Universidade do Minho
Escola de Engenharia
Departamento de Informática

Rui Fernando Carvas dos Santos

Microservices architectures in healthcare with Apache Kafka

December 2021

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Rui Fernando Carvas dos Santos

Microservices architectures in healthcare with Apache Kafka

Master dissertation
Master's in Informatics Engineering

Dissertation supervised by
Professor Doutor António Carlos da Silva Abelha
Professor Doutor Hugo Daniel Abreu Peixoto

December 2021

ACKNOWLEDGEMENTS

I consider myself lucky for the people who crossed my path along the way. The elaboration of this work would not be feasible without their collaboration, encouragement and commitment. In this regard, I would like to express my thanks and consideration to all those who contributed.

To Professor Dr. Carlos Abelha and Professor Dr. Hugo Peixoto, for the indispensable accompaniment and encouragement throughout the journey, as well as for the cordiality with which they welcomed me.

To the University of Minho and to all the professors, especially those who are part of the Masters in Informatics Engineering, for the excellence of the teaching and the support provided.

To all my friends who made the process easier, making the less good moments a little better and the rest memorable.

To my sister, for her precious advice, for her high competence, total availability and encouragement at this crucial moment in my life.

To my parents, for their unconditional support, for their patience and, above all, for the effort made, allowing me to get here.

Finally, I would like to thank EVERYONE who is directly or indirectly part of my life and, in particular, of my academic path.

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

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

ABSTRACT

Over the past few years, we have seen an exponential increase in the amount of data produced. This increase in data is due, in large part, to the massive use of sensors, as well as the immense amount of existing applications. Due to this factor, and in order to obtain relevant information through the data, companies, institutions and the scientific community are constantly looking for new solutions to be able to respond to the challenges.

One of the areas where evolution is most needed is the area of healthcare, an area on which we all depend as a society. Every day, traditional healthcare information systems produce a large amount of data, making it complex to manage. Much of this data is produced by IoT devices, such as vital signs monitors, and in many cases can be critical to the patient's health, as in the case of Intensive Care Units.

In this sense, the main objective of this dissertation is to expose the advantages and disadvantages of the applicability of microservices architectures and the use of Apache Kafka in the health area, more specifically in Intensive Care Units where the information flow is critical. In order to support these objectives, a Proof of Concept was developed, based on a future real applicability, which will support the carrying out of analyzes and tests.

KEYWORDS Apache Kafka, Microservices Architectures, Intensive Care Units, Big Data, Internet of Things, Health Information Systems.

RESUMO

Durante os últimos anos, temos assistido a um aumento exponencial da quantidade de dados produzida. Este aumento de dados deve-se, em grande parte, à utilização massiva de sensores, assim como à enorme quantidade de aplicações existentes. Devido a esse fator, e de forma a conseguir obter informações relevantes através dos dados, empresas, instituições e comunidade científica, estão constantemente à procura de novas soluções para conseguir responder aos desafios.

Uma das áreas onde a evolução é mais necessária é a área da saúde, uma área da qual todos dependemos enquanto sociedade. Todos os dias, os sistemas tradicionais de informação em saúde produzem uma grande quantidade de dados, tornando-os complexos de gerir. Muitos destes dados são produzidos pelos dispositivos IoT, como os monitores de sinais vitais, e em muitos dos casos podem ser fulcrais para a saúde do paciente, como é o caso das Unidade de Cuidados Intensivos.

Neste sentido, a presente dissertação tem como objetivo principal expor as vantagens e desvantagens da aplicabilidade das arquiteturas de microservices e da utilização do Apache Kafka na área da saúde, mais concretamente nas Unidades de Cuidados Intensivos onde o fluxo de informação é crítico. De forma a auxiliar estes objetivos, foi desenvolvido uma Prova de Conceito, tendo por base uma futura aplicabilidade real, que servirá de suporte à realização de análises e testes.

PALAVRAS-CHAVE Apache Kafka, Arquiteturas de microservices, Unidades de Cuidados Intensivos, Big Data, Internet of Things, Sistemas de Informação na saúde.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACRONYMS

ABP Arterial Blood Pressure. 31, 32
API Application Programming Interface. 2, 11, 13, 26, 33, 35, 36, 37, 39, 42

BSON Binary Javascript Object Notation. 28, 29

CD Continuous Delivery. 15, 52
CI Continuous Integration. 15, 52
CSS Cascading Style Sheets. 42

DevOps Development and Operations. 14, 15, 52, 57
DSRM Design Science Research Methodology. 3

EHR Electronic health record. 7
ESB Enterprise Service Bus. 9
ETL Extract, transform, load. 52, 56

gRPC Google Remote Procedure Calls. 17

HL7 Health Level 7. 45, 50, 52
HTTP Hypertext Transfer Protocol. 39, 40

ICU Intensive Care Unit. iv, 3, 31, 34, 35, 42, 45, 46, 47, 48, 49, 50, 55, 56
IoT Internet of Things. 1, 6, 33, 34, 51, 52
IS Information Systems. 3

JS JavaScript. 17
JSON JavaScript Object Notation. 23, 28, 29, 34, 41, 45, 50, 52

ML Machine Learning. 52

PoC Proof of Concept. 4, 5, 30, 37, 45, 55, 56

RDBMS Relational Database Management System. 38
REST Representational State Transfer. 17, 35, 36, 37, 38, 39, 40, 42

SOA Service-Oriented Architecture. iii, 3, 6, 7, 9, 10, 11
SOAP Simple Object Access Protocol. 9
SPA Single Page Application. 42

STOMP Simple Text Orientated Messaging Protocol. 42
SWOT Strengths, Weaknesses, Opportunities and Threats. ii, iv, 4, 5, 45, 50, 51, 52, 53, 54, 55

TTL Time-to-Live. 29, 42

UI User Interface. 8, 13, 38

VM Virtual Machine. 15, 16, 37

WS Web Services. 9

XML Extensible Markup Language. 6, 50

1

INTRODUCTION

This chapter is dedicated to the presentation of the context and motivation of the theme of the dissertation, later addressing its purpose and objectives to be achieved with its development. This is followed by an exposition of the methodology used, including the different phases, as well as the literature review process. Finally, the organization of this document is described.

1.1 CONTEXT AND MOTIVATION

Information from the most diverse types of sources has been one of the key points for a better organization and development of society. The more information with quality, the better the results that will be achieved. For this reason, companies and organizations from different areas are increasingly recognizing its importance and need. (Hassan et al., 2020)

One of the areas where there is still space for evolution is healthcare, an area that we all depend on as a society. Every day, traditional health information systems produce a huge amount of data, making it complex and costly to manage. Additionally, new technologies are developing, and Internet of Things (IoT) has been taking over this area as well, generating even more data, thus making the whole process of collecting, storing, and making data available more difficult and complex. The growth of IoT has made it so that more traditional methods face more difficulties, which contributes to the emergence of the term Big Data. (Dash et al., 2019)

Data arises from various devices, such as the vital signs monitors in intensive care units, representing the health of a user at a given time. Data obtained by the various monitors must be presented in real-time, so that health professionals (doctors, nurses, etc.) can act accordingly, especially in an emergency situation.

However, to achieve better results, in addition to obtaining patient data, it is necessary to carry out several innovative strategies throughout the development cycle, namely in **data ingestion**, that is, in the way the data reaches the processing; in **verifying the data**, in order to certify that there is no invalid or fraudulent data; in the **processing of data** and, finally, in it's **presentation**.

Furthermore, when developing healthcare software, which may be responsible for processing large amounts of information, it is important that an architecture is prepared to deal with the concept of Big Data but also with scalability issues, such as the introduction of new features and/or increased load/response capacity.

A solution provided and adopted in other industries, has been to change systems based on monolithic architectures to systems based on microservices architectures. The latter has been gaining prominence in scientific

articles, blogs and conferences. A microservices architecture is based on a distributed development, where an application is divided into small services that communicate with each other through Application Programming Interface (API)s. Each service can be developed, implemented, and scaled completely independently without directly affecting other services in the application. (Richardson and Smith, 2016) Figure 1 represents the difference between monolithic architectures and microservices architectures.

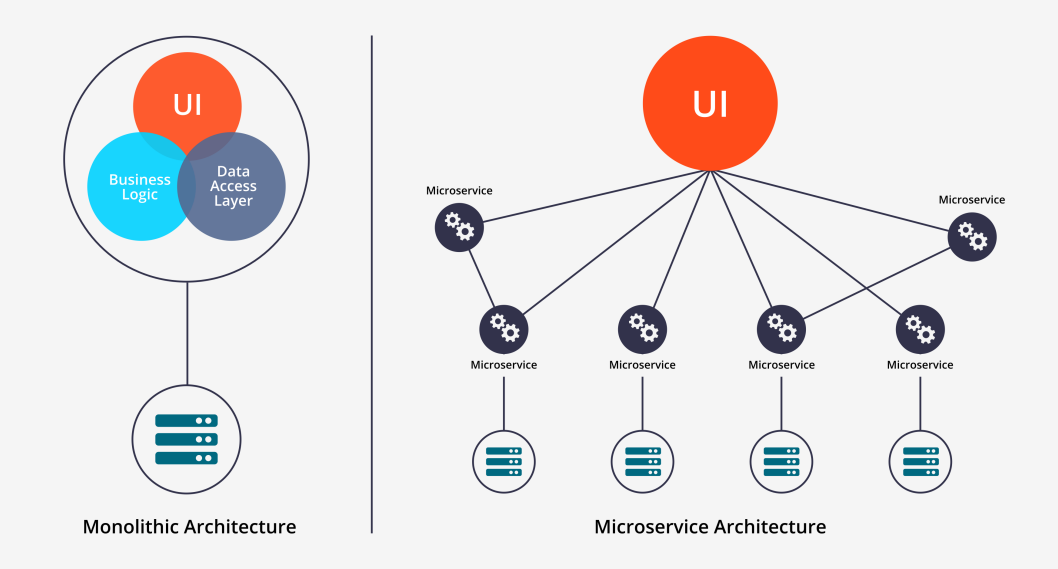

Figure 1: Monolithic Architecture versus Microservices Architecture. Source:(Sanjaya, 2020)

Allied to microservices architectures, there are several problems that can become complex, namely the communication between the services of an application. There are different forms of communication, however, a bad approach can make an application based on microservices become a monolithic distributed solution, thus not enjoying its advantages. Among the various types of communication, the Messaging that employs a Message Broker stands out. (Newman, 2020)

One of the most used Message Brokers, employed by 35% of Fortune 500 companies, is Apache Kafka[1], an open-source platform for event streaming, initially developed by LinkedIn[2]. It allows communications with latencies below 2ms, is tolerant to the most diverse failures that may occur during communication and can be scaled horizontally. Apache Kafka has evolved and today has a complete ecosystem, which can be used to help in various areas, such as Apache Kafka Streams and Apache Kafka Connect. (Narkhede et al., 2017b)

Assuming this context, the motivation inherent to this research is to understand the scenarios and contexts of microservices architectures and Apache Kafka in healthcare, analyzing their advantages and disadvantages. To this end, and to aid the research, the author participated in a research lab, with a team that was already developing projects and articles in this area.

1 https://kafka.apache.org/
2 https://www.linkedin.com/

1.2 OBJECTIVES

This dissertation has as its main objective the investigation of the following possibility: Is the use of architectures based on microservices and technologies such as Apache Kafka a viable alternative for the processing of messages in healthcare, with special focus in the transfer of information in Intensive Care Unit (ICU) where the flow of information is rather critical? Through the construction of a prototype with real data, it is intended to be able to demonstrate potential advantages, namely in the reception of information from the most diverse health providers, which may later be used.

Therefore, the formal objectives of this dissertation are:

- Review the main concepts associated with architectures based on microservices;
- Understand Apache Kafka technology and its main contexts of use;
- Design, plan and develop a real-time ICU data visualization prototype, using real data available in a dataset;
- Evaluate the prototype developed in order to assess the suitability of the proposal.

1.3 METHODOLOGY

The present dissertation is methodologically based on Design Science Research Methodology (DSRM), a well-known and accepted framework for implementing Design Science research in Information Systems (IS). Given the adopted methodology, the following steps result:

1. **Problem identification and motivation:** This consists of defining the target research problem and its importance, as well as clarifying whether the proposed solution has value in this context. This includes the elaboration of the Work Plan, in which the dissertation topic is identified, as well as its framework.
2. **Define the objectives:** Corresponds to the definition of the objectives based on the problem described above. This requires the study of various approaches, as well as their advantages/disadvantages for a correct choice of objectives. These can be quantitative (how a specific solution may be more desirable than others) or qualitative (how a particular artifact may be the solution to a particular problem). The following activities are included in this step:
    a) State of the Art, represented in chapter 2, where the relevant literature is identified, analyzed and understood, exposing the fundamental concepts involved in this research. In the case of this work, the concepts of Big Data, Microservices Architectures in comparison with Monolithics Architectures and Service-Oriented Architecture (SOA), and Inter-Process Communication are addressed.
    b) Technological Context, described in chapter 3, where the technologies used are analyzed and described for design and development.

