numerous types of objects, namely Pods, Replica Set Controller, Deployment Controller, Daemon Set Controller, Stateful Set Controller, and many others. As seen previously, Pods are the smallest deployable units of computing that can be created and managed in K8S. Replica Set Controller is an abstraction of Pods that maintain a stable set of replica Pods running at any given time, maintaining its availability. Above the latter is the Deployment Controller, which is the most important one as it represents the deployment of an application and it abstracts the Replica Set. Additionally, it also defines the number of replicas, the strategy of updates and rollbacks, allowing to manage multiple versions of the same application simultaneously, among many other features. The Daemon Set Controller and Stateful Set Controller derive from the Deployment Controller (Burns et al., 2018). They do almost the same but with more complexity in certain areas and are better, for example, to guarantee data persistence at the container level.

The one that fits better the current Primavera BSS reality is the Deployment Controller, which is also the most used one in general. It allows configuring everything needed without the extra complexity that the company does not need, or has specific use cases that can fully benefit from the other controllers.

Figure 27 represents the deployment manifest for the Nitrogen microservice.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ntr-deployment
  labels:
    app: ntr
spec:
  replicas: 3
  selector:
    matchLabels:
      app: ntr
  template:
    metadata:
      labels:
        app: ntr
    spec:
      containers:
      - name: ntr
        image: primaverabss.azurecr.io/dev/ntr:v4
        ports:
        - containerPort: 80
        volumeMounts:
        - name: config-volume
          mountPath: NTR/app/config
      volumes:
      - name: config-volume
        configMap:
          name: demo-config
      imagePullSecrets:
      - name: primaverabss
```

Figure 27: Deployment manifest for the Nitrogen microservice

In the manifest, we describe the desired state, and the Deployment Controller changes the actual state to the desired one at a controlled rate. We can define Deployments to create new Replica Sets or to remove existing Deployments and adopt all their resources with new ones (de la Torre, 2018). They have several fields inside their manifests to define the desired state and configurations.

For the company, we are creating the first version of a Deployment, named ntr-deployment, as indicated by the first three fields. It creates three replicated pods with the selector field of ntr. This field defines how the Deployment finds which Pods to manage and represents their identification in the cluster. Inside the template field, we have sub-fields to define the content of the Pods themselves. First, we define that the Pods are labeled ntr. After that, we indicate which container image the Pods are going to run, which is the Nitrogen microservice we previously uploaded to the Azure Container Registry using Docker, and, as we can see, we are using the fourth version of this image.

Since we used a private container registry, we need to create a secret to pull our image from there, otherwise, anyone would be able to pull it and they would not be private. For that, we can use, for example, the K8S CLI (kubectl) to create, store, and manage our container registries connection secrets. The last

main step to have a functional deployment is to define the port that we want to assign to our containers, in this case, port 80.

We also defined volumes for our Pods because they prevent our containers from losing on-disk files and information when they crash and K8S has to restart them. When running containers together in a Pod it becomes easier to share files between those containers. Apart from all the technical details, in short, in this example, we are creating three Nitrogen replicated Pods.

Once the deployment manifest was structured as desired, we have to create it by running it on the K8S CLI. Henceforth, our microservice was deployed and we did not have to do anything more because everything was managed by K8S. For example, if a node fails it will instantly be replaced by another functional one with the same characteristics. It automatically handles scalability, load balancing, failover, keeps the deployment standards, logging, monitoring, network requirements, and many other aspects. There are some critical cases where K8S can fail, but with the proper precaution, we can prevent that.

If we want to update our deployment, we can do it the same way we do to create one, by creating the desire fields to structure our manifest and run it with the K8S CLI. It is smart enough to detect the changes between deployments and modify solely that part. For example, if we update our deployment from three to five Pods, it will not stop the existing three Pods to create the new desired five, it will only create two new ones since there are already three Pods functioning properly. Although, this can change if we modify, for example, the version of the image, in which case, five new ones with the updated image are going to be created.

4.5.4 Decision about the service to be used

After we chose our container orchestrator technology, and we know how to work with it, we needed to decide on which service to implement and manage it. Since Primavera BSS uses the Azure ecosystem, we explored it more in-depth in this dissertation. This cloud provider is one of the biggest worldwide and has plenty of offers concerning containerization. It has three main options to deploy our apps using container orchestration: Azure Web Apps, Azure Container Instances (ACI), and Azure Kubernetes Service (AKS).

Azure Web Apps is the most simple of the three, and the one Primavera BSS uses at the moment. It deploys web apps to containers which allow new teams in the containerization world to start deploying their first containers more easily (Microsoft, 2019d).

The ACI's main value proposition is the fully managed hosting environment for container workloads that are billed only for the processing time it takes to start them and run the workload. It allows us to right away deploy containers to it without the need to maintain that environment or to upgrade/patch the underlying

operating system or VMs and without having to adopt a higher-level service. All that is transparent, and we just deploy containers into a ready-to-use environment, enabling short-lived containers that respond to on-demand events (Microsoft, 2019e).

Finally, the AKS is the managed K8S hosting environment supported by Azure. It makes it quick and easy to deploy and manage containerized applications without container orchestration expertise while also giving operational control. The K8S master nodes are managed by Azure, and we only need to operate and maintain the agent nodes (Microsoft, 2019c).

Taking into account price, the ACI presents significant cost savings for services that do not require constant up-time charges at an always-on rate. The AKS is free, and we only pay for the agent nodes within our clusters, not for the masters. In Azure Web Apps, we have to pay per App, and it can vary depending on the App plan.

Regarding scalability, ACI provides a group of instances that is a collection of containers that get scheduled on the same host machine (similar to the concept of Pod in K8S). AKS is a scalable solution by design and meets growing demands with its built-in application autoscaling. The Azure Web Apps is also a scalable solution, but we have to pay for a special App plan.

When it comes to monitoring and diagnostic tools, ACI provides container logs and few alert diagrams, but it is not enough because if something happens with a hosted machine, it is hard to detect the problem. AKS provides full control to monitor the state of our application. Azure Web Apps also provides a few alerts and tools to analyze our solution, but again, we have to pay for a special App plan to have these functionalities.

Concerning features, AKS is the most complete one because ACI lacks some, namely SSL encryption. As for Azure Web Apps, it is robust in terms of features, and they are easy to implement since we only need to use the Azure Portal UI to configure them. However, we have to pay for most of every new functionality we want to adopt.

After defining the three main options to orchestrate containers, and comparing them in different aspects, we could decide which one fitted better for Primavera BSS. Teams that are starting on the cloud-native journey should opt for either Azure Web Apps or ACI since they have a great environment for migrating to containers without plenty of expertise. However, since Primavera BSS's cloud team already had expertise in the cloud world and needed a service that allowed them to fully control and configure their deployments, the AKS was the best service to adopt. Moreover, AKS is the best option for large Production environments and the one that works better with microservices (de la Torre, 2018).

4.5.5 Pipeline proposal

Containers make it easy to continuously build and deploy applications. By orchestrating the deployment of those containers using K8S in AKS, we can achieve replicable, manageable clusters of containers. By setting up a continuous build to produce container images and orchestration, Azure DevOps increases the speed and reliability of deployment (Microsoft, 2019b)). That said, the CI, CDEP, and CDEL flow can be improved with containers.

Since Primavera BSS still did not have containerization implemented in its Continuous Model workflow, a new pipeline model needed to be developed, taking into account the technologies and services previously chosen. Figure 28 presents the proposed Azure DevOps deployment pipeline implementing the Continuous Model.

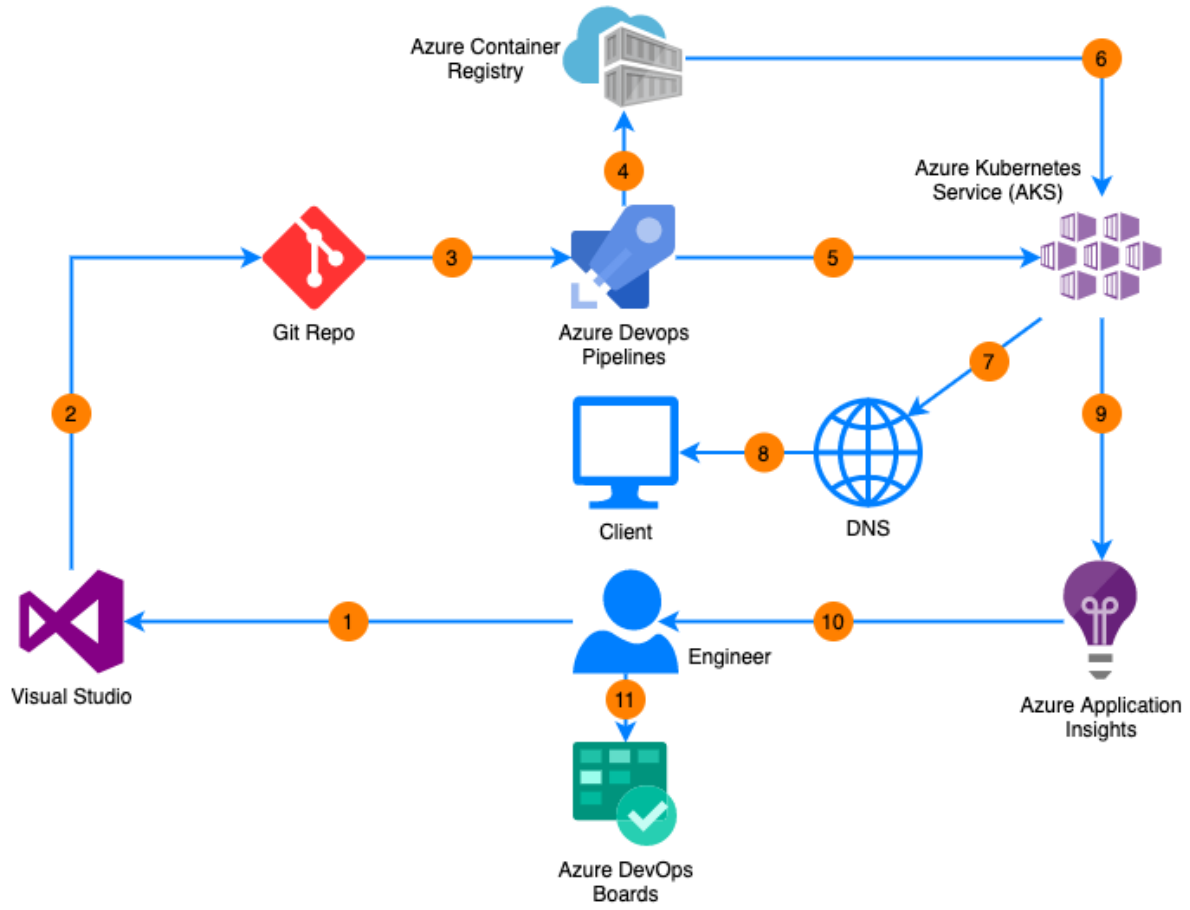


Figure 28: Proposed deployment pipeline (Adapted from (Microsoft, 2019b))

The flow starts with the engineer creating or making changes to an application source code. Once the code is stable and working the desired way, it is committed to the respective source code repository, in this case, Git Repo, but we can use Azure Repos as well. With the code committed, the CI triggers the application build, container image build, and unit tests. If tests are passed successfully, the updated

container image is pushed to the Azure Container Registry. Simultaneously, the CDEP trigger orchestrates the deployment of application artifacts with environment-specific parameters.

Once the last steps are completed, the artifacts are deployed to the AKS, and the container is launched using the updated container image pushed to the Azure Container Registry. Here, the AKS does all the work to put our microservice up and running following our manifest file configurations, and it is now available to everyone that has access to it, depending on the intention and type of the microservice. If the microservice is meant to be accessed directly by clients, it has an external IP address it can be reached with and is now available. Independently of its purpose, it is ready to be connected and integrated with other microservices to create an MSA. Afterward, it is connected to the Azure Application Insights, which collects and analyses health, performance, and usage data. Then, this backlog information is monitored by engineers, who use it to prioritize new features and bug fixes using Azure Boards.

4.6 Configuration Management

Configuration Management is a simple concept, yet it streamlines and saves developers plenty of additional work. Primavera BSS realized that it was an easy technique to implement in their architecture that could bring plenty of benefits to the company's software, given their current MSA.

One of the use cases that Primavera BSS benefited from with this approach was the fact that they have more than one server for some of the environments (development, staging, preview, and production) in a microservice release process. These settings were changed manually for each server in each microservice. When applying the concept of external configurations, we started to be able to change the configuration only once, and it implemented the changes in all the microservices on the different servers to which it applies.

To contextualize in practice, if a microservice in total has five servers throughout the release process, it will be necessary to change the settings five times in the web apps of the different servers and stages of release. Considering the time necessary to change these settings, the number of people needed to do it, and their respective salary, the benefits continue to increase if we multiply this process by the more than twenty microservices that Primavera BSS currently has.

All this, combined with the functional organizational structure of the teams, which leads to the development and deployment being carried out by different teams, generates the potential to bring many benefits and streamline the process of updating the settings. We concluded that this is a lengthy and expensive operation to do, and we could save plenty of money, time, and avoid additional errors if we adopted a centralized configuration management service. This had a monetary cost, but it is lower than the current

approach and brought other benefits.

4.6.1 Decision about the tool to be used

Within the external configuration tools, and taking into account the current organizational reality of Primavera BSS, there were two that stand out more for the implementation of this functionality. They are the Microsoft Azure App Configuration Service because the company uses the Microsoft stack (C#, .NET Core, ASP.NET Core, Azure), and the K8S ConfigMaps, because we previously studied deployment tools.

The Microsoft Azure App Configuration Service provides a service to centrally manage application configurations and feature flags through key-value pairs with a few more configurable parameters. It is ideal for microservices based on AKS, Azure Service Fabric, or other container applications deployed in one or more geographies, serverless applications, which include Azure Functions or other event-driven stateless applications, and continuous deployment pipelines (Microsoft, 2020).

ConfigMaps allows disassociating configuration artifacts from the image content to keep applications in portable containers. The contents of the ConfigMap can be injected as environment variables or mounted files. It is possible to achieve this through individual configuration settings, reading configuration files, or reading configuration files directories (Kubernetes, 2020). This approach makes more sense when there is already greater adoption and maturity of container technology by the organization because its implementation becomes simpler and more effective.

Figure 29 is an example of a ConfigMap applied to a Primavera BSS microservice. Here, the Logging settings for the microservice were externalized, which allowed us to change its values and debug the microservice the way we want.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: demo-config
data:
  appsettingsteste.json: |-
    {
      "Logging": {
        "LogLevel": {
          "Default": "Error",
          "System": "Error",
          "Microsoft": "Error"
        }
      }
    }
```

