INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA INFORMÁTICA




Microsserviços em Jolie: uma experiência

RUI MIGUEL MADUREIRA ASSIS
Outubro de 2022

POLITÉCNICO
DO PORTO

Instituto Superior de Engenharia do Porto

Jolie Microservices

An Experiment

Rui Miguel Madureira Assis

Dissertation to obtain the Master's Degree in Informatics Engineering, Area of Expertise in Computer Systems

Supervisor: Isabel de Fátima Silva Azevedo

Porto, October 2022

Dedication

This work is dedicated to my beloved parents, my true heroes.

Resumo

Os microsserviços estão cada vez mais presentes no mundo das tecnologias de informação, por providenciarem uma nova forma construir sistemas mais escaláveis, ágeis e flexíveis. Apesar disto, estes trazem consigo o problema da complexidade de comunicação entre microsserviços, fazendo com que o sistema seja difícil de manter e de se perceber. Linguagens de programação específicas a microsserviços como Jolie entram em cena para tentar resolver este problema e simplificar a construção de sistemas com arquiteturas de microsserviços.

Este trabalho fornece uma visão ampla do estado da arte da linguagem de programação Jolie onde é primeiramente detalhado o porquê de surgirem linguagens específicas a microsserviços e como a linguagem Jolie está construída de maneira a coincidir com as arquiteturas de microsserviços através de recursos nativos.

Para demonstrar todas as vantagens de usar esta linguagem em comparação com as abordagens mais *mainstream* é pensado um experimento de desenvolvimento de um sistema de microsserviços no âmbito de uma aplicação de e-commerce. Este sistema é construído de forma igual usando duas bases tecnológicas – Jolie e Spring Boot. O Spring Boot é considerado a tecnologia mais usada para desenvolver sistemas de microsserviços sendo o candidato ideal para comparação. É pensada toda a análise e design deste experimento.

Em seguida, a implementação da solução é detalhada a partir das configurações do sistema, escolhas arquitetónicas e como elas são implementadas. Componentes como API gateway, mediadores de mensagens, bases de dados, orquestração de microsserviços, e conteinerização para cada microsserviço e outros componentes do sistema.

Pol último as soluções são comparadas e analisadas com base na abordagem Goals, Questions, Metrics (GQM). São analisadas relativamente a atributos de qualidade como manutenção, escalabilidade, desempenho e testabilidade. Após esta análise pode-se concluir que a solução construída com Jolie apresenta diferenças na manutenção sendo significativamente superior à solução baseada em Spring Boot e apresenta diferenças em termos de performance sendo ligeiramente inferior à solução construída com Spring Boot. O trabalho termina com a indicação das conquistas, dificuldades, ameaças à validade, possíveis trabalhos futuros e observações finais.

Palavras-chave: microsserviços, Jolie, linguagens específicas de microsserviços, Spring Boot

Abstract

Microservices are increasingly present in the world of information technologies, as they provide a new way to build more scalable, agile, and flexible systems. Despite this, they bring with them the problem of communication complexity between microservices, making the system difficult to maintain and understand. Microservices-specific programming languages like Jolie come into play to try to solve this problem and simplify the construction of systems with microservices architectures.

This work provides a broad view of the State of Art of the Jolie programming language, where it is first detailed why microservices-specific languages emerge and how the Jolie language is built to match microservices architectures through native resources.

To demonstrate all the advantages of using this language compared to more mainstream approaches, an experiment is designed to develop a microservices system within an e-commerce application. This system is built equally using two technological foundations – Jolie and Spring Boot. Spring Boot is considered the most used technology to develop microservices systems and is an ideal candidate for comparison. The entire analysis and design of this experiment are thought through.

Then the implementation of the solution is detailed from system configurations, architectural choices, and how they are implemented. Components such as API gateway, message brokers, databases, microservices orchestration, and containerization for each microservice and other components of the system.

Finally, the solutions are compared and analyzed based on the Goals, Questions, Metrics (GQM) approach. They are analyzed for quality attributes such as maintainability, scalability, performance, and testability. After this analysis, it can be concluded that the solution built with Jolie presents differences in maintenance being significant superior to the solution based on Spring Boot, and it presents differences in terms of performance being slightly inferior to the solution built with Spring Boot. The work ends with an indication of the achievements, difficulties, threats to validity, possible future work, and final observations.

Keywords: microservices, Jolie, microservices specific languages, Spring Boot

Acknowledgements

I would like to thank my advisor Isabel Azevedo for all the revisions, help, and tips to increase the quality of my work. I also want to thank Professor Susana Nicola for the help given in the early stages of the thesis when big decisions needed to be made. Thank you to all the teachers at ISEP for always providing the best learning materials and directions to all the student's best interests.

Finally, I want to thank my entire family, my girlfriend, friends, all the teachers, colleagues, and everybody that passed through my personal, academic, and professional life as I believe every one of them was essential for completing this challenge.

Table of Contents

List of Figures

List of Tables

Acronymous and Symbols

List of acronymous:

AHP	Analytic Hierarchy Process
API	Application Programming Interface
APP	Application
DB	Database
ESB	Enterprise Service Bus
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IP	Internet Protocol
JSON	JavaScript Object Notation
L2CAP	Logical Link Control Adaptation Protocol
LOC	Lines of Code
MS	Microservice
MSA	Microservice Architecture
MDE	Model-Driven Engineering
MDD	Model-Driven Development
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
SODEP	Simple Orientation Data Exchange Protocol
RCP	Remote Procedure Call
RMI	Remote Method Invocation
QA	Quality Attribute
TCP	Transmission Control Protocol
XML	Extensible Markup Language
UI	User Interface

1 Introduction

This first chapter contextualizes and presents the problem to be solved, addressing the main objectives of the work to be developed and the approach to reach a solution. Last, the structure of this document is presented.

1.1 Problem

Microservices have been adopted to take advantage of their promised benefits like agility and scalability [1]–[3]. The tendency is for more businesses to move to microservices architecture (MSA) in future years due to the emerging future of the Cloud, the Internet of Things, and mobile computing [4], [5].

Microservices architectures bring complexity as other distributed systems, due to communication between processes, testing, debugging, and deploying the system [6]. MSA systems also lead to some problems regarding high memory use, the time required to fragment different microservices, the complexity of managing many services, and developers that need to solve problems such as network latency or load balancing. Also, if a big corporation already has a monolith system the cost and complexity to move to a microservices architecture is even higher [7] since their systems run on different types of technologies, platforms, and programming languages that can be an incredibly hard task to change everything to MSA.

The emergence of programming languages such as Jolie [8] that include primitives to deal with the programming of communications, monitoring, fault management, and architectural patterns (like API gateway) [9], issues that are relevant in applications with multiples microservices, leads to question how applications that use them compare with other more conventional approaches in

terms of some quality attributes. However, there is a lack of work regarding these technological choices.

1.2 Objectives

In the field of microservices development, complex, error-prone and communication situations abound. The specific languages of microservices, such as Jolie, due to abstraction and its ease of development can help to obtain systems with better handling and communication management and less complexity.

The main objective is to evaluate a microservice solution in terms of some attributes seen as important in MSA systems and understand how a language-based approach can support software architects in specific scenarios.

The presented work is studied and presented framed in the context of distributed systems, mainly in the microservices architecture domain. The motivation behind the problem is that this architecture is one of the current most adopted technology approaches in the world of software development, being more and more popular each day. The proposed solution portrayed in the thesis aims to highlight the benefits and drawbacks of the service-oriented programming language Jolie.

1.3 Methodology

To achieve the objectives defined the adopted methodology englobes all phases from the definition of the problem to the developed solution and further evaluation. For this, the Design Science Research Methodology (DSRM) was used [10].

With the following six activities:

1. Problem identification and motivation
    a. Define the specific research problem
    b. Collect information about the problem, namely what different people have already addressed informally in relation with the problem
    c. Look at microservices field knowledge regarding their actual design patterns, techniques, frameworks, languages, and challenges looking at the advantages they bring but also the disadvantages
2. Define the objectives for a solution

   a. Propose the objectives of a solution from the problem definition and knowledge of what is possible and feasible
   b. Collect information about the state of problems, possible solutions, and their efficacy
3. Design and development
   a. Design the artifact: an application to be developed using Jolie and in a traditional approach
   b. Determine the experiment desired functionality and its architecture
   c. Create the artifact (based on microservices architecture)
4. Demonstration
   a. Demonstrate the construction of the solution in two different ways
5. Evaluation
   a. Measure how well the experiment supports a solution to the problem. This activity involves comparing the objectives of a solution to actual observed results from the use of the artifact in the demonstration
   b. Present quantitative performance measures relative to some Quality Attributes namely scalability, maintainability, and performance, as these are the main quality attributes seen as important and problematic in the microservices architecture that a language-based approach can handle
   c. Compare the two different ways on which the solution is built
6. Communication
   a. Communicate the problem and its importance, the artifact, its utility and novelty, the accuracy of its design, and its effectiveness to researchers. Moreover, the comparison results and what was learned with the experiment.

1.4 Structure

This chapter concludes with the document structure of the thesis that aim at describing all chapters that compose this document, divided, and described as:

1. **Chapter 1 – Introduction:** this is the current chapter, and it provides an introduction for the thesis describing the problem, the objectives and how will the problem and objectives be solved.

2. **Chapter 2 – Background:** provides general knowledge around microservices, namely what are microservices, how they communicate with each other and some of the challenges that

these types of systems bring with them. It also mentions and describes the most important quality attributes for such systems.

3. **Chapter 3 – State of Art:** presents the state of the art on microservices specific languages – programming languages made just for microservice programming, and it details Jolie programming language state of the art.

4. **Chapter 4 – Analysis and design:** presents the analyze and design choices for a solution, introducing the requirements, use cases, architecture, and technology choices.

5. **Chapter 5 – Implementation:** provides the detailing of the solution implementation, namely how API gateway and containerization is done, what distinctions exist between implementing the solution with Jolie and a traditional approach, how the microservices communicate with each other, and lastly the verification and testing of the solution.

6. **Chapter 6 – Evaluation:** this chapter does experiments to measure the quality attributes define on chapter 2 in the solution developed with Jolie and with the traditional approach, finishes by comparing both.

7. **Chapter 7 – Conclusions:** describes the conclusions of this thesis, namely the achieved goals, found difficulties, the threads to validity of the findings, future work suggestions and some final remarks.

8. **Annex A – Microservices development:** this chapter is a supplement to understand the current most used frameworks, languages and libraries used for developing microservices, it also gives insight into the lifecycle phases of such development and the design patterns commonly use to develop the systems.

9. **Annex B – Value analysis:** this other supplemental chapter contains the value analysis, since it decreases the ease of read of the document it is opted to put it as an annex. The chapter emphases on the value that is projected with Jolie programming language, it is done and opportunity identification and analysis to obtain a better overview in which market Jolie fits. Then the Analytic Hierarchy Process (AHP) method is compared Jolie with other microservice specific programming languages, and Jolie is evaluated as its business value.

2 Background

This chapter provides background about microservices and the microservices architecture, first, it is provided in a summarized way the why and how microservices emerged, how they work in terms of communication, and the challenges that came along. Secondly, the technologies and frameworks that have been developed, the development lifecycle as well as the design patterns used in microservices programming.

2.1 Microservices

A microservice is a program that offers functionalities to other components of the system (other microservices, databases, user interfaces) via message passing [11] and this concept applies in distributed computing. Microservices have the goal of eliminating unnecessary levels of complexity to focus on simpler services and do just one single thing [11].

Even though there is not a concrete definition of what a microservice is, the literature defines it in relatively similar ways. In a revision of the state-of-art of microservices (Dragoni et al. 2017), a microservice is defined to be “a cohesive, independent process interacting via messages” [11], this is the definition to consider in the entirety of the thesis. A collection of these processes constitutes a collection of microservices creating the structure of a system with a microservice architecture.

Software development evolved through the years of the digital world and systems became more and more distributed. These new connected and distributed systems naturally created the necessity for new architectures and approaches in software engineering. This originated the appearance of the microservices architectures.

The microservices architecture has a lot of definitions reported in the literature as reviewed in [5] where the most recurring definition by (Fowler et al.) is that a microservice architecture is "a particular way of designing software applications as suites of independently deployable services" [12] since this is the most recurring definition for the entirety of the thesis is the one considered. Another popular definition is the one of (Dragoni et al. 2017) which defines that "a microservice architecture is a distributed application where all its modules are microservices" [11], this tends to show that all definitions give the impression that instead of having a single application the system is composed of multiple applications and multiple databases that communicate with each other like a multi-component that are isolated from each other but communication between them. In this thesis, the definition given by (Fowler et al.) will be used when referring to a microservice.

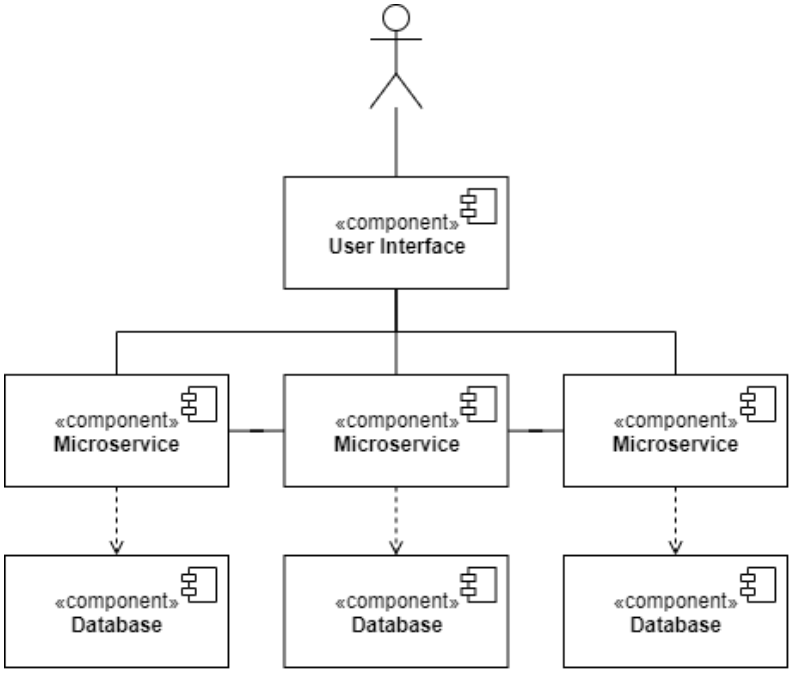

Figure 1 – Microservice system architecture (MSA) UML diagram

Microservices architecture has some changes concerning service-oriented architecture. In Figure 1 we can see that the services are still present and are now called microservices, each one has its database, and the UI communicates directly with the needed microservices for the feature that is performing.

MSA changes the way systems are architected because as (Guidi et. al, 2017) says, "microservices support a different view, enabling the organization of systems as collections of small independent components. Independent refers to the capability of executing each microservice on its own machine" [13]. We now think of systems of interconnected machines communicating by exchanging data (in formats like e.g., JSON or XML, supported in HTTP and other protocols or simpler database

records) running simple services that have a single job and that can be anywhere, causing a complete distribution of concerns making the programs very smaller in size compared to SOA [13], [14].

Another interesting aspect of microservices is the use of containerization, a technology that allows running a program (e.g.: a microservice) inside a “block” that contains the source code together with all the needed libraries and resources into a final package. The container is then deployed in any machine generating a running instance as an isolated process. This process can be replicated anywhere [15] corroborating the statement of (Guidi et. al, 2017) that where they say that the independency of a microservice “refers to the capability of executing each microservice on its own machine” [13].

Important features and reasons why MSA triumphed in distributed systems are [14]:

- easy to find functionalities [14]
    - related requirements are merged into a single business capability
- the small size of each component’s program [14]
    - again, always providing a single business capability
    - developers need to rethink the service if it starts to get too large
- the independence of each component’s program [14]
    - service communicates only via published interfaces
    - essential for easy bug fixing without impacting the entire system
    - since they are deployed separately, it allows for the independent management of components’ lifecycles as new versions of components can be gradually introduced in a system, by deploying them side to side with previous versions [9]
- components can be more specialised, since they can be written in different technologies – this happens due to message passing support of technologies used [9]
- scalability is different in MSA relatively to SOA since MSA does not imply duplication of all the components and developers can conveniently deploy/dispose of instances of services concerning their load [9]

2.1.1 Orchestration and choreography

To implement large application processes, microservices must communicate and collaborate between them for the system to work flawlessly. Microservices collaborate in a business process by

implementing and modeling distributed system workflows using sagas. Sagas help to model microservice workflows in a more adequate approach [16].

The most known and discussed patterns both in the industry and the literature are (1) orchestration and (2) choreography [17], [18].

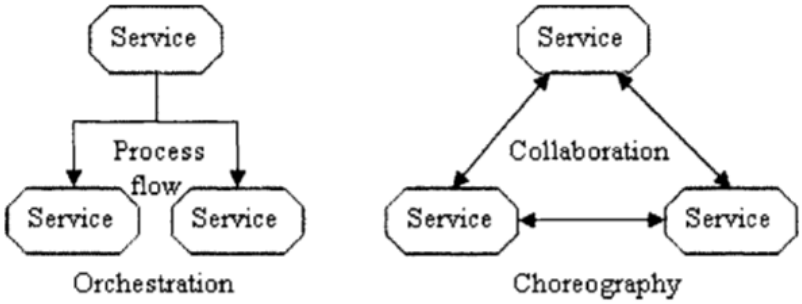

Figure 2 – Orchestration versus choreography [18]

1. Choreography
    a. communication happens asynchronously via message passing [17]
    b. deals with service collaboration and interaction tracking the messages from the multiple services involved in the message exchange [18]
    c. normally developers interpret choreography as a sequence of execution of services [18]
    d. do not depend on centralized coordinators [16]
    e. support a more loosely coupled model [16]
2. Orchestration
    a. communication happens with a central controller [17] [16]
    b. a central controller that is responsible to manage the entire flow of processes [17], [16] and their behavior [18] expressed in Figure 2
    c. all steps are business services [18]
    d. it is the term for workflow in the formal specification of a business process [18]
    e. describes a process flow and includes execution order of service interactions [18]
    f. the control flow must always be controlled by one party (centralized control) [18]

(Montesi et. al, 2016) refer that “development methodologies for service communications typically employ choreographies for the description of service protocols. Choreographies do not require central control, a critical feature for the scalability of MSA systems” [9]. On the other hand, some features like circuit breakers, service discovery, load balancing, and API gateways can’t be

implemented [9]. Also, orchestration provides better error handling alongside less event handling leading to simpler code.

The choice between each pattern is business-dependent. It depends on what the business requires and the system in question. The business logic is key, when there is a single point of logic, a centralized one, orchestration is used, when the problem has a decentralized logic choreography is used. In short for multiple business domains, a developer can consider more choreography while for a single business domain he/she may be more on the side of orchestration.

2.1.2 Challenges

Microservices even though solved a lot of problems in distributed systems scope some challenges came across also, some of those challenges are described in [19] being performance, orchestration, and, cyber-security all related to the current approach of containerization.

The most targeted and worked problems in the literature are complex [4], [5] because of the hard task that is to put every service communicating with each other, and the high number of distributed services to operate. Microservices solve a lot of problems but to solve them they bring with them the challenge of complexity which requires more attention to the way systems are designed and documented before being implemented. By making sure the design is correct the system can avoid future problems with the complexity issue [4].

2.2 Quality Attributes

The microservices architecture has some quality attributes (QA) important to this software development approach. In the literature [20]–[24] the most referred are scalability, performance, and maintainability. This makes sense regarding that companies move to the microservices architecture to benefit from the system scalability aspect as referred to in section 1.1. Also, performance is always a critical factor for systems to satisfy the user and the requirements in an effective way. Maintainable system source code allows for quick error fixing and can be achieved for example by decreasing the level of the system's technology heterogeneity or reducing the lines of code in the system.

The complexity of software affects multiple tasks of maintenance activities such as testability, reusability, understandability, and modifiability for example [25]. Software complexity can be defined as — "the degree to which a system or component has a design or implementation that is

difficult to understand and verify" [26]. Complexity is present in MSA not in a single microservice itself but in the inter-service communication [27]. Thus, it will also be analysed.

2.2.1 Scalability

Scalability is a measure of a system's ability to add resources to handle a varying amount of requests and can be divided into horizontal scalability (sometimes called elasticity) and vertical scalability [20]. The resources can be added to logical units like servers of a cluster or adding resources to physical units like more CPU or memory to a computer.

Horizontal scalability is the most important property of MSA since it decomposes the monolith into independent microservices and enables microservice instances scale out conveniently. The tactic to comply with this scalability is by increasing or decrease of microservice instances responding to the changing amount of requests [20].

Scalability may seem one of the minimum QAs due to its importance in distributed systems. One can measure this attribute by assessing the distribution of dependencies and diversity of synchronous requests made by the microservice [23]. It also can be measured by checking if horizontal and vertical scalability when applied does come with performance penalties [20].

2.2.2 Performance

Performance is one of the critical quality attributes in every software system, not only distributed systems. It is a measure of a system's ability to meet timing requirements when responding to an event. MSA systems often target performance by using lightweight and REST-based mechanisms for microservices' communication [20].

Is important to note that performance sometimes needs to be abdicated to favor scalability, the smaller the microservices the better the scalability and less performance whereas the bigger the microservice (reducing communication costs) the better the performance with the cost of less scalability. Companies need to make a trade-off when architecting MSA systems since containers and VMs can influence response times but the communication complexity of trying to make everything small can come with impacts on the performance of MSA systems [20]. This quality attribute can be measured using the response time of a microservice [23].