3. **Design and development:** Includes the identification of the functionalities and architecture, taking into account the inherent quality attributes and challenges.
4. **Demonstration:** Demonstrates the effectiveness of the previously developed artifact for solving the problem. This may involve using it in experiments, simulation, a case study, proof, or other appropriate activity.
5. **Evaluation:** Measurement of the results obtained in the demonstration and analysis. This phase will be executed using test and performance metrics.
6. **Communication:** Writing and presentation of the dissertation, and the publication of related articles in journals/conferences is also feasible.

In addition to the applied methodology and for a better implementation of the methodology, the Proof of Concept (PoC) method was used. The PoC method is usually related to the area of Software development and is used as a demonstration to test the applicability and potential of concepts or theories in a real context, therefore it is a practical model. The use of PoC also allows the identification of weaknesses and potential weaknesses. In the scope of this dissertation a Strengths, Weaknesses, Opportunities and Threats (SWOT) Analysis was also developed to understand the feasibility of the solution.

1.3.1 *Literature Review Process*

The literature review is a process of collecting and selecting relevant literature that allows the author of the dissertation to increase knowledge in the area of study in question, as well as establish the relevance of the study to be developed by comparing it with existing works.

Literature identification and selection were performed in the period between October 2020 and January 2021. For this, the following keywords were used: "Apache Kafka", "Microservices Architectures", "Big Data" and "Health Information Systems", searched individually or in combination. Furthermore, the bibliographical searches were carried out using the following databases: "repositoriUM", "Google Schoolar", "Scopus", "ResearchGate", and "IEEE Xplore". In a first phase, the articles were selected according to the title and abstract, but they could have been discarded after a quick reading. The articles that were considered important were carefully analyzed and are referenced throughout the dissertation document. In addition, articles provided by the dissertation's co-supervisor were also used, as well as websites of the mentioned technologies and WEB articles. E-books developed by companies of the technologies used were also consulted.

1.4 DOCUMENT STRUCTURE

This document is divided into chapters to easily systematize the matters discussed. There are a total of eight chapters, including:

1. Introduction

In this first chapter, the purpose of this dissertation is exposed, by which the Context and Motivation that led to the choice of the topic are presented, as well as its objectives, the methodological approach to be used and, finally, the structure of this document.

2. State of the Art

   The second chapter presents the literature review considered essential for a better understanding of the present work, such as: Big Data, Microservices Architectures and Inter-Process Communication. Several works related to the context of the project are also exhibited.

3. Technological Context

   The third chapter describes the main technologies used, namely, Apache Kafka and its ecosystem and MongoDB.

4. Infrastructure and Materials

   The fourth chapter is dedicated to the characterization of the infrastructure and the dataset used to support the development.

5. The Prototype

   The fifth chapter shows the solution designed in order to successfully achieve the proposed objectives.

6. Development

   The sixth chapter reveals the entire PoC development process, explaining from the secondary technologies involved to the final demonstration.

7. Evaluation

   The seventh chapter refers to the analysis of the data obtained with the PoC together with a SWOT analysis.

8. Conclusions and Future Work

   The eighth and final chapter seeks to summarize the main conclusions drawn from the work carried out. Proposals for future work are also defined.

2

STATE OF THE ART

This dissertation focuses on the study of a new type of architecture in healthcare, namely Microservices architectures. However, the scope of this work is in fact larger than the study of this architecture, since, in addition to the architectural component, it is also important to find strategies that can help in the processing and analysis of a large demand for data in short periods of time.

2.1 BACKGROUND

The first part of this chapter presents the most relevant concepts associated with **Big Data**, specifically addressing healthcare, with **microservices architectures**, making a brief comparison with two other architectures, namely monolithic and SOA, as well as exposing its main characteristics and possible techniques used in the employment of this architecture. Finally, different types of **Inter Process Communication** will be analysed with special emphasis on the Message Systems/Event Based component.

2.1.1 *Big Data*

The term Big data has grown immensely in recent years and has become recurrent in several sectors, mainly those related to information and communication technology. This concept refers to the effort required to handle the processing and storage of large amounts of data. It is often data with a high size and complexity, which more traditional technologies have difficulty responding to. This high amount of data produced by organizations is directly related to the various sources existing today, such as IoT devices (sensors, mobile devices), banking transactions, industrial equipment, amongst others. (SAS, 2018)

One of the most challenging difficulties in managing big data is how data can surge. There are three types of classification for data (Ohri, 2021), namely:

- Structured data, which follows a structure that is known in advance;
- Unstructured data, where its form or structure is totally unknown, making it difficult for organizations/companies to extract knowledge; and
- Semi-structured data, which can be a mixture of the two previous types, such as Extensible Markup Language (XML) files.

In the early 2000s, Doug Lanei, a data & analytics strategy innovation fellow with the consultancy West Monroe, established the now dominant definition of big data as the three Vs: Volume, due to the collection of data from multiple sources; Velocity, since data arrives at companies at a high speed and must be processed in a timely manner; and Variety, as data can be in different types of formats. (SAS, 2018)

Big Data in healthcare domain

Healthcare as a broad concept can be defined has a set of healthcare services aimed at treating diseases and preserving the health of human beings. This sector, which is in constant evolution, generates a large amount of data from the large number of sources present in hospital units. These data sources are varied and may include: Electronic health record (EHR); medical imaging; wearables; and medical devices, such as the intensive care unit monitors present in the intensive care units. The ability to process Big Data, not just in healthcare, brings several benefits. With the information from the data analysis, healthcare professionals can have the help of an external intelligence to assist in decision making. (Catalyst, 2018)

The purpose of processing these data focuses on a wide variety of medical functions such as: diagnostics, preventative medicine, precision medicine, medical research, among others. (Catalyst, 2018)

2.1.2 *Microservices Architectures*

For a better understanding of this type of architecture, as well as the inherent characteristics, that will be exposed in the document, it is relevant to explain two other types of architectures. One, monolithic, more traditional, and SOA, a style often referred to as the predecessor of microservices.

Monolithic Architecture

The main programming languages used in the development of modern apps such as Python and Java, provide abstractions in order to divide the project in modules, facilitating the developers. However, these same languages are projected for the creation of single executable, commonly designed as monolithics. These are characterized for sharing all the modules in the same machine that is running the app, turning the app extremely coupled. (Dragoni et al., 2017)

Using a monolithic architecture in the development of apps can be beneficial, mainly due to the following advantages: (Ianculescu and Alexandru, 2020)

- Easy and fast initial development;
- Single location for management of resources;
- Universal programming language (no need for additional programming skills);
- Simple communication between the various modules;
- Easy deployment.

Nonetheless these advantages can in reality turn to disadvantages with time. New functionalities and updates add a set of obstacles to this architecture, as such: (Dragoni et al., 2017)

- Extensive Source-Code are increasingly harder to maintain;
- A single failure point;
  - If there is a problem that hasn't been detected in the compilation or realization of tests, the app can turn offline.
- A single alteration implies an restart of the entire monolithic, therefore increasing the time of developmen-t/tests;
- Complete deployment in each update of the source-code, even in modules that have not suffered any alterations;
- Limited scalability of the app;
  - Usually to solve the overload of the app, a new instance of the same app is created, dividing the load into two parts. However, this excess of load can occur in a specific part of the app and not in its entirety, causing, nonetheless, the replication of the app in its entirety.
- Using a single technology can be seen as a limitation from a development standpoint, when, at times, the best technology for a certain requirement is not used.

In figure 2, an example of how an app can be developed using a monolithic architecture is presented, where a single executable, contain the layer of business logic, User Interface (UI) and data interface. In addition, the data is all stored in the same database.

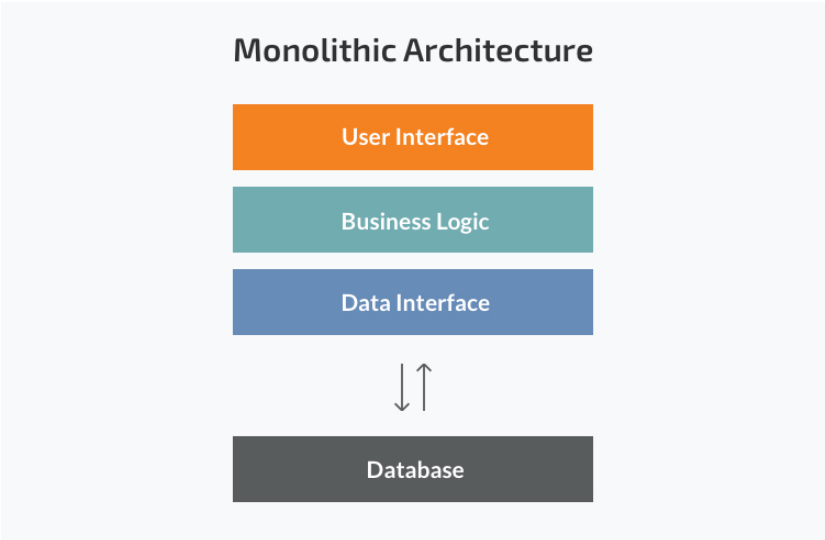

Figure 2: Monolithic Architecture. Source: (Gnatyk, 2018)

Service-Oriented Architecture

SOA is an architecture style based on a distributed system that allows the construction of complex apps through services that communicate between each other via a mutual language. Typically, Simple Object Access Protocol (SOAP) or Web Services (WS) is used. (Papazoglou and van den Heuvel, 2007)

A service, in this software architecture, is an independent unit that is projected to conclude a complete commercial function, having at its disposal all the data needed. (Papazoglou and van den Heuvel, 2007)

The employment of Enterprise Service Bus (ESB) is fairly typical in SOA. It allows for integration between the several services of an app. Furthermore, one or more databases are commonly used and shared amongst the several services of the app. (Papazoglou and van den Heuvel, 2007)

Some of the main advantages of this architecture are: (Papazoglou and van den Heuvel, 2007)

- Re-utilization of services;
- Easy maintenance;
- Better availability and scalability;
- Increased productivity;
- Platform independence.

Despite the previously shared advantages, this architecture presents some disadvantages, such as: (Paul, 2020)

- Single failure point;
- Shared databases increase the coupling of apps;
- Services can become large monolithics.

In figure 3 an example of how an app can be developed using a SOA is presented.

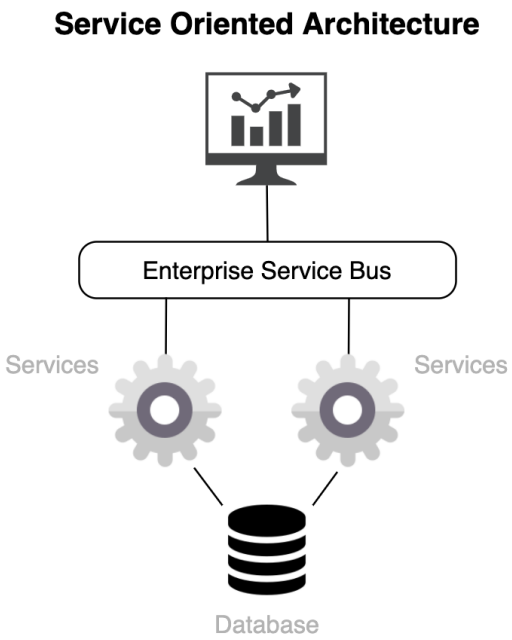

Figure 3: Service-Oriented Architecture (SOA) Ad: (Gill, 2020)

Microservices Architectures: Definition, Characteristics and Patterns

The term microservices, that ultimately has gained attraction from companies and organizations such as Netflix[1], Amazon[2] and UBER[3], and the scientific community, is, in fact, related to two key definitions presented below. (Richardson and Smith, 2016)

Definition 1 (Microservice) - *"A microservice is a minimal independent process interacting via messages."* (Dragoni et al., 2017)

Definition 2 (Microserive Architecture) - *"A microservice architecture is a distributed application where all its modules are microservices."* (Dragoni et al., 2017)

Having as a basis the previous definitions is possible to comprehend that a microservices based architecture is the decomposition of an entire app by basic functions, where each function is named microservice and can be created and run in a totally independent manner.