Figure 29: Example of a ConfigMap applied to a Primavera BSS microservice

After comparing the two approaches, the adopted tool was the Microsoft Azure Application Configuration Service. This was the best option because, at the time, Primavera BSS had not implemented and matured container technologies. Plus, they already used the Microsoft stack and several Azure services, which gives them a company price plan for those services.

Another key factor was that, currently, the internal refresh on changes to the .NET main file provider does not work with symlink files. The ConfigMap does not trigger the refresh of the configurations as expected. This appears to be because the discovery of .NET Core changes depends on the date the file was last modified. As the file we are monitoring has not been changed (the symlink reference has changed), no changes were detected. To date, this problem has not yet been solved, however, for now, we can solve this problem by taking advantage of the .NET Core extensible configuration system. We can implement a file-based configuration provider that detects changes based on the contents of the file, as presented in Figure A.10.

It is important to note that, even if container technology is adopted in the future, this solution remains valid and, thus, does not need to be changed and can still be the best option.

4.6.2 Microsoft Azure App Configuration Service

After the tool has been chosen, we can start its implementation. The first task to execute is to go to the Azure portal, select the "App Configuration" service, and create a new App Configuration Store. Here, we can create our configurations manually or import them either from other configuration stores or files from our application, for example, the famous appSettings.json. In the future, we can export those configurations from the store to other ones or even to configuration files.

After this first step is completed, we can start using this service in our microservices by connecting and configuring it as presented in Figure A.11. We have to configure parameters like the connection to the App Configuration store instance that is being used. Since we are dealing with important information that we do not want to share with anybody, we need to use a secret manager to store this connection string.

Another useful method to configure this service is the Select that allows selecting only the configurations that we want between all the available ones, by key and label filter. It is at this point that it will be checked whether the implementation of the namespaces for the configurations was successful or not. To take advantage of the full potential of this feature, in the case of Primavera BSS, and in addition to the good practice exposed in the literature review, it would be wise to define the configuration labels with the release process to which it applies (dev, stg, pre or prod).

ConfigureRefresh is another method that we can use to configure our App Configuration. It specifies

the settings used to update the configuration data in the App Configuration store when an update operation is triggered. However, to trigger an update operation, it is necessary to configure an update middleware so that the application updates the configuration data when any changes occur. This way, we keep the configurations updated and avoid many calls to the configuration store, using the cache for each configuration. Until a configuration's cached value expires, the update operation does not update the value, even when the value changes in the configuration store. The default expiration time for each request is thirty seconds, but can be overridden if necessary. This way, we grant that the configurations are always up to date, with a maximum of thirty seconds of delay between updates, and we save our Azure App Configuration service from redundant and unnecessary calls, which cost money. Moreover, since excessive requests to the App Configuration Store can result in throttling or overage charges, we are preventing them also.