2.2.3 Maintainability

In terms of software engineering maintainability is “the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment” [26].

This quality attribute can be extracted from others like decreased level of system’s technology heterogeneity. And also, good scalability of the system normally indicates good maintainability too [23]. It seems that one of the best, most used, and most simple ways to analyze maintainability on its own is to use the metric of lines of code (LOC). LOC metrics can tell the size of the source code immediately, and it seems that a big LOC metric will lead to less maintainability in the system [25], [28].

2.2.4 Testability

Testability refers to the ability of the system to determine faults using testing mechanisms, this is normally done through some sort of execution-based testing. Testing has been seen as one of the most important quality attributes in MSA because of the nature of such systems, microservices are frequently modified and if testability is not present it can affect all the other quality attributes. This quality attribute can be measured by checking the presence of Automating Test Procedures and API Documentation and Management [20].

Multiple microservices are integrated between them creating large number of tests to perform, doing this by hand would be time-consuming and not efficient, for this reason there are multiple solutions in the market that provide the tools necessary to automate test procedures in MSA systems.

API specification is necessary for testing to provide an overview of a microservice. APIs are updated regularly and can turn the testing to be mor complicated. If the developers and testers have API documentation generated automatically at their disposable every time a new version of an API is created the testing procedures become more efficient. One can note that API documentation and testing procedures are the two measurements of testability, and they work hand in hand since the documentation generates the specification of the API reducing time testing that API.

3 State of Art

This chapter provides a state-of-the-art about the Jolie programming language, as well as other microservice-specific languages that support the same approach and concept of Jolie. Additionally, it is explained the reason behind the emergence of these languages.

3.1 Microservices specific languages

The field of microservices always makes developers think of complexity. To reduce the complexity is shown that the Model-Driven Engineering (MDE) or Model-Driven Development (MDD) method has become a vigorous methodology to design an MSA system [29], [30] reducing the complexity and effort of development. Microservice-specific languages enter the domain of MDE because they are formal languages, providing formalization for workflows and service composition. Their goal is to make the most out of the microservices architectural style and distributed systems possibilities [31].

These languages have the benefits of MDE but solve some problems intrinsic to MDE-based solutions like the fact that MDE is a long-term investment, needs customization of environment, tools, and processes, requires training, and creation of metamodels in complex systems leading to the use of several languages. In sum MDE comes with the problem of the difficulty of developing an MDE environment tailored to the company's needs [32] and microservice-specific languages can be seen as a new paradigm for abstraction in MSA system development.

Generally, microservice applications use object-oriented (C#, Java, etc) or functional languages (Haskell, Scala, F#) [11]. Even though these languages provide solutions for programming microservices, orchestrating communications between them, and taking care of the monitoring,

observability, and deployment, it is still quite time-consuming, more prompt to errors and bugs making the system very complex and prone to failures overall.

The creation and evolution of microservice-focused languages aim to solve this complexity and effort intrinsic to MSA systems taking advantage of MDE foundations but solving problems regarding MDE and traditional languages. To implement or move to an MSA system the standard industry approach is already done using deployment paradigms, but the linguistic paradigm and abstraction are unknown to the industry. Programming languages that natively support the microservices paradigm is something quite new and unexplored – since services are built using functional or object-oriented languages and then deployed and scaled using containerization.

Jolie and Ballerina programming languages are the two main players to take a step further in the programming world of microservice-specific languages. They do not eliminate the infrastructure solutions but they compete with other functional or object-oriented languages that are among the many players to do it [31], [33]. Such languages are born to embrace languages that have as the core technology the service-oriented paradigm [34].

So, in sum languages like Jolie and Ballerina appeared as an attempt to make a language that provides primitives to deal with concerns regarding microservices, eliminating the need for frameworks or libraries at the same time giving developers a way to produce and manage MSA systems more efficiently [35] as well as bridge the gap between integration and general programming languages providing agility and ease of integration [36]. The concept of Jolie for example - programming native microservices, already attracted the attention of some companies [33].

The microservices development came as an adaptation of this new architectural style of building software and instead of new languages developed for this purpose frameworks and libraries were created. Since these frameworks do the job the programming community is still starting to see the advantages and the power of a language that is focused natively on microservices. Once these languages start to be used in big companies or projects more and more microservices-directed languages will probably appear.

3.2 Jolie

Jolie project started in 2005, and the language was created with concepts related to more legacy approaches, like SOA and modeling of concurrency [37]. Since then, the language evolved and according to (The Jolie Team, 2021) it is now a “service-oriented programming language: (…)

designed to reason effectively about the key questions of (micro)service development including the following" [35]:

- "What are the APIs exposed by services?"
- "How can these APIs be accessed?"
- "How are APIs implemented in terms of concurrency, communication, and computation?"

But one can ask why a new programming language when there are a lot of frameworks that take care of building MSA systems, that work and are referred to in Table 11. When Jolie emerged in 2005 distributed systems were still too focused on SOA and not on MSA, the language later become to follow the microservice paradigm with the industry and literature adoption of microservices. The biggest reason Jolie is in constant development still today is not only to make the development of microservices more effective but in sum to minimize code-model distance [13], [38] for model-driven engineering to be optimized in MSA [29]. The frameworks referred on Table 11 use functions and objects modeling the services using these concepts, but in Jolie, the language has code to mimic in a direct way. Concepts are implemented directly with the approaches referred on 3.3.1 like APIs, access points, services, and behaviors [38].

For the Jolie team [39], this language is needed because of the central abstraction that it has. They argue that the build and deployment are done using native language primitives. Since loose coupling is an inherent problem in distributed systems architectures [40] the Jolie team also refers that in Jolie the programmer can't break loose coupling. Because two Jolie programs do not share data, they only exchange data by communicating. This communication supports every data protocol and communication technology presented on Figure 6. The share of data only happens in processes inside the same program/microservice. This way thread-safe programming [41] and reusability can't be broken by hidden shared data structures.

Also, Jolie allows for better coordination. The translation from MSA to code without changing the model is impossible. Jolie comes to support this translation reducing the risk of introducing errors or unexpected behaviors since complex message exchange structures are used in MSA. Jolie's key goal is to tame this complexity, that is the goal and why the language exists in the first place. When the merge between language and microservice technologies is thought one realizes that it can't be obtained by just using a framework on top of already existing mainstream programming languages like Java or Python. It requires the design of a new language from the foundation [13].

In the Code Snippet 1 is an example of a hello world program in Jolie, one just needs to create a file named for example *greeter.ol* and write the following code [35]:

```
// Some data types
type GreetRequest { name:string }
type GreetResponse { greeting:string }

// Define the API that we are going to publish
interface GreeterAPI {
    RequestResponse: greet( GreetRequest )( GreetResponse )
}

service Greeter {
    execution: concurrent // Handle clients concurrently

    // An input port publishes APIs to clients
    inputPort GreeterInput {
        location: "socket://localhost:8080"        // Use TCP/IP
        protocol: http { format = "json" }        // Use HTTP
        interfaces: GreeterAPI                    // Publish GreeterAPI
    }

    // Implementation (the behaviour)
    main {
        /*
        This statement receives a request for greet,
        runs the code in { ... }, and sends response
        back to the client.
        */
        greet( request )( response ) {
            response.greeting = "Hello, " + request.name
        }
    }
}
```

Code Snippet 1 – Hello world program in Jolie

This creates a new service, to run it one just needs to open the OS-native terminal and run the following:

```
jolie greeter.ol
```

To test a client request, one opens another terminal and runs the following:

```
curl http://localhost:8080/greet?name=Jolie
```

The output will be:

```
{"greeting":"Hello, Jolie"}
```

(The Jolie Team, 2021) say that Jolie allows "to build complex systems without having to worry about the underlying communication details of the included microservices" [39]. The Jolie team also does not want to reinvent the wheel when not necessary. The language can be used and run inside a Java program due to the offered Jolie's Java API. This way the language can be used as a Java library with Jolie programming syntax if needed [39]. Not is the flexibility immense but also the simplicity is as shown in the snippet of Code Snippet 1, even if one never handled microservices before one can understand the code being written since it provides a very formal abstraction. This formal abstraction forces the developer to focus on what matters and lead him/her to fewer errors codifying the system by being less complex code that he/she needs to write. Another interesting and important aspect of the language shown in the snippet of Code Snippet 1 is the separation of concerns between behavior and deployment, being deployment information the addresses at which functionalities are exposed, communication technologies, and data protocols used in the interaction between services [13].

To demonstrate a better perspective between microservice-oriented languages like Jolie and microservices frameworks the same hello world program developed in Java using in Spring Boot framework as a dependency. One first creates three files, the build automation tool file in this case using Maven, the controller, and the main application executor as shown in Figure 3.

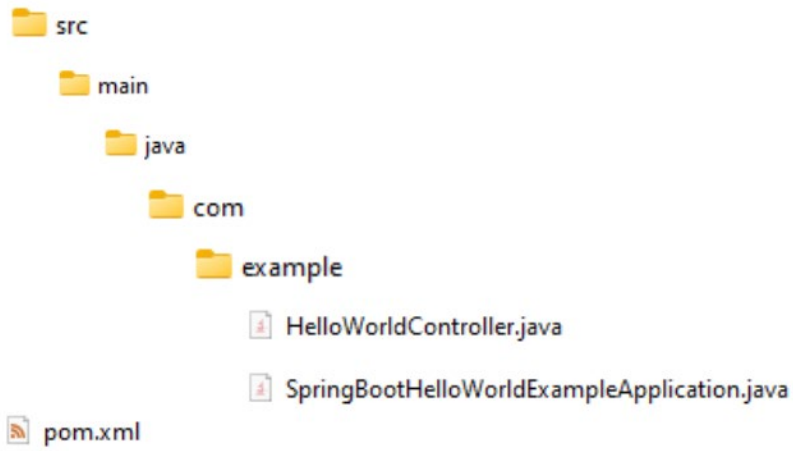

Figure 3 – Files needed for Spring Boot hello world program

The file *SpringBootHelloWorldExampleApplication.java* is the main executor of the application and is written as shown in Code Snippet 2.

```
package com.example;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication
public class SpringBootHelloWorldExampleApplication
{
    public static void main(String[] args)
    {
        SpringApplication.run(SpringBootHelloWorldExampleApplication.class, args);
    }
}
```

Code Snippet 2 – Spring Boot hello world program main executor

Then the controller that represents the behavior of the service is presented in the Code Snippet 3.

```
package com.example.controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloWorldController
{
    @RequestMapping("/")
    public String hello()
    {
        return "Hello world";
    }
}
```

Code Snippet 3 – Controller for the Spring Boot hello world program

Finally, Spring Boot requires many dependencies and libraries to work. These libraries can be added or deleted according to the necessity of the developer. The build automation tool Maven writes the entire build on a *pom.xml* file which is normally referred to as the POM file and is represented for this specific program on the Code Snippet 4.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.2.BUILD-SNAPSHOT</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
```

```
<artifactId>spring-boot-hello-world-example</artifactId>
<version>0.0.1-SNAPSHOT</version>
<name>spring-boot-hello-world-example</name>
<description>Demo project for Spring Boot</description>
<properties>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.1.RELEASE</version>
        <type>pom</type>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>

    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
        <exclusions>
            <exclusion>
                <groupId>org.junit.vintage</groupId>
                <artifactId>junit-vintage-engine</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

```
    <repositories>
        <repository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
        </repository>
        <repository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>spring-milestones</id>
            <name>Spring Milestones</name>
            <url>https://repo.spring.io/milestone</url>
        </pluginRepository>
        <pluginRepository>
            <id>spring-snapshots</id>
            <name>Spring Snapshots</name>
            <url>https://repo.spring.io/snapshot</url>
            <snapshots>
                <enabled>true</enabled>
            </snapshots>
        </pluginRepository>
    </pluginRepositories>

</project>
```

Code Snippet 4 – POM file for the Spring Boot hello world program

From this simple analysis is possible to conclude that Jolie is more straightforward in terms of understanding and reading the code. Spring Boot requires a developer to know the libraries to use, and what they do and the annotations and code itself have more lines of code making it harder to read.

3.2.1 Orchestrator

Jolie is not only a programming language but also an orchestrator and interpreter for service-oriented programs that instead of an XML-based syntax follows a programming-like syntax [8]. Thus, providing an easy way to learn the language for a developer who is costumed with orchestration.

This can be because the language took inspiration and can be seen as a descendant of orchestration languages such as WS-BPEL and XLANG [8], [13]. It is a candidate to apply certain techniques: runtime adaptation; process-aware web applications [31] using provide-until construct (that allows a developer to program behaviors that are quite repetitive in the MSA system and are driven by external calls and other services) [13]; correctness-by-construction in concurrent software [31].

3.2.2 Native microservices

The language contains native constructs that can point to communication between services/processes and native support for distributed workflows programming [33]. Provides novel workflow primitives since each Jolie program is a workflow – the blueprint of the program behavior, has receive operations from input ports and send operations to output ports, has a sequence, conditional and loop constructs. Provides parallel composition to enable concurrency, and input-guarded choice to wait for multiple incoming messages, this means that input ports are available only when a corresponding receive is enabled, otherwise messages to this port are buffered [13]. It is a simple way to design orchestrators and coordinators [33].

These characteristics make Jolie heavily MSAs focused, even a simple variable fit into the microservice paradigm with Jolie. The current approach for programming microservices is that it does not matter the technology (programming language) used provided the approach that the deployment is made in containers, Jolie reverses this idea by presenting that the deployment does not matter how one does it, provided that a microservice programming language is used, this is the main key idea behind Jolie [13] and the native approach the language takes into microservices.

3.2.3 Advantages

The Jolie developers argue that with Jolie one does not have to deal with multiple technologies when a new MSA system is built as represented in Figure 4. Multiple technologies can lead to the slow and complex development of microservices and with Jolie higher speeds of development are achieved because Jolie lowers the level of knowledge for developing an MSA project. The Jolie authors claim that Jolie is easy to learn and the communication is facilitated by sharing a common linguistic reference and uniform technological approach helps to a common understanding between developers [33]. All of Jolie's advantages in terms of development can be summed up in the gains of speed, sharing, and simplicity.

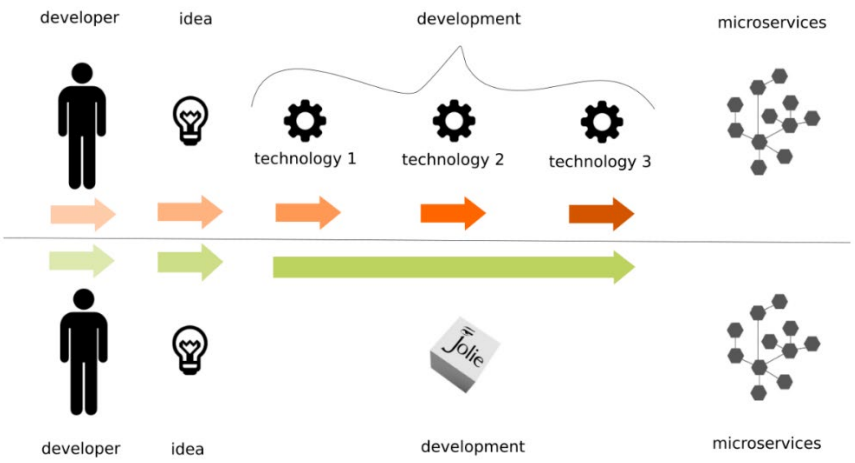

Figure 4 – Simplicity and speed of Jolie development [33]

Jolie has a deep integration between language and microservice technologies by using tree structure variables [13]. Jolie code is simple that can be easily understood how the service can be invoked and which service(s) it requires to work without prior knowledge of the language [9], [13]. This is accomplished using Jolie *Ports* and *Interfaces* mentioned in section 3.3.1 as first-class entities/citizens. Repeated functionality is exposed differently using different *InputPorts*, this way a service is never replicated. Supports the programming of protocols and workflows that are not natively supported by mainstream languages, it can have a new *Port* providing the same functionality using the SOAP protocol and another using HTTP protocol for example. Also, since *Port* (using location and protocol) can be changed dynamically by the behavior this improves flexibility. The language allows changing the communication of services when the environment changes without changing the behavior of the service [13].

For design patterns in microservices referred to in Annex B, Jolie implements all the main patterns. In terms of circuit breakers, Jolie can implement a circuit breaker that can be deployed as a client-side circuit breaker, service-side circuit breaker, or proxy circuit breaker and a Jolie circuit breaker can be adapted to microservice interfaces changes [9]. Other solutions like Hystrix [42] require an additional implementation of a HystrixCommand [9] and even though Jolie is not as mature as the Netflix solution for circuit breakers it has the adaptability advantage [9]. In terms of load balancing and service discovery which are both supported by Jolie, Netflix tools provide client-side solutions and Amazon tools provide a server-side solution, and both can be achieved with Jolie [9].

Two programming languages very similar to Jolie are Erlang and Golang. Erlang comes close to Jolie as one of the most mature implementations of processes and support workflows for the actor

model. The problem with these languages is the non-separation of behavior and deployment and explicit ports are not present in both languages. WS-BPEL is another very similar language to Jolie as it provides ports, interfaces, workflows, and processes but is just a composition language and cannot be used to program single services, and those services when programmed are heavy [13].

3.2.4 Drawbacks

In the literature, Jolie is referred to as having the drawback of an API gateway implemented with the language like the one in [9] where (Montesi et. al, 2016) refer that "redirections in Jolie (…) do not take care of extra features such as security and monitoring by themselves" and that an API gateway implemented with Jolie "should be composed with other patterns e.g., circuit breakers, to achieve this extra features, keeping concerns separate" [9].

Also, The Jolie Team indicates on the official Jolie page that this type of programming has a performance cost (even though not been noticed in production environments so far) [39]. This performance issue is noted in other languages where the use of abstraction makes performance decrease, and with time the language gets optimised for the gap between performance with abstraction and without abstraction is not relevant. Jolie is a good option for communication handling but not for critical algorithms where performance is key. That's why the language allows the embedding of other languages like Java, C, and JavaScript [35] as shown in the Code Snippet 5 and Code Snippet 6.

```
embedded {
JavaScript :
    "MyService.js" in MyService
}
```

Code Snippet 5 – Embedded JavaScript code in Jolie

```
embedded {
    Java: "mypackage.MyService" in MyService
}
```

Code Snippet 6 – Embedded Java code in Jolie

3.3 Jolie microservices

3.3.1 Architecture

Jolie as a microservice-focused programming language has as main blocks for programming the following components:

- Interfaces (APIs)
- Ports (access points for deployment)
- Workflows and Processes (behaviour)
- Types

They are architected and related as shown in Figure 5.

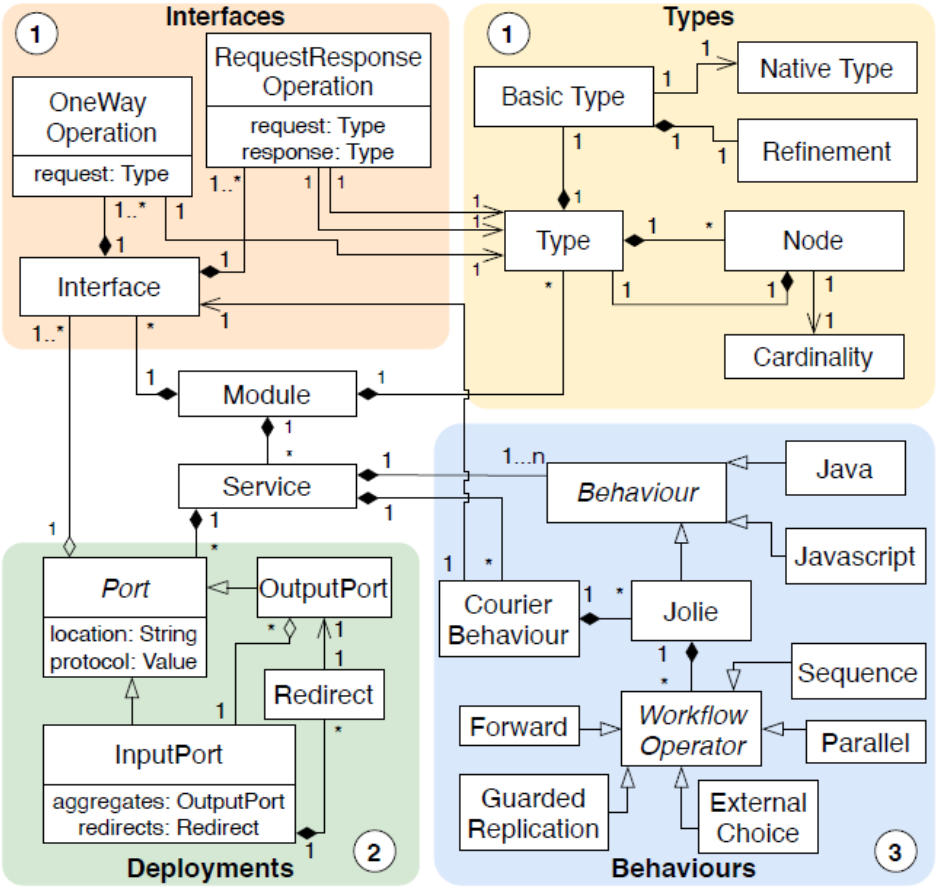

Figure 5 – Jolie metamodel architecture [29]

3.3.1.1 Interfaces

As referred to in Annex B a component's program in an MSA architecture needs to be independent to compose it in large systems. To achieve this independence services communicate using published interfaces, hiding the implementation [13] for better bug fixing, management, and updating of the services [9], [14]. The interfaces also offer losing coupling, and contribute to technology agnosticism,

minimizing this way the assumptions made on the technologies used to implement behaviors of the microservice program [29].