Making each microservice autonomous, allows for it to be developed, implemented, run and scaled without affecting the functioning of the other services of the app. Moreover there is not an excess of coupling, since the

1 https://www.netflix.com/
2 https://www.amazon.com/
3 https://www.uber.com/

other services do not share any type of codes amongst themselves. The communications through APIs are well defined.

Another important fact is that microservices are specialized, meaning each service is projected for a specific functionality dedicated to the solution of a problem. In case there is a need to add new functionalities to a specific service increasing its complexity, it can be later divided into smaller services allowing for a horizontal scaling.

In figure 6, the example of how an app can be developed using a microservice architecture is presented.

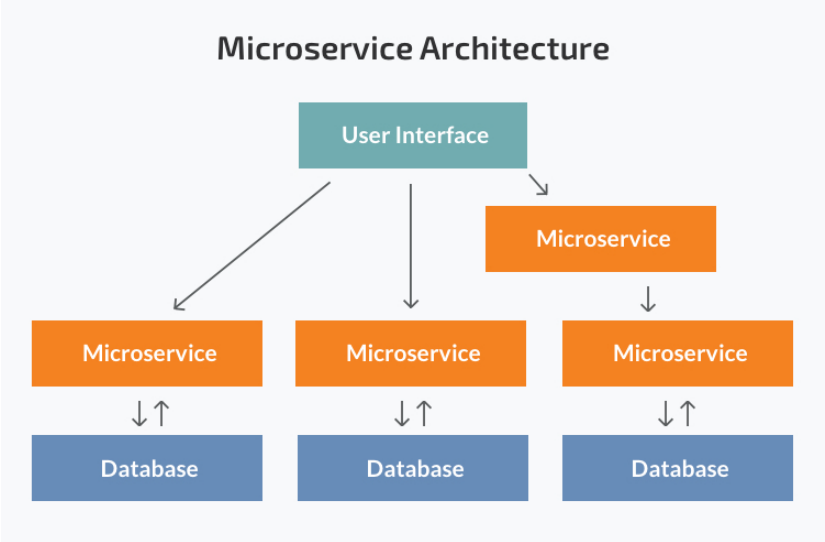

Figure 4: Microservices Architecture. Source: (Gnatyk, 2018)

Choreography vs. Orchestration

Associated with microservices architectures, there is much discussion regarding the type of coordination logic in the communication between the various services of an architecture. There are two types of approaches: Orchestration and Choreography. (Dragoni et al., 2017)

Orchestration promotes the coordination of logic through a centralized approach using a central service. This central service is primarily responsible for communicating with each of the services, then monitoring the results and returning the response to the front-end. This coordination technique is quite common in SOA architectures through the use of the ESB. Although with Orchestration you get rigider control of all the steps, this more centralized approach makes the services dependent on each other and has a single point of failure. The following image represents the Orchestration approach in microservices architectures. (Heusser, 2020)

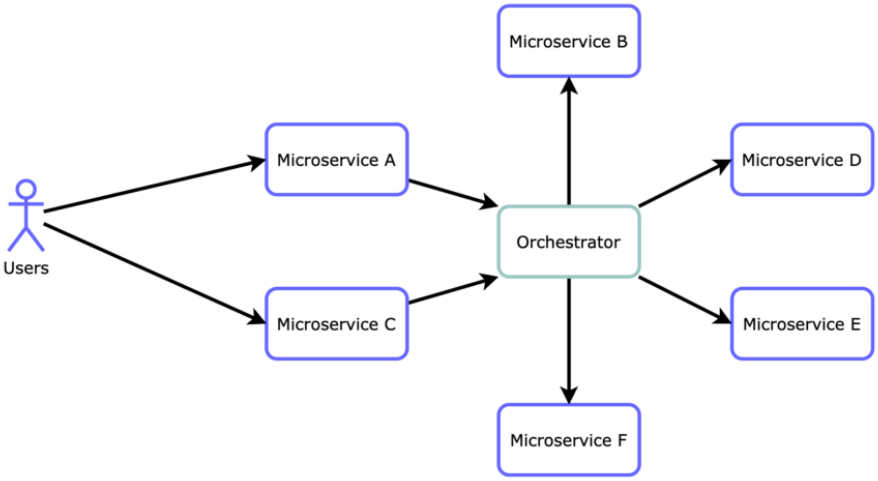

Figure 5: Orchestration Approach. Source: (Sucaria, 2021)

In the Choreography approach there is no longer a central service responsible for coordinating communication between the various microservices. In this approach, the application services work independently and are not blocked waiting for information from other services. It is common to use an Event Broker that distributes the work among the various components. In this sense, there is no longer a single point of failure but, the development complexity increases. The following image represents the Choreography approach in microservices architectures. (Heusser, 2020)

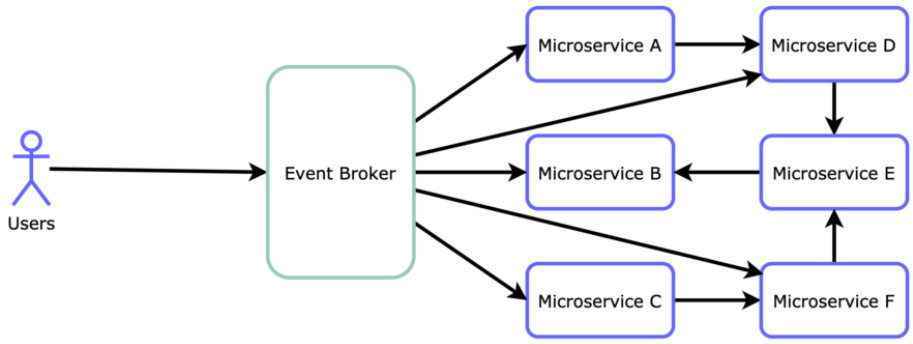

Figure 6: Choreography Approach. Source: (Sucaria, 2021)

Characteristics of a Microservice Architecture

There is a set of characteristics that are directly linked with microservices architecture, even though not all app's that employ this architecture share all the characteristics detailed below.

Componentization via Services

One of the main characteristics of microservices architecture is the componentization via services, meaning, the possibility to create a complete app composed by microservices, totally independent, that communicate amongst each other through well defined API's. These services are divided in such a way that each of them perform a functionality, and at the same time, cooperating to run a complete app. (Lewis and Fowler, 2014)

Organized around Business Capabilities

During the construction of an app, it is usual to separate the app in different development parts. The most common are: UI team; server-side logic team; and the database team, meaning, the architecture of the app organized around the technological resources. (Lewis and Fowler, 2014)

In a microservices based approach, the division must be made around the business. A development team should be able to alone be responsible for one or more products, through all stages of development (UI, Server-Side Logic, Database, amongst others). Each team member should have a specific ability, managing to supply a finished product. The development team needs to be multidisciplinary. (Lewis and Fowler, 2014) Figure 7 illustrates how the division of microservices should be done within a company. In this case there are two microservices, one responsible for payments and the other for medical appointments. Each team has at its disposal all the members responsible for the development cycle, and there is no division between the different areas of the product (frontend, backend).

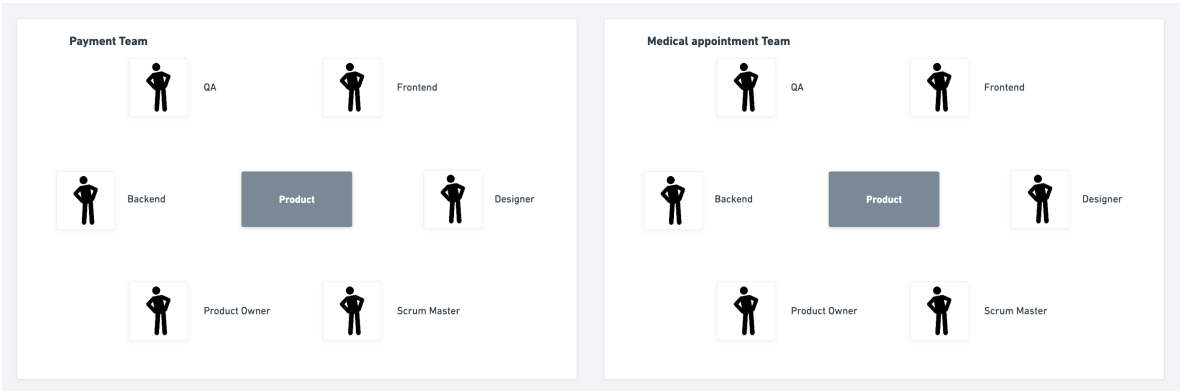

Figure 7: Organization based on Business Capabilities

Products not Projects

It is common in the development of an app, to employ a project model. The main goal of the team is to deliver an operating software that meets the required demands. After this stage, the software is managed by a team in charge of its maintenance. (Lewis and Fowler, 2014)

In a microservice based approach, this tends to not happen. Following Amazon's slogan "You build, you run it", a team is in charge of a product during its development and operating process, remaining in permanent contact with the product to improve it for the final clients. (Lewis and Fowler, 2014)

Smart Endpoints and Dumb Pipes

The communication between services of a microservice based app should be simple, quick and uncomplicated. To do so this architecture's communication is inspired in the UNIX classic system. The goal is: receive a requisition; process it; and respond. Simple synchronous or asynchronous protocols such as Request/Response and Publish/Subscribe are employed. (Lewis and Fowler, 2014)

By using these communication methods, in which the containers that send and receive messages are smart and the pipes are dumb, it allows for a decentralization of the architecture. (Lewis and Fowler, 2014)

Decentralized Governance

Another main characteristic of this architecture is the division of the app in small services. By doing this division there is no longer the need to restrict technologies that happens in other architectures such as monolithic architectures, where all logic is implemented in a single programming language. (Lewis and Fowler, 2014)

With this freedom of choice, the development teams can then adapt to each service the appropriate technology for the functionality. (Lewis and Fowler, 2014)

Decentralized Data Management

Each microservice should have its own database, in opposition to a centralized database that is used in monolithic architectures. (Lewis and Fowler, 2014)

The use of a database per service, makes it so that there are no shared tables. Consequently it increases cohesion and diminishes coupling, since alterations in the databases will not affect the data model of other services. (Lewis and Fowler, 2014)

Furthermore, it allows to use the type of storage best indicated for the service. E.g., relational databases, document store database, graph database, among others. (Lewis and Fowler, 2014)

Infrastructure Automation

When working with microservices architecture, the process should be agile and quick, including the Development and Operations (DevOps) parts. Since there are many microservices in an app, the manual deployment of each one can be arduous. (Lewis and Fowler, 2014)

To make the process more agile and quick at the DevOps level, it is common to employ varied techniques such as: (Lewis and Fowler, 2014)

- Cloud Computing;
- Automated testings;
- Continuous Integration;
- Continuous Delivery;
- Load Balancer / Auto Scaling.

Design for failure

Since an app is composed of several services, the app should be prepared for eventual failures in one or more services, therefore augmenting potential clients confidence. For such, secondary plans must exist to come into action when one or more services fail. After the occurrence of fails, it should be quickly detected, and if possible, the service should restart automatically. To allow for a quick detection and correction, the use of logging systems and real time monitoring is common. (Lewis and Fowler, 2014)

The characteristics mentioned above are only possible if the organization is composed by disciplined and qualified teams. In addition, a very coherent company organization is important. (Lewis and Fowler, 2014)

Deployment

Associated with the deployment process of a microservice architecture app, there is a set of good practices that should be implemented, such as Continuous integration and Continuous delivery. (Ska and P, 2019)

Continuous Integration (CI) is the process of integrating source-code modifications in a continuous automatic way, in a single software project. One of the checkups that can be performed is the execution of tests and the verification of the quality of the code, thus preventing human error of validation, and guaranteeing more agility and safety in the process of software development. Currently, CI is a valuable practice, of high performance, and well established in modern organizations. (Ska and P, 2019)

Another good practice in the DevOps process is the Continuous Delivery (CD). The development teams produce a valid software in short cycles. Allows to guarantee that the software can be continuously deployed in a trusted way. (Ska and P, 2019)

There are different types of patterns associated with the deployment of Microservices applications, each of which has its advantages and disadvantages. The different types of patterns are: (Richardson and Smith, 2016)

- Multiple Service Instances per Host Pattern
  - Run multiple instances of different services on a host (Physical or Virtual Machine (VM)).
- Service Instance per Host Pattern;
  - Service Instance per VM

    * Package the service as a VM image and deploy each service instance as a separate VM
  - Service Instance per Container
    * Package the service as a (Docker) container image and deploy each service instance as a container

2.1.3 *Inter-Process Communication*

Since microservices architecture is a distributed system, one of the focal points is precisely the communication between the various services. In monolithic services, the communication is made through call functions between modules. The same is not possible for microservices. (Richardson and Smith, 2016)

Microservices need to communicate amongst each other through a mechanism of communication between processes. There are several approaches that can be used for the interaction between microservices. They can be classified resorting to a two dimensions table. (Richardson and Smith, 2016)

Table 1: Inter-Process Communication Styles. Source: (Richardson and Smith, 2016)

	One to One	**One to Many**
Synchronous	Request/Response	-
Asynchronous	Request/Async Response	Publish/Subscribe

One of the dimensions refers to the number of destination services that should receive a solicitation.