With the initial set up of the tool completed, we needed to start filling the Configuration Store. As said previously, it stores configuration data as a key-value pair. To create an efficient way of using this service, we needed to practice good management of the key namespaces, otherwise, it would get confusing and we could have problems in the long run. To take advantage of the full potential of this feature, in the case of Primavera BSS, the good practices previously exposed were taken into consideration and, the best approach was to define the configuration key as a composition of the microservice and configuration name (for example "OrderService:ConnectionString") and the labels as being the environment to which it applies (for example "Development"). Adding this to the fact that we can use a method that allows us to select only the settings we want from all the available and we have an agile system of configurations that can respond to different use cases and environments.

Inside the microservice code, ASP.NET Core supports linking configuration settings to strongly typed .NET classes, such as the one presented in Figure A.12. Here, we are talking specifically about ASP.NET Core and .NET as it is part of the Primavera BSS stack, so they were the languages used to develop the prototypes. However, most languages, especially object-oriented ones, can replicate the same. The ASP.NET Core injects the configuration settings into the code using the various IOptions patterns. One of these standards, specifically IOptionsSnapshot, automatically reloads the application's configurations when the underlying data changes. Through Dependency Injection, we can inject the IOptionsSnapshot into the application's controllers to access the most recent configuration stored in the App Configuration Store, as seen in the controller example in Figure A.13.

To be able to use the App Configuration Service and to register the model classes created for the configurations in our microservices, we have to create a method as the one presented in Figure A.14. It uses the IServiceCollection.Configure<T> method to bind configuration data to the Settings model class.

Lastly, to add the App Configuration Service to the middleware pipeline, and allow the configuration settings registered to be updated, while the ASP.NET Core web app continues to receive requests, we have to use the method UseAzureAppConfiguration, as in Figure A.15. This method must be registered before the MVC, otherwise, it will fail. The middleware uses the update configuration specified in the AddAzureAppConfiguration method in Figure A.14 to trigger an update for each request received by the web app. For each request, an update operation is triggered, and the client library checks whether the cached value of the registered configuration settings has expired. For expired cached values, configuration values are updated with the value that is in the App Configuration Store, and the remaining values remain unchanged.

A final prototype using the Microsoft Azure App Configuration service was developed and presented to the team responsible to implement new technologies in the company. The proposal was reviewed and approved, and a guideline document with the steps to implement successfully this tool, and its best practices, was developed. It was already implemented successfully in Primavera BSS's MSA and is now being used in production by several microservices. However, the pieces of code presented here are only a small portion of the entire prototype that was developed to the company, which can not be entirely shown for privacy reasons

4.7 Webhooks

Based on the literature review and aforementioned good practices for webhooks, plus some from other technological areas, a webhooks architecture was created to solve the APIs limitations and to help to implement and manage webhooks in Primavera BSS's MSA. It serves as an intermediary that registers and manages all available events and subscriptions. It also manages the database and supports the task of sending notifications for all subscribers when events are triggered. In the event of a failure, it implements a system to send the notification again, with a variable difference of time between retries, until the subscriber successfully receives it or we reach a certain number of attempts.

There are two different approaches to implement this solution in an MSA. The first is to create an independent microservice that centrally controls all webhooks operations and is part of the architecture. Each microservice that wants to use webhooks will communicate with it to execute the desired operation. The second approach is through the implementation of these same functionalities, but in a library directly in each microservice. Thus, every operation that they want to execute is within its scope, and they do not need to communicate with other microservices to use this functionality.

There are some differences between both implementations. One of them concerns the databases. In

the first case, the database is shared between all the microservices that are interacting with the webhooks microservice, whereas in the second case, each microservice only works with its database, but we can still have a shared webhooks database in this case. Depending on the use case, it might be preferable to choose one or another. The first approach is better if we want to centralize the webhooks, it will make it easier to manage them, and it does not overload the microservices itself because it isolates an operation in an individual one. The second approach is preferable if we have light microservices that do not require plenty of processing power. Since this architecture creates a standard in webhooks, the second approach gives the same results as the first one but without the ability to centrally manage all of them.

Figure 30 presents the proposal for a general and versatile architecture that can be applied in both approaches, and that implements a standard to both produce and consume the webhooks. This proposal was developed taking into account the best practices and concepts for webhooks described in the literature review and uses technologies, design patterns, and best practices from other areas, namely REST and object-oriented programming. Moreover, by applying a standard between provider and consumer, we can solve some of the disadvantages presented previously.

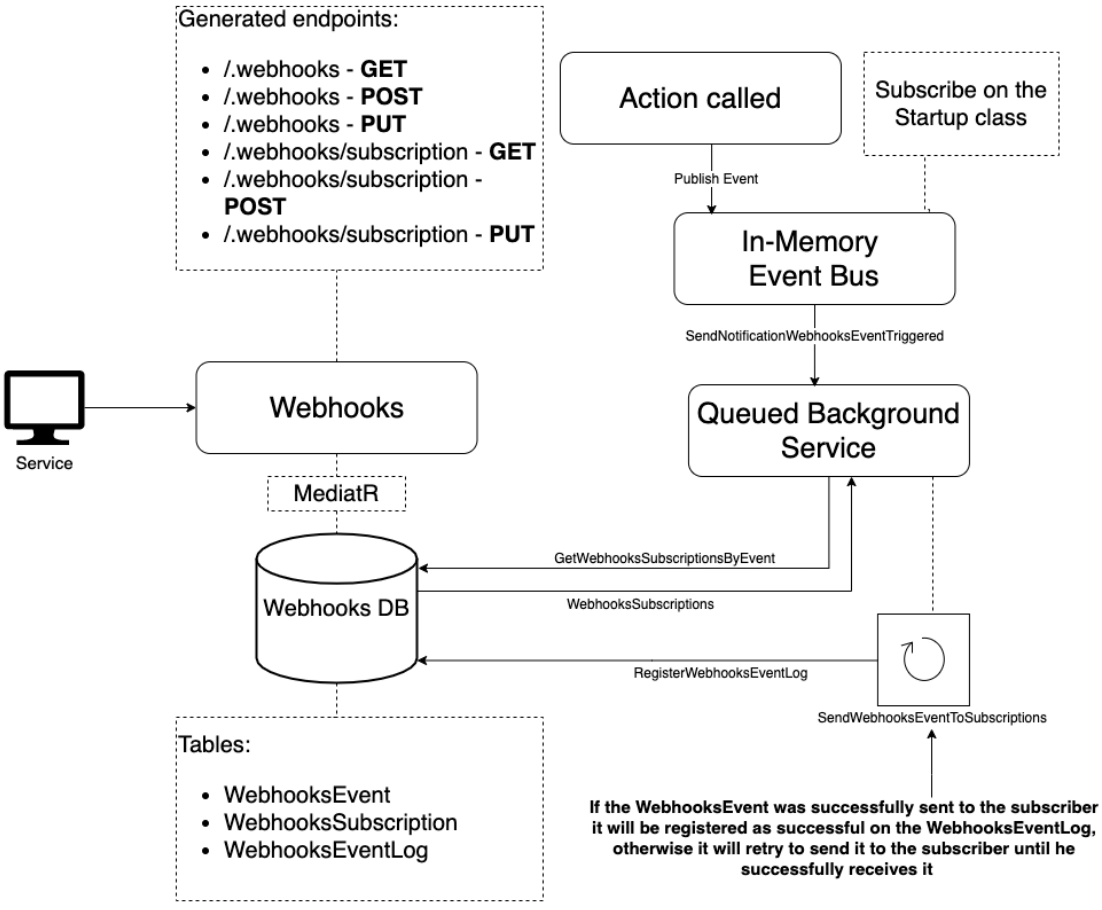


Figure 30: Webhooks proposed architecture

A service can interact with the webhooks, independently if they are implemented as a microservice or a library. Inside, the endpoints and database necessary to fully manage the events and subscriptions, and interact with them, are generated on the startup of the microservice. Figure A.16 exemplifies how it automatically generates the endpoints and maps them to work with the events and subscriptions middlewares.

To fully encapsulate the interactions with the database, and to not create dependencies on the implementation based on the database technology we are using, the mediator pattern and CQRS were implemented using the mediatR library. Figure A.17 exhibits how a CQRS Command class to update a webhook subscription is structured using the mediatR library. This class is meant to declare all the inputs needed to perform a specific action, in this case, the object WebhooksSubscription. Figure A.18 represents the Handle method that is responsible to implement the logic behind the database operation. More specifically, here, it inserts the subscriber in the database if he does not exist, otherwise, it updates him.

The event bus subscription is also done on the Startup and, in this case, we use an in-memory event bus. It is used to create a pipeline between an action that is called and the subscribers of that action. When an event is triggered by the event bus, it is published using a method like the one presented in Figure A.19. To create the event payload, it gets the event that was triggered and puts it in its body, adding more information to it, such as the product (microservice) which produced it and the date in which it was triggered. Moreover, the service namespace is added as a property to delineate which event bus subscription is going to manage this operation.