These published interfaces are known as APIs and describe the functionalities that a microservice provides to clients to be remotely invoked and the ones it requires. Jolie's Interfaces are the way Jolie conceptualizes APIs in the language and they consist of first-class citizens in Jolie microservices [13], [29].

As shown in Figure 5 a Jolie Interface is a collection of Operations that can be *OneWay* or *RequestResponse* operations [29], [35]. A *OneWay* operation receives messages [35] and the sender delivers the message to the service but does not wait for the processing by the service's behavior [29]. A *RequestResponse* operation replies or receives messages and sends back a response [35], this means that the sender delivers its message and waits for the processing by the service's behavior to receive a reply with some response [29].

3.3.1.2 Ports

Ports are access points for the API/interface that a microservice implements. Is the where and how the clients interact with the API, which communication technologies, data protocols, and interfaces are required and offered? As microservices communicate via message passing with other components of the MSA is important to know how data is structured for transmission (for example using JSON) and transmitted (where microservices can contact each other and how data are transported among them (for example using IP addresses) [29].

In other words, it represents how the functionalities in terms of the stated characteristics of a microservice are made available into the network. Ports are important because of the need to separate the implementation of the service from what it provides and/or needs, this way one can look at a service and automatically know what and how it works without looking at the implementation [13].

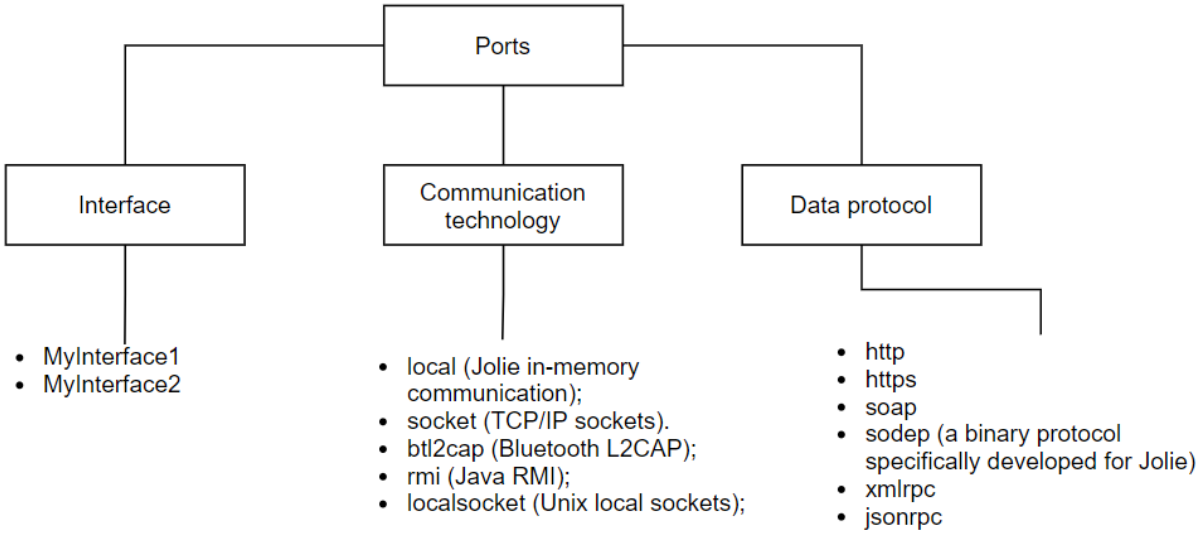

Figure 6 – Jolie *Ports*

A Jolie Port access point in terms of communication technology is defined by a URI that can have any of the types defined in Figure 6; a data transfer protocol and it also can cluster one or more Jolie Interfaces [29].

On Figure 5 is presented that a Jolie Port can be an *InputPort* or *OutputPort* [29], [35]. *InputPort* describes the functionalities that the service provides to the rest of the MSA [13] by exposing a public contract to clients [29]. *OutputPort* describes the functionalities that the service requires from the rest of the MSA [13] by defining access points used in behaviors to invoke other microservices [29].

3.3.1.3 Workflows and processes

(Guidi et. al, 2017) refers that a "workflow defines the blueprint of the behavior of a service" [13]. In Jolie, the workflow is represented in code by the main keyword [29], [43].

This blueprint is the business logic or the implementation of the microservice. Jolie allows developers to use Jolie, Java, or JavaScript to write the microservice behavior [29].

In runtime the workflow is then called a process that can interact with multiple other components of the MSA (services or clients) and the number of processes running can always change in runtime [13], [44]. The behavior when running can behave in various execution modalities that in Jolie can be single, concurrent, or sequential [44] to align with the principle that microservices can be used in different manners and are completely independent, this means that Jolie processes run independently of the others avoiding interference [13].

Single

- Used by default, running the program behavior (main construct) once [44]

Concurrent

- Runs all the time when the first input statement can receive a message [44]
- Message exchanging can happen more frequently

Sequential

- When the currently running instance terminates the program behaviour (main construct) is made available again [44]
- This provides exclusive access to the resource (a component of the MSA) [44]

3.3.1.4 Types

Microservices exchange messages and these messages need to be in a specific format/type, Jolie exchanges these messages as data trees [13], [29], [31].

Jolie supports seven basic data types [29]:

- bool: booleans;
- int: integers;
- long: long integers (with L or l suffix);
- double: double-precision float (decimal literals);
- string: strings;
- raw: byte arrays.
- void: the empty type.

It also supports custom data types very similar to mimic the popular Data Transfer Objects (DTO) [45]. Known in distributed systems world for being the most used approach for handling data and reducing communication latency. Jolie data types and even interfaces are technology agnostic [29] modelling a DTO that is built on native types and are used in *Operations* (see Figure 5).

3.3.2 Development

The development of a Jolie system starts with the definition of *Interfaces* and *Types*, to proceed with the *Ports*. They are written where locations and protocols are defined, and the *Interfaces* are used (see Figure 5). Lastly, behavior of each service must be specified using the *main* keyword.

The example provided in [35] where a basic calculator service is built is here detailed. The *Types* in Jolie are the models that will be used in a specific service, and four *Types* are used (see Code Snippet 7). Each of these *Types* can be used in the *Interface* of the service. The Code Snippet 7 shows the calculator example *Types*.

```
type SumRequest: void {
    term[1,*]: int
}

type SubRequest: void {
    minuend: int
    subtraend: int
}

type MulRequest: void {
    factor*: double
}

type DivRequest: void {
    dividend: double
    divisor: double
}
```

Code Snippet 7 – Use of *Types* in Jolie

These *Types* can then be used in *Interfaces* making it easy to write the blueprint of an operation.

```
interface CalculatorInterface {
    RequestResponse:
        sum( SumRequest )( int ),
        sub( SubRequest )( int ),
        mul( MulRequest )( double ),
        div( DivRequest )( double )
}
```

Code Snippet 8 – *Interface* code example in Jolie

Types and *Interfaces* are saved in the same file for example *CalculatorInterfaceModule.ol* then this file can be used on multiple services separating behavior and model. After defining the *Types* and *Interface* in their respective file for the service the developer can then pass to the next phase which is programming and running the service. For this a new file called *CalculatorService.ol* can be created, the file will have the workflow and *Ports* of the service and starts like portraited on Code Snippet 9 where the *Ports* are defined.

```
from CalculatorInterfaceModule import CalculatorInterface
```

```
service CalculatorService {
  inputPort CalculatorPort {
      location: "socket://localhost:8000"
      protocol: http { format = "json" }
      interfaces: CalculatorInterface
  }
}
```

Code Snippet 9 – Service behavior initial syntax in Jolie

The programmer can now go inside the *CalculatorService* block and type the *Ports* and behavior of the service.

```
from CalculatorInterfaceModule import CalculatorInterface

service CalculatorService {

  inputPort CalculatorPort {
      location: "socket://localhost:8000"
      protocol: http { format = "json" }
      interfaces: CalculatorInterface
  }

  main {
      [ sum( request )( response ) {
          for( t in request.term ) {
              response = response + t
          }
      }]

      [ sub( request )( response ) {
          response = request.minuend - request.subtraend
      }]

      [ mul( request )( response ) {
          response = 1
          for ( f in request.factor ) {
              response = response * f
          }
      }]

      [ div( request )( response ) {
          response = request.dividend / request.divisor
      }]
  }

}
```

Code Snippet 10 – Service workflow and *Ports* in Jolie

The development in Jolie is done always with the described four main steps in mind: create the *Types* as portraited on Code Snippet 7; create the *Interfaces* as portraited on the snippet Code Snippet 8; create the service *Ports* as portraited on the Code Snippet 9; create the *Service* behavior as portraited on Code Snippet 10. Meaning that supposing that there is a service one must build it. First one thinks about the model to be used, and what the data model will be both requested and/or responded back to. Then how many operations will this service allow and support, and which models are used in which one. Next, proceed with identifying the infrastructure that will be used like protocols and addresses. Finally, write the service behavior for each operation. These four steps make it possible to quickly build a microservice with minimal effort.

3.3.3 Containerization

Section 3.2 refer to that one of its advantages and goals of Jolie is to tame the complexity of MSA. A common technology used to do this is container technology where the most popular one is Docker [46]. Docker appeared to handle the multiple possible machines where the services are deployed since these machines could have a different OS, installed libraries, and configuration. It was hard to justify using virtual machines to handle all this, an abstraction was needed. In 2013, Docker was the pioneer and standardized the industry with what is known as containers. These containers are a unit of software where developers isolate an application (that can be a microservice or any program, known as a containerized app) from the environment of the machine where the application is deployed which solves the problem of the application working on a machine and not on others when deployed [46]. In the MSA context, Docker isolates the execution of services from other programs that might be running on the same machine.

The literature is often focused-on containerization and deployment solutions without consideration of linguistic ones [31], [33] due to the emergence of the Cloud and the Internet of Things where the deployment and relocation of containers are important. Jolie is not incompatible with the trend of containerization, but it moves the focus from deployment to development in order to reduce and tame complexity by providing interfaces to manage communication [13]. The mainstream languages using their frameworks referred to in Table 11 do not handle this problem already as they do not provide enough support for mastering communication since service coordination is programmed in an unstructured and ad-hoc way hiding communication structure with less relevant low-level details [13].

4 Analysis and design

This chapter introduces the experiment solution by describing its characteristics, providing functional and non-functional requirements, and the use cases. The architecture of the solution is presented and explained within the architectural choices using diagrams and tables when needed. Also, the technology stack is described, and alternatives are explored leading to the final technology choices.

4.1 Scenario

The proposed experiment is an e-commerce application where users can browse items, add them to the cart, and purchase them. The application is built based on the microservices architecture style.

Two different solutions are to be built:

- One that will be built using Jolie
- Another will be built using Spring Boot since it is recognized that Java is the most popular technology for MSA-based system development (see Table 11) and Spring Boot currently leads the framework market for MSA systems [47]

These two technical possibilities have the goal to ponder the implications and limitations of Jolie as well as the benefits of such technology.

The MSA prototype system for this scenario has multiple services running and the need for interservice communication is required. The communication between microservices will be done using Jolie as an orchestrator for the Jolie solution. For the Spring Boot solution, the communication

is implemented using Apache Kafka [48] for the reason that is the technology the developer is more efficient and knowledgeable at.

To interact and test the microservices the software Postman [49] will be used. The graphical user interface (GUI) of Postman looks simple and effective and for this reason, a front end will not be built for the application. Also, since the purpose of this work is in the context of microservices, and distributed systems, the Postman GUI by itself allows the interaction with microservices APIs and performs tests to the microservices developed.

4.2 Requirements

Table 1 shows the functional requirements for the project to be developed. The requirements have the goal to provide users with all the functionalities to support the entire process of online product buying from browsing, purchasing, and shipping.

Table 1 – Functional requirements of the application

Identifier	Functional requirement	Description
FR1	Cart management	System provides users to add and remove items from their cart
FR2	Product lookup	System provides users with the ability to browse and search products
FR3	Product buying	System provides users the ability to buy products
FR4	Email management	System sends emails to users from transaction info and status
FR5	User management	System allows the user to edit personal information

Table 2 shows the non-functional requirements for the project which must be built having the microservices architecture as the focus where communication is handled using sagas and design patterns considered.

Table 2 – Non-functional requirements of the application

Identifier	Non-functional requirement	Description
NFR1	Technology choice	The system/solution must use Jolie and/or Spring Boot
NFR2	Use of a microservice database	Every microservice as their own database
NFR3	Use of MSA	Solution has the focus on the microservice system architecture, and their principles need to be considered
NFR4	API gateway use	The microservices are called through an API gateway

NFR5	Saga pattern use	Saga transactions are implemented between the microservices, more specifically saga orchestration for interservice communication since the developer is familiarized with the approach
NFR6	Use of service discovery	Microservices are registered through a service discovery system

4.3 Usage

The system's usage can be detailed by presenting the actors of the system and the use cases that these actors can perform.

The system is composed of the following actors:

- Customer
    - This actor represents the customer of the e-commerce application
    - This actor browses and buys products from the system, adds the products to the cart, and places orders for those products
- Moderator
    - This actor represents the moderator of the e-commerce application
    - The moderator can perform operations that are specific to modify the system standards – for e.g.: create, edit, or delete a product and edit specific statuses of orders
    - Has access to features that a normal customer should not have access to

Table 3 shows the use cases for the system providing all the user actions and essentially what the user can do with the built system.

Table 3 – Use cases of the application

Identifier	Actor	Description
UC1	Customer	Search and/or browse a product
UC2	Customer	Add product(s) to the cart
UC3	Customer	Remove product(s) from the cart
UC4	Customer	Place an order from the products of the cart
UC5	Customer	Manage their profile information
UC6	Customer	Consult their own cart
UC7	Customer	Consult the status of a placed order
UC8	Customer	See the history of purchases/orders
UC9	Moderator	Change the status of an order
UC10	Moderator	Manage the product catalogue
UC11	Customer	Manage their orders

The use case diagram represents the dynamic behavior of the system when it is running and can be represented in Figure 7.

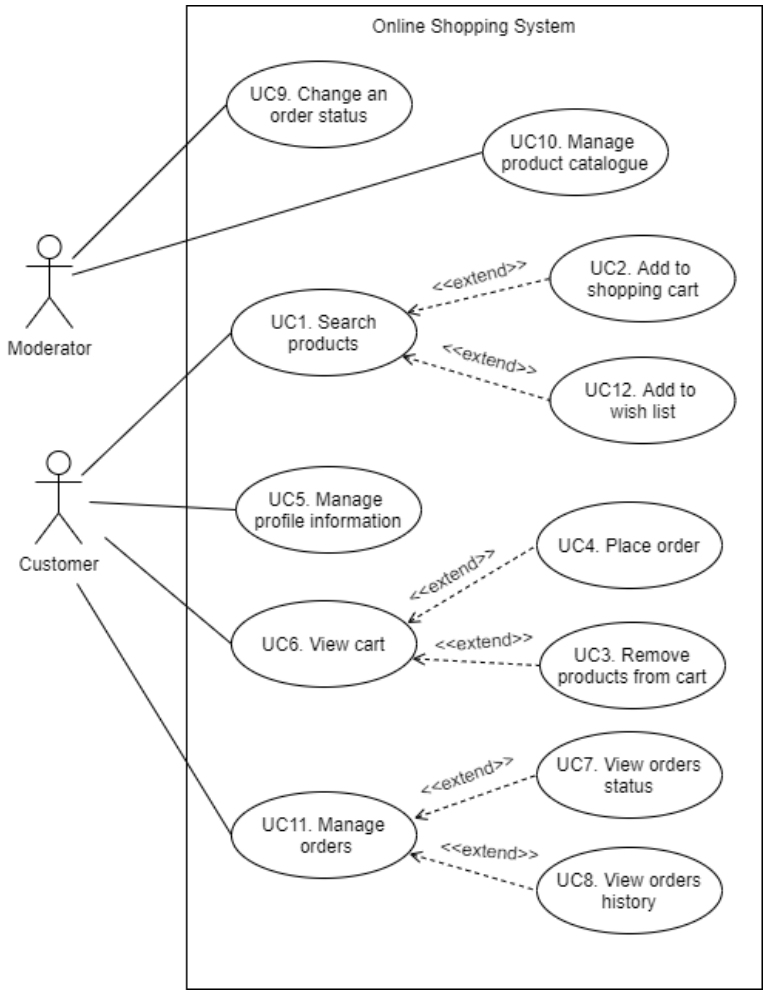

Figure 7 – System's use cases diagram

4.4 Architecture

As in every other informational system, the architecture plays a very important role because it represents the foundation of the system. To ensure a software system has a solid foundation it can be proven by the software's architecture. In MSA-based systems, the principle still applies. It is applied with more intensity because the entire system is distributed and can communicate with other external systems, which makes it very fault prominent. Every aspect of the architecture and documentation needs to be done a priori to the development of the system itself.

The architecture will be described in diagrams using Unified Modelling Language (UML) and tables with the following points:

- System components diagram
- Microservices description table
- Saga transactions diagram

4.4.1 Components

The system consists of a group of microservices that together provide all the features necessary to have a complete e-commerce application. As seen in Figure 8 the system has 3 main components – the client module, the API gateway, and all the microservices each one with its database. All these components are deployed in different servers including each microservice and its database.

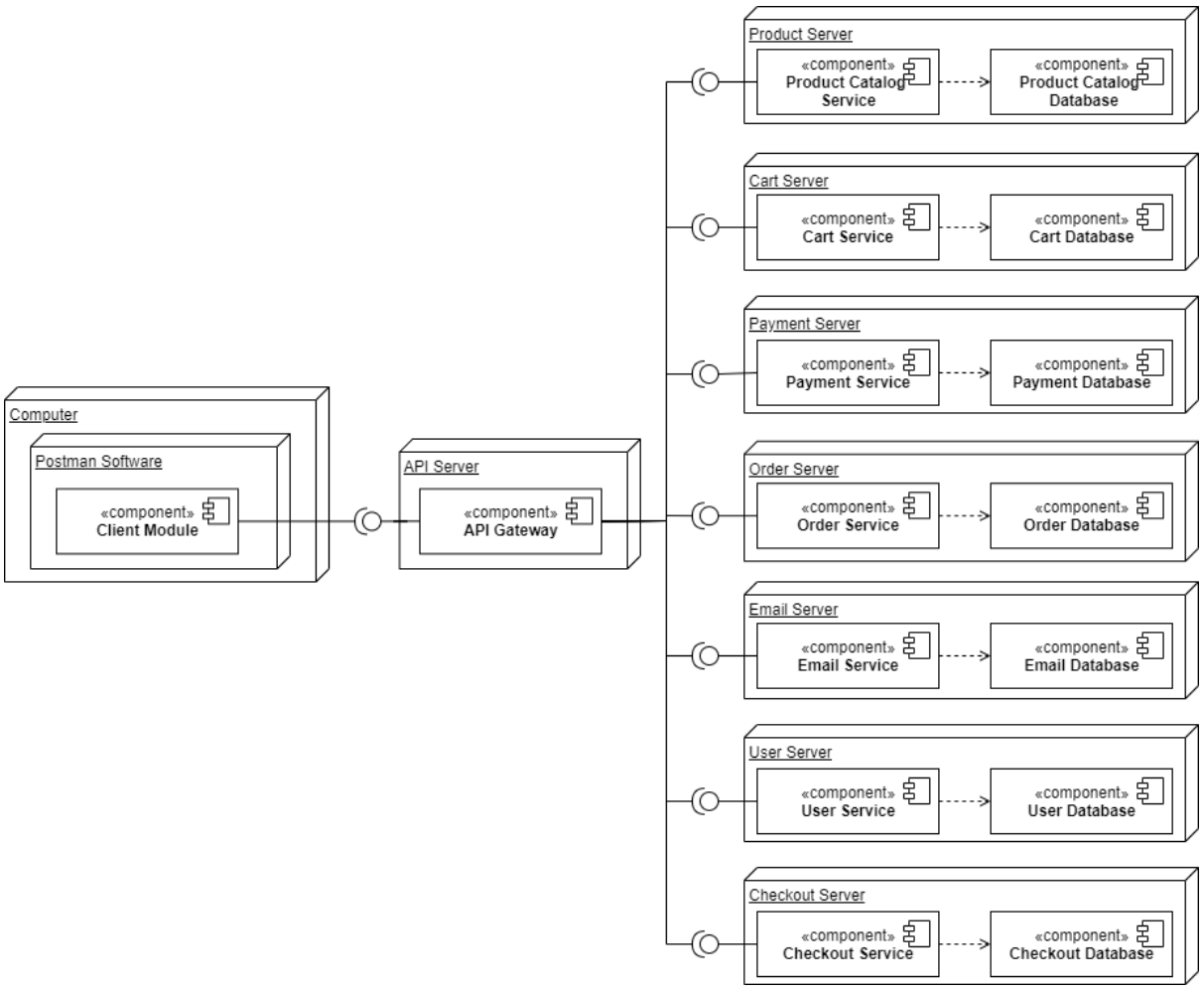

Figure 8 – System component diagram

The components are divided into different servers, and the client communicates with an API Gateway that is present on another server.

4.4.2 Microservices description

Table 4 describes every microservice and its purpose. Shows what every microservice will be responsible for and the general notes of each one.