- One to One;
  - Each client request is processed by exactly one service instance.
- One to Many.
  - Each request is processed by multiple service instances.

The other dimension is related to the answer time of the solicitation.

- Synchronous;
  - The service that made a request blocks waiting for a response.
- Asynchronous.
  - The service that made a request does not block waiting for a response.

Synchronous Communication

As previously described, synchronous communication has as a goal to assure that the service that made the request is blocked until the service to which it was sent, answers. To do so, and as demonstrated in the Table 1, there is a way to employ a synchronous communication:

Request/Response

A service makes a request to another service, blocks, and waits for a possible answer. An example of a Request/Response synchronous request is Representational State Transfer (REST) or Google Remote Procedure Calls (gRPC), when done synchronously. (Richardson and Smith, 2016) Figure 8 represents an example of a request/response operation where the requesting Service 1 is blocked waiting for a response.

Figure 8: Example of Request/Response Communication

Asynchronous Communication

On the contrary of what is verified in a synchronous communication, a asynchronous communication implies that the service that made the request, doesn't block whilst it waits for an answer. It may not even bother if the operation was or not concluded.

Request/Async Response

A service makes a request to another service that will answer asynchronously. The service that makes the request does not block whilst it waits for the answer, and will receive the answer later on. Example: JavaScript (JS) Callbacks. (Richardson and Smith, 2016) Figure 9 represents an example of a request/ async response operation where the requesting Service 1 is not blocked waiting for a response from Service 2.

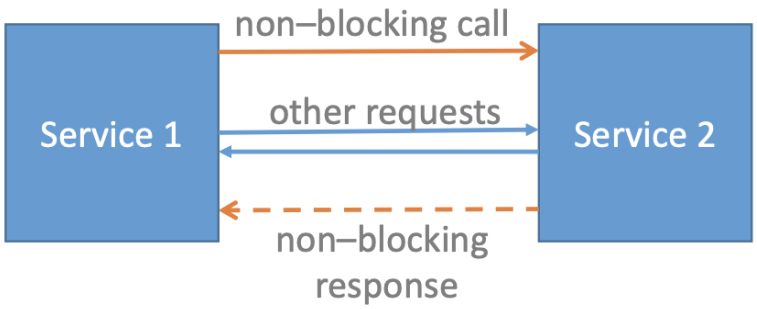

Figure 9: Example of Request/Async Response Communication

Event based

A service publishes a message that will be consumed by one or more services. An example of this type of communication is RabbitMQ and Apache Kafka. (Richardson and Smith, 2016) The later one will be discussed in section 3.1 of this dissertation.

As seen in figure 10, when Microservice A wants to send information to two other microservices, namely B and C, it publishes a message in a Broker. Later it will then be consumed by Microservice B and C. In this way, it is possible to guarantee that, even if Microservice B and C are offline, the message will not be lost, being consumed when the microservices are back online.

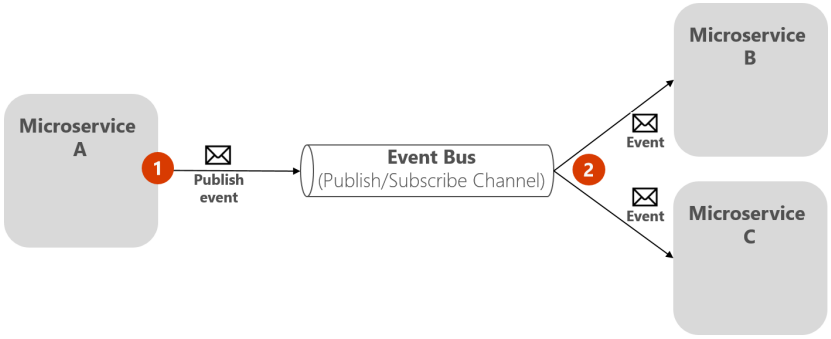

Figure 10: Event based Communication. Source: (de la Torre et al., 2019)

2.2 RELATED WORK

In this subsection, related projects of companies that apply microservices approaches together with Apache Kafka in healthcare are presented. The first company is Humana Inc. and the second is Babylon Health, both with healthcare as its main market.

2.2.1 *Humana Inc.*

Humana Inc. is a healthcare-related company, founded by David A. Jones and Wendell Cherry in 1961. It is currently headquartered in Louisville (United States of America) and integrates insurance, health, and wellness services as main areas of its job market. This company works with several segments, including: Retail, Employer Group and Health and Well Being Services. (Forbes, 2021)

The Health and Well-Being Services segment involves services offered to health plan subscribers, as well as third parties, which include pharmacy solutions, provider services, clinical care, predictive modeling and informatics services to other Humana businesses, as well as external health plan members, external health plans, and other employers. (Forbes, 2021)

Due to the multiple areas involved in Humana Inc.'s job market, it is difficult to link all clinical data of patients, so that they are updated. This data may be present in several providers, such as: the pharmacy record, the home healthcare visit, the specialists' recommendations, the electronic medical record at the local hospital, and others. If the data is out of date, it may trigger an incomplete perception of health professionals with regard to the clinical data of patients, implying their decision-making. Ultimately it could have serious repercussions, inclusively fatal. (Combs, 2020)

Levi Bailey, current Associate Vice President and Cloud Architecture at Humana Inc., said: *"When we think of a better healthcare ecosystem, we need to think about the opportunity to exchange data in a seamless way, where all participants can freely integrate that data to help drive the outcomes and the experience within their organizations."*. (Confluent, 2020) In order to overcome this adversity, the solution found by Levi Bailey and Humana Inc., went through the use of Event-driven mechanisms in the cloud, namely Apache Kafka. This type of mechanism changed the concept of certain actions, such as sales or a customer service interaction for events. Data from these events can be used to provide real-time recommendations and decisions. (Combs, 2020)

The use of this type of mechanisms allows Humana Inc. the ability to scale to a hyperconnected data ecosystem, thus managing to gather all data, optimize it and subsequently make it available for the intended services. In addition, it provides another benefit, allowing the existence of a history of events, which makes it easier to understand the implications of changes. (Combs, 2020)

All the changes made allowed Humana Inc. and its patients to obtain a more complete view of their information, including lab data and diagnostic codes, ensuring, above all, an accurate and current assessment by healthcare professionals. (Combs, 2020)

2.2.2 *Babylon Health*

Babylon Health is a healthcare company, founded in 2013 in the UK by Ali Parsa. The main objective of the company is to make healthcare accessible to all citizens worldwide. In order to achieve this, it applies innovative technologies and Artificial Intelligence, in order to relieve the doctors work, allowing them to focus on the patients who need it most. (Babylon Health, 2013)

In addition to developing its products, Babylon Health is present at several conferences such as Big Data LDN[4] and KafkaSummit (led by Confluent)[5]. During these events, Babylon Health explains how it applies architectures based on Microservices and Event Streaming to develop its products. At the Freeing up Kafka conference engineering resources to scale with DataOps for Big Data LDN, Jeremy Frenay, the Data Operation Engineer of Babylon Health, showcased Babylon Health's evolution from a low-service startup to a healthcare unicorn as well as the resulting challenges. (Stevenson and Frenay, 2019)

Since Babylon Health intends to revolutionize the healthcare area with Artificial Intelligence techniques, the need arose to aggregate a set of data present in various services, as represented in Figure 11. (Stevenson and Frenay, 2019)

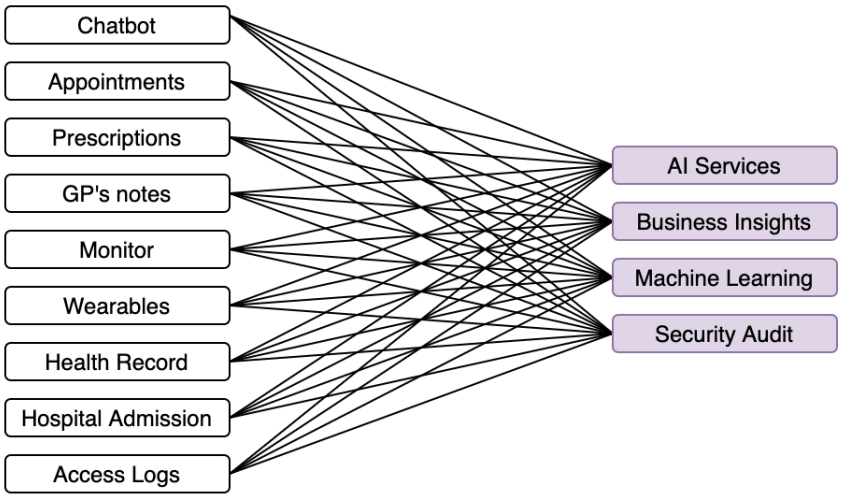

Figure 11: Babylon architecture before the introduction of Apache Kafka. Ad: (Stevenson and Frenay, 2019)

To be able to aggregate, filter and anonymize data from the various services, in order to obtain better insights, models and security, it was necessary to outline a new style of development based on the development of microservices. Furthermore, it chose to use Apache Kafka to aggregate all the data quickly and efficiently. Figure 12 represents the role of microservices and Apache Kafka at Babylon Health. (Stevenson and Frenay, 2019)

4 https://bigdataldn.com/
5 https://www.kafka-summit.org/

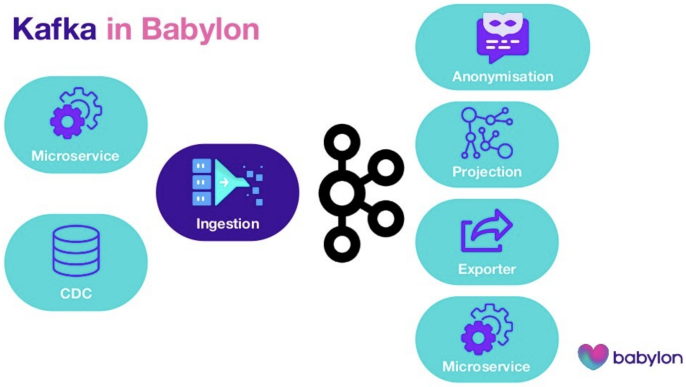

Figure 12: Babylon architecture after the introduction of Apache Kafka. Source: (Noble and Nobilia, 2019)

3

TECHNOLOGICAL CONTEXT

One of the main objectives of this dissertation is the evaluation of an emerging technology in the IT market, Apache Kafka. To this end, this chapter explores the main concepts of Apache Kafka, as well as two of the various tools present in its ecosystem. Additionally, a description of MongoDB is also provided.

3.1 APACHE KAFKA

Apache Kafka is an open-source platform of distributed transmission of events, developed in the programming languages Java and Scala. Initially developed by Linkedin[1], with the goal of processing 1.4 trillion message a day, it was launched in January 2011. Currently, it is kept and explored by the company Confluent[2], under the stewardship of the Apache foundation[3]. (Narkhede et al., 2017a, 14–16)

Of the many characteristics of the Apache Kafka, four can be highlighted: **High Performance**, since the delivery of messages has a latency of around 2 milliseconds; **Horizontal Scalability**, it is possible to expand up to hundreds of brokers and process millions of messages by the second; **Fault-tolerant**, thanks to its distributed system, in which messages are duplicated and stored in different locations; and **High availability**, since it is possible to have clusters in several parts of the world.