The event is then published to a queued background service that will get all the subscriptions for that event from the database and will send the notification with the payload to the subscribers, as seen in Figure A.20. All these notifications are individually saved to a webhooks log table in the database. A timed background service will pick up the ones that were not successfully sent and will retry to send them, with a variable difference of time between retries, to the subscribers until they receive them or they reach a certain number of attempts.

To explore the architecture more in-depth, Figure 31 shows the entity-relationship diagram for the database tables. One of the best practices mentioned previously was to use a type attribute for the events to help consumers manipulate them. However, we did not include it here because we aimed to a more standard solution that does not require extensive setup. Instead, a general class to manage all the notifications was developed.

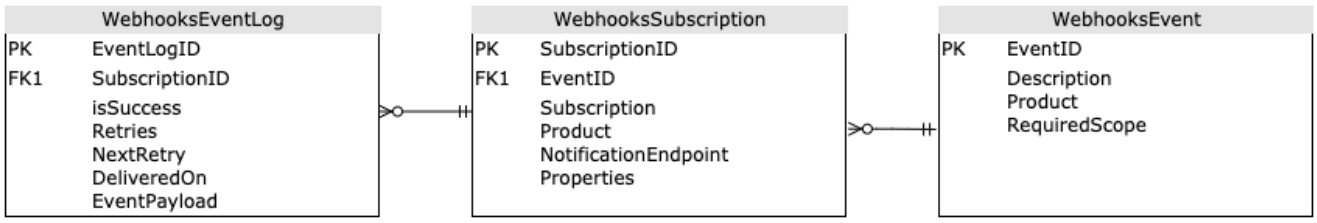


Figure 31: Entities-Relationship Diagram for the proposed architecture

Concerning security, an attribute named RequiredScope was added to the event. This way, we can manage the webhooks permissions, allowing only the users that are authorized to access that scope to subscribe to that event, and consequently, receive it. For that, we can use tools like the Identity Server to help with the authorization process, scopes management, secret generation, and API security in general.

Once the architecture is well implemented, depending on the kind of use case that the webhooks are going to be applied to, it could be a good option to create a UI to help handle and manage events and subscriptions.

4.7.1 Implementation at Primavera BSS

This architecture was already tested at Primavera BSS. They have recently migrated from an MA to an MSA and had no way of making their microservices act whenever new interesting actions occurred, either inside or outside the architecture. Moreover, since the company is not going to work with event-driven communication technology, the simpler and more efficient the solution the better, because those technologies require plenty of process power and it can overload the architecture. However, if, or when, they end up using event-driven communication technology, this solution is still applicable to separate concerns.

Notably, since the company develops business management software, there are plenty of good use cases for the webhooks because there are numerous interesting events that, once triggered, can create plenty of chained functions to act upon that. Plus, being able to act upon those events in real-time is an enormous market advantage and benefits both the company and its clients.

The webhooks were implemented internally in the company for their use and to test if they solved this necessity. To ease their adoption, the company incorporated them using the library in its SDK, which automatically generates the code for all the microservices created. This approach guarantees that, even if different developers create different microservices, the webhooks are always going to be implemented the same way.

Middleware is software that is assembled into an app pipeline to handle requests and responses

(Anderson & Smith, 2020). The order in which the middleware is built matters because the next middleware will receive the state from the previous one, and depends on it to work properly. Figure 32 represents the location of the webhooks in the ASP.NET Core middleware pipeline.

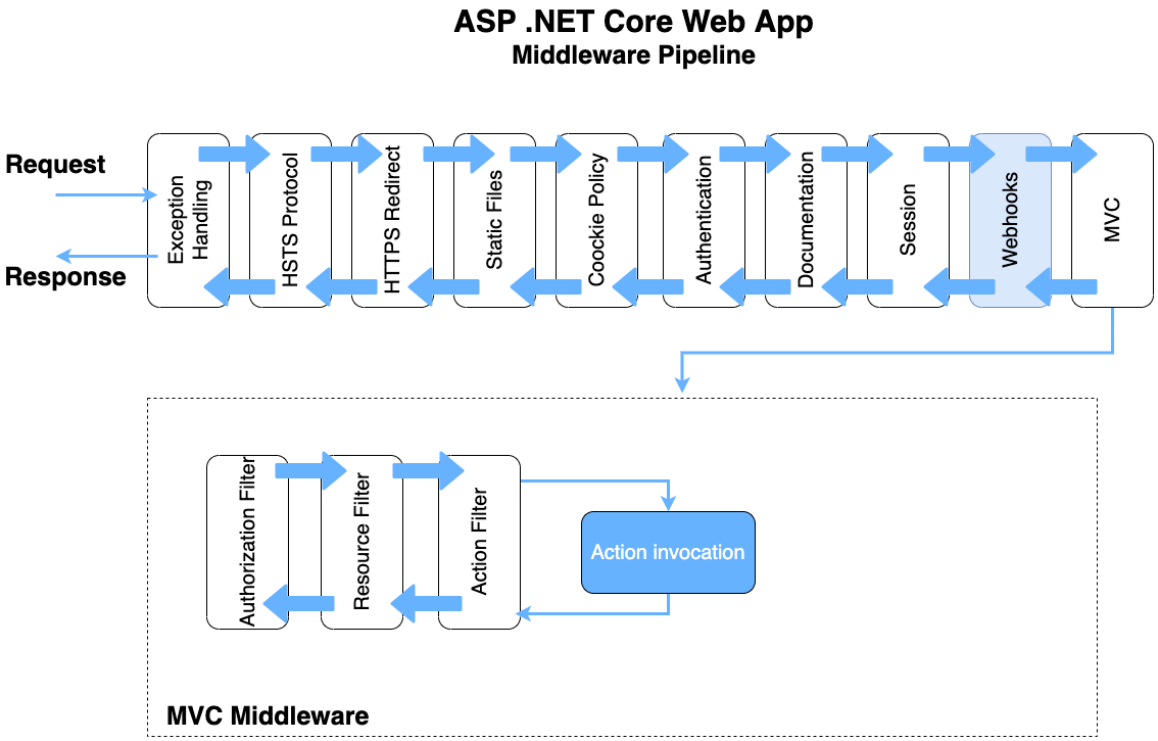


Figure 32: Standard ASP .NET Core Web App Middleware Pipeline (Adapted from (Anderson & Smith, 2020))

The webhooks' middleware was placed after the "Session" middleware because, in this case, it was a company requirement. Although, it would work if it was placed right after the "Authentication" middleware. Independently of the technology stack, if it uses the Model View Controller (MVC) pattern and has a middleware pipeline, it can be implemented the same way.

After the implementation in the development environment and first tests, the proposed solution fulfilled the expectations and was approved. A guideline document with the steps to implement it successfully and its best practices was developed. It is now being used in production by several microservices. However, the pieces of code presented here are only a small portion of the entire prototype that was developed to the company, which can not be entirely shown for privacy reasons.

4.7.2 Future scalability improvements

Over time it is normal for the solution to grow a lot, either if we opt for the first or second approach. In order not to overload the microservice and create bottlenecks, it is necessary to scale it. Figure 33

presents a solution to scale the microservice through data partition, using the product (microservice) ID. This way we manage to scale the webhooks microservice efficiently, and we take more advantage of this model because we can also play with the databases to make this flow even more flexible.