Table 4 – Microservices description [50]

Service	Description
CartService	Stores the items in the user's shopping cart and retrieves it.
ProductService	Provides the list of products and the ability to search, browse products and get individual products.
PaymentService	Charges a given credit card (mock) with the given amount.
OrderService	Ships items to the given address (mock).
EmailService	Sends users an order confirmation email (mock).
CheckoutService	Retrieves user cart, prepares order, and orchestrates the payment, shipping, and the email notification.
UserService	Registers and holds information about the system users.

4.4.3 Orchestrated saga

The central coordinator is commonly called the orchestrator - it defines the order of execution of actions and triggers required compensative actions. The orchestrator controls what happens, therefore allowing for a good extent of visibility into what is happening with any given saga. In general, orchestrated sagas heavily use request-response interactions between services. An important caveat to avoid too much centralization with orchestrated flows is to ensure different services play the role of the orchestrator for different workflows.

In this application, the two orchestrator services are the checkout and cart services. In the Spring Boot solution, this orchestration is made using Apache Kafka and in the Jolie solution, the orchestration is built-in within the Jolie programming language.

As Figure 9 portrays in a diagram how the Kafka cluster can handle the orchestration in the Spring Boot solution.

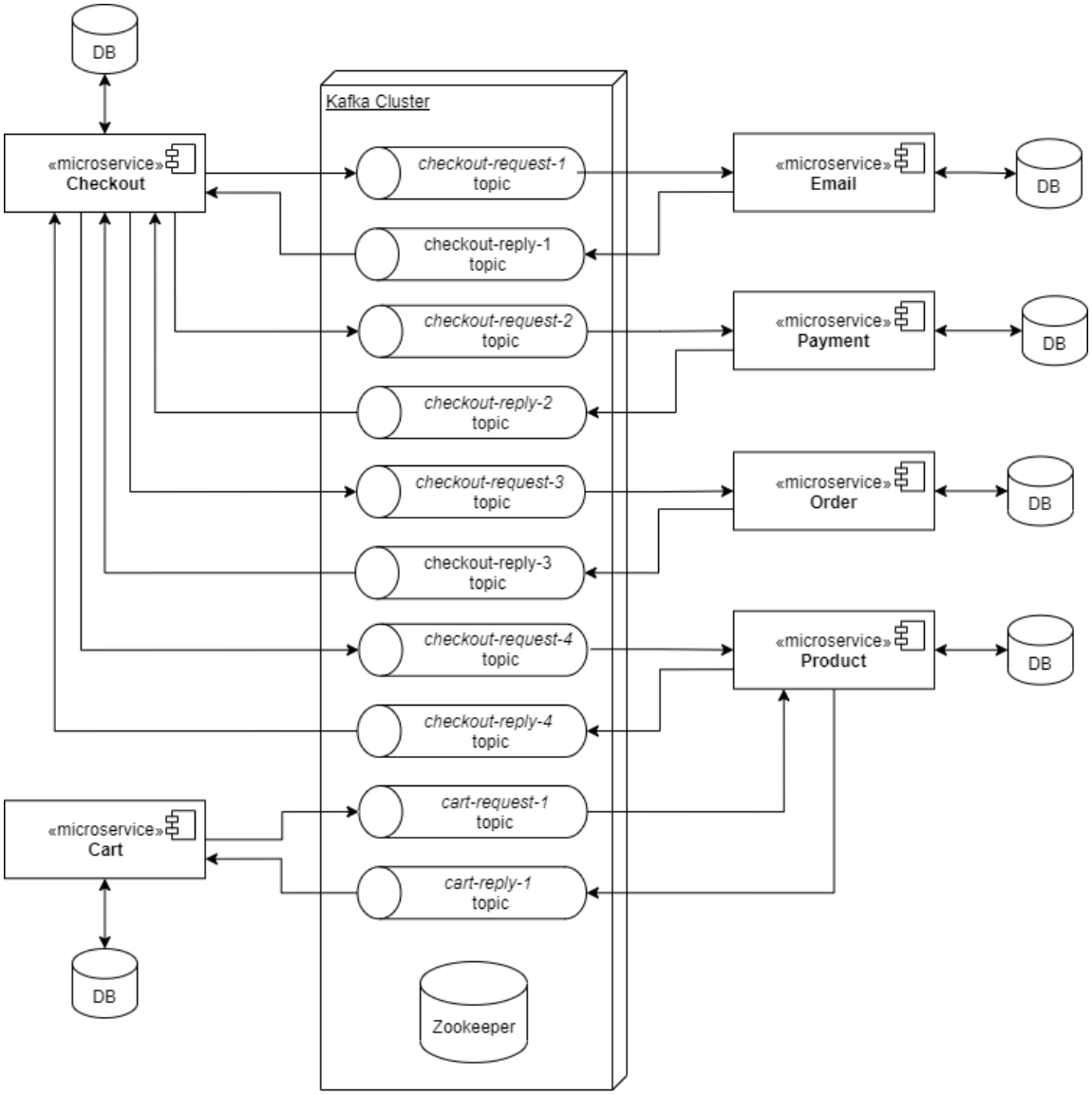

Figure 9 – Spring Boot solution saga orchestration

Whenever the checkout service is called it sends a new message in the related Kafka broker, then the order, email, payment, and cart services are listening to the respective topic and reading the messages there, when they notice a new message, they trigger their respective actions. The cart service also sends a message to another topic that the product service is listening for checking the total amount in the cart is up to date with the product prices. Zookeeper [51] is responsible for making the service discovery aspect of the architecture.

For the Jolie solution, there is no need of using external solutions like Kafka, even though it would be possible since Jolie integrates with Java [52], by using embedding mechanisms. Jolie contains native support for orchestration as explained in section 3.2.1 of the thesis. In Jolie a workflow can be implemented by calling interfaces that are known to a service, working like APIs. The communication between services can be achieved inside the codification of the service itself without the need for

external tools. Section 3.3.1.1 explains that the interfaces hide the implementation or behavior of the service, which allows for this to be achievable. A service can call another service like an API call, and it can know what to send and what it receives without even knowing the implementation or behavior at the called service. Figure 10 illustrates the process for the Jolie solution in the checkout service and cart service which are both orchestrators in the application.

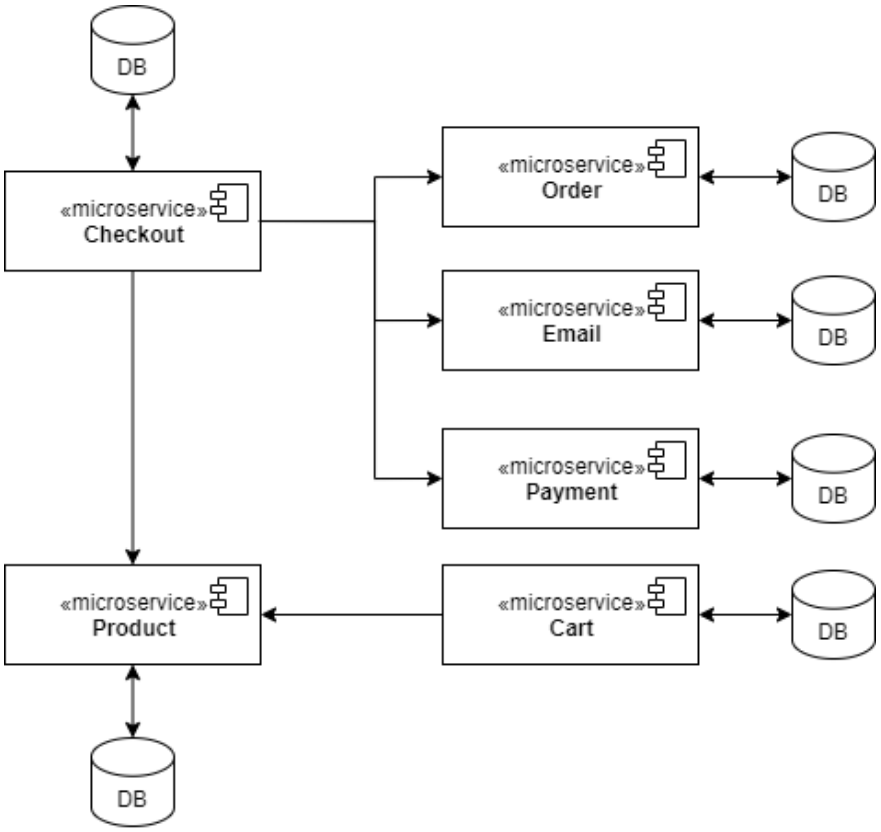

Figure 10 - Jolie solution saga orchestration

As one can compare in the Jolie solution the microservices themselves implement the orchestration saga required for the application. This is an important difference in Jolie because it reduces the number of technologies necessary to develop the system.

4.5 Technology stack

The use cases are mapped into a specific technology stack that was chosen. For the communication technology, the choices are HTTP as the communication protocol and JSON as the syntax of the messages between components. The database of each microservice can be implemented in theory with any database engine system. For the solution prototype, only open-source databases were considered. The project does not have a dimension so large that requires commercial products, that intend to produce more robust database systems. For this reason, for each component of the

database shown in Figure 8 only open-source database management systems are considered. Due to the author's experience with PostgreSQL, this technology was chosen to increase the speed of development in terms of database components.

The client module is made using the Postman API client platform that is normally targeted for developers to design, build, test, and iterate their APIs. The microservices as referred on section 4.1 will be developed two times each, one with Jolie programming language and another with Spring Boot framework for comparison reasons.

Comparison of technology stack between the two solutions:

- Jolie based solution:
    - Jolie programming language
    - PostgreSQL database management system
    - Docker OS-level virtualization platform
    - Kong API gateway
- Spring Boot based solution:
    - Java programming language
    - Spring Boot framework
    - PostgreSQL database management system
    - Docker OS-level virtualization platform
    - Kong API gateway
    - Apache Kafka event store and stream-processing platform
    - Apache Maven build automation tool
    - Apache Zookeeper distributed coordinator

5 Implementation

This chapter provides an extensive description of how the solution is implemented. For this an initial overview is presented, then each microservice is explained on how it is implemented and how the data flows in the system. The source code of the implementation can be consulted on GitHub [53].

5.1 Overview

The solution experiment consists of a prototype MSA-based system as stated in previous sections of the thesis. The microservices developed in the experiment are both developed in Jolie and Spring Boot. Currently, microservices development as shown in Annex B is framework heavy and the currently accepted standard in the industry seems to share this idea. For this reason, Spring Boot is used as a comparison to Jolie because it is the most used framework for microservice development. Although it may seem to be a standard in microservices is still a relatively new concept and some companies are still trying to migrate their system. There is a high chance of these companies choosing other frameworks other than Spring Boot because the trend is not consistent. Spring Boot is used because it is the most used one and it shares some similarities with other popular frameworks.

Each microservice will communicate to its database which stores data relative to the entities used by the microservice. In microservices there is a relation between data handled differently – since different databases are used the concept of intermediate tables for multiple-to-multiple relations between data is not existent. A microservice's database contains data relative to the entity of that microservice – e.g., the product microservice will have a database that stores product information, the orders microservice will contain a database that stores order information so on, and so forth.

5.1.1 API Gateway

An API gateway is a management tool contained between the consumer/client of the system and a collection of backend microservices. It can be interpreted as the single point of entry for a defined group of APIs that the microservice programs expose. The client of these API gateways varies, for example, mobile devices, web browsers, API clients, internal systems, or any third-party application that can communicate with the system. From a network perspective the API gateway behaves like a reverse proxy to accept requests from clients and returns the appropriate result [54].

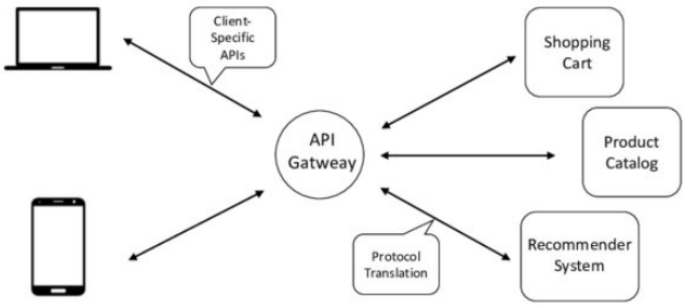

Figure 11 - The API Gateway architectural pattern [55]

Each microservice and its own database are deployed on different servers, therefore the communication between the client and the microservices should pass through an API Gateway, as a good practice and from a practical perspective. The existence of an API Gateway is a popular feature in MSA, as referred to in the design pattern section of Annex B, where the API Gateway is seen as a common design pattern for MSA.

The market of API Gateways is already robust and presents a great number of good solutions. Even though Jolie can be used to implement an API Gateway on its own, since the focus of the thesis is the microservices development with Jolie and not the design patterns of microservices, the API Gateway will be an existent one. The developer already has experience with API gateway platforms therefore for this experiment Kong [56] is used. Kong is one of the world's most popular API gateway which is built with microservices and distributed systems as its target, so it is the choice for the system.

Both solutions (Jolie-based one, and Spring Boot-based one) have an API gateway running with Kong. Docker simulates the containerized production environment, so in each solution, on the *docker-compose.yml* file, Kong is installed as a service (see Code Snippet 1) providing all the features necessary for Kong to be accessed by clients and to communicate with the microservices.

```
version: "3.7"
```

```yaml
services:
  kong:
    image: kong:latest
    volumes:
      - ./kong.yml:/usr/local/kong/declarative/kong.yml
    environment:
      - KONG_DATABASE=off
      - KONG_DECLARATIVE_CONFIG=/usr/local/kong/declarative/kong.yml
      - KONG_PROXY_ACCESS_LOG=/dev/stdout
      - KONG_ADMIN_ACCESS_LOG=/dev/stdout
      - KONG_PROXY_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_ERROR_LOG=/dev/stderr
      - KONG_ADMIN_LISTEN=0.0.0.0:8001, 0.0.0.0:8444 ssl
    ports:
      - "8000:8000"
      - "8443:8443"
      - "127.0.0.1:8001:8001"
      - "127.0.0.1:8444:8444"
```

Code Snippet 11 - docker-compose file for Kong API gateway machine

In the *volumes* entry of the Code Snippet 11, the container named *kong* is trying to copy a *kong.yml* file in the same directory as the *docker-compose.yml* file. This file is where the entire API Gateway definition is written as shown on Code Snippet 12.

```yaml
_format_version: "2.1"

services:
  - name: product-service
    url: http://host.docker.internal:9051
    routes:
      - name: product-route
        paths:
          - /product

  - name: user-service
    url: http://host.docker.internal:9052
    routes:
      - name: user-route
        paths:
          - /user

  - name: order-service
    url: http://host.docker.internal:9053
    routes:
      - name: order-route
        paths:
          - /order
```

```
- name: checkout-service
  url: http://host.docker.internal:9054
  routes:
    - name: checkout-route
      paths:
        - /checkout

- name: email-service
  url: http://host.docker.internal:9055
  routes:
    - name: email-route
      paths:
        - /email

- name: payment-service
  url: http://host.docker.internal:9056
  routes:
    - name: payment-route
      paths:
        - /payment

- name: cart-service
  url: http://host.docker.internal:9057
  routes:
    - name: cart-route
      paths:
        - /cart
```

Code Snippet 12 - *kong.yml* file

An important point on Code Snippet 12 is that the URL *http://host.docker.internal:[PORT] is used.* The host machine where Docker is running has a changing IP address for networking reasons. For this reason, Docker offers a special DNS name - *host.docker.internal*, which resolves to the internal IP address used by the host machine. This does not work in a distributed environment where different host machines are used (in that case the URL of the machine that has the Docker container needs to be used). In order to develop the project and simulate a production environment, this can be done using Docker Desktop software [57].

5.1.2 Containerization and deployment

Containerization is a standard approach for microservices-based systems as detailed in section 2.1. The solution is implemented using containerization to mock the production environment of such a system. The deployment logic is divided into layers – the layer that interacts directly with the actors

of the system, the layer of the microservices themselves, and the layer of all the external dependency tools needed to make the system work.

Deployment architecture is different for the two solutions. The Jolie-based solution deployment diagram as shown in Figure 12 shows that the microservices communicate with each other directly and, as shown in Figure 13 the Spring Boot-based solution uses Kafka which itself requires Zookeeper as a service discovery mechanism.

5.1.2.1 Jolie solution

The Jolie solution deployment diagram contains several containers holding a database for each microservice, containing several microservice containers each one representing a microservice, and exposes an API to the API gateway container that is the only container exposed to the clients of the system. Also, in Figure 12 it is possible to see that in Jolie a microservice exposes APIs to other microservices via interfaces, not requiring external tools for interservice communication.

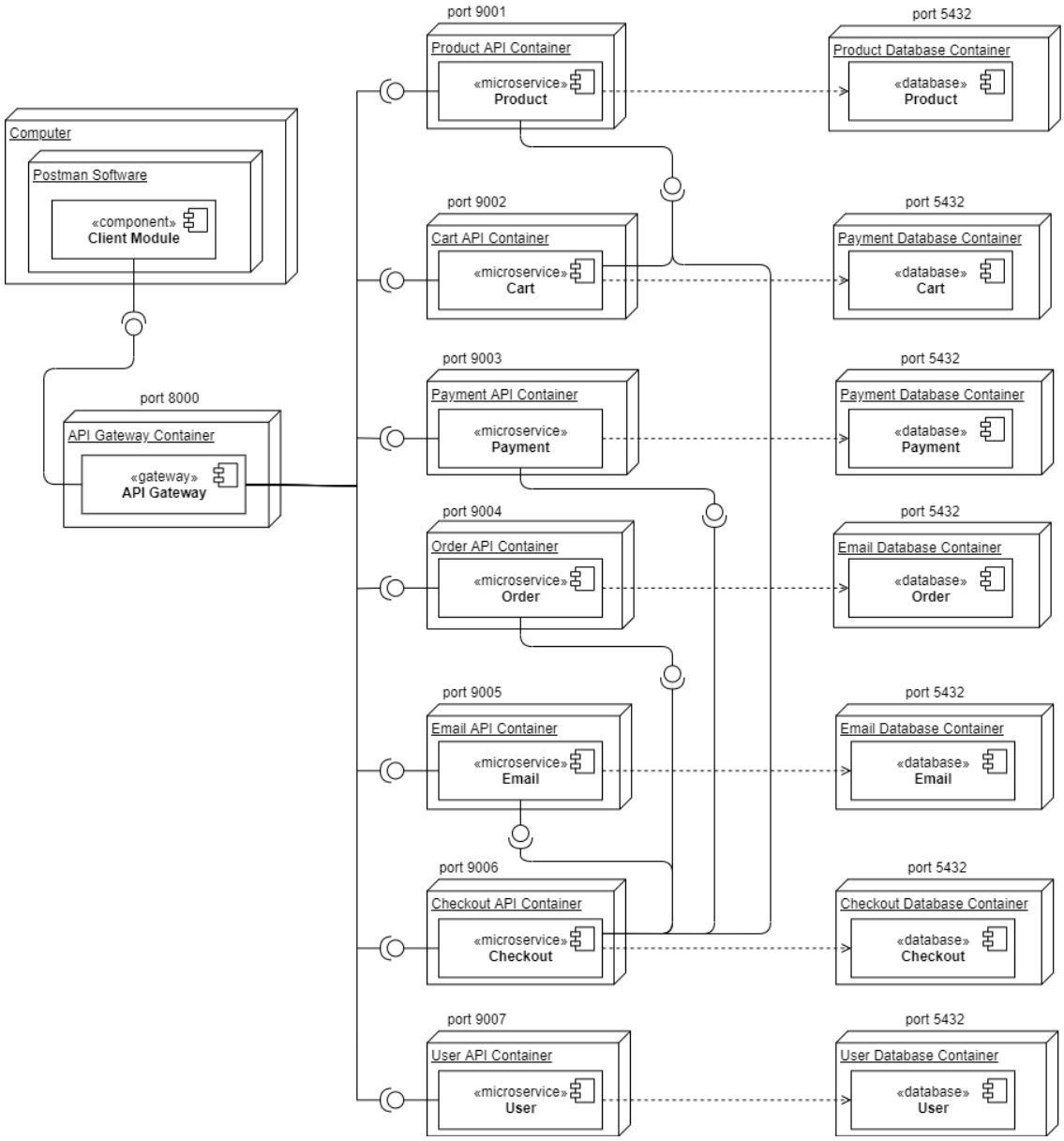

Figure 12 - Deployment diagram for the Jolie solution

5.1.2.2 Spring Boot solution

The Spring Boot solution deployment diagram contains several containers, holding a database for each microservice. It contains several microservice containers, each one representing a microservice and exposing an API to the API gateway container. The API gateway container is the only container exposed to the clients of the system. Also, in Figure 13 it is possible to see that in Spring Boot, microservices requires Apache Kafka for orchestration saga and interservice communication between the microservices.