Apache Kafka supports different types of use cases categories, such as: the *publish-subscribe message system*; *Website Activity Tracking; Gather Metrics from many different locations; Log Aggregation; Stream Processing; Event Sourcing; and Commit Log*. (Narkhede et al., 2017a, 11–13)

Due to the previously mentioned characteristics, Apache Kafka is present in the most diverse business areas such as Banks, hospitals and telecommunications. Currently, it is employed by more than 2000 companies, including 80% of all Fortune 100 companies. Companies such as Netflix, Uber, Airbnb and PayPal all use Apache Kafka. (Apache Kafka)

1 https://www.linkedin.com/
2 https://www.confluent.io/
3 https://www.apache.org/

3.1.1 *Key Concepts*

Kafka Cluster and Kafka Broker

A Kafka cluster consists of one or more Kakfa servers (also denominated Kafka brokers). Despite being possible to initiate a Kafka cluster with only one Kafka broker, it is not recommended since it does not allow to take advantage of all the qualities of Apache Kafka. (Narkhede et al., 2017a, 7–9)

Inside the Kafka brokers, the topics are located. Producers publish messages in these topics so that later on consumers can read them. (Narkhede et al., 2017a, 7–9)

To initiate a Kafka cluster, at least one Zookeeper is needed, that will have as main function the management of all the Kafka brokers of one cluster. (Narkhede et al., 2017a, 7–9)

Zookeeper

In order for a Kafka cluster to run properly, at least one zookeeper is needed. The zookeeper's main goal is to manage the several brokers of a cluster. (Vinka, 2018)

Besides managing the several brokers, the zookeeper has other functionalities, such as: (Vinka, 2018)

- Aiding in the election of a partition leader in a topic;
- Send notification to Kafka when:
  - A new topic is created;
  - An existing topic is eliminated;
  - A broker becomes unavailable;
  - A broker becomes available again.

The existence of more than one Zookeeper is possible. However, only one will be the leader. The others will act as alternate, and can only exist in odd numbers. For example, there can be 1,3,5,7 etc zookeepers.

Messages

Messages is the main resource of Apache Kafka. It's where the data that is published by the producers and consumed by the Consumers, is located. A Message is composed by a Key, that identifies in which partition the message will be stored, a Value, where the content is stored (it can be a simple number or a string to an object in JavaScript Object Notation (JSON), and a timestamp, that identifies the hour in which a message was sent to a specific topic. (Narkhede et al., 2017a, 4)

Topic, Partitions and Offset

The term topic in Apache Kafka signifies a category or a name of a flow in which messages are published. It is identified through a single name that usually references the flow of data for which it was created. It is composed

by a specific number of partitions that should be introduced by the user when it creates the topic. The messages are stored in the partitions in an ordained and immutable way, where each is attributed a single incremental identifier called Offset. (Narkhede et al., 2017a, 5–6)

When a message arrives to a topic, it is sent randomly to a partition, unless it carries a specific key. In this case, all the messages with the same key will be stored in the same partition. (Narkhede et al., 2017a, 5–6)

By definition of the Apache Kafka, a message is retained in a topic during a week. However, the time period can be altered. For example, to one day or one year. (Narkhede et al., 2017a, 5–6)

The topics may possess a replication factor that makes it so that partitions are replicated by the several brokers of a cluster. In this sense, if a broker has problems or is unavailable, the topic messages can be consulted through other brokers, therefore increasing the availability of the software. (Narkhede et al., 2017a, 5–6)

Apache Kafka uses the concept of partition leader. There is only one leader partition, independently of the quantity of replicas. At the same time, it is exclusively responsible for receiving and providing the various messages. The replicas should only have as function to synchronize the information in order to prevent errors. (Narkhede et al., 2017a, 5–6)

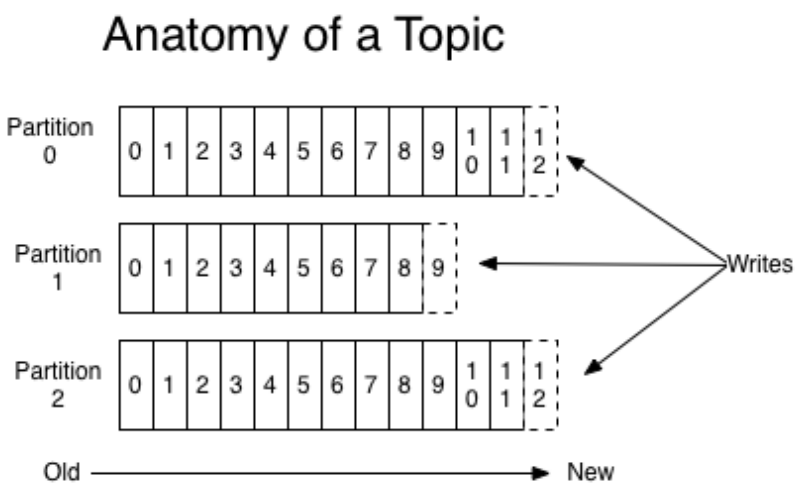

Figure 13: Anatomy of a Topic in Apache Kafka. Source: (Lawlor et al., 2018)

Producer

In Apache Kafka, the producers are responsible for publishing messages in a topic. A producer knows which broker to communicate to when it first calls (this is done automatically through Apache Kafka). In case the broker is unavailable, Kafka will redirect to another broker. (Narkhede et al., 2017a, 41–62)

By definition, a producer can not choose which topic partition it wants to publish the message. This can be altered by adding a key in the message. A key can be a number, a set of characters, etc. (Narkhede et al., 2017a, 41–62)

When the producer publishes a message, an acknowledgement of receipt is sent to assure there aren't failures. There can be three types of confirmations: (Narkhede et al., 2017a, 123–124)

- Acknowledgment = 0;
  - Producer won't wait for acknowledgment;
  - Possible data loss;
  - Faster mode.
- Acknowledgment = 1;
  - Producer will wait for leader acknowledgment;
  - Subject to the possibility of loss of information if the process fails in the synchronization of replicas. It is, however, safer than in Acknowledgment =0.
- Acknowledgment = All.
  - Producer will wait for leader and replicas acknowledgment;
  - No data loss;
  - Slower process compared to Acknowledgment = 0 and = 1.

Consumer and Consumer Groups

The main goal of a consumer in Apache Kafka is to consume the data of a topic, with the role of treating and/or analysing it. As it happens with the Producers, the consumers know, right at the initial communication, to which broker they should communicate. (Narkhede et al., 2017a, 63–93)

A consumer can subscribe to one or more topics, that are identified by their name, and consume stored messages in the various partitions in the order through which it was produced by the Producers. Despite the messages being consumed by order within a partition, it is not possible to guarantee the order of consumption through the several partitions of a topic. (Narkhede et al., 2017a, 63–93)

In the case of an overwhelming demand of messages that the Consumer can not keep up with there is an additional feature in the Apache Kafka that allows for scaling and following of the execution of the publication of messages, called Consumer Group. A consumer group is formed by consumers in which the various partitions of a topic are divided through the members of the group. In case there are more consumers in a consumer group then partitions in a topic, one of the members of the group will have to be inactive. It will act as an alternate, so in case one fails, it can immediately substitute it, without losing productivity in the software. (Narkhede et al., 2017a, 63–93)

Similarly to Producers, Apache Kafka has a mechanism to safeguard eventual failures during the consumption of the various messages. As soon as a consumer receives a message, it automatically communicates to Apache Kafka the Offset that it just received. If in case of failure the consumer becomes unavailable, the Apache Kafka will know what message it was in and as soon as the consumer becomes available, it will continue reading

the mentioned message. (Narkhede et al., 2017a, 63–93) There are three ways to do the safety commit of the reception of a message: (Vasquez, 2020)

- At Most Once;
  - Commit is realized as soon as the message is received;
  - In case something goes wrong after the reception, in other words, in its possible storage, the message will be marked as read, and it will not be read again.
- At Least Once;
  - Commit is realized after the reception and processing of the message;
  - In case something goes wrong after the message is received, there won't be a problem, since the message will be consumed again;
  - There is a possibility of the duplication of messages, in which case it should be saved by the source-code.
- Exactly Once.
  - Used by Apache Kafka on the Apache Kafka Streams.

3.1.2 *Apache Kafka Ecosystem*

Initially Apache Kafka was essentially, what was previously described, there being, the existence of Brokers, Producers and Consumers. However, throughout its existence it has evolved to a complete ecosystem, built and sustained to help companies that use this system. Some of the products that have helped build the ecosystem are the Apache Kafka Connect and the Apache Kafka Streams. (Sax, 2018)

The Apache Kafka Connect was created in November 2015 during an update of the Apache Kafka. It's goal is to simplify what, often times, is very repetitive. It allows to fetch data from a source (e.g. MongoDB, MySQL, Twitter API), and publish in one or more Kafka topics, without there being a need to type a single line of code. Furthermore, another product was created, Apache Kafka Sinks, that does the exact opposite. It fetches data from a topic and places it in a Sink, that can be anything from an ElasticSearch or a text file. (Sax, 2018) Figure 14 exemplifies Apache Kafka Connect. On the left side are examples of Apache Kafka Connect Source (Cassandra, MySQL, Oracle Database and MongoDB) and on the right side are examples of Apache Kafka Connect Sink (Hadoop, ElasticSearch, Cassandra).

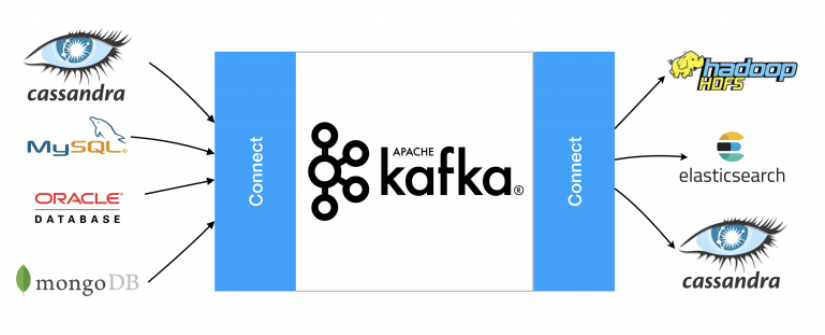

Figure 14: Apache Kafka Connect. Source: (Malik, 2018)

There are several connectors developed by Apache Kafka and Confluent. However, and similarly to what happens with Docker, companies and developers are free to provide its own connectors, creating a hub of Connectors.[4]

Besides Apache Kafka Connect there is also the Apache Kafka Streams, a library produced in Java and launched in 2016. It aims to help in the processing and transformation of data in real time. In other words, Apache Kafka Streams is a library capable of consuming data from a topic, processing it, and, per a condition, sending it a specific topic. (Sax, 2018) Figure 15 demonstrates an example of the various Apache Kafka products linked together. On the left side the data sources (Apache Kafka Connect Source and Apache Kafka Producer) are represented, in the middle are Apache Kafka Streams responsible for transforming data into real time, and on the right side are Apache Kafka Consumer and Apache Kafka Connect Sink.

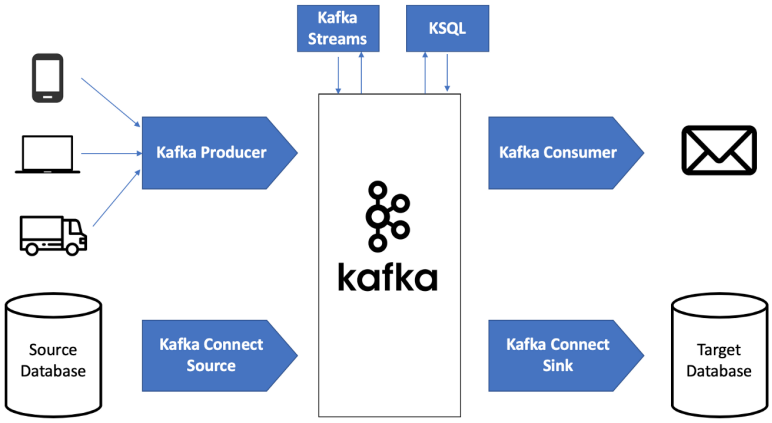

Figure 15: Apache Kafka Streams and Apache Kafka Connect. Source: (Maarek, 2019)

4 https://www.confluent.io/hub/

In addition to the two variables presented previously, Apache Kafka contains a wide range of options that can help companies in the most varied of occasions.[5]

3.2 MONGODB

MongoDB is an open-source document database that provides high performance, high availability, and easy scaling. It is one of the NoSQL databases available on the market, which is currently part of the group of the most used. Applies to many use cases, such as: Big Data; Content Management and Delivery; Mobile and Social Infrastructure; User Data Management; and Data Hub. Furthermore, this database is known for its simplicity of use when it comes to data representation, since its key concepts are related to: Document, where the data is stored; and Collection, a set of several Documents. (Maksimovic, 2017)

A Document in MongoDB is a data structure made up of field and value pairs. The representation format of a Document is very similar to the JSON format, however, MongoDB stores the information internally in Binary Javascript Object Notation (BSON) format. BSON, in turn, represents a JSON in binary format and its use in MongoDB allows it to be analysed faster. (MongoDB - JSON and BSON)

A Collection in MongoDB is the location where Documents are stored. Each Document in the Collection, unless specified, has an "_id" automatically assigned by MondoDB. Inside a Collection, contrary to what happens in a traditional SQL database, there is no specific format, that is, a Collection can have different Document formats, which simplifies the storage of unstructured or semi-structured data. By default, all Documents in a Collection have similar or directly related purposes. (TutoialPoint) The example Document of figure 16 is composed of an _id, two fields (firstname and lastname), a sub-document address with four fields, and a field of type array with two values.

```
{
  "_id": "5cf0029caff5056591b0ce7d",
  "firstname": "Jane",
  "lastname": "Wu",
  "address": {
    "street": "1 Circle Rd",
    "city": "Los Angeles",
    "state": "CA",
    "zip": "90404"
  }
  "hobbies": ["surfing", "coding"]
}
```

Figure 16: Example of a MongoDB document. Source: (What is a Document Database?)