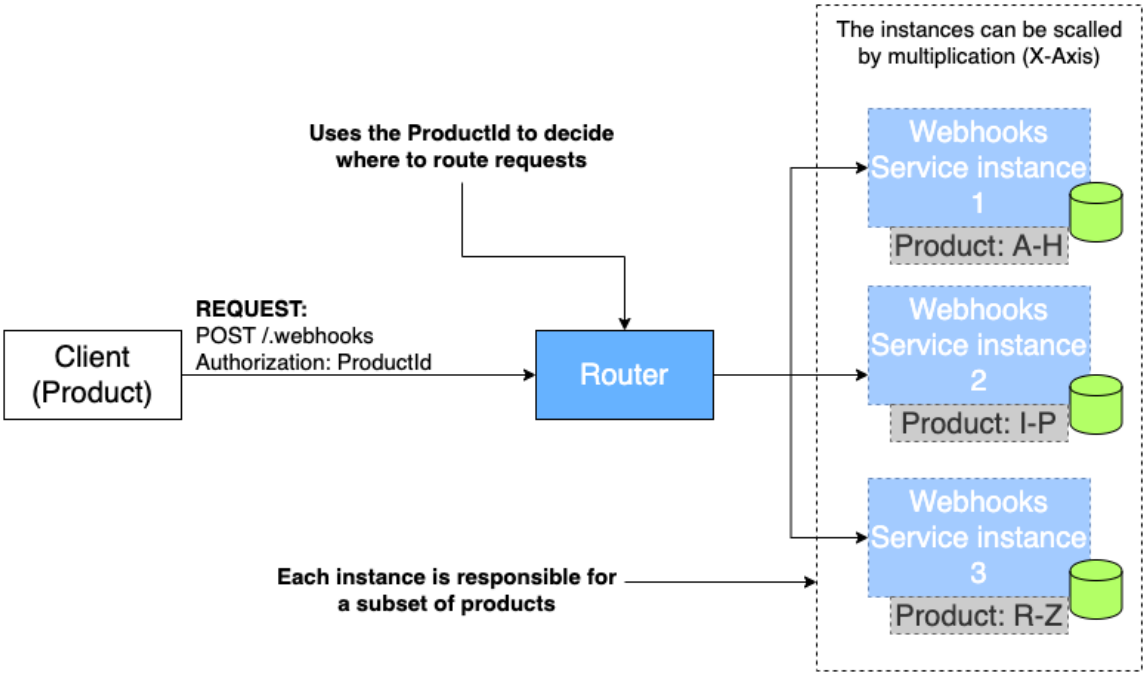


Figure 33: Webhooks architecture with data partitioning

5 Conclusion

The goal of this dissertation was to make an in-depth investigation of the MSA, regarding its concept, principles, patterns, advantages, and disadvantages. Other software architectures were also individually analyzed, and after comparing them with the MSA, we were able to delineate the best use cases for each. Given that this dissertation took place in the context of an internship at Primavera BSS, a project to improve their MSA was also carried out, in parallel with the theoretical research, to apply and validate the scientific knowledge acquired and, thus, contribute to the company's MSA. To be noted that the existing architecture in the company already included much of what has been studied, namely some principles, best practices, and patterns. Here, the objective was to improve the parts already existing and contemplate possible alternatives.

The company's objectives in its microservices project were discussed and aligned. The five topics proposed by the company, which were considered priority interest, were the documentation of microservices, gRPC, deployment platforms, configuration management, and webhooks. When developing the proposed topics, the integration with the Primavera Lithium Framework was taken into consideration, to be able to generate the microservice's code by interpreting model diagrams with the new functionalities. Apart from the main five topics, other smaller ones were also analyzed to complement them, like background workers and services, CQRS, and RESTful APIs. Even though they did not have their own literature review, an in-depth study about them was taken, not only to be able to deepen more the knowledge about a topic but also to help to produce an output with better quality.

In the literature review for the MSA, the main studied authors were Chris Richardson, Martin Fowler, and Sam Newman. When compared with the other studied software architectures, it was concluded that the MSA provides great advantages, especially when we talk about large applications, whose lifecycle is long term. Yet, these do not come without costs, also presenting major challenges that should not be taken lightly and imply a need to assess whether the adoption is justified or not. To note that, there are some use cases where the other options are preferable, for example, when we have a smaller solution it does not benefit from the MSA because it is more complex and harder to implement when compared to the MA. Concerning the other concepts, there were diversified authors, not existing no one that stood out. This variety of scientific information available is, in some cases, because they are more mature concepts and have more research about them.

Apart from the software architectures, the literature review also included the research of the five proposed topics by the company.

Concerning the practical component of this dissertation, apart from the software architectures, the

literature review also included the research of the five proposed topics. This step helped significantly making decisions backed off by a lot of knowledge to enhance Primavera BSS's MSA. Over the process of developing this chapter, to analyze the topics, some fundamental concepts were introduced, and the state of the art and best practices were exposed. This process allowed us to solve the company's challenges and deliver outputs with better quality. It is necessary to take into consideration that some of the subjects around MSA were searched and explored more in-depth than others because or they were more complex and required more time and research, or the company required them.

Initially, the current Primavera BSS MSA was described to get to know the internal structure of their microservices. Afterward, the five topics that the company proposed were individually studied, developed, and tested during the internship at Primavera BSS. From the proposed topics, almost all of them were implemented in the company, namely the documentation of microservices, deployment platforms, configuration management, and webhooks. These topics are now being used by the company in their microservices, and some are already being used in production.

Despite searching the topic, the gRPC was not implemented in the company. After its presentation, it was mutually agreed that this technology is still giving its first steps and is not mature enough to be adopted. Characteristics, namely the limited browsers support and lack of tools to streamline the process of testing the services, are deal-breakers to the company. Moreover, since this technology requires a restructuring of the communication method used by Primavera BSS's microservices to communicate with each other, the company would need to change its SDK to generate their microservices. This task involves plenty of hours of work and, considering that the company is still in an embryonic phase with microservices, they would not benefit enough from this migration to be worth adopting it. Nonetheless, in the future, this topic can be beneficial for the company, and it should be analyzed again due to its potential.

Having reached the end of this dissertation, there were several difficulties, mainly regarding the scope of the investigation. It was often necessary to leave the context under study, and research themes to complement it, which, in themselves, would make an entire dissertation. The MSA is an abstract and complex theme, which implies the knowledge of several topics regarding development, software architecture, deployment, and also organizational practices and structure. Thereby, there was some difficulty in following all these themes and technologies initially. Fortunately, after the study of all the literature and with the help of Primavera BSS's staff, it became easier to understand the full scope of the MSA.

Another difficulty felt was the shortage of literature review for some topics that needed to be mastered, specifically for the gRPC and webhooks. For those, it was necessary to resort to different blogs and articles to complete the information provided from an industry perspective, creating scientific research with a more

practical basis. Even though they were not directly scientific researches, they had plenty of information about the state of the art and best practices for those fields and were mostly written by the companies' research and development teams.

There was an unexpected incident during the internship that became, at some point, an obstacle. The global pandemic anticipated the termination of the internship by almost three months. Despite this, luckily, the topics that the company wanted to see investigated were already fully developed and presented to them and were ready to be implemented. Therefore, if this external obstacle did not exist, this dissertation would have the same content and output because all the efforts during the last months of the internship were going to be directed into the development of something outside the microservices topic.

Concerning Primavera BSS's MSA, the perception is that it is on the right track regarding the transition to this architecture, in a phased and intelligent manner. Although, it still has a long way to go and several challenges to solve, for example, the effective implementation of the remain principles and deployment platforms for the containerization, which may be based on the work carried out in this dissertation. Here, the topics developed are an asset to the company that leads to the improvement of their current MSA. However, they are just proof of concept and need to be tweaked to bring more robustness to Primavera BSS's microservices.

For future improvements, within the scope of the organization, some topics should be researched and implemented. It should be considered the implementation of the remaining patterns and principles, improvement of the continuous model, and the analysis of new technologies and architectures that are emerging, associated with the cloud and the MSA.

For future investigation in the scientific world, research into new technologies of cloud providers and new software architectures remain open. Also, it would be interesting to explore new approaches to make MSAs even simpler to develop and deploy, reducing some of its complexity to increase its appeal.

To complete the research and investigation carried out of both internship and dissertation, it would be interesting to deepen a comparison between MAs and MSAs. For example, for certain project sizes, measure all the performance differences, costs associated with development and deployment, time spent, and other comparison variables. Then, this could result in the creation of a framework to help companies decide if the migration between technologies is worth it.

It was concluded that, as the MSA is a recent architecture, there is still a great need for the availability and evaluation of case studies such as the one developed at Primavera BSS. The success or failure of the change process can be verified and lessons learned that explain the results. This way, it is possible to provide examples of this transition process, making it easier for companies that have difficulties or a lower

budget for research and development but want to do this transition.

Bibliography

Abbott, M. L., & Fisher, M. T. (2009). *The art of scalability: Scalable web architecture, processes, and organizations for the modern enterprise* (1st ed.). Addison-Wesley Professional.

Anderson, R., & Smith, S. (2020, May). *Asp.net core middleware.* Retrieved from https://docs.microsoft.com/en-us/aspnet/core/fundamentals/middleware/?view=aspnetcore-3.1

Belshe, M., Peon, R., Thomson, M., & Ed. (2015, May). *doc: Rfc 7540: Hypertext transfer protocol version 2 (http/2).* Retrieved from https://www.hjp.at/doc/rfc/rfc7540.html

Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, *1*(3), 81-84.

Biehl, M. (2017). *Webhooks–events for restful apis* (Vol. 4). API-University Press.

Burns, B., Beda, J., & Hightower, K. (2018). *Kubernetes*. Dpunkt.

Conway, M. E. (1968). (Unpublished doctoral dissertation). Datamation, Vol. 14, No. 4.

Crane, S., Dulay, N., Fosså, H., Kramer, J., Magee, J., Sloman, M., & Twidle, K. (1995). Configuration management for distributed software services. In A. S. Sethi, Y. Raynaud, & F. Faure-Vincent (Eds.), *Integrated network management iv: Proceedings of the fourth international symposium on integrated network management, 1995* (pp. 29–42). Boston, MA: Springer US. Retrieved from https://doi.org/10.1007/978-0-387-34890-2_3 doi: 10.1007/978-0-387-34890-2_3

Davison, R. M., Martinsons, M. G., & Kock, N. (2004). Principles of canonical action research. *Inf. Syst. J.*, *14*, 65-.