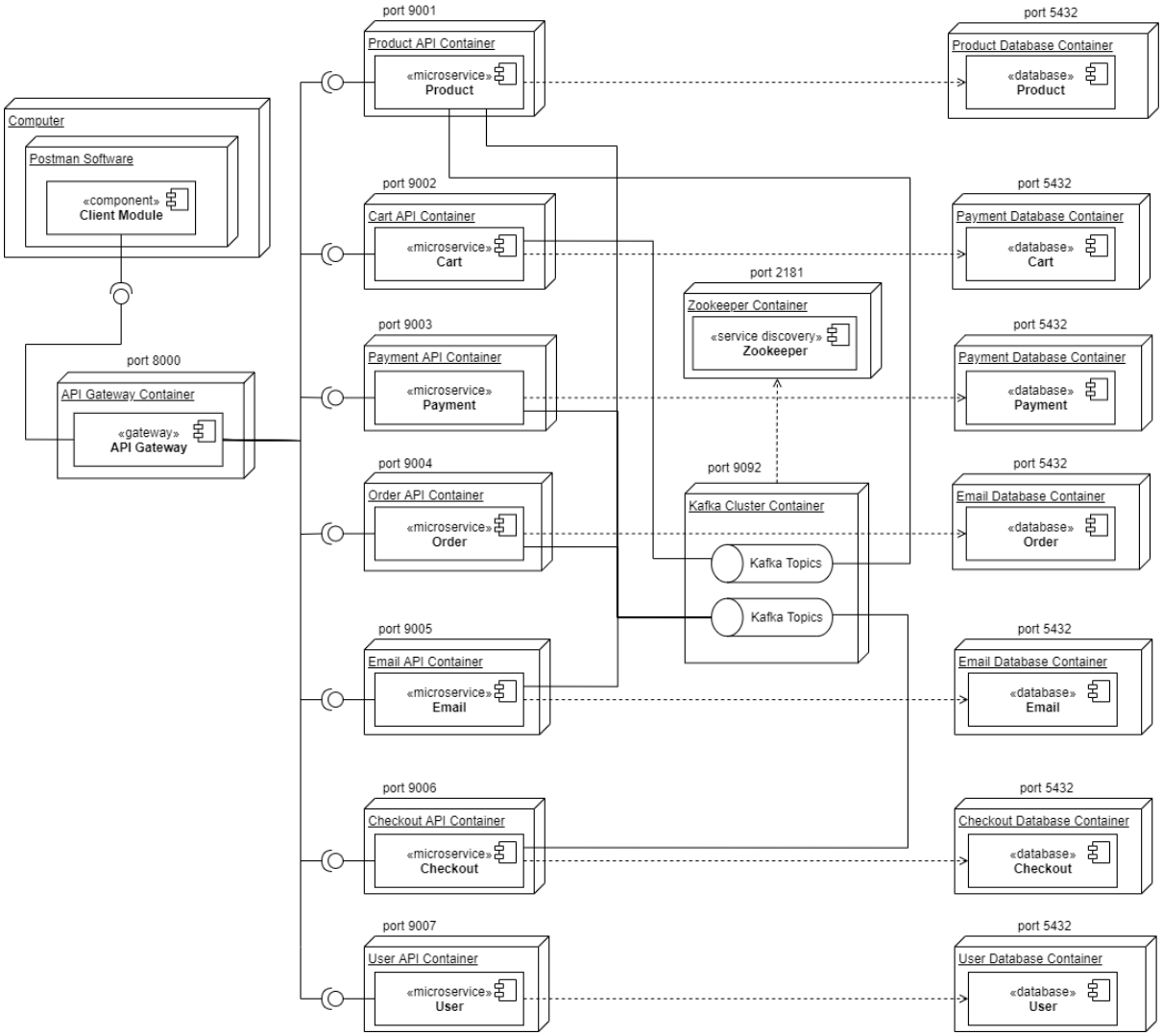

Figure 13 - Deployment diagram for Spring Boot solution

Essentially both deployment diagrams can be divided into four logical layers: the application management that consists of the API gateway, and the Zookeeper service discovery tool (in the Spring Boot solution); the microservices layer that consists of all the microservices; the database layer that contains all the databases; and the orchestration layer that uses Apache Kafka in the Spring Boot-based solution and native provision for the Jolie-based one.

5.1.2.3 Dockerfile and Compose

Dockerfile is a set of instructions that allow Docker to build images automatically [58] and Compose is a tool for specifying multi-container application's services, it allows with a single command the creation and starting of all the services defined in the configuration written in a YAML file [59]. Both these approaches are used in the application since in some cases Compose is better that Dockerfile for dependency management of services, e.g., Kafka's dependency on Zookeeper.

The Compose file used for Apache Kafka can be checked on the Code Snippet 13, here two containers are going to be created, based on the official images of Zookeeper and Kafka. The string "*depends_on*" states that the Kafka container depends on the Zookeeper one. The Kafka container exposes port 9092, which is the port the other containers will use to communicate with this one, and Zookeeper container exposes port 2181 to the outside, in this case to be used by the Kafka container.

```
version: '3'
services:
  zookeeper:
    image: 'bitnami/zookeeper:latest'
    ports:
      - '2181:2181'
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes
  kafka:
    image: 'bitnami/kafka:latest'
    ports:
      - '9092:9092'
    environment:
      - KAFKA_BROKER_ID=1
      - KAFKA_LISTENERS=PLAINTEXT://:9092
      - KAFKA_ADVERTISED_LISTENERS=PLAINTEXT://127.0.0.1:9092
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes
    depends_on:
      - zookeeper
```

Code Snippet 13 - *docker-compose.yml* file for Apache Kafka and Zookeeper containers

For the Jolie microservices *Dockerfile* as one can see on the Code Snippet 14 the image is based on the official Jolie image on Docker Hub. In addition, the code will download the jar necessary for the PostgreSQL database, then port 9001 is exposed for the Product service, and for other services instead of the 9001, other ports are used. For simplicity and non-replication, the Product service is only one used as example. Another important point is that when the container launches *ProductService.ol* (file that contains the behavior of the microservice itself) is run. For other microservices not only has the port to be changed but also the name of the microservice pointing to the file name of that microservice behavior.

```
FROM jolielang/jolie

# Set the working directory to /app
WORKDIR /app
```

```
# Copy the current directory contents into the container at /app
ADD . /app

# Install any needed packages (PostgreSQL JDBC)
RUN wget https://jdbc.postgresql.org/download/postgresql-42.4.0.jar
RUN mv postgresql-42.4.0.jar jdbc-postgresql.jar
RUN cp jdbc-postgresql.jar /usr/lib/jolie/lib/jdbc-postgresql.jar

# Make port 9001 available to the world outside this container
EXPOSE 9001

# Run ProductService.ol when the container launches
CMD jolie ./ProductService.ol
```

Code Snippet 14 - *Dockerfile* for a Jolie microservice

For Spring Boot microservices the Dockerfile can be seen on the Code Snippet 15 where it will be based on the official Open JDK image on Docker Hub. Then it will copy the jar resultant of the project build and place it on the container running as the entry point, which means that when the container starts running, the jar file is run using Java.

```
FROM openjdk:11

VOLUME /tmp

ARG JAR_FILE=./target/ProductService.jar

COPY ${JAR_FILE} ProductService.jar

ENTRYPOINT ["java","-jar","/ProductService.jar"]
```

Code Snippet 15 - *Dockerfile* for a Spring Boot microservice

5.2 Jolie and Spring-based project distinction

It is relevant to know how to distinguish projects that are made using Jolie versus projects using Spring Boot as the main technology.

In Jolie, it is considered a good practice to write every microservice using two files: a *.iol* and a *.ol* file – the first is the interface file, and the second is the implementation file. Each microservice only needs these two files to run on a system with Jolie installed.

Spring Boot projects require more files, most of the time 8 files minimum: one *pom.xml* file for the build tool and dependency management; one *application.properties* file where all the project properties like ports, locations, database drivers and database connection information; and six Java

files that together make the bulk of the program. Spring Boot being a Java framework normally requires the usage of a build automation tool like Apache Maven [60] or Gradle [61], and this makes it one of the factors to immediately identify if a project is built using Spring Boot.

To better understand these differences one can, look at the file structure of the *ProductService* implementation in Jolie on Figure 14 and on Spring Boot on Figure 15.

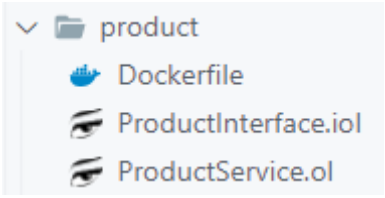

Figure 14 - Product service file structure in Jolie implementation

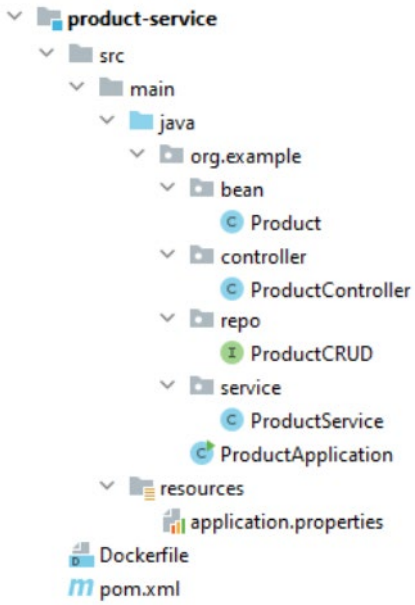

Figure 15 - Product service file structure in Spring Boot framework implementation

The file structure shows a great difference in terms of organization and nomenclature. Jolie only divides its logic of files into interfaces and behavior/implementation, and Spring Boot has a division on the same logic – resources and behaviors/implementation, on the first are stated constants like ports, drivers, database connections, etc, and in the second one are the controllers, entities, repositories, and services divided in a specific logic. The controller can be seen as the surface and the services as a deep implementation that integrates with the JPA repository defined in the repository folder, the entities folder works as a DAO for the entity in the question of the specific microservice. As showed in Figure 15 is the product entity, this entity mirrors the data object of the object stored

in the microservice respective database. This way of developing in Spring Boot is a typically used layout for Spring Boot projects, the framework does not require any specific code layout to work but these best practices of structuring the code can help developers' productivity and code universal readability [62].

5.3 Microservices implementation

As referred to in previous sections the experiment microservices solution is implemented using Spring Boot framework and Jolie programming language creating two separate solutions that are explained in this section.

5.3.1 High level view

Both solutions have the same microservices and API paths communicating via HTTP. In Table 5 are displayed in a more structured way all the paths and to which microservice they belong.

Table 5 - High level view of the microservices implementation

<table>
<tr><th>Topic</th><th>Rationale (why)</th></tr>
<tr><td>Services</td><td>• Product service
• Cart service
• Order service
• Payment service
• User service
• Email service
• Checkout service</td></tr>
<tr><td>Assignment of operations to services</td><td>
<table>
<tr><td>Product service</td><td>/all
/product
/create
/update
/delete</td></tr>
<tr><td>Cart service</td><td>/all
/cart
/create
/update
/delete
/addProduct
/removeProduct</td></tr>
<tr><td>Order service</td><td>/all
/order
/create
/update
/delete</td></tr>
<tr><td>Email service</td><td>/sendEmail</td></tr>
<tr><td>Payment service</td><td>/withdrawlAccount</td></tr>
<tr><td>User service</td><td>/all
/user
/create
/update
/delete</td></tr>
<tr><td>Checkout service</td><td>/pay</td></tr>
</table>
</td></tr>
<tr><td>Communication technologies</td><td>• HTTP
• JDBC driver for PostgreSQL</td></tr>
</table>

5.3.2 Jolie

In Jolie, a microservice implementation begins by creating the interfaces. The behavior is the last step in writing a microservice since the interfaces are APIs that will be consumed by clients and other microservices.

Code Snippet 16 showcases that it is possible to understand how to interact and use the Product service just by reading the interface. First, the types are defined to be used as the request body or the response body. Then interfaces are created using those types when needed, here the methods are defined, and they all are *RequestResponse* type meaning that a response will always be sent back to the client. Finally, the constants represent persistent data relative to the location of the service on the network as well as the database location and respective connection information. It also contains the code for database table(s) creation.

```
type Product {
    .id: string
    .description: string
    .product: string
    .type: string
    .price: double
}

type Products {
    .products[1, *]: Product // an array of Products
}

type CreateRequest {
    .description: string
    .product: string
    .type: string
    .price: double
}

type DeleteRequest {
    .id: string
}


interface ProductInterface {
    RequestResponse:
        all(void)(Products),
        product(undefined)(undefined),
        create(CreateRequest)(Product),
        update(Product)(Product),
        delete(DeleteRequest)(string)
```

```
}

constants {
    LOCATION_SERVICE_PRODUCT = "socket://host.docker.internal:9051",

    SQL_USERNAME = "postgres",
    SQL_PASSWORD = "welcome1",
    SQL_HOST = "host.docker.internal",
    SQL_DATABASE = "postgres",
    SQL_DRIVER = "postgresql",

    SQL_CREATE_TABLE_PRODUCT = "CREATE TABLE product (
                                    id UUID,
                                    description VARCHAR(50),
                                    product VARCHAR(50),
                                    price DECIMAL(20,2),
                                    type VARCHAR(50),
                                    CONSTRAINT product_pkey PRIMARY KEY (id)
                                );

                                ALTER TABLE IF EXISTS public.product
                                    OWNER to postgres;

                                COMMENT ON TABLE public.product IS 'Table that holds
all the products of the application.';",
}
```

Code Snippet 16 – Snippet of Product service interface

All the microservices in Jolie have their respective interface. Then on the microservice behavior *.ol* file is imported the interface and defined as *inputPort* (see Code Snippet 17). This specifies that the Product service will have a port opened to client requests that need to use the *ProductInterface* contract.

```
include "console.iol"
include "database.iol"
include "string_utils.iol"
include "time.iol"

include "./ProductInterface.iol"

execution { concurrent }

// deployment info
inputPort ProductPort {
    Location: LOCATION_SERVICE_PRODUCT
    Protocol: http { .format = "json" }
```

```
    Interfaces: ProductInterface
}
```

Code Snippet 17 - Product service behavior snippet

5.3.2.1 Interservice communication

In Jolie, interservice communication happens via simple API calls as explained in section 4.4.3. Therefore, for example, the cart service requires communication with the product service to retrieve the total price of a product present in the cart. To accomplish this on the behavior of the operation that adds a product to the cart an API call (see Code Snippet 18) needs to be added.

```
requestProduct.id = request.productId
product@ProductService( requestProduct )( responseProduct )
```

Code Snippet 18 – Calling product service 'product' operation from cart service

The call on Code Snippet 18 will return *responseProduct*. This returns the same data as if the 'product' operation was called via a different client via URL. So, e.g., to use the product price, one would write in the code *responseProduct.price*.

To call 'product' operation from 'ProductService' as showed on Code Snippet 18, the product interface needs to be imported to the Cart service and defined as an output port (see in Code Snippet 19).

```
include "../product-service/ProductInterface.iol"

. . .

outputPort ProductService {
    Location: LOCATION_SERVICE_PRODUCT
    Protocol: http { .format = "json" }
    Interfaces: ProductInterface
}
```

Code Snippet 19 - Product interface import on Cart service

5.3.3 Spring Boot

In the Spring Boot solution, the microservices implementation starts by creating a Maven project, this includes the creation of a *pom.xml* file and the standard directory structure for such projects [63]. One could also use Gradle instead of Maven as the build automation tool, the reason Maven is used is due to the experience of the developer with the tool. The *Dockerfile* for a Spring Boot

microservice is explained in section 5.1.2.2 and placed in the same directory as the *pom.xml* file. The initial structure of each microservice project is displayed in Figure 16.

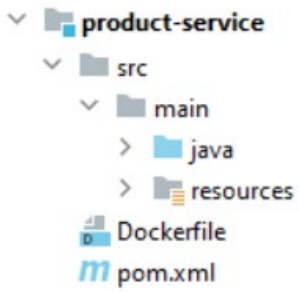

Figure 16 - Initial Spring Boot microservice project structure

On the resources folder, one adds an *application.properties* file. This file is responsible for configuring some aspects of the Spring Boot framework. It also allows the developers to define custom properties of their own, working as a built-in mechanism for application configuration. The initial setup for the *application.properties* is present on snippet Code Snippet 20.

```
# APP CONFIG
server.port:9001


# POSTGRES CONFIG
spring.datasource.url=jdbc:postgresql://host.docker.internal:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=welcome1
spring.jpa.generate-ddl=true
spring.jpa.hibernate.show-sql=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQL81Dialect
```

Code Snippet 20 – application.properties initial content

The *server.port* contains the port on which the microservice API exposes itself to its clients. Database connection URL, username, and password are settled on properties starting with the prefix *spring.datasource*. The properties with the prefix *spring.jpa* are the properties defined for Spring Data JPA [64] commonly used on Spring-powered applications. This technology makes it easier to build applications that use data access technologies. In sum, it supports the database connection and configuration providing a built-in system used by the framework to perform database queries and operations. These properties are what allow this project auto-create database tables, as well as reading and writing operations on the database.

The *main* folder contains the basic Spring Boot project structure and follows the same structure explained in 5.2. Each microservice was implemented with the previously described steps, also, it was first developed all requirements that do not involve interservice communication.

5.3.3.1 Interservice communication

When the bulk of the microservice development is concluded and requirements can only be satisfied by using interservice communication between microservices, then Kafka needs to be added to the code of the microservices that need to talk between them.

To establish communication between microservices Kafka relies on four concepts – messages, topics, producers, and consumers. A message is some data, e.g., simple strings or JSON objects. The producer is responsible for producing messages on a specific topic. The consumer is responsible to read the messages that the producer puts on a topic. A topic is a group of categorized messages. What happens when interservice communication is established is that the producer produces a message that is annexed to a categorized topic, and the consumer receives that message followed by the execution of some code.

On each microservice project that requires Kafka a package named *kafka* is added for ease of understanding. Also, some properties on the *application.properties* file are added (see Code Snippet 21). A property for the Spring application to know the location of the Kafka server and another property to disable the auto-creation of topics in Kafka.

```
# KAFKA PROPERTIES
spring.kafka.bootstrap-servers=host.docker.internal:9092
auto.create.topics.enable=false
```

Code Snippet 21 – Kafka properties on *application.properties* file

Regarding the package, this one contains child packages two of them which are used on every microservice that communicates with other(s), and another one that is used on those microservices that need to listen to Kafka topics for new messages.

The package named *bean* contains the objects that are passed from a microservice to the Kafka topic, and the ones that come from the topic to the microservice. The parsing of the objects before being sent to any topic is made using Gson [65] - an open-source Java library used to serialize and deserialize Java objects into JSON. For example, the product service receives a request from the cart service for the total price of a product that is present on the cart and answers it back, therefore it needs two beans, one for the request (see Code Snippet 22) that needs a product id and the quantity and another one for the response (see Code Snippet 23) that responds with the total price.

```java
package org.example.kafka.bean;

import lombok.*;

@Getter
@Setter
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class RequestProductPrice {

    private String id;
    private Integer quantity;

}
```

Code Snippet 22 - Product price topic request bean

```java
package org.example.kafka.bean;

import lombok.*;

@Getter
@Setter
@Data
@Builder
@NoArgsConstructor
@AllArgsConstructor
public class ReplyProductPrice {

    private Double totalPrice;

}
```

Code Snippet 23 - Product price topic response bean

The package named *config* contains the Kafka configurations including base Kafka configuration and the configuration of the topics. This package has a file – *KafkaConfig.java*, that contains producer and consumer configuration, templates for the messages to be sent and received, and creation and configuration of the topics on which these messages are passing.

The last package named *listener* is a package that contains the topic listener, a method that is listening to a topic for new messages. In this case, all methods that are listening also reply to another topic (see Code Snippet 24) using the *@SendTo* annotation on the method. The *@KafkaListener* defines the topic on which this method is listening to.

```
@Slf4j
@Component
public class KafkaListenerTopic {

    @Autowired
    ProductCRUD productCRUD;

    @KafkaListener(id = "server", topics = "cart-request")
    @SendTo
    public String listenAndReply(String message) {
        log.info("Received message: " + message);

        RequestProductPrice topicRequest = new Gson().fromJson(message,
RequestProductPrice.class);

        UUID productId = UUID.fromString(topicRequest.getId());
        Integer productQuantity = topicRequest.getQuantity();

        Product productDB = productCRUD.findById(productId).get();
        Double productSingleUnitPrice = productDB.getPrice();

        Double productPrice = productSingleUnitPrice * productQuantity;

        ReplyProductPrice topicResponse =
                ReplyProductPrice
                        .builder()
                        .cartTotalPrice(productPrice)
                        .build();

        log.info("Sending message: " + topicResponse.toString());

        System.out.println("Sending message: " + topicResponse.toString());

        return new Gson().toJson(topicResponse);
    }

}
```

Code Snippet 24 – Listener of cart-request Kafka topic on product service

5.4 Solution verification and tests

To automate the verification of the solution Postman Automated API Testing feature [66] was used. Postman software allows for the creation of a test suite of multiple types of tests that can run again and again. These tests will be run against Docker containers pointing to the Kong API Gateway.

To start the process a Postman workspace is created which provides a way to organize the API work by organizing the tests into collections, each collection can have a set of requests with tests that can be run. Figure 17 displays an example of the cart service collection flow covering all possible resources and success scenarios on the cart service. Since the cart service has interservice communication with product service this entire collection can also provide feedback on the

communication between the services. The same is done for all the other collections and microservices.