5 The complete ecosystem of Apache Kafka can be consulted in https://cwiki.apache.org/confluence/display/Kafka/Ecosystem.

MongoDB has indexes that support efficient query execution. Furthermore, it features special single-field indexes such as Time-to-Live (TTL), which allow the removal of Documents from a Collection after a period or at a certain point in time. This type of Index is useful in files that only need to be persisted in the database for a finite period, such as Logs, machine generated event data and others. (MongoDB - TTL Indexes)

3.2.1 *MongoDB in healthcare domain*

Databases in the healthcare sector provide a suitable system for storing, organizing, and managing information related to several areas, such as: patient and healthcare professionals' information, data related to laboratories and medical exams, management of healthcare units, among others. This information contains data and different specialties, and that must be made available in a fast and simple way. (Rajendran, 2012)

MongoDB is widely used in healthcare by several companies, such as Greenline Senergy, Cure.fit, Apervita, Humana and Doctoralia. (MongoDB for Healthcare) This is due to the fact that MongoDB is a document-oriented database that allows the storage of information without any limitations or schemas. In addition, the use of the BSON/JSON format presents a significant advantage, since in addition to the speed of storage, it also allows for simple data manipulation. (Rajendran, 2012) Another advantage of using MongoDB in healthcare is that it allows simple and fast querying, eliminating the performance overhead of JOIN operations that occurs in relational databases. (Groce, 2014)

4

INFRASTRUCTURE AND MATERIALS

Starting from the objectives of the dissertation, which mainly consist of understanding the inherent advantages and disadvantages in the use of Microservices Architectures and Apache Kafka, this chapter will present the infrastructure and dataset used during the development of the PoC.

4.1 INFRASTRUCTURE

The infrastructure used to run the tests is configured on the Confluent platform, which enables the creation of an Apache Kafka cloud cluster. The cloud part used in the Confluent platform was Google Cloud Platform. The configuration used in Confluent was:

- Basic Cluster
  - **Broker:** 1
  - **Uptime SLA:** 99.5%
  - **Ingress:** up to 100 MB/s
  - **Egress:** up to 100 MB/s
  - **Storage:** up to 5,000 GB
  - **Client Connections:** up to 1,000
  - **Based on Google Cloud Platform with:**
    - **Region:** us-west4 (Las Vegas)
    - **Availability:** Single Zone

4.2 MIMIC DATABASE

To assist the study presented in this dissertation, a public dataset that is available on PhysioNet[1] was used. This platform is managed by members of the MIT Laboratory for Computational Physiology, whose purpose is to

1 https://physionet.org/

offer large datasets of clinical data for research and studies. (Goldberger et al., 2000) The selected dataset was the MIMIC Database version 1.0.0, which contains data from 72 patients present in the ICU. (Moody and Mark, 1992) Each patient in the dataset has a corresponding folder that is identified by a unique ID (ex. 301, 209) and where all their files available in different formats, such as: txt; dat; and hea. Each file represents ten minutes of measurements, resulting in twenty or more hours of periodic signals and measurements that may vary between patients. Thus, there are a total of 200 days of vital signs recorded in the dataset.

In the current dissertation, txt files were used, which have the following format:

```
[11:27:45 14/08/1994]   03700001.txt
RESP    46
ABP     64      82      53
HR      122
SpO2    0
PULSE   0
```

Figure 17: Example of a txt file format. Source: (Goldberger et al., 2000)

The first line of each file corresponds to the identification of the date and time of the first measurement. Subsequently, the periodic measurements of the patient are found, where, in this specific case, each 5 lines represent 1 second of the 10 minutes presented in each file. Each line displays a measurement of the sensors present on the monitor, initially identified by the signal type name, followed by one or more values. Values can be simple (1 single value) or composite (+ than one value). The types of sensors shown may vary by patient.

There are several types of data present in the files, such as:

- NBP – Noninvasive blood pressure;
- Arterial Blood Pressure (ABP)
- HR – Heart rate;
- SpO2 – Peripheral Oxygen Saturation.

In addition to the above-mentioned data types, notes corresponding to changes in the monitor's operation as well as changes in the patient's condition are also presented. Changes can be of various types. One of the changes present in several dataset files refers to anomalies in the monitor's operation, such as measurement failures. Changes that indicate measurement failures are initiated by identifying the sensor name, followed by the status and status value. Figure 18 exemplifies the failure of a sensor, namely the Sp02 sensor.

```
[21:48:00 14/08/1994]   03700063.txt
RESP    21
ABP     51      66      44
HR      122
SpO2  [inactive, status = 31a2]
PULSE   0
```

Figure 18: Example of an inactive sensor. Source: (Goldberger et al., 2000)

Another change that can occur in the measurements presented in this dataset is the occurrence of an inoperative sensor, that is, it indicates useless or ineffective values. This type of change is initially identified by a keyword INOP (Inoperative), followed by the sensor that is inoperative and finally the reason. Figure 19 demonstrates a case where a sensor situation happens to be inoperative.

```
RESP    13
SpO2    0
INOP    SpO2 NON-PULSATILE
```

Figure 19: Example of an incorrect sensor reading. Source: (Goldberger et al., 2000)

Finally, the occurrence of alarms happens when a certain sensor indicates a value above the recommended or previously defined value. There are several types of alarms present in the dataset and the structure is similar in all. It is initially identified by the Keyword ALARM, followed by two asterisks. After that, the sensor is identified, as well as the maximum recommended value and the measurement obtained. Figure 20 demonstrates an alarm situation, where the ABP sensor indicates an anomaly in the measurements.

```
PULSE   70
HR      70
RESP    15
PAP     39      59      27
ABP     57      90      41
SpO2    95
NBP     0       0       0
ALARM   ** ABP   85 < 90
```

Figure 20: Example of a high measurement alarm in the sensor. Source: (Goldberger et al., 2000)

5

THE PROTOTYPE

This dissertation consists of the study and exploration of a new approach and a technology, which, later, can be applied in future projects in healthcare. For this, and in order to be able to explore this new approach, the need arose to create a prototype that would represent a real-life use case, and that, simultaneously, could help in the analysis and conclusion of this type of approach.

In this chapter, an overview of the prototype that will be developed is presented, followed by the proposed approach and architectural description.

5.1 OVERVIEW

The architecture of a healthcare-related system should address the purpose of its creation, methods that facilitate the inclusion of future functionalities, as well as communication with external elements from multiple service providers. In this sense, it is important to develop a prototype that presents an organized, well-designed, and efficient structure, so that it can respond to problems and facilitate future changes. In this way, it is possible to promote a better flow of data in the system and an efficient use by the end user.

Consequently, a prototype was created, which represents a real-time data visualization system of patients in an intensive care unit.

5.1.1 *Challenges*

The prototype to be developed must be able to respond to a set of challenges that are often present in healthcare-related architectures, namely:

- The presence of **external suppliers**, that is, companies that provide services and that provide data through public APIs, making the communication process with the system difficult;
- The existence of **large amounts of data in short periods of time (Big Data)**, due to the excess of IoT devices (Medical Devices, bedside monitors) and users (Doctors, Nurses);
- The **reception, analysis and processing of data in real-time**, in order to prevent any anomalies/faults;

- **Large-scale data storage**, as they deal with clinical data, so it is extremely important to have a data history;
- The **representation of data through real-time graphics**, in order to support the medical decision and present alerts of possible changes (Alarms, Errors, etc.);
- The **horizontal scalability must be previously weighed** (instead of the vertical one) due to the high probability of a significant increase of devices and users.

5.1.2 *Functionalities*

The architecture of an ICU real-time clinical data visualization system comprises the communication between the IoT devices and the application responsible for the visualization, as well as the communication with other services for data analysis and storage. Thus, it is necessary to organize a structure capable of managing the data communication flow between the various components of the system.

The main features are described in the following steps:

1. The System must be able to carry out communication between the IoT devices (originated or not from external services) and the application itself. The data resulting from these devices can appear in different formats (structured, semi-structured or unstructured).
2. The application must be able to receive data quickly and securely and then present tools capable of processing and analysing it. Later, they can be provided for other resources (Front-end application, Data Knowledge) in a specific and known format: JSON.
3. Storage must be carried out in a database capable of quick and simple queries. At the same time, it must also adapt to the different types of data provided by the numerous sensors present.
4. After processing and storing the data, they must be made available to the end user through a WEB application that presents them in an intuitive and dynamic way.

5.2 PROPOSED APPROACH

During this study, several challenges were encountered in relation to the prototype to be built, as mentioned in section 5.1.1, among which the following stand out: the difficult ingestion of data, arising from the large amount that exists; their demanding Treatment and Processing, due to the frequent presence of anomalies or failures; the complex data storage; and finally the need for real-time visualization.

Therefore, the approach proposed to build this application, in order to facilitate the research, was the creation of an architecture based on a set of interconnected components, supported by Apache Kafka and MongoDB technologies, as well as by the concepts Microservices Architectures and Inter-Process Communication.

5.2.1 *Architecture Diagram*

This section provides a detailed explanation of the system's workflow. The diagram in Figure 21 illustrates the architecture proposed to solve the problem and demonstrates the sequence of interactions between the different components of the system, as well as the data flow that occurs in the process. This architecture was developed based on the proposal in article (Peixoto et al., 2020), where the main change is the replacement of Mirth Connect Engine Software for Apache Kafka. In addition, several microservices are used in order to support data analysis and visualization in real time.

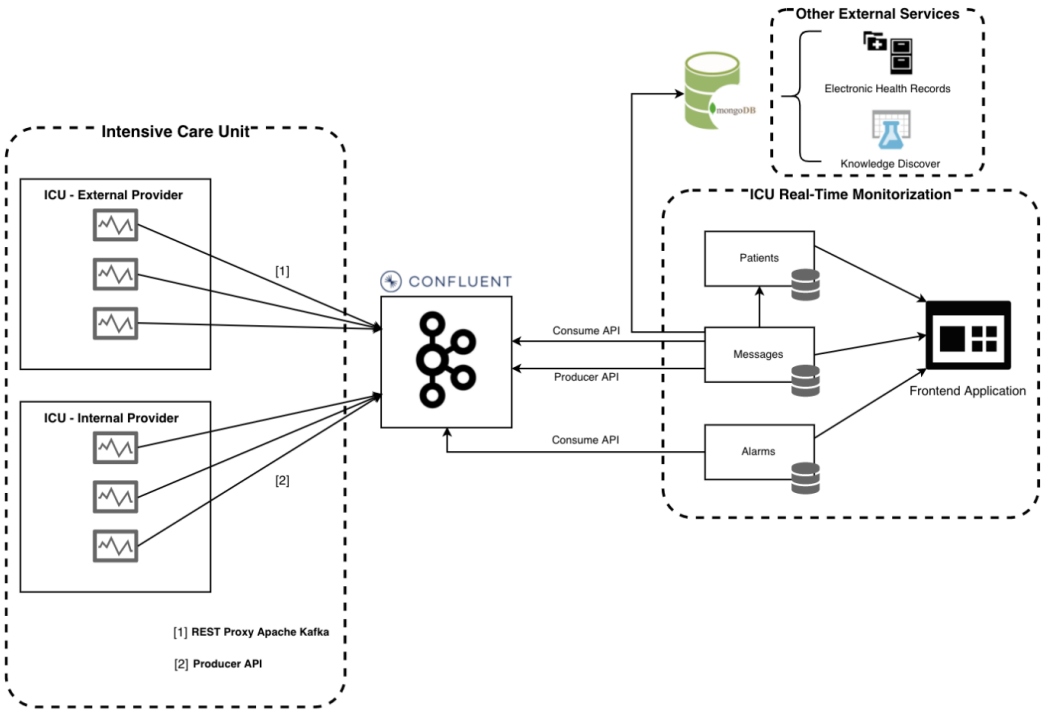

Figure 21: Architecture Diagram

This architecture is based on event streaming technology, Apache Kafka, and microservices architectures. On the left side of figure 21, the monitors used in the Intensive Care Units are represented, which may come from external suppliers, and which are constantly producing new data. Then, this data is sent to a topic previously generated in the Apache Kafka cluster called "icu_messages". For this there are two possibilities: the REST Proxy provided by Apache Kafka or the Publisher-Subscriber API. Apache Kafka brokers are located on the Confluent platform in order to get an overview of what is happening in the cluster.