De, B. (2017). Api documentation. In *Api management* (pp. 59–80). Springer.

de la Torre, C. (2018). *Modernize existing .net applications with azure cloud and windows containers* (2nd ed.). Microsoft Press and Microsoft DevDiv.

Docker. (2017a). *Docker overview.* Retrieved from https://docs.docker.com/get-started/overview/

Docker. (2017b). *What is a container?* Retrieved from https://www.docker.com/resources/what-container

Docker. (2020a, May). *Dockerfile reference.* Retrieved from https://docs.docker.com/engine/reference/builder/

Docker. (2020b, May). *How services work.* Retrieved from https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/

Docker. (2020c, May). *Swarm mode key concepts.* Retrieved from https://docs.docker.com/

engine/swarm/key-concepts/

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., & Safina, L. (2017). Microservices: Yesterday, today, and tomorrow. In M. Mazzara & B. Meyer (Eds.), *Present and ulterior software engineering* (pp. 195–216). Cham: Springer International Publishing. Retrieved from https://doi.org/10.1007/978-3-319-67425-4_12 doi: 10.1007/978-3-319-67425-4_12

Evans, E. (2003). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley.

Foote, B., & Yoder, J. (1999, Jun). *Big ball of mud.* Retrieved from http://www.laputan.org/mud/

Fowler, M. (2006, May). *Continuous integration.* Martin Fowler. Retrieved from https://martinfowler.com/articles/continuousIntegration.html

Fowler, M. (2014, Jan). *Boundedcontext.* Retrieved from https://martinfowler.com/bliki/BoundedContext.html

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

Freeman, E. (2019). *Devops.* John Wiley Sons, Inc. Retrieved from https://aws.amazon.com/devops/what-is-devops/

Gabbrielli, M., Giallorenzo, S., Guidi, C., Mauro, J., & Montesi, F. (2016). Self-reconfiguring microservices. In *Essays dedicated to frank de boer on theory and practice of formal methods - volume 9660* (pp. 194–210). Berlin, Heidelberg: Springer-Verlag. Retrieved from https://doi.org/10.1007/978-3-319-30734-3_14 doi: 10.1007/978-3-319-30734-3_14

Gao, G. R., Hum, H. H. J., Theobald, K. B., Xin-Min Tian, & Maquelin, O. (1996). Polling watchdog: Combining polling and interrupts for efficient message handling. In *23rd annual international symposium on computer architecture (isca'96)* (p. 179-179).

Google. (2019, Sep). *Introduction to http/2 | web fundamentals | google developers.* Author. Retrieved from https://developers.google.com/web/fundamentals/performance/http2

gRPC. (2020a, May). *Core concepts, architecture and lifecycle.* Retrieved from https://grpc.io/docs/what-is-grpc/core-concepts/

gRPC. (2020b, May). *Introduction to grpc.* Retrieved from https://grpc.io/docs/what-is-grpc/introduction/

Iacobelli, G. (2016, Nov). *Webhooks do's and dont's: what we learned after integrating 100 apis.* Retrieved from https://restful.io/webhooks-dos-and-dont-s-what-we-learned-after-integrating-100-apis-d567405a3671

Iivari, J., & Venable, J. R. (2009). *Action research and design science research - seemingly similar but decisively dissimilar.* Retrieved from https://aisel.aisnet.org/ecis2009/73/

Kissami, I. (2017, Feb). *High performance computational fluid dynamics on clusters and clouds: the adapt experience.* Retrieved from https://www.researchgate.net/figure/Separation-of-responsibilities_fig9_323108484

Kruchten, P. (1995, Nov). *Architectural blueprints—the "4 1" view model of software architecture.* Philippe Kruchten. Retrieved from https://www.cs.ubc.ca/~gregor/teaching/papers/41view-architecture.pdf

Kubernetes. (2019, Nov). *Kubernetes components.* Retrieved from https://kubernetes.io/docs/concepts/overview/components/

Kubernetes. (2020, May). *Configure a pod to use a configmap.* Retrieved from https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/

Leach, B. (2017, Sep). *Should you build a webhooks api?* Retrieved from https://brandur.org/webhooks

MacKenzie, C. M., Laskey, K., McCabe, F., Brown, P. F., Metz, R., & Hamilton, B. A. (2006). Reference model for service oriented architecture 1.0. *OASIS standard*, *12*(S 18).

Martin, R. C. (2003). *Agile software development: Principles, patterns, and practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR.

McKay, J., & Marshall, P. J. (2001). The dual imperatives of action research. *IT People*, *14*, 46-59.

Merkel, D. (2014, March). Docker: Lightweight linux containers for consistent development and deployment. *Linux J.*, *2014*(239). Retrieved from http://dl.acm.org/citation.cfm?id=2600239.2600241

Microsoft. (2019a, Feb). *Azure app configuration best practices.* Retrieved from https://docs.microsoft.com/en-us/azure/azure-app-configuration/howto-best-practices

Microsoft. (2019b, Dec). *Ci/cd for containers - azure solution ideas.* Retrieved from https://docs.microsoft.com/en-us/azure/architecture/solution-ideas/articles/cicd-for-containers

Microsoft. (2019c, May). *Introduction to azure kubernetes service - azure kubernetes service.* Retrieved from https://docs.microsoft.com/en-us/azure/aks/intro-kubernetes

Microsoft. (2019d, Apr). *Overview - azure app service.* Retrieved from https://docs.microsoft.com/en-us/azure/app-service/overview

Microsoft. (2019e, Apr). *Serverless containers in azure - azure container instances.* Retrieved from https://docs.microsoft.com/en-us/azure/container-instances/container-instances-overview

Microsoft. (2020, Feb). *What is azure app configuration?* Retrieved from https://docs.microsoft.com/en-us/azure/azure-app-configuration/overview

Modak, A., Chaudhary, S. D., Paygude, P. S., & Ldate, S. R. (2018). Techniques to secure data on cloud: Docker swarm or kubernetes? In *2018 second international conference on inventive communication and computational technologies (icicct)* (p. 7-12).

Naik, N. (2016). Building a virtual system of systems using docker swarm in multiple clouds. In *2016 ieee international symposium on systems engineering (isse)* (p. 1-3).

Namiot, D., & Sneps-Sneppe, M. (2014, 09). On micro-services architecture. *Interenational Journal of Open Information Technologies*, *2*, 24-27.

Newman, S. (2015). *Building microservices: designing fine-grained systems* (1st ed.). O'Reilly Media, Inc.

Newton-King, J. (2019, May). *Compare grpc services with http apis.* Retrieved from https://docs.microsoft.com/en-us/aspnet/core/grpc/comparison?view=aspnetcore-3.1

OASIS. (2007, Apr). *Web services business process execution language.* Retrieved from http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007, December). A design science research methodology for information systems research. *J. Manage. Inf. Syst.*, *24*(3), 45–77. Retrieved from https://doi.org/10.2753/MIS0742-1222240302 doi: 10.2753/MIS0742-1222240302

PrimaveraBSS. (2019a, Nov). *Primaveradeveloper/lithium.* Retrieved from https://github.com/PrimaveraDeveloper/lithium/blob/master/vision/3-lithium-architecture.md

PrimaveraBSS. (2019b, Nov). *Primaveradeveloper/lithium.* Retrieved from https://github.com/PrimaveraDeveloper/lithium/blob/master/vision/4-microservice-architecture.md

Rapoport, R. N. (1970). Three dilemmas in action research: With special reference to the tavistock experience. *Human Relations*, *23*(6), 499-513. Retrieved from https://doi.org/10.1177/001872677002300601 doi: 10.1177/001872677002300601

Richardson, C. (2014a). *Microservices adoption anti-pattern: Red flag law.* Retrieved

from https://chrisrichardson.net/post/antipatterns/2019/06/07/antipattern-red-flag-law.html

Richardson, C. (2014b). *Pattern: Microservice architecture.* Retrieved from https://microservices.io/patterns/microservices.html

Richardson, C. (2014c). *Pattern: Monolithic architecture.* Retrieved from https://microservices.io/patterns/monolithic.html

Richardson, C. (2018). *Microservices patterns: with examples in java*. Manning Publications.