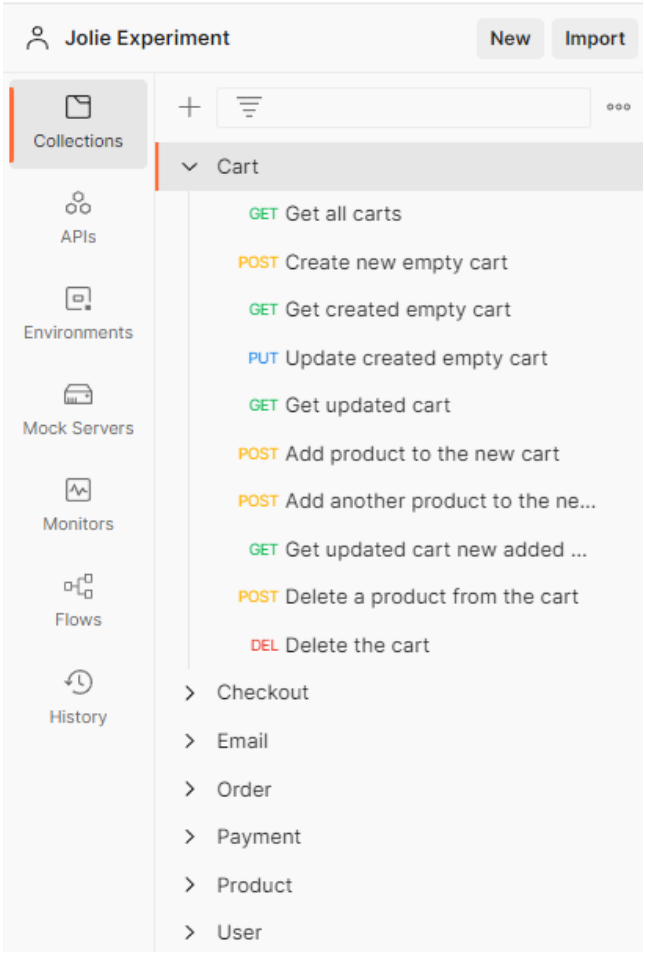

Figure 17 – Cart collection flow tests

Figure 18 shows the Cart collection run results. It is a reliable automatic way to verify both service basic requirements and the requirements that integrate multiple microservices.

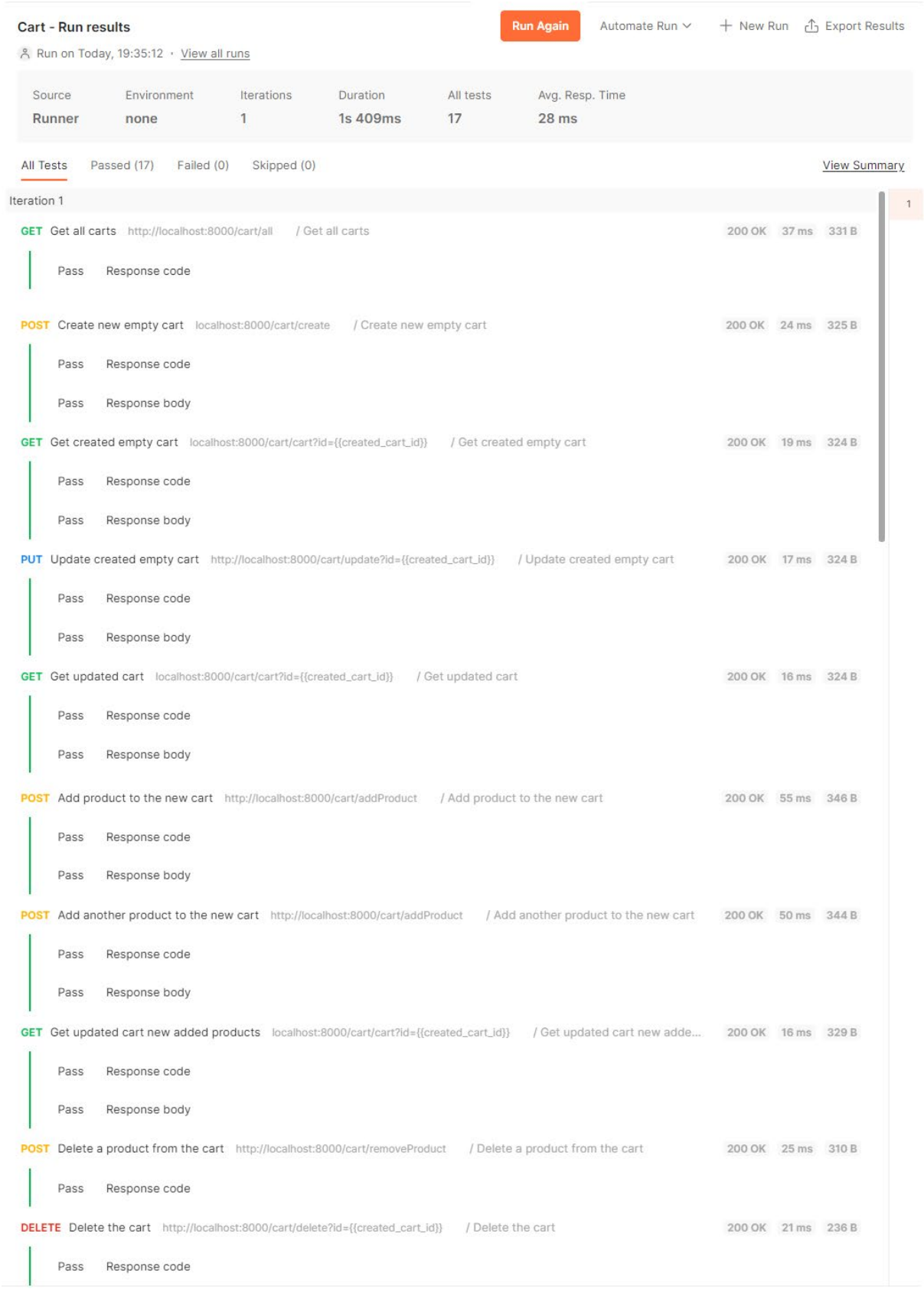

Figure 18 - Cart collection run results

The same process is made on all the other collections providing a mechanism to test the entire flow of a microservice.

6 Evaluation

In this section, the solutions are evaluated according to the most relevant quality attributes in microservices architecture defined in Section 2.2: Scalability, Performance, Maintainability, and Testability. First, the Goals, Questions, and Metrics (GQM) approach are explained and analyzed how to use it in these solutions. Then the development of the experiments to perform is explained. Finally, the conclusions of each experiment using GQM are taken.

6.1 Design

The evaluation of both solutions used to develop the experiment is based on the Goal, Question, Metrics (GQM) paradigm [67], this approach is a method for driving goal-oriented measurements. One starts by defining the goals to achieve, then clarifying the questions to answer with the data collected, and lastly performs mapping between business outcomes and goals to data-driven metrics. By doing this one can obtain a great overview of the entire span of the software [68].

The result of the implementation of the GQM approach is the specification of a measurement system targeting a particular set of quality attributes and a set of rules for the interpretation of the measurement data. The resulting measurement model has three levels [68]:

- Conceptual level (goal): Define a goal for each quality attribute presented, all having in account both solutions developed
- Operational level (question): Define how the goals can be achieved using some questions

- Quantitative level (metric): Define the metrics that reply to the questions and determine if the goals are achieved

First, the goals and questions are defined for each quality attribute (see Table 6).

Table 6 - Quality attributes, goals, and questions

Quality Attribute	Goals	Questions
Scalability	The experiment solutions should be easily scalable.	Can the experiment solutions scale by having low distribution of dependencies, and can the system scale horizontally and vertically?
Maintainability	The experiment solutions must have a low number of lines of code and low technology heterogeneity.	Can the experiment solutions be maintainable easily?
Performance	The experiment solutions work under heavy loads and the response times are acceptable.	The developed microservices have decent response times when running under heavy loads?
Testability	The experiment solutions provide testing features.	Does the experiment solutions provide a way to execute automatic tests and generation of API documentation?

The metrics for each quality attribute are in a generalized way defined on Section 2.2. However, for these solutions in specific, one needs to be more particular in the way to establish the metrics.

6.2 Experiments

To understand each quality attribute for both solutions built the experiments are carried out separately on each quality attribute for both solutions.

6.2.1 Scalability

Regarding scalability, the following components need to be addressed as referred on section 2.2.1 - distribution of dependencies, diversity of synchronous requests, and horizontal and vertical scalability. Both solutions, Jolie-based, and Spring Boot-based ones, need to comply with the scalability components and requirements to be considered scalable.

Distribution of dependencies is the measurement of the percentage of microservices operations that call other microservices. The lower the total percentage more scalable the entire system.

Table 7 - Distribution of dependencies on the microservices of the system

Microservice	Operations number	Calls to other microservices
Cart	7	2
Checkout	1	4
Email	1	0
Order	5	0
Payment	1	0
Product	5	0
User	5	0

From the analysis of Table 7, the total amount of operations is 25 and the calls to other microservices are 6 making it 4.17% of the operations calling other microservices, which is a low number, meaning the system is scalable in this regard.

Both solutions can use horizontal and vertical scalability, the first being referring to adding more resources to logical units like servers of a cluster and the second being adding more physical unit resources like memory to a machine [20].

6.2.2 Maintainability

Measurements of maintainability are done using lines of code (LOC) as previously mentioned in section 2.2.3 and looking at the technology heterogeneity. To accomplish the LOC measurement the tool cloc [69] was executed on the folder of each microservice achieving the results presented in Table 8.

Table 8 - LOC metrics for the solution

Microservice	LOC in Spring Boot based project	LOC in Jolie based project
Cart	638	231

Checkout	831	143
Email	423	66
Order	810	204
Payment	485	96
Product	695	160
User	429	169
Total LOC	**4311**	**1069**

Section 2.2.3 states that the lower the technology heterogeneity the more maintainable the system is. From the analysis of both solutions' technology stack (see section 4.5) it is possible to conclude that the Jolie-based solution is less heterogeny, therefore more maintainable in this regard.

Also, from the analysis of Table 8 Jolie-based solution has around 75.2% less LOC than the Spring Boot-based solution. It's considered that the Jolie-based solution has better maintainability since the LOC measurement and the technology heterogeneity are lower than the Spring Boot-based one.

6.2.3 Performance

For performance metrics, the tool used to automate the process is Apache JMeter [70]. The scenarios were adapted to the developer machine resources and are tested for the cart service flow and checkout service flow since these are the two microservices that are saga orchestrators and contain interservice communication with other microservices.

The testing plan scenarios are the following for both flows:

- 15 virtual users performing the test plan 10 times
- 150 virtual users performing the test plan 10 times
- 1500 virtual users performing the test plan 10 times

Both solutions are tested by running the microservices, API gateway, and databases with Docker containers. The Docker host is present on the developer's machine.

6.2.3.1 Cart service flow

The cart service flow can be seen on Apache JMeter graphical user interface (GUI) as shown in Figure 19 – every used executes all the HTTP requests in the *Thread Group* several times defined in the configuration.

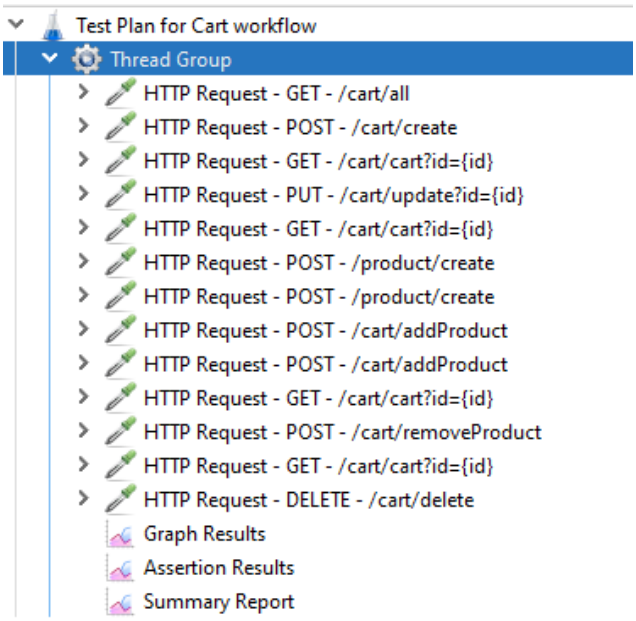

Figure 19 - Apache JMeter test plan for cart service flow

Table 9 shows the performance metrics like throughput, which consists of the number of requests handled per second, of each scenario of testing – 15, 150, or 1500 virtual users performing the cart service flow for 10 times each user.

Table 9 – Performance table report for cart service flow

Solution	Number of users	Throughput (requests/s)	Deviation	Time elapsed (hh:mm:ss)
Jolie based	15	100.9	29	00:00:19
Jolie based	150	98.2	333	00:03:18
Jolie based	1500	93.4	4028	00:34:48
Spring Boot based	15	101.2	31	00:00:19
Spring Boot based	150	119.2	91	00:02:43
Spring Boot based	1500	116.1	2092	00:27:59

6.2.3.2 Checkout service flow

The checkout service flow can be seen on Apache JMeter graphical user interface (GUI) as shown in Figure 20 – every used executes all the HTTP requests in the *Thread Group* several times defined in the configuration.

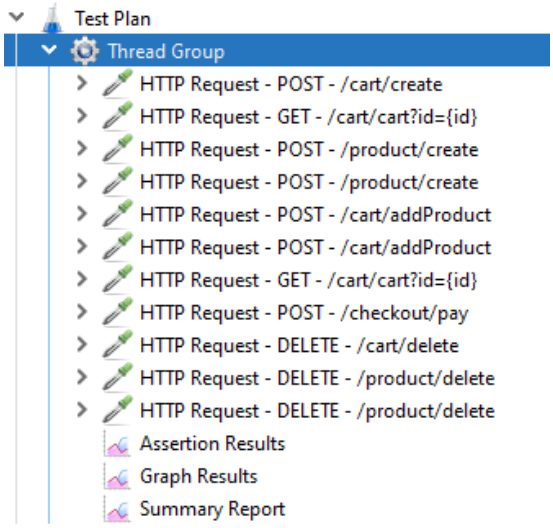

Figure 20 - Apache JMeter test plan for checkout service flow

Table 10 shows the performance metrics like throughput, which consists of the number of requests handled per second, of each scenario of testing – 15, 150, or 1500 virtual users performing the checkout service flow for 10 times each user.

Table 10 - Performance table report for checkout service flow

Solution	Number of users	Throughput (requests/s)	Deviation	Time elapsed (hh:mm:ss)
Jolie based	15	98.2	25	00:00:16
Jolie based	150	109.3	21	00:02:30
Jolie based	1500	79.6	12703	00:34:33
Spring Boot based	15	100.9	19	00:00:16
Spring Boot based	150	107.6	50	00:02:33
Spring Boot based	1500	109.8	278	00:25:02

6.2.4 Testability

For testability, in terms of API documentation Spring Boot supports the automatic generation of OpenAPI documentation with the usage of annotations [71]. OpenAPI is the standardization of how

APIs are described [72], Spring Boot allows the usage of the Swagger plugin to create an HTML page to explore the API. Jolie programming language on the other hand comes with Joliedoc [73] - a tool that comes with the installation of Jolie programming language and can automatically generate documentation in the format of an HTML page. Even though the two technologies (Jolie and Spring Boot) use different tools to achieve the automatic generation of documentation, they both have testability features.

For testing the application in unit, integration, and acceptance tests Spring Boot has unit tests and integration tests provision using Java that can be easily automated. Jolie because of the nature of the language the tests are done using a tool called Joliemock [74]. This tool also comes native with the Jolie language installation. It is used since, in Jolie a service interface is firstly defined providing an automatic way to mock services starting from an input port. Also, both languages use Postman to create collections to automatically execute the entire bulk of microservices built.

6.3 Summary

The evaluation results are overall acceptable, and one can conclude that both technologies (Jolie and Spring Boot) are viable to comply with quality attributes seen as most important in MSA systems.

Regarding similarities both solutions provide similar scalability and testability. Both technologies (Jolie and Spring Boot) provide native resources for testing and have a low number of calls to other services. Also, scalability and testability can be partially obtained using external tools (Docker and Postman), this can make one can conclude that they are similar in terms of these two quality attributes.

On performance Spring Boot-based solution ranks slightly higher as both cart and checkout flow performed better with a high number of virtual users (1500) performing the entire flow. If a low number of users are performing the flow, there are no significant changes in performance. It is important to note that this performance results could change if more containers (horizontal scalability) were used to multiply the number of instances of each microservice or if machines with better hardware were used, but for this test scenario and resources available only one container per microservice is used on the developer machine and results interpreted as such. As for maintainability, the Jolie-based solution ranks significantly higher as it contains a lower heterogeneity of technologies and a lower LOC compared to the Spring Boot solution.

7 Conclusions

7.1 Achievements

Some objectives were defined at the beginning of this thesis and presented in section 1.2. This thesis provides insights on the current knowledge of MSA-based systems focusing on language-based approaches found in the literature using Jolie. Two experiment solutions are developed with source code available on GitHub [53] and analyzed in this document. The experiment demonstrates how to implement an e-commerce MSA-based system using Jolie programming language face to the same system using Spring Boot framework as the base technology. Also, it was demonstrated how to apply interservice communication, testing, and containerization on both solutions. The focus is to evaluate both solutions based on some quality attributes – scalability, maintainability, performance, and testability.

The output of this document is an assessment of part of the benefits and possible limitations to pay attention to when using Jolie versus using Spring Boot to build the same solution. The document is also a solid starting point when using Jolie programming language to build an MSA system.

7.2 Difficulties

The development of the thesis brought some challenges mainly because the language-based approach to microservices development is a recent topic and in its only being started to be used by companies and portrayed more in the research community. Therefore, most of the literature and community projects and discussions are around framework solutions for MSA systems.

Jolie is a product of renowned research authors and its presence in the literature is extensible. It is important to praise the authors of the programming language for the effort made to provide great formal insight into the language. Also, the official documentation on the language is user-friendly and understandable being a great source of information. Nevertheless, the development is still hard because whenever some issue is faced by unfamiliarity with the programming language the community answers for the problems are quite inexistent and some issues took the developer a great amount of time to figure out how to overcome them.

7.3 Threads to Validity

Testing was performed in a local environment which brought limitations due to resource limitations and unavailability to make more large tests and experiments. This is important for testing the system at a large scale as using thousands of users. On this work only a maximum of 1500 virtual users were used for performance testing. A real production environment could bring different results. Also, if container orchestration was applied, and the containers multiplied themselves accordingly to the load on the service the performance could have a different result too.

It is important to note that the author is more familiarized with Spring Boot than with Jolie which can also impact the results.

7.4 Future work and final remarks

In the future, it would be interesting to see how container orchestration can impact the performance of both solutions. Also, documentation must be generated for the two solutions.

A final improvement is to compare more frameworks and microservice-specific languages to Jolie to further investigate the difference in quality attributes between more different technologies.

The overall development of the project was positive. It was a satisfaction to the author to discover and use language-based approaches for microservices development, an approach that the author did not know that it existed. It is an area of interest, and it hopefully gets more explored by the author, research community, and businesses.

Is satisfactory to conclude a challenging work where every aspect of an MSA system is developed. The work provided the author with much valuable experience in microservices for future career paths and opportunities.

References

[1] Karma, 'How we build microservices at Karma', *Medium*, Apr. 04, 2016. https://blog.karmawifi.com/how-we-build-microservices-at-karma-71497a89bfb4 (accessed Oct. 28, 2021).

[2] E. Haddad, 'Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow', *Uber Engineering Blog*, Sep. 08, 2015. https://eng.uber.com/service-oriented-architecture/ (accessed Oct. 28, 2021).

[3] 'Amazon Architecture - High Scalability -'. http://highscalability.com/amazon-architecture (accessed Oct. 28, 2021).

[4] P. D. Francesco, I. Malavolta, and P. Lago, 'Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption', in *2017 IEEE International Conference on Software Architecture (ICSA)*, Apr. 2017, pp. 21–30. doi: 10.1109/ICSA.2017.24.

[5] P. Di Francesco, P. Lago, and I. Malavolta, 'Architecting with microservices: A systematic mapping study', *J. Syst. Softw.*, vol. 150, pp. 77–97, Apr. 2019, doi: 10.1016/j.jss.2019.01.001.

[6] *Advantages and Disadvantages of Microservices Architecture*, (Nov. 13, 2019). Accessed: Oct. 31, 2021. [Online Video]. Available: https://cloudacademy.com/blog/microservices-architecture-challenge-advantage-drawback/

[7] 'Microservices definition, advantages and disadvantages', *Chakray*, May 15, 2019. https://www.chakray.com/what-are-microservices-definition-characteristics-and-advantages-and-disadvantages-2/ (accessed Oct. 31, 2021).

[8] F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro, 'JOLIE: a Java Orchestration Language Interpreter Engine', *Electron. Notes Theor. Comput. Sci.*, vol. 181, pp. 19–33, Jun. 2007, doi: 10.1016/j.entcs.2007.01.051.

[9] F. Montesi and J. Weber, 'Circuit Breakers, Discovery, and API Gateways in Microservices', *ArXiv160905830 Cs*, Sep. 2016, Accessed: Dec. 08, 2021. [Online]. Available: http://arxiv.org/abs/1609.05830

[10] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, 'A Design Science Research Methodology for Information Systems Research', *J. Manag. Inf. Syst.*, vol. 24, no. 3, pp. 45–77, Dec. 2007, doi: 10.2753/MIS0742-1222240302.

[11] N. Dragoni *et al.*, 'Microservices: yesterday, today, and tomorrow', *ArXiv160604036 Cs*, Apr. 2017, Accessed: Dec. 05, 2021. [Online]. Available: http://arxiv.org/abs/1606.04036

[12] 'Microservices', *martinfowler.com*. https://martinfowler.com/articles/microservices.html (accessed Feb. 06, 2022).

[13] C. Guidi, I. Lanese, M. Mazzara, and F. Montesi, 'Microservices: a Language-based Approach', *ArXiv170408073 Cs*, Apr. 2017, Accessed: Oct. 26, 2021. [Online]. Available: http://arxiv.org/abs/1704.08073

[14] N. Dragoni, I. Lanese, S. Larsen, M. Mazzara, R. Mustafin, and L. Safina, *Microservices: How To Make Your Application Scale*. 2017.