Regarding the ICU Real-Time Monitorization(Backend and Frontend), which is the main focus of these thesis, there are a total of three microservices, as shown in Figure 21. The "Messages" microservice aims to consume the data published by bedside monitors and then validate the existence of any type of anomaly. In the event of an anomaly, the "Messages" microservice publishes a new message in an Apache Kafka topic called "icu_alarms".

Later, this message will be consumed by the "Alarms" microservice and persisted in a database for this purpose. Regardless of the existence of an anomaly, the microservice “Messages” stores all data in a NoSQL database. In addition, this same data is also sent, through WebSockets, to a Frontend application, which allows it to be visualized in real time in the form of a graph or table.

The application responsible for displaying patient information, real-time value graphs and alarms that have already occurred, will consume the data through REST API and through WebSockets.

Through the storage of data, it is possible that it allows to assist in other aspects, given that, in addition to presenting data in real time, it allows for future availability for Data Science and Machine Learning techniques.

6

DEVELOPMENT

In this chapter, the stages of development and implementation of the proposal of this dissertation are described. Initially, the technologies used are presented in addition to those presented in chapter 3. Subsequently, a description of each part of the developed application is exposed.

6.1 TECHNOLOGY USED

This section presents a brief description of the technologies and frameworks used in the PoC development, with the exception of the technologies already explored in section 3.

Java

Java (Java) is an object-oriented programming language and is employed in a variety of use cases. Codes developed in this language need a VM to be compiled, the JDK. Furthermore, it is common to use a tool to manage the use of dependencies and optimize builds. In the development of this project, the Java language was used in the development of microservices, together with JDK 11. The Apache Maven, a project management and comprehension tool software, was adopted and became essential for the use of libraries such as those provided by Apache Kafka.

Java Spring

Java Spring (Java Spring) is a Java platform that provides comprehensive infrastructure support for the development of Java applications. It aims to facilitate the development, thus exploring the concepts of Inversion of Control and Dependency Injection. The Java Spring platform has at its disposal several modules that simplify the development in the creation of Microservices and Web Applications. Spring is composed of modules such as Spring Security, which creates secure applications based on authentication and authorization; Spring Data, which flexibilizes the use of various databases such as NoSQL and SQL; and Spring for Apache Kafka, which simplifies the process of communicating with Apache Kafka. During the PoC, the use of Java Spring was adopted to perform the treatment, validation and processing of data published by bedside monitors simulators. Additionally, it also functioned as a REST API in the management of patients and beds.

VueJS

VueJS (VueJS) is a progressive framework created for building user interfaces. The core library is focused on the view layer only, and is easy to pick up and integrate with other libraries or existing projects. This JavaScript

framework was used in the development of the WEB application. The choice of technology was based on five main factors: Learning curve; Speed of development; Speed of the app; and Code structure and Documentation. This framework allowed the addition of libraries such as Vuetify, the complete UI library, and Apex Charts.

Microsoft SQL Server

Microsoft SQL (Microsoft SQL Server) is a Relational Database Management System (RDBMS) developed and marketed by Microsoft that supports a wide variety of transactions. In the course of this dissertation, it was used to store information regarding the patients, the anomalies that occurred and the base values.

6.1.1 *Development Support*

In order to simplify the application execution, management and distribution process, all components of the architecture were developed using a Docker Stack. Docker is an open source project for automating the deployment of applications as portable, self-sufficient containers that can run on the cloud or on-premises. (de la Torre et al., 2019) Each component of the architecture was built inside its own container, completely isolated from the others, to avoid coupling and potential conflicts. In developing this project, the following docker images were used:

Table 2: Version table of docker images used in the project

Docker Compose	**Software**	**Docker Image (image:version)**
Mongo-express[1]	Mongo	mongo:4.4.4
	Mongo-express	mongo-express:0.54.0
cp-all-in-one [2] [3]	Zookeeper	confluentinc/cp-zookeeper:5.5.1
	Apache Kafka Broker	confluentinc/cp-server:5.5.1
	Apache Kafka Control Center	confluentinc/ cp-enterprise-control-center:5.5.1
	Apache Kafka REST Proxy	confluentinc/cp-kafka-rest:5.5.1
	Sql Server	mcr.microsoft.com/mssql/ server:2019-latest

6.2 ICU SIMULATOR APPLICATION

In order to continue the study of the surrounding architectural methods and technologies, the need arose to create applications that could reproduce the simulation of bedside monitors that are present in intensive care units. The application created to reproduce the simulators has the objective of reading the files of a specific

1 https://hub.docker.com/_/mongo
2 https://github.com/confluentinc/cp-all-in-one
3 An infrastructure on Confluent was used for testing. This docker-compose was only used for the development and not the production.

patient from the MIMIC dataset, separating the data from each file, creating the message and, subsequently, sending it to the Apache Kafka cluster.

For the development of the simulation application, an object-oriented language, namely Java using JDK 11 together with Maven, was used. The choice of language was based on the fact that the support/documentation language used in Apache Kafka is Java.

In order to run the same application several times and for different patients, a mechanism was created that internally differentiates patients. To do so, when starting the application, it is necessary to indicate the path relative to the patient's folder in MIMIC Database v1.0. It is possible to indicate the path through arguments or, in the event that there are no arguments, indicate the path at the beginning of the execution.

After validating the path of a specific patient, it is run through to start the process of reading each file. The files are in the format indicated above (section 4.2), where the first line has the indication of the time of the first measurement, the patient indicator and the name of the file being read. Since initially only the timestamp of the first measurement in the file was present, it was necessary to find a formula to calculate the timestamp of the remaining measurements. Each file has a total of ten minutes of measurements and the number of data may vary. The formula found was: 1, where x corresponds to the number of registers (lines) of the file and y the value in milliseconds that must be added to the timestamp of message to message. In addition, the values can be of different types, so it was necessary to carry out several validations, namely, with regard to warnings, the composition of sensor values and alarms. At the end, the message is constructed in the following format:

$$y = \left(1 + \frac{600 - x}{x}\right) * 1000 \tag{1}$$

Afterwards, with the message already created, it is necessary to send it to the Apache Kafka cluster. Apache Kafka offers two ways of sending messages: one through a REST API (Apache Kafka Rest Proxy); and another through the Producer/Consumer APIs.

Apache Kafka REST Proxy works as a REST API, which consists of sending an Hypertext Transfer Protocol (HTTP) request with the POST method to a specific endpoint. This endpoint varies depending on the topic the message is sent to *(https://ip_address/topic/topic_name)*. In addition, it is necessary to add headers to the request, namely Accept and Content-Type. The request body must follow a specific format. Listing 6.1 represents the example of sending a message through the Apache Kafka REST Proxy.

```
POST /topics/icu_messages HTTP/1.1
Host: http://localhost:8082
Content-Type: application/vnd.kafka.binary.v2+json
Accept: application/vnd.kafka.v2+json, application/vnd.kafka+json, application/
    json
Body:
{
  "records": [
    {
      "key": "061",
      "value": "patient_id timestamp file\nHR 70\nRESP 65\nABP 57 90 41"
    },
  ]
}
```

Listing 6.1: Example of a Hypertext Transfer Protocol (HTTP) request to Apache Kafka through Representational State Transfer (REST) Proxy.

The other option, the most common one, works using the Apache Kafka library. For this, it is necessary to add this library to the project and perform the initial configuration indicating several properties, such as the IP Address of one of the Brokers in the Cluster, as well as the part related to authentication. After successful configurations, it is possible to use the various methods available in the library, such as the methods of class *KafkaProducer*, namely the *java.util.concurrent.Future<RecordMetadata> send(ProducerRecord<K,V> record, Callback callback)*. This method allows to send a message to the Apache Kafka cluster asynchronously. Since the messages can be distributed in different partitions of the Apache Kafka topic, it is necessary to add the Key, in order to guarantee the reading order. The Key used refers to the ID of each patient.

6.3 ANALYSIS AND PROCESSING APPLICATIONS

To perform the reception, analysis, processing and availability of application data, the Java programming language was used together with the Spring Boot framework. The choice of this framework derives from the ease of integration of different modules (Apache Kafka, Databases, among others) and also to the use of concepts such as Inversion of Control and Dependency Injection that simplify and protect the way we program.

The data is published in the Apache Kafka cluster topics through the producers, which in the scope of this project are the bedside monitor simulators. Then, the consumers responsible for validating and processing the data - Message Service, start consuming them.

Through the use of the Spring Boot Framework, it is possible to connect to the Apache Kafka cluster in a simple and fast way, by installing the Spring for Apache Kafka module[4]. Subsequently, and with the connection made to the cluster, it was necessary to create *KafkaListerner's* for the topics in which the information is intended

4 https://spring.io/projects/spring-kafka

to be consumed. Listing 6.2 represents an example of a *KafkaListener* used during development for the topic named "icu_messages".

```
@KafkaListener(topics = "icu-messages", groupId = "icu-group")
public void consumeMessages(String message) throws JsonProcessingException {
    ObjectMapper mapper = new ObjectMapper();
    Message m = mapper.readValue(message, Message.class);
    this.messageService.analyzeMessage(m);
}
```

Listing 6.2: Example of a KafkaListerner in Java Spring Boot

After receiving the data, there was the need to process the message in order to be able to verify the existence of anomalous values or alarms. For this purpose, tools such as ModelMapper were used, which allows mapping objects in a simple and practical way. After that, the treated message is stored in the MongoDB database and sent in JSON format through WebSockets to the Web interface. In the event of anomalies, it was important that they be stored permanently. For this, they were published in a topic of the Apache Kafka cluster with an extensive durability. Later, they will be consumed and persisted by the "Alarms" microservice. The message is sent to the web interface in the following format:

```
{
   patientId: "037",
   timestamp: "1994-08-14T09:27:45",
   file: "03700001.txt",
   values: [
        {
            typeOfValue: "RESP",
            values: [46]
        },{
            typeOfValue: "ABP",
            values: [64,82,53]
        },{
            typeOfValue: "HR",
            values: [122]
        },{
            typeOfValue: "SpO2",
            values: [0]
        },{
            typeOfValue: "PULSE",
            values: [0]
        }
   ],
   isAlarm: false,
}
```

Listing 6.3: JavaScript Object Notation (JSON) structure with a patient's data

Finally, for all messages in which the existence of an anomaly is not verified, an expiration TTL is applied, which can be changed by the user through the Single Page Application (SPA).

6.4 FRONTEND APPLICATION

Regarding the WEB client, a SPA was used, where users can observe in real-time the graphics related to the sensors of the bedside monitors. This was developed using the JavaScript programming language together with the VueJS Framework. The choice of these technologies was based not only on the advantages already presented above (section 6.1), but also due to the author's participation in a research lab that employed these technologies. Additionally, and due to the speed of styling compared to other tools or Cascading Style Sheets (CSS) in its pure state, the Vuetify library was applied. Furthermore, and in order to develop intuitive, responsive and customizable charts, the Apex Charts library was chosen.

The application responsible for frontend communicated with the backend services through two methods: REST APIs and WebSockets. Since not all browsers support the WebSockets technology, it was necessary to overcome this problem using an emulator, in this case SockJS. In this sense, it was essential to make changes on the server side, using the Simple Text Orientated Messaging Protocol (STOMP) protocol, which implied the use of a new library on the application side responsible for demonstrating the graphics. Webstomp-client was the library chosen.

For the construction of the charts and, as mentioned above, the Apex Charts library was used. It offers different types of charts, such as: Line Charts, Bar Charts, Area Charts, among others. In this specific case, we resorted to the use of Line Charts, adding DataLabels and Annotations. DataLabels are necessary to specify the values in more detail in a certain period, while Annotations serve to indicate the limit of values in the acceptable level.

In the following sections, the result of the application responsible for displaying the sensor graphics for each patient registered in the system will be demonstrated.

6.4.1 *Demonstration*

Using the microservices and the technologies discussed above, and together with the mockups developed (Appendix A), it was possible to develop the application as shown in the following figures.

Figure 22 illustrates the Home page of the application. On this page all the beds registered in the ICU are represented. Each card in the central layout is composed by its ID and when pressed it opens a new page with more details. In addition, on this page it is also possible to register new beds through an ID.