Richardson, C. (2019, Feb). *Blog.* Retrieved from http://crichardson.com/post/microservices/general/2019/02/27/microservice-canvas.html

Rowell, L. L., Polush, E. Y., Riel, M., & Bruewer, A. (2015). Action researchers' perspectives about the distinguishing characteristics of action research: a delphi and learning circles mixed-methods study. *Educational Action Research*, *23*(2), 243-270. Retrieved from https://doi.org/10.1080/09650792.2014.990987 doi: 10.1080/09650792.2014.990987

SendGrid. (2013, Jun). *Why every api needs webhooks.* Retrieved from https://sendgrid.com/blog/why-every-api-needs-webhooks/

Specification, O. (2020, Feb). Retrieved from http://spec.openapis.org/oas/v3.0.3

Stamat, D. (2019, Apr). *7 reasons webhooks are magic.* Retrieved from https://blog.iron.io/7-reasons-webhooks-are-magic/

Stonebraker, M. (2010, April). Sql databases v. nosql databases. *Commun. ACM*, *53*(4), 10–11. Retrieved from http://doi.acm.org/10.1145/1721654.1721659 doi: 10.1145/1721654.1721659

Susman, G. I., & Evered, R. D. (1978). An assessment of the scientific merits of action research. *Administrative Science Quarterly*, *23*(4), 582–603. Retrieved from http://www.jstor.org/stable/2392581

Szyperski, C. (1998). *Component software: Beyond object-oriented programming*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.

Tay, B. H., & Ananda, A. L. (1990, July). A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, *24*(3), 68–79. Retrieved from https://doi.org/10.1145/382244.382832 doi: 10.1145/382244.382832

TechNavio. (2017, Jun). *Global system infrastructure revenue market 2017-2021.* Retrieved from https://www.technavio.com/report/global-system-infrastructure-revenue-market

TOGAF. (2014). *Using togaf to define govern soas.* Retrieved from https://pubs.opengroup.org/architecture/togaf91-doc/arch/chap22.html

W3C. (2014, Feb). *Web services architecture.* Retrieved from https://www.w3.org/TR/ws-arch/

Webster, J., & Watson, R. T. (2002). Analyzing the past to prepare for the future: Writing a literature review. *MIS Quarterly*, *26*(2), xiii–xxiii. Retrieved from http://www.jstor.org/stable/4132319

Wiggins, A. (2017). *The twelve-factor app.* Retrieved from https://12factor.net/

Attachments

Name:	**Order Service**	
Description:	The Order Service provides an API for creating, revising, and cancelling orders.	
Capabilities		
Order Management		
Service API		
Commands	Queries	Events Published
Synchronous: • createOrder() • reviseOrder() • cancelOrder() Asynchronous: • N/A	getOrder()	Order event channel: • Order Created • Order Authorized • Order Revised • Order Cancelled • ...
Non-functional requirements	• 99.95% availability • 1000 orders/second	
Observability		
Key metrics		
• placed_orders • approved_orders • rejected_orders • ...		
Health check endpoint	/actuator/health	
Implementation		
Domain Model		
• Order aggregate		
Dependencies		
Invokes	Subscribes to	
Consumer Service: • validateOrder() Kitchen service: • createTicket() • confirmCreateTicket() • cancelCreateTicket() Accounting Service • authorize()	Restaurant Service • Restaurant Created event • Restaurant Menu Revised event Saga reply channels: • Create Order Saga • Revise Order Saga • Cancel Order Saga	

Figure A.1: Example of a microservice canvas Richardson (2019)

```
protected virtual void AddDocumentation(IServiceCollection services)
{
    // NSwag
    // Register the Swagger services
    HostConfiguration hostConfiguration = services.GetHostConfiguration();

    services.AddOpenApiDocument(config =>
    {
        config.FlattenInheritanceHierarchy = true;
        config.PostProcess = document =>
        {
            document.Info.Version = hostConfiguration.Information.Version;
            document.Info.Title = hostConfiguration.Information.HostTitle;
            document.Info.Description = hostConfiguration.Information.ProductName;
            document.Info.TermsOfService = new Uri("https://pt.primaverabss.com/pt/site/termos-de-utilizacao/").ToString();
            document.Info.Contact = new NSwag.OpenApiContact
            {
                Name = hostConfiguration.Information.Company,
                Email = string.Empty,
                Url = new Uri("https://pt.primaverabss.com/pt/").ToString()
            };
            document.Info.License = new NSwag.OpenApiLicense
            {
                Name = hostConfiguration.Information.Company + " " + hostConfiguration.Information.Copyright,
                Url = new Uri("https://pt.primaverabss.com/pt/").ToString()
            };
        };

        config.AddSecurity("bearer", Enumerable.Empty<string>(), new OpenApiSecurityScheme
        {
            Type = OpenApiSecuritySchemeType.OAuth2,
            Description = "Lithium Sample API",
            Flow = OpenApiOAuth2Flow.Implicit, //Application,
            Flows = new OpenApiOAuthFlows()
            {
                Implicit = new OpenApiOAuthFlow()
                {
                    Scopes = new Dictionary<string, string>
                    {
                        { "lithium-sample", "Access lithium-sample API." }
                    },
                    AuthorizationUrl = $"{hostConfiguration.IdentityServerBaseUri.Trim()}/connect/authorize",
                    TokenUrl = $"{hostConfiguration.IdentityServerBaseUri.Trim()}/connect/token",
                    RefreshUrl = $"{hostConfiguration.IdentityServerBaseUri.Trim()}/connect/token"
                },
            }
        });

        config.OperationProcessors.Add(
                new AspNetCoreOperationSecurityScopeProcessor("bearer"));
    });
}
```

Figure A.2: Method to add the documentation to the microservice

```
protected virtual void UseDocumentation(IApplicationBuilder app)
{
    // NSwag
    // Register the Swagger generator and the Swagger UI middlewares
    // Add Swagger 2.0 document serving middleware
    app.UseOpenApi();
    app.UseSwaggerUi3(settings =>
    {
        settings.DocExpansion = "list";
        settings.OAuth2Client = new OAuth2ClientSettings
        {
            ClientId = "lithium-sample-implicit",
            ClientSecret = ""
        };

        settings.OAuth2Client.AdditionalQueryStringParameters.Add("redirect_uri", "http://localhost:4200");
    });
}
```

Figure A.3: Method to use the documentation in the microservice's middleware pipeline

```
public void ConfigureServices(IServiceCollection services)
{
    services
        .AddMvc(config =>
        {
            config.InputFormatters.Add(new CustomTextInputFormatter());
            config.OutputFormatters.Add(new CustomTextOutputFormatter());
        })
        .SetCompatibilityVersion(CompatibilityVersion.Version_2_1);

    services.AddApiVersioning(options =>
    {
        options.AssumeDefaultVersionWhenUnspecified = true;
        options.ApiVersionReader = new UrlSegmentApiVersionReader();
    })
    .AddMvcCore()
    .AddVersionedApiExplorer(options =>
    {
        options.GroupNameFormat = "VVV";
        options.SubstituteApiVersionInUrl = true;
    });

    services
        .AddOpenApiDocument(document =>
        {
            document.DocumentName = "v1";
            document.ApiGroupNames = new[] { "1" };
        })
        .AddOpenApiDocument(document =>
        {
            document.DocumentName = "v2";
            document.ApiGroupNames = new[] { "2" };
        })
        .AddOpenApiDocument(document =>
        {
            document.DocumentName = "v3";
            document.ApiGroupNames = new[] { "3" };
        });
}
```

Figure A.4: Method to define different versions of the API to be displayed in the microservice's documentation

```
/// <summary>
/// Defines the class for the gRPC Alive service.
/// </summary>
public class AliveService : Alive.AliveBase
{
    private readonly ILogger<AliveService> logger;

    /// <summary>
    /// Initializes a new instance of the <see cref="AliveService" /> class.
    /// </summary>
    /// <param name="loggerService">LoggerService</param>
    public AliveService(ILogger<AliveService> loggerService)
    {
        this.logger = loggerService;
    }

    /// <summary>
    /// Implements the Sum function
    /// </summary>
    /// <param name="request">gRPC Sum Request</param>
    /// /// <param name="context">gRPC Context</param>
    /// <returns>
    /// The sum of 2 numbers
    /// </returns>
    ////[Authorize]
    [System.Diagnostics.CodeAnalysis.SuppressMessage("Design", "CA1062:Validate arguments of public methods", Justification = "<Pending>")]
    public override Task<SumReply> Sum(SumRequest request, ServerCallContext context)
    {
        var user = context.GetHttpContext().User;

        SumReply reply = new SumReply();
        reply.Message = request.Number1 + request.Number2;
        return Task.FromResult(reply);
    }
}
```

Figure A.5: Example of a gRPC service

```
// This method gets called by the runtime. Use this method to add services to the container.
public void ConfigureServices(IServiceCollection services)
{
    services.AddGrpc();

    services.AddSingleton<CinemaRepository>();

    services.AddAuthorization(options =>
    {
        options.AddPolicy(JwtBearerDefaults.AuthenticationScheme, policy =>
        {
            policy.AddAuthenticationSchemes(JwtBearerDefaults.AuthenticationScheme);
            policy.RequireClaim(ClaimTypes.Name);
        });
    });
    services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
        .AddJwtBearer(options =>
        {
            options.TokenValidationParameters =
                new TokenValidationParameters
                {
                    ValidateAudience = false,
                    ValidateIssuer = false,
                    ValidateActor = false,
                    ValidateLifetime = true,
                    IssuerSigningKey = SecurityKey
                };
        });
}
```