[15] H. Zhu and I. Bayley, 'If Docker is the Answer, What is the Question?', in *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, Mar. 2018, pp. 152–163. doi: 10.1109/SOSE.2018.00027.

[16] S. Newman, *Building Microservices, 2nd Edition*, 2nd ed. O'Reilly Media, Inc., 2021. Accessed: Aug. 30, 2022. [Online]. Available: https://learning.oreilly.com/library/view/building-microservices-2nd/9781492034018/

[17] A. Megargel, C. M. Poskitt, and V. Shankararaman, 'Microservices Orchestration vs. Choreography: A Decision Framework', in *2021 IEEE 25th International Enterprise Distributed Object Computing Conference (EDOC)*, Oct. 2021, pp. 134–141. doi: 10.1109/EDOC52215.2021.00024.

[18] F. B. Vernadat, 'Reengineering the Organization with a Service Orientation', in *Service Enterprise Integration: An Enterprise Engineering Perspective*, C. Hsu, Ed. Boston, MA: Springer US, 2007, pp. 77–101. doi: 10.1007/978-0-387-46364-3_3.
[19] E. Casalicchio and S. Iannucci, 'The state-of-the-art in container technologies: Application, orchestration and security', *Concurr. Comput. Pract. Exp.*, vol. 32, no. 17, p. e5668, 2020, doi: 10.1002/cpe.5668.
[20] S. Li *et al.*, 'Understanding and addressing quality attributes of microservices architecture: A Systematic literature review', *Inf. Softw. Technol.*, vol. 131, p. 106449, Mar. 2021, doi: 10.1016/j.infsof.2020.106449.
[21] J.-P. Gouigoux, D. Tamzalit, and J. Noppen, 'Microservice Maturity of Organizations: towards an assessment framework', *ArXiv210514935 Cs*, vol. 415, pp. 523–540, 2021, doi: 10.1007/978-3-030-75018-3_34.
[22] L. Chen, M. Ali Babar, and B. Nuseibeh, 'Characterizing Architecturally Significant Requirements', *IEEE Softw.*, vol. 30, no. 2, pp. 38–45, Mar. 2013, doi: 10.1109/MS.2012.174.
[23] M.-D. Cojocaru, A. Uta, and A.-M. Oprescu, 'Attributes Assessing the Quality of Microservices Automatically Decomposed from Monolithic Applications', in *2019 18th International Symposium on Parallel and Distributed Computing (ISPDC)*, Jun. 2019, pp. 84–93. doi: 10.1109/ISPDC.2019.00021.
[24] F. H. Vera-Rivera, C. Gaona, and H. Astudillo, 'Defining and measuring microservice granularity—a literature overview', *PeerJ Comput. Sci.*, vol. 7, p. e695, Sep. 2021, doi: 10.7717/peerj-cs.695.
[25] Y. Tashtoush, M. Al-Maolegi, and B. Arkok, 'The Correlation among Software Complexity Metrics with Case Study', *ArXiv14084523 Cs*, Aug. 2014, Accessed: Apr. 09, 2022. [Online]. Available: http://arxiv.org/abs/1408.4523
[26] 'IEEE Standard Glossary of Software Engineering Terminology', *IEEE Std 61012-1990*, pp. 1–84, Dec. 1990, doi: 10.1109/IEEESTD.1990.101064.
[27] F. Tapia, M. Á. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, 'From Monolithic Systems to Microservices: A Comparative Study of Performance', *Appl. Sci.*, vol. 10, no. 17, Art. no. 17, Jan. 2020, doi: 10.3390/app10175797.
[28] L. Ardito, R. Coppola, L. Barbato, and D. Verga, 'A Tool-Based Perspective on Software Code Maintainability Metrics: A Systematic Literature Review', *Sci. Program.*, vol. 2020, p. e8840389, Aug. 2020, doi: 10.1155/2020/8840389.
[29] S. Giallorenzo, F. Montesi, M. Peressotti, F. Rademacher, and S. Sachweh, 'Jolie & LEMMA: Model-Driven Engineering and Programming Languages Meet on Microservices', *ArXiv210402458 Cs*, Apr. 2021, Accessed: Oct. 26, 2021. [Online]. Available: http://arxiv.org/abs/2104.02458
[30] F. Rademacher, J. Sorgalla, P. N. Wizenty, S. Sachweh, and A. Zündorf, 'Microservice architecture and model-driven development: yet singles, soon married (?)', in *Proceedings of the 19th International Conference on Agile Software Development: Companion*, New York, NY, USA, May 2018, pp. 1–5. doi: 10.1145/3234152.3234193.
[31] A. Bandura, N. Kurilenko, M. Mazzara, V. Rivera, L. Safina, and A. Tchitchigin, *Jolie Community on the Rise*. 2016. doi: 10.1109/SOCA.2016.16.
[32] P. Mohagheghi, M. Fernández, J. Martell, M. Fritzsche, and W. Gilani, *MDE Adoption in Industry: Challenges and Success Criteria*, vol. 5421. 2008, p. 59. doi: 10.1007/978-3-642-01648-6_6.
[33] 'italianaSoftware - Jolie, il linguaggio di programmazione'. http://www.italianasoftware.com/jolie.html (accessed Oct. 26, 2021).
[34] A. Fernando *et al.*, 'Ballerina and Jolie: Connecting Two Frontiers of Microservice Programming', p. 3.
[35] 'Introduction · Jolie Documentation'. https://docs.jolie-lang.org/v1.10.x/ (accessed Dec. 28, 2021).
[36] D. Madushan, *Cloud Native Applications with Ballerina*. Packt Publishing, 2021. [Online]. Available: https://www.oreilly.com/library/view/cloud-native-applications/9781800200630/

[37] I. Lanese, F. Montesi, and G. Zavattaro, 'The Evolution of Jolie: From Orchestrations to Adaptable Choreographies', in *Software, Services, and Systems*, vol. 8950, R. De Nicola and R. Hennicker, Eds. Cham: Springer International Publishing, 2015, pp. 506–521. doi: 10.1007/978-3-319-15545-6_29.
[38] 'In a nutshell: because we want to minimise code <-> model distance. Other langua... | Hacker News'. https://news.ycombinator.com/item?id=27183206 (accessed Oct. 31, 2021).
[39] 'Jolie, the service-oriented programming language'. https://www.jolie-lang.org/faq.html (accessed Dec. 31, 2021).
[40] C. Pautasso and E. Wilde, *Why is the Web Loosely Coupled? A Multi-Faceted Metric for Service Design*. 2009, p. 920. doi: 10.1145/1526709.1526832.
[41] V. Alessandrini, 'Chapter 4 - Thread-Safe Programming', in *Shared Memory Application Programming*, V. Alessandrini, Ed. Boston: Morgan Kaufmann, 2016, pp. 83–99. doi: 10.1016/B978-0-12-803761-4.00004-6.
[42] 'Hystrix: Latency and Fault Tolerance for Distributed Systems'. Netflix, Inc., Dec. 30, 2021. Accessed: Dec. 30, 2021. [Online]. Available: https://github.com/Netflix/Hystrix
[43] 'Microservices Applications' Life Cycle | LinkedIn'. https://www.linkedin.com/pulse/microservices-applications-life-cycle-sam-gabrail/ (accessed Dec. 29, 2021).
[44] S. Giallorenzo, M. Gabbrielli, and F. Montesi, 'Service-Oriented Architectures: from Design to Production exploiting Workflow Patterns', *Adv. Distrib. Comput. Artif. Intell. J.*, vol. 3, no. 2, pp. 26–52, Mar. 2015, doi: 10.14201/ADCAIJ2014322652.
[45] Archiveddocs, 'Data Transfer Object'. https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649585(v=pandp.10) (accessed Dec. 30, 2021).
[46] 'Empowering App Development for Developers | Docker'. https://www.docker.com/ (accessed Dec. 31, 2021).
[47] T. Colanzi *et al.*, 'Are we speaking the industry language? The practice and literature of modernizing legacy systems with microservices', in *15th Brazilian Symposium on Software Components, Architectures, and Reuse*, New York, NY, USA, Sep. 2021, pp. 61–70. doi: 10.1145/3483899.3483904.
[48] 'Apache Kafka', *Apache Kafka*. https://kafka.apache.org/ (accessed Aug. 26, 2022).
[49] 'Postman API Platform | Sign Up for Free', *Postman*. https://www.postman.com/ (accessed Apr. 07, 2022).
[50] 'GoogleCloudPlatform/microservices-demo'. Google Cloud Platform, Feb. 24, 2022. Accessed: Feb. 24, 2022. [Online]. Available: https://github.com/GoogleCloudPlatform/microservices-demo
[51] 'Apache ZooKeeper'. https://zookeeper.apache.org/ (accessed Aug. 30, 2022).
[52] F. Montesi, C. Guidi, and G. Zavattaro, 'Service-Oriented Programming with Jolie', in *Web Services Foundations*, A. Bouguettaya, Q. Z. Sheng, and F. Daniel, Eds. New York, NY: Springer New York, 2014, pp. 81–107. doi: 10.1007/978-1-4614-7518-7_4.
[53] R. Assis, 'jolie-experiment', *GitHub*. https://github.com/ruimiguel98/jolie-case-study (accessed Jul. 17, 2022).
[54] J. Gough, D. Bryant, and M. Auburn, *Mastering API Architecture*. O'Reilly Media, Inc., 2022. Accessed: Aug. 31, 2022. [Online]. Available: https://learning.oreilly.com/library/view/mastering-api-architecture/9781492090625/
[55] D. Taibi, V. Lenarduzzi, and C. Pahl, 'Continuous Architecting With Microservices and DevOps: a Systematic Mapping Study', 2019. doi: 10.1007/978-3-030-29193-8_7.
[56] A. Kuhn, 'Kong Open-Source API Management Gateway for Microservices', *Kong Inc.* https://konghq.com/products/api-gateway-platform (accessed Jul. 17, 2022).
[57] 'Explore networking features', *Docker Documentation*, Aug. 30, 2022. https://docs2.docker.com/desktop/networking/ (accessed Aug. 31, 2022).

[58] ‘Dockerfile reference’, *Docker Documentation*, Aug. 30, 2022. https://docs2.docker.com/engine/reference/builder/ (accessed Aug. 31, 2022).
[59] ‘Overview of Docker Compose’, *Docker Documentation*, Aug. 30, 2022. https://docs2.docker.com/compose/ (accessed Aug. 31, 2022).
[60] ‘Maven – Welcome to Apache Maven’. https://maven.apache.org/ (accessed Jul. 19, 2022).
[61] ‘Gradle Build Tool’, *Gradle*. https://gradle.org/ (accessed Jul. 19, 2022).
[62] ‘14. Structuring Your Code’. https://docs.spring.io/spring-boot/docs/2.0.0.RELEASE/reference/html/using-boot-structuring-your-code.html (accessed Jul. 19, 2022).
[63] ‘Maven – Introduction to the Standard Directory Layout’. https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html (accessed Oct. 02, 2022).
[64] ‘Spring Data JPA’. https://spring.io/projects/spring-data-jpa (accessed Oct. 02, 2022).
[65] ‘Gson’. Google, Oct. 02, 2022. Accessed: Oct. 02, 2022. [Online]. Available: https://github.com/google/gson
[66] ‘Automated API Testing | Postman’, *Postman API Platform*. https://www.postman.com/automated-testing/ (accessed Jul. 19, 2022).
[67] V. R. Basili, ‘Software Modeling and Measurement: The Goal/Question/Metric Paradigm’, Sep. 1992, Accessed: Sep. 28, 2022. [Online]. Available: https://drum.lib.umd.edu/handle/1903/7538
[68] V. R. Basili, G. Caldiera, and H. D. Rombach, ‘THE GOAL QUESTION METRIC APPROACH’, p. 10.
[69] AlDanial, ‘cloc’. Oct. 05, 2022. Accessed: Oct. 05, 2022. [Online]. Available: https://github.com/AlDanial/cloc
[70] ‘Apache JMeter - Apache JMeter™’. https://jmeter.apache.org/ (accessed Oct. 07, 2022).
[71] O. 3 L. for spring-boot B. B. N. LAHSEN and L. for O. 3 with spring-boot B. B. N. LAHSEN, ‘OpenAPI 3 Library for spring-boot’, *OpenAPI 3 Library for spring-boot*. http://springdoc.org/ (accessed Sep. 28, 2022).
[72] ‘OpenAPI - About’, *OpenAPI Initiative*. https://www.openapis.org/about (accessed Sep. 28, 2022).
[73] ‘Documenting APIs · Jolie Documentation’. https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/documenting-api.html (accessed Sep. 28, 2022).
[74] ‘Mock Services · Jolie Documentation’. https://docs.jolie-lang.org/v1.10.x/language-tools-and-standard-library/mock/ (accessed Sep. 28, 2022).
[75] C. Nokleberg and B. Hawkes, ‘Application Frameworks: While powerful, frameworks are not for everyone’, *Commun. ACM*, vol. 64, no. 7, pp. 42–49, Jul. 2021, doi: 10.1145/3446796.
[76] G. Márquez and H. Astudillo, ‘Actual Use of Architectural Patterns in Microservices-Based Open Source Projects’, in *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, Dec. 2018, pp. 31–40. doi: 10.1109/APSEC.2018.00017.
[77] P. Koen *et al.*, ‘1 Fuzzy Front End : Effective Methods , Tools , and Techniques’, 2002. https://www.semanticscholar.org/paper/1-Fuzzy-Front-End-%3A-Effective-Methods-%2C-Tools-%2C-and-Koen-Ajamian/b6731a73075c82622ad9babe296f853fce62bf71 (accessed Feb. 08, 2022).
[78] F. B. Insights, ‘Cloud Computing Market to Reach USD 791.48 Billion by 2028; Microsoft Corporation Launches Cloud for Healthcare Organizations Set to Offer Growth Prospect: Fortune Business Insights™’, *GlobeNewswire News Room*, Jan. 11, 2021. https://www.globenewswire.com/news-release/2021/11/01/2324091/0/en/Cloud-Computing-Market-to-Reach-USD-791-48-Billion-by-2028-Microsoft-Corporation-Launches-Cloud-for-Healthcare-Organizations-Set-to-Offer-Growth-Prospect-Fortune-Business-Insights.html (accessed Feb. 08, 2022).
[79] F. B. Insights, ‘IoT Market Size, Share, Growth, Trends, Business Opportunities, IoT Companies, Statistics, Report 2028 | Internet of Things Industry Report- Fortune Business Insights’, *GlobeNewswire News Room*, Oct. 11, 2021. https://www.globenewswire.com/news-

release/2021/11/10/2331267/0/en/IoT-Market-Size-Share-Growth-Trends-Business-Opportunities-IoT-Companies-Statistics-Report-2028-Internet-of-Things-Industry-Report-Fortune-Business-Insights.html (accessed Feb. 08, 2022).
[80] R. and Markets, 'Global Smartphone Market (2020 to 2027) - by Operating System, Display Technology, Screen Size, RAM Capacity, Price Range, Distribution Channel and Region', *GlobeNewswire News Room*, Sep. 17, 2021. https://www.globenewswire.com/news-release/2021/09/17/2298924/28124/en/Global-Smartphone-Market-2020-to-2027-by-Operating-System-Display-Technology-Screen-Size-RAM-Capacity-Price-Range-Distribution-Channel-and-Region.html (accessed Feb. 08, 2022).
[81] C. Xu, H. Zhu, I. Bayley, D. Lightfoot, M. Green, and P. Marshall, 'CAOPLE: A Programming Language for Microservices SaaS', in *2016 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, Mar. 2016, pp. 34–43. doi: 10.1109/SOSE.2016.46.
[82] D. Jordan *et al.*, 'Web Services Business Process Execution Language', p. 264, 2007.
[83] A. Osterwalder and Y. Pigneur, *Business Model Generation: A Handbook for Visionaries, Game Changers, and Challengers*. John Wiley and Sons, 2010.
[84] 'How SWOT (Strength, Weakness, Opportunity, and Threat) Analysis Works', *Investopedia*. https://www.investopedia.com/terms/s/swot.asp (accessed Feb. 04, 2022).
[85] J. Sorgalla, *AjiL: A Graphical Modeling Language for the Development of Microservice Architectures*. 2017.
[86] 'JHipster - Full Stack Platform for the Modern Developer!' https://www.jhipster.tech/ (accessed Feb. 04, 2022).
[87] 'Welcome', *Context Mapper*, Dec. 17, 2021. https://contextmapper.org/docs/home/ (accessed Feb. 04, 2022).

Annex A – Microservices Development

Frameworks, libraries, and languages

When one talks about the development of microservices it always leads to some framework or library that can help developers in the process.

Libraries are essentially pieces of code written by others that allow developers to reuse code, be consistent in their teams and improve the speed and quality at which a system is coded. Frameworks on the other hand use these libraries too but also configure them and wire everything together to make it even easier for developers to code the system. Frameworks have preinstalled and preconfigured libraries providing a simplified, allowing for more consistency and therefore more simplified and greater development experience which offers large productivity gains. Frameworks avoid developers the hard task of having to choose which are the right libraries to use, the configurations to use, and how to incorporate everything together to make them work flawlessly. Also, frameworks boost security, because as they are universalized and widespread through multiple companies the share of security concerns and fixed multiplies among all teams make systems more secure without developers even realizing it. Another interesting fact that happens with the universalization of frameworks usage is that the structure of frameworks provides a foundation for building higher-level abstractions like microservices platforms allowing new architectures and automation. Not everything is perfect, and frameworks have some disadvantages like the cost of flexibility as the maintainers of such frameworks need to provide standards and defined behaviours without being too restrictive on developers' creativity [75].

Currently, there are plenty of frameworks and libraries for microservice development but not so many programming languages microservice oriented.

Microservice development is still too focused-on frameworks and libraries using other technology stacks like Java, Python, etc because these frameworks have been around for a long time and are mature in the industry already. Programming languages oriented to microservice which have as only focus the area of microservice are still emerging and capturing the attention of developers in an initial phase. Ballerina and Jolie are the two main popular languages at the frontier of microservice programming [34].

Table 11 — List of frameworks and programming languages to develop microservices

Name	Type of technology	Used for
Spring Boot	Framework	Java
Helidon	Framework	Java
AxonIQ	Framework	Java
Micronaut	Framework	Java
Lagom	Framework	Multi-language
Dropwizard	Framework	Java
Restlet	Framework	Java
Quarkus	Framework	Java
GoMicro	Framework	Golang
Kite	Framework	Golang
Django	Framework	Python
Flask	Framework	Python
Falcon	Framework	Python
Bottle	Framework	Python
Nameko	Framework	Python
CherryPy	Framework	Python
Moleculer	Framework	Node.js
.NET	Framework	C#
Spark	Framework	Multi-language
Slim	Framework	PHP
Ballerina	Programming language	—
CAOPLE	Programming language	—
Jolie	Programming language	—
WS-BPEL	Programming language	—

Table 11 shows that there are so many frameworks to choose from when developing microservices that enterprises always face challenges to choose the right one. They need to consider the following factors:

- Speed of development
- Cost efficiency
- Community support
- Developers' skillset
- Learning time
- Future business needs analysis
- Industry acceptance

A recent study showed that Spring Boot dominates the technology adoption for microservices framework choice followed by ASP.NET [47].

Lifecycle phases

In the programming of an MSA system questions like these are raised:

- What the microservice will do?
- What programming language to use?
- Which framework to use?
- Where is data stored?
- How will the microservice scale?
- How it will be deployed?

Each microservice lifecycle is mainly constituted by the build phase where the service is programmed, then the running phase is when the service is running and the system can monitor, secure, bug fix, or change update the source code with a new version in case the business logic change.

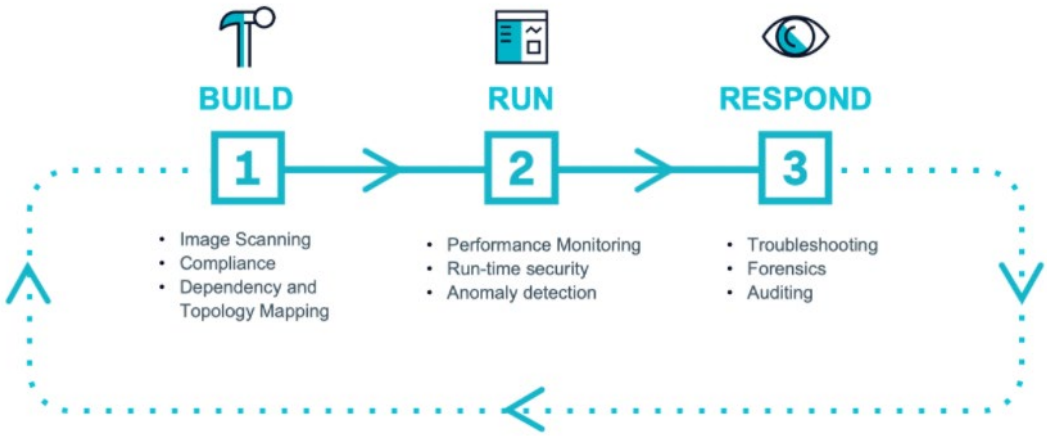

Figure 21 – Microservice lifecycle [43]

One needs to remember that a system developed on top of MSA has specific characteristics and needs that make it different from a monolith architecture or SOA, namely because of the nature of the architecture. The infrastructure is super distributed, scalability in real-time is performed because of the load on each component, since some systems have like hundreds of microservices we need to remember that to rearchitect such a system and better allocate resources there is a constant need for monitoring and changes to be applied over time.