Figure A.6: ConfigureServices method with the configuration to support gRPC and authentication

```
private string GenerateJwtToken(string name)
{
    if (string.IsNullOrEmpty(name))
    {
        throw new InvalidOperationException("Name is not specified.");
    }

    var claims = new[] { new Claim(ClaimTypes.Name, name) };
    var credentials = new SigningCredentials(SecurityKey, SecurityAlgorithms.HmacSha256);
    var token = new JwtSecurityToken("ExampleServer", "ExampleClients", claims, expires: DateTime.Now.AddSeconds(60), signingCredentials: credentials);
    return JwtTokenHandler.WriteToken(token);
}

private readonly JwtSecurityTokenHandler JwtTokenHandler = new JwtSecurityTokenHandler();
private readonly SymmetricSecurityKey SecurityKey = new SymmetricSecurityKey(Guid.NewGuid().ToByteArray());
```

Figure A.7: Method to configure and generate a JWT Token

```
// This method gets called by the runtime. Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        // Communication with gRPC endpoints must be made through a gRPC client.
        endpoints.MapGrpcService<AliveService>();

        endpoints.MapGet("/generateJwtToken", context =>
        {
            return context.Response.WriteAsync(GenerateJwtToken(context.Request.Query["name"]));
        });
    });
}
```

Figure A.8: Configuration of the microservice's middleware pipeline to use and map the gRPC and JWT Token

```
static async Task Main(string[] args)
{
    // The port number(20001) must match the port of the gRPC server.
    var channel = GrpcChannel.ForAddress("https://localhost:20001");
    var client =  new Alive.AliveClient(channel);

    Console.WriteLine();
    Console.WriteLine("Unary Call Test");
    Console.WriteLine("Write the first number");
    var number1 = Console.ReadLine();

    Console.WriteLine("Write the second number");
    var number2 = Console.ReadLine();
    Console.WriteLine();

    var aliveRequest = new SumRequest { Number1 = Convert.ToInt32(number1), Number2 = Convert.ToInt32(number2) };

    var alive = await client.SumAsync(aliveRequest);
    Console.WriteLine($"{number1} + {number2} = {alive.Message}");
}
```

Figure A.9: Example of a simple gRPC client application

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration(c =>
        {
            c.AddJsonFile(ConfigMapFileProvider.FromRelativePath("config"),
                "appsettings.json",
                optional: true,
                reloadOnChange: true);
        })
        .UseStartup<Startup>();
```

Figure A.10: CreateWebHostBuilder method with capability to detect changes based on the file's content

```
public static IWebHostBuilder CreateWebHostBuilder(string[] args) =>
    WebHost.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((hostingContext, config) =>
        {
            var settings = config.Build();
                config.AddAzureAppConfiguration(options =>
                {
                    options.Connect(settings["ConnectionStrings:AppConfig"])
                           .ConfigureRefresh(refresh =>
                           {
                               refresh.Register("TestApp:Settings:testeExternalConfig");
                               refresh.Register("TestApp:Settings", refreshAll: true);
                               //refresh.SetCacheExpiration(TimeSpan.FromMinutes(5));
                           })
                           .Select(KeyFilter.Any, LabelFilter.Null)
                           .Select(KeyFilter.Any, "prod");
                });
            });
        })
        .UseStartup<Startup>();
```

Figure A.11: CreateWebHostBuilder method to connect and configure the App Configuration

```
namespace Primavera.Lithium.Nitrogen.WebApi.CustomCode.Models
{
    public class Settings
    {
        /// <summary>
        /// Configuration testeExternalConfig
        /// </summary>
        public string testeExternalConfig { get; set; }
    }
}
```

Figure A.12: Settings model class

```
public class HomeController : Controller
{
    private readonly Settings _settings;
    public HomeController(IOptionsSnapshot<Settings> settings)
    {
        _settings = settings.Value;
        Console.WriteLine(_settings.testeExternalConfig);
    }
}
```

Figure A.13: Example of the use of App Configuration in a controller

```
public virtual void AddAppConfiguration(IServiceCollection services)
{
    if (Configuration.GetValue<bool>("UseAzureAppConfiguration"))
            services.Configure<Settings>(Configuration.GetSection("TestApp:Settings"));
}
```

Figure A.14: Method to add the App Configuration to the microservice and get configurations

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    app.UseAzureAppConfiguration();

    services.Configure<CookiePolicyOptions>(options =>
    {
        options.CheckConsentNeeded = context => true;
        options.MinimumSameSitePolicy = SameSiteMode.None;
    });

    app.UseMvc();
}
```

Figure A.15: Configuration of the microservice's middleware pipeline to use the App Configuration and configure cached values

```
namespace Primavera.Lithium.Webhooks
{
    public static class WebhooksBuilderExtensions
    {
        #region Public Constants

        /// <summary>
        /// Defines the route for the following action: Create Invoice.
        /// </summary>
        public const string GetWebhooks = "/.webhooks";

        /// <summary>
        /// Defines the route for the following action: Get Invoices.
        /// </summary>
        public const string GetSubscriptions = "/.webhooks/subscriptions";

        #endregion

        public static IApplicationBuilder UseWebhooks(this IApplicationBuilder builder)
        {
            builder.Map(GetSubscriptions, b => b.UseMiddleware<WebhooksSubscriptionsMiddleware>());
            builder.Map(GetWebhooks, b => b.UseMiddleware<WebhooksMiddleware>());

            return builder;
        }
    }
}
```

Figure A.16: WebhooksBuilderExtensions class

```
namespace Primavera.Lithium.Webhooks.Application
{
    public class UpdateWebhooksSubscriptionCommand : IRequest<BaseResponse>
    {
        public WebhooksSubscription WebhooksSubscription { get; set; }
    }
}
```

Figure A.17: Example of a Command class to Update webhooks subscriptions

```
/// <summary>
/// Defines base class for the controller that provides monitoring routes.
/// </summary>
/// <param name="request">qewe</param>
/// <param name="cancellationToken">ad</param>
/// <returns><see cref="Task"/> representing the asynchronous operation.</returns>
public async Task<BaseResponse> Handle(UpdateWebhooksSubscriptionCommand request, CancellationToken cancellationToken)
{
    SmartGuard.NotNull(() => request, request);

    WebhooksSubscription webhooksSubscription = request.WebhooksSubscription;

    await this.table.InsertOrReplaceEntityAsync<WebhooksSubscription>(webhooksSubscription).ConfigureAwait(false);

    return new BaseResponse { IsSuccess = true, Message = "WebhooksSubscription updated with success!" };
}
```

Figure A.18: Example of the Handle method of a Command class to Update webhooks subscriptions

```
public async Task Publish(EventTriggeredDto eventTriggeredDto, Dictionary<string, string> additionalProperties = null)
{
    IEventBusEvent<EventTriggeredDto> @event = new AzureEventBusEvent<EventTriggeredDto>()
    {
        Body = eventTriggeredDto
    };

    @event.Body.Product = Product;
    @event.Body.TriggerDate = DateTime.Now;

    @event.Properties.Add("Service", ServiceNamespace);

    this.EventBus.Publish(@event);
}
```

Figure A.19: EventBus Publish method

```
public async Task Executar(EventTriggeredDto eventTriggered)
{
    try
    {
        IEnumerable<WebhooksSubscription> eventSubcriptions = await this.GetWebhooksSubscriptionsByEvent(eventTriggered.Product, eventTriggered.Event).ConfigureAwait(false);

        await SendEventToSubscribers(eventSubcriptions, eventTriggered.Product, eventTriggered.Event, eventTriggered);
    }
    catch (Exception e)
    {
        Console.WriteLine($"Message handler has encountered an exception: '{e.Message}'");
    }
}
```

Figure A.20: Example of a method in a background service that sends events to its subscribers

5.1 Abstract from the developed article

During the development of this dissertation, an article was developed in parallel and submitted for the Conference of the Portuguese Information Systems Association. Its abstract is as follows:

"Microservices are a software architecture inspired by service-oriented computing that has recently started gaining popularity and has been showing remarkable results in the development of large-scale systems. Contrarily, webhooks have been around for some time but they do not have a big adoption for the amount of potential they have, mainly because solutions use Application Programming Interfaces (API) to do their job, which is inefficient and does not give the same result. APIs have natural limitations to receive real-time events that webhooks solve and when embedded together, allow providers to offer a better and richer service to the customers. Microservices can also benefit from the low latency, lightweight, event-driven communication that webhooks provide. This allows microservices, that know little or nothing about each other, to communicate and act when an event happens and to act upon that without having to use heavy third-party technologies and dependencies.

Keywords: Software Architecture; Software Development; Software Engineering; Distributed Systems;"