Because of this and to avoid rearchitecting such systems the focus when programming an MSA system should always be the design of the system as shown in the literature being the most

predominant researched lifecycle phase the design one [4], [5]. By providing a great and solid design the maintenance, operation, implementation, and testing phases are less prone to errors.

Design patterns

Some strategies are defined to allow the design lifecycle phase to elapse flawlessly, these are known in the programming world as design patterns or architectural patterns, which are strategies to solve common problems. With the inherent complexity of MSA systems, a lot of design patterns appeared to mitigate some issues allowing for the code to be more flexible. Is worth mentioning some of these strategies as they are very common and almost like a foundation of microservices.

The most used design patterns are API gateway, publish/subscribe, circuit breaker, proxy, load balancer, discovery pattern, service registry, and service bus [4], [5], [76] handling most of the concepts of communication and orchestration. Also, the infrastructure needs constant monitoring to make sure services do not fail and if they do are replaced immediately.

These design patterns mainly appeared because of the already stated complexity, to handle communication between hundreds of services that need to communicate between them and handling millions of requests from clients having to provide responses as fast as possible led the programming community to achieve such strategies that allow MSA systems to be more maintainable, trustable, error-free, and flexible. Microservices can then provide the benefits of scalability that every system is after.

Annex B – Value analysis

Value analysis

This chapter emphases on the value that is projected with Jolie, by means of the opportunity identification and analysis is possible to obtain a better overview in which market Jolie fits, then using the Analytic Hierarchy Process (AHP) method is compared Jolie with other programming languages and which one is the better decision, lastly a more business focused approach is done to evaluate Jolie in terms it's proposed value, in which points it shines and which ones is not that great.

Opportunity identification

As a new programming language, to enter the market for the industry and developers to embrace it is not easy. There are an immense number of languages already in use for various types of requirements. Jolie has particularities that can make it fit in and one needs to think of it as a service-oriented programming language so if this type of languages become more and more popular instead of frameworks the language might have a change in the market for wide adoption of it, but this is a slow process.

The key for success in the market and developer usage of Jolie is being really purpose specific and at the same time providing some benefits over the frameworks that fulfil the same purpose. Microservices always emerge the word complexity with them, and Jolie can be the solution for that.

Thus, if the programming community and industry starts to see the advantages of Jolie in practice and not only in the academia and research probably Jolie will have a good position in the microservices development space.

If one decides to use Jolie to build a MSA system several benefits can be reached like:

- Codebase easy to read and understand as Jolie code is formal
- Support for programming of protocols and workflows that are not natively supported by mainstream languages
- No need to change behaviour of services when service communication changes
- Deep integration between language and microservice technologies
- Allows one to clearly understand how a service can be invoked, and which services it requires by just reading the code

- Allows the focus to be shifted into architecture and model of the system
- Reduced gap between model and code

Opportunity analysis

After identified an opportunity is of extreme importance to assess the opportunity and analyse it to confirm the worth if it's worth pursue. Thus, to confirm the value of the opportunity there is the need of additional information to transform the opportunity identification into specific business and technology opportunities. All the information can be obtained with extensive technology and marketing assessments like groups, market studies, and/or scientific experiments.

Information technology never stops evolving as new trends will reach top market growth in the future like ubiquitous computing, Internet of Things, datafication or big data and artificial intelligence [53]. Thus, as being technologies that require a lot of the features that microservices provide like scalability and agility. It can be seen as an even more promising world for microservices programming.

To analyze this opportunity three assessments are done: market, customers, and competitors. The market is assessed doing a detailed description of the market segment, showing why it represents a great opportunity, this is done providing a market size analysis, growth rates, and market share. For customers the assessment is done by determining what major customer needs are not being met by current solutions. The competitor's assessment determines who the major competitors are in the identified market segment and how to gain advantage over them [77].

Market

The areas of industry where microservices has more research perspective are Cloud, Internet of Things and mobile oriented technology as showed in [4], [5]. Thus, there must be analysed the market share of all these three areas to have a better perception if the opportunity is worth pursuing.

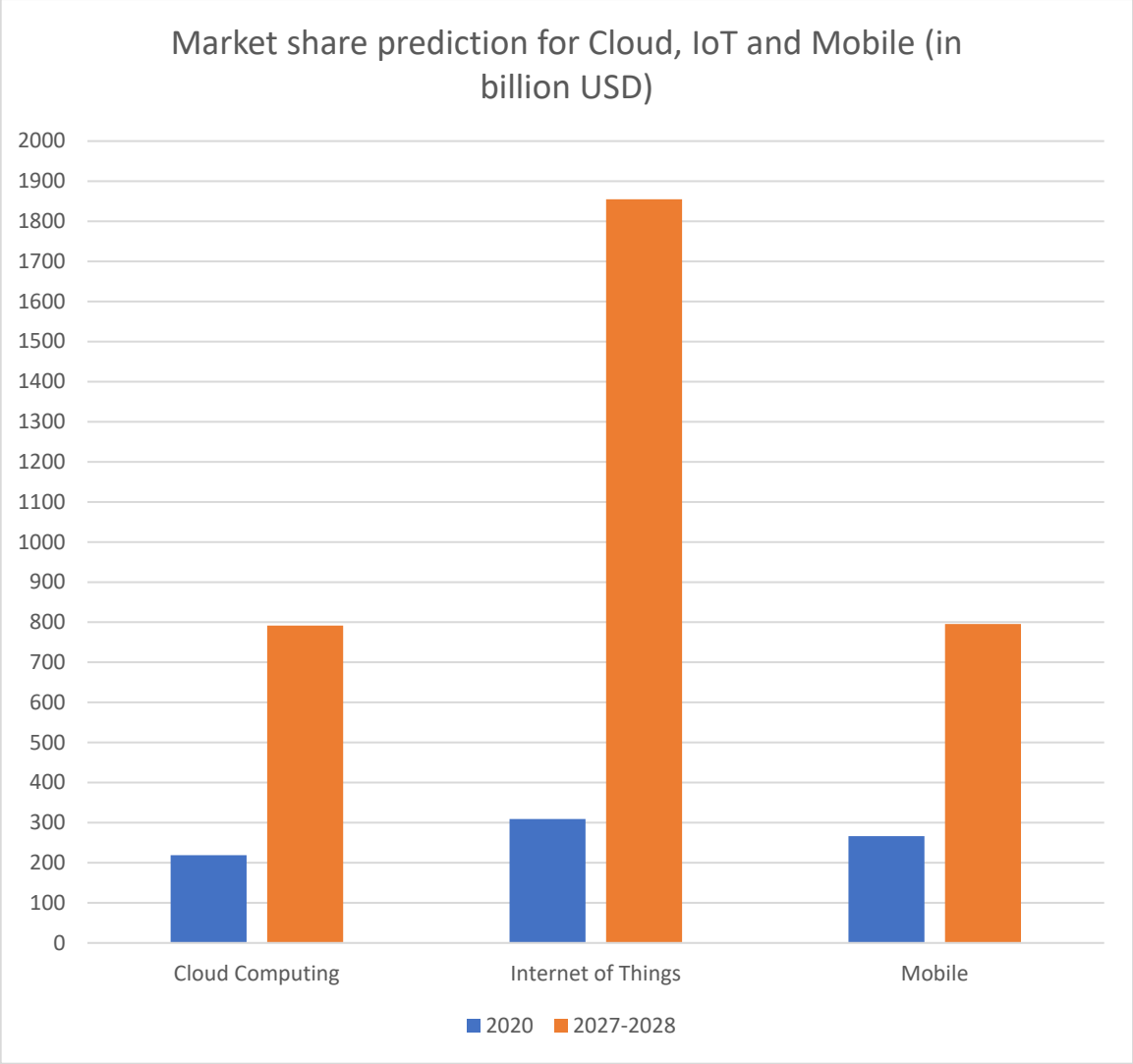

Figure 22 – Market share prediction for cloud, IoT and mobile between 2020 and 2027/2028 [78]–[80]

As Figure 22 shows the grow of these three sectors will be super high as predictions suggest. Cloud and mobile areas will grow pretty much the same amount, but Internet of Things will explode beating mobile and cloud market share combined. This is an excellent sign for Jolie since microservices will be even more needed than ever as they already are in these sectors.

Customers and competitors

To place Jolie against his major competitors in the microservices programming world one must consider the Jolie biggest opponents and the current market leaders in providing solutions to develop microservices. Since frameworks are the most used type of technology provided to the market the ones identified in [47] refer that the most used ones are Spring Boot and ASP.NET.

These competitors have all kinds of customers in the technology field like the technologic giants that moved to MSA systems [1]–[3] as being the biggest since their systems serve millions and millions of users. To be more likely to get these customers Jolie needs to appeal to his strong points and perhaps make more promotional content showing the difference between all the frameworks and the Jolie programming language. Customers might not even know what they can gain and expect using a formal language for microservices programming since this is a quite new approach for it.

The opportunity here is to provide the biggest value customers want: reduce complexity together with all the challenges brought with them as mentioned in 2.1.2 and reduce costs by providing an easier way to manage and integrate all the services and not needing developers with high-level experience [27] making them easy to read and develop.

Analytic Hierarchy Process (AHP)

The Analytic Hierarchy Process known as the AHP method was created by Thomas L. Saaty in 1980. AHP is a multicriteria decision and analysis method that uses numerical techniques to help decision makers choose an option from a discrete set of alternatives. This process is carried out based on crossing the alternatives with the existing criteria. Is important that the criteria get structured in such way that all the different criteria get the same level of importance.

This method allows for the use of qualitative and quantitative methods in the process evaluation. Main key of the method is to divide the decision problem in hierarchy levels allowing for better comprehension and evaluation.

Problem

The AHP method provides help in deciding something. But, to do so the problem/question needs to be declared explicitly and directly. This AHP will compare four microservice specific oriented programming languages: Jolie, Ballerina, WS-BPEL and CAOPLE. Thus, the question can be defined as - which service-oriented programming language to use?

Criteria

Some criteria used to evaluate each one of the programming languages and choose the best one accordingly.

- Documentation – the amount of documentation each language has on various web resources

- Community support – GitHub project stars can help to decide the biggest community of the programming language
- Performance – Here a hello world program performance can be done to measure the fastest language
- Versatility – What Jolie allows to do more than Ballerina and other languages? With this we can see which language is more versatile

Alternatives

For the alternatives, it will be used Jolie, Ballerina, WS-BPEL, and CAOPLE as they are the most used service-oriented programming languages in the community accordingly to [13], [34], [81], [82]. These programming languages are the ones that are on the frontier of the microservices, and cloud technology-focused programming languages.

The data for each alternative in relation to the criteria is searched in different places, namely: for documentation is searched on the programming languages' official websites and other documentation sources that appear on a Google search, even without official support from the programming language development team. For community support, the GitHub repositories of the languages are consulted if they exist (WS-BPEL and CAOPLE do not have an official GitHub repository) and the stars of the repositories set the scale for community support criteria. Performance is one of the least critical measures of the thesis since service-oriented programming languages are still emerging into the industry and literature, and because that do not dictate the full potential of performance in the programming language, performance is a measure outside of the programming language spectrum alone and other variables come into account like network latency, the architecture of the system, and things outside of the language control. Versatility approached with the data existing in the industry and literature about the programming languages and what they can provide in terms of different design patterns implementations, architectural styles, ways of programming, and any other important characteristics and features of the language, the one that has more features and allows for more diversity has better versatility.

Process

To start the process there is the need to define the problem and structure it in a hierarchical diagram. This is done by decomposing the problem/decision into a hierarchy, composed of an objective, criteria, and alternatives.

Figure 23 shows the hierarchy levels defined for the decision problem:

- General decision objective statement
- Criteria associated with the decision problem
- Available and more suitable alternatives

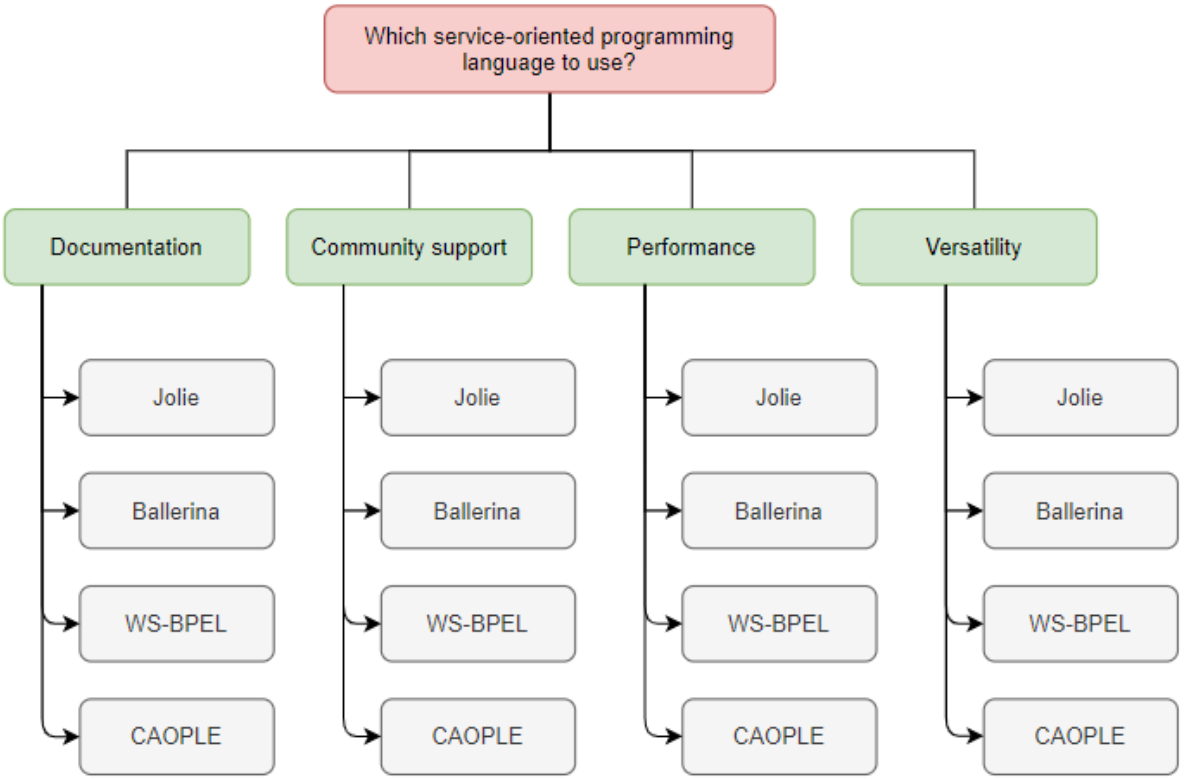

Figure 23 – Hierarchy tree of the AHP method

Now that the tree is completed the next phase consists in establishing priorities among each level of the criteria using the Fundamental Scale developed by Thomas L. Saaty represented in the Table 12.

Table 12 – Fundamental scale [60]

Level of importance	Definition	Explanation
1	Equal importance	Two activities contribute equally to the objective
3	Moderate importance of one over another	Experience and judgment strongly favour one activity over another
5	Essential or strong importance	Experience and judgement strongly favour one activity over another
7	Very strong importance	An activity is strongly favoured, and its dominance demonstrated in practice
9	Extreme importance	The evidence favouring one activity over another is of the

		highest possible order of affirmation
2, 4, 6, 8	Intermediate values between the two adjacent judgments	When compromise is needed
Reciprocals	If activity *i* has one of the above numbers assigned to it when compared with activity j. then j has the reciprocal value when compared with *i*	
Rationales	Ratios arising from the scale	If consistency were to be forced by obtaining n numerical values to span the matrix

These criteria comparison can be seen in Table 3.

Table 13 – Comparison of criteria

	2.1.	**2.2.**	**2.3.**	**2.4.**
2.1.	1	3	2	1
2.2.	1/3	1	1/2	1
2.3.	1/2	2	1	1/3
2.4.	1	1	3	1
SUM	2 5/6	7	6 1/2	3 1/3

To obtain the relative priority of each criteria it is necessary to:

a) Weight each criteria in relation to the problem/question
b) Obtain the normalized matrix values, done by dividing each matrix value by the total of the respective column
c) Obtain the priority vector, done by calculating the arithmetic mean of the values of each row of the normalized matrix

Dividing each value by the SUM value present on Table 13 at the end of the column. Normalized matrix is presented in Table 13. Next the priority vector containing all the relative priority of each criteria can be obtained. Table 14 presents the relative priority percentages.

Table 14 – Normalized matrix with relative priority

	2.1.	**2.2.**	**2.3.**	**2.4.**	**Relative priority**
2.1.	0.3529	0.4286	0.3077	0.3000	34.73%
2.2.	0.1176	0.1429	0.0769	0.3000	15.94%
2.3.	0.1765	0.2857	0.1538	0.1000	17.90%
2.4.	0.3529	0.1429	0.4615	0.3000	31.43%

After the relative priority the AHP method instructs to calculate the Consistency Ratio (CR) to measure how much the judgements where consistent when compared to big samples of random judgements. Thus, checking if the level of importance of each criteria is not done randomly. If the CR is greater than 0.1 the judgments made by the decision maker are not reliable because they are too close to the comfort of randomness.

The obtained CR value is 0.09146086 which is less than 0.1, therefore the obtained results do not present consistent values.

After the CR value is obtained there are left three more phases for the process to be completed. The construction of the comparation matrix peer to peer for each criteria as well as obtain the normalized matrices for each alternative considering the criteria are defined on the Attachment A.

In the Attachment A are all procedures for the construction of the comparison matrix and for the determination of the relative priority of each criteria that must be carried out again, observing now the relative importance of each of the alternatives that make up the hierarchical structure of the problem in question. Then one obtains the composite priorities of the alternatives, multiplying the previous values and those of the relative priorities, obtained at the beginning of the method. After this the choice of the alternative is made.

Table 15 – Alternative composite priority and choice

	2.1.	**2.2.**	**2.3.**	**2.4.**		**Best alternative**
3.1.	0.4774	0.2765	0.2786	0.5408		42.9690
3.2.	0.2969	0.5739	0.4690	0.3212		37.9471
3.3.	0.1810	0.1141	0.1484	0.0884		13.5381
3.4.	0.0448	0.0355	0.1040	0.0497		5.5458

Based on the AHP process, the Jolie programming language is the best service-oriented programming language solution to develop microservices followed by Ballerina, WS-BPEL and CAOPLE respectively.

Value proposition

CANVAS model

The business model is the way a company is structured to generate and capture value. Every company needs to know how to combine the means to deliver value to the relevant stakeholders and capture that value to the organization [83].

For this purpose, a practical and very system focused way to do it is to use a Business Model CANVAS that creates a template board to overview the business, it is important to not confuse the business model with product or service. Figure 24 portraits the model developed for the Jolie programming language.

<table>
<tr><th>Key Partners</th><th>Key activities</th><th>Value proposition</th><th>Customer relationships</th><th>Customer segments</th></tr>
<tr><td rowspan="3">Docker

HashiCorp

WSO2

Solo.io

Weaveworks

Epsagon

Tyk Technologies

Kong

Aqua Security

Sysdig</td><td>Programming

Microservices development

API gateway development

Service discovery mechanism development

Circuit breaker development</td><td rowspan="3">Jolie reduces the gap between model and code which allows for easier development of microservices. It reduces a lot of the complexity intrinsic to communications in microservice architecture-based systems</td><td>Every component of a microservice architecture system with less complexity and correct-by-construction meaning less errors</td><td rowspan="3">Developers

Software companies</td></tr>
<tr><td>Key resources</td><td>Channels</td></tr>
<tr><td>Software

Cloud providers

Hardware</td><td>Conferences on microservices

Web pages of partner companies

Workshops on microservices</td></tr>
<tr><td colspan="3">Cost structure</td><td colspan="2">Revenue streams</td></tr>
<tr><td colspan="3">Hardware

Developers</td><td colspan="2">Paid support</td></tr>
</table>

Figure 24 – Business model CANVAS

S.W.O.T. analysis

S.W.O.T. (strengths, weaknesses, opportunities, and threats) analysis is a process where a framework is used to evaluate a company, product, or service at a competitive level. Allows for a better strategic plan and decisions of the future. It also assesses the internal and external factors that can provide an overview of the potential of a product or service [84].

The S.W.O.T. analysis here is used to provide an overview of the Jolie programming language in terms of is current and future positive aspects and negative aspects. Provides a better perspective

that can be retained about the programming language. Figure 25 presents the complete Jolie S.W.O.T. analysis.

Strengths	**Weakness**
• Jolie code is simple • Supports programming of protocols and workflows that are not natively supported by mainstream languages • Changes on the communication of a service when environment changes without changing the behaviour of the service • Deep integration between language and microservice technologies • Allows one to clearly understand how a service can be invoked, and which services it requires by just reading the code	• Monitoring microservices is not easy using Jolie • Security is a problem with Jolie specially when communicating with external services • Performance cost by using such a formal language
Opportunities	**Threats**
• Allow to build a complete ecosystem based on a programming language • Compete against other tools for service discovery, load balancing, circuit breakers, API gateways	• Consolidation and wide adoption from the software development community of new programming paradigms and architectures • Implementation of the formal language approach used by Jolie by other languages • Appearance of more languages that want to reduce the gap between model and code and ease the development of MSA using MDE approaches like AjiL [85], JHipster [86] or Context Mapper [87]

Figure 25 – Jolie S.W.O.T. analysis