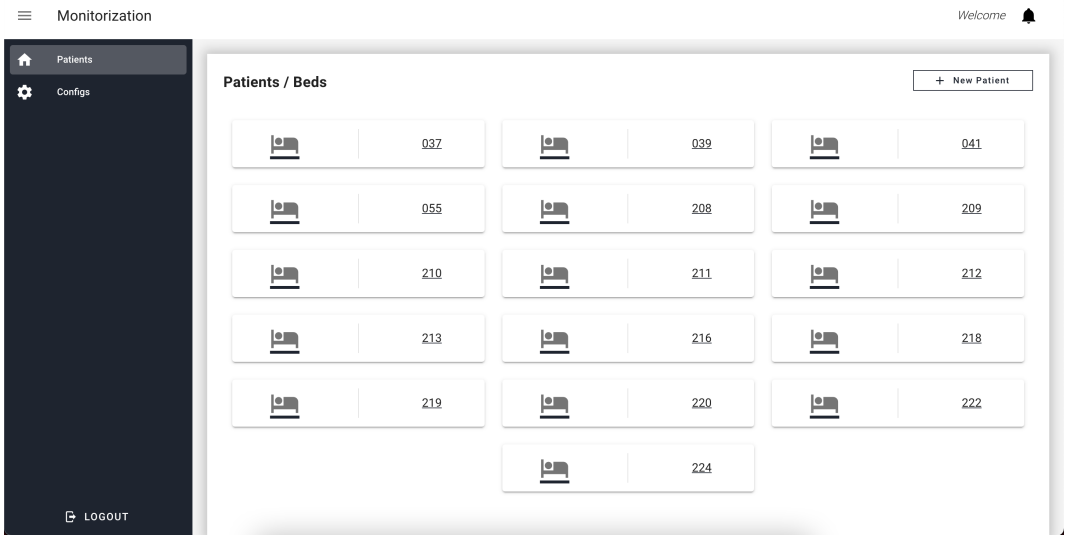

Figure 22: Main page of the application

After choosing a bed represented in figure 22, a new page will open where all the patient's details, charts depicting the patient's current health data and a table with the history of occurred alarms, will be represented. In the event that there is no data the page will appear as shown in figure 23 and 24.

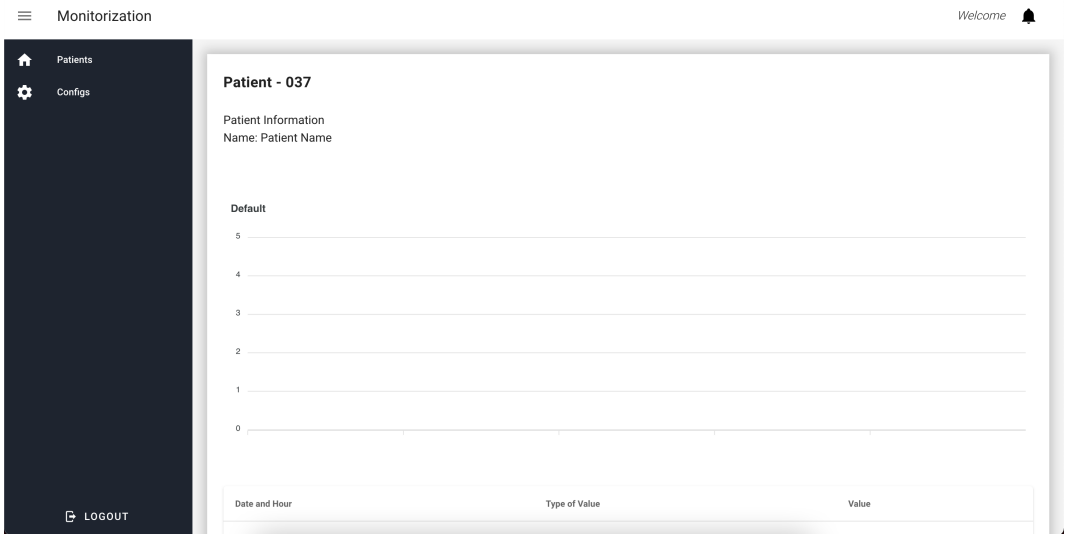

Figure 23: Page responsible for a patient's information

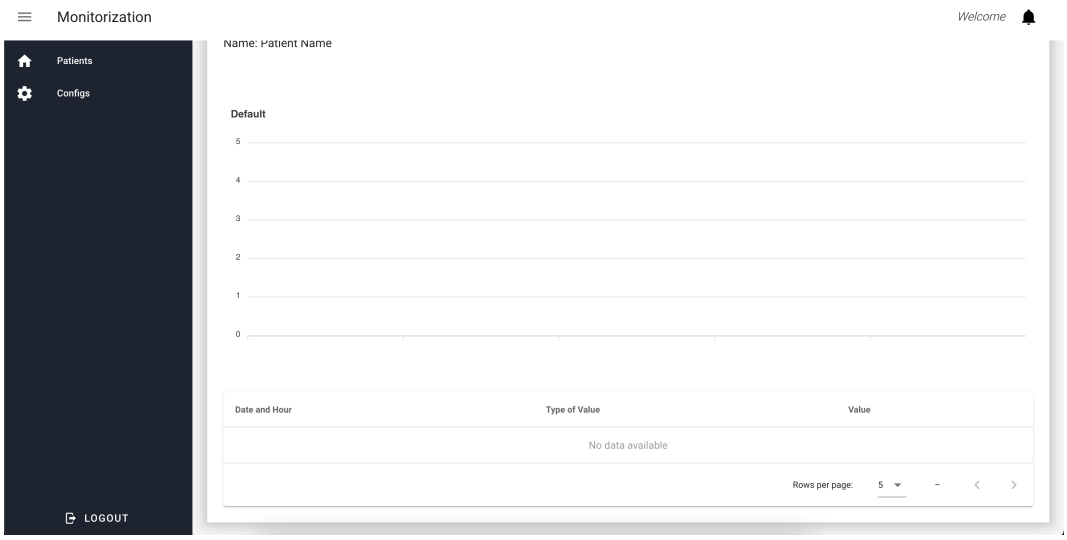

Figure 24: Cont. - Page responsible for a patient's information

In the case the patient already has data the page will be relatively different since an additional section with separate data will appear. Figure 25 represents the data of a patient (in this case 037) in real-time via a manipulated graph and cards.

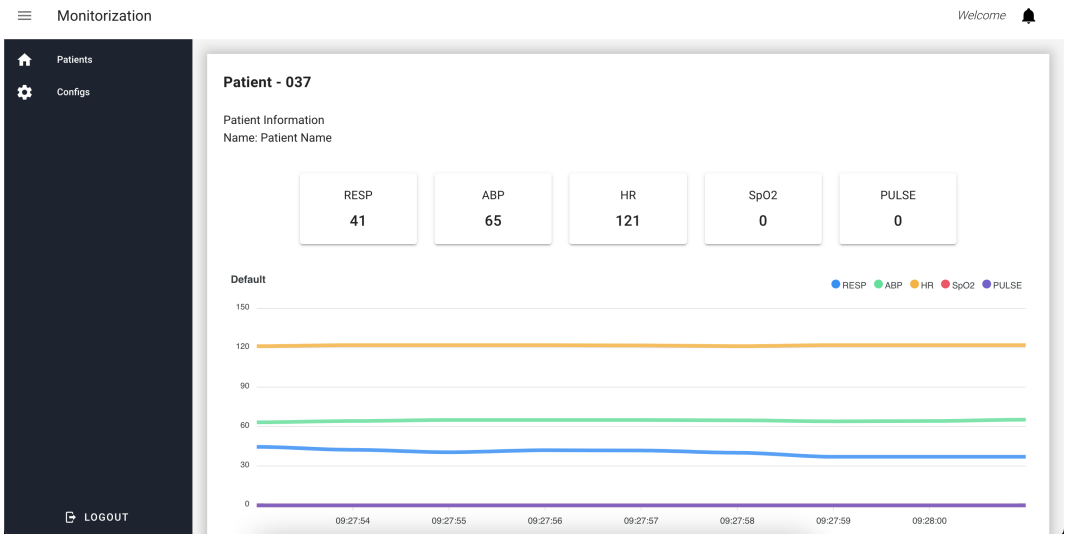

Figure 25: Page responsible for the information of a patient with data

7

EVALUATION

This chapter aims to evaluate the solution that was portrayed in the previous chapters. The test scenarios performed on the PoC, the results from these tests and the discussion, are described. Finally, a SWOT analysis of the main themes of this dissertation is prepared.

7.1 TESTS

As mentioned earlier, this dissertation aims to understand what the advantages and disadvantages of using microservices architectures and Apache Kafka in healthcare are, specifically in the ICU. To better understand these advantages and disadvantages, two types of tests were devised:

- **Performance Testing**, which aims to evaluate the developed prototype at the level of reception, availability and message handling;
- **Message format testing**, which aims to see if Apache Kafka can receive different types of formats such as JSON, TXT, Health Level 7 (HL7) and others.

For both types of tests the infrastructure described in chapter 4 was used, and specifically for the performance tests, the MIMIC Database dataset was also used.

7.1.1 *Performance*

This subsection contains the performance tests performed on the developed prototype, along with the MIMIC Database dataset. For the development of these tests, a specific script was created to run several ICU simulators simultaneously. Tests were performed with 1, 3, 5, 20 and 40 simulators. To elaborate the discussion, during the tests different types of metrics were collected through the Apache Kafka dashboard and MongoDB. The metrics collected were:

- Production (bytes/min);
- Consumption (bytes/min);

- Storage;
- Messages save;
- Messages saved/sec.

Image 26 represents the dashboard made available by Confluent to receive the Production (bytes/min), Consumption (bytes/min) and Storage data.

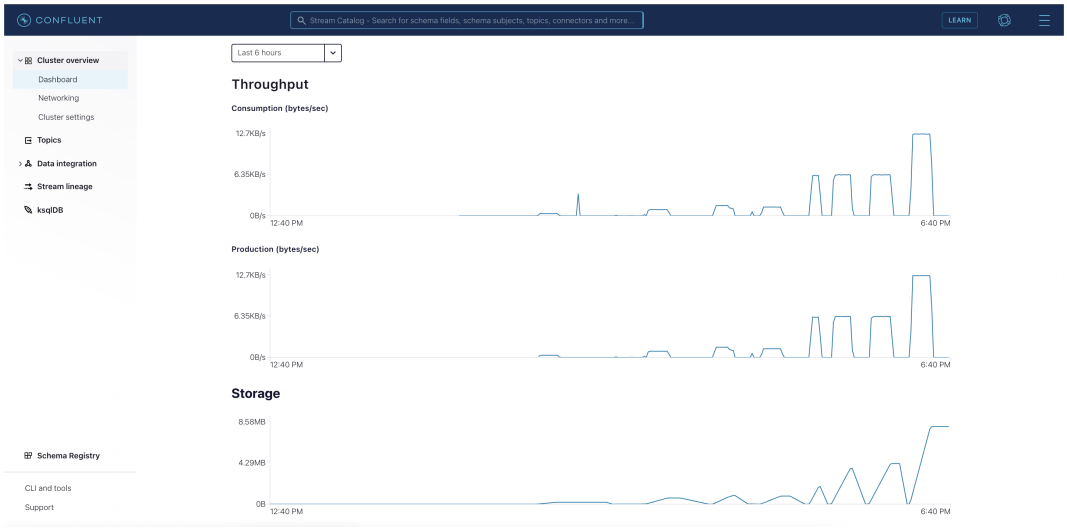

Figure 26: Confluent Center Dashboard

Table 3 represents the data obtained with 1 ICU simulator. It is possible to see that the production values and the consumption values follow each other at the bytes/min level. That is, there was no difficulty in receiving the data from the consumers. Besides the production and consumption of data, and since it was only a simulator, the number of messages received and the average per second was as expected.

Table 3: Performance tests with 1 ICU simulator

Production (bytes/sec avg over 1 min)	Consumption (bytes/sec avg over 1 min)	Storage	Messages	Messages/sec
116	116	202.43 kb	614	0.9
327	383			
327	338			
332	327			
327	332			
332	327			
327	332			
332	327			
327	327			
330	375			
327	327			
51	51			

Table 4 illustrates the data obtained from the performance behavior of the prototype developed with 3 ICU simulators. Once again, the production and consumption values were similar, which means that there was no difficulty in making the data available for processing. The number of messages stored was higher in relation to the previous one, and the number of messages stored per second corresponds to the number of sends, i.e., 3 simulators correspond to 3 messages stored per second.

Table 4: Performance tests with 3 ICU simulator

Production (bytes/sec avg over 1 min)	Consumption (bytes/sec avg over 1 min)	Storage	Messages	Messages/sec
835	846	653.21 kb	2045	3.1
959	948			
954	965			
959	948			
949	965			
966	949			
949	949			
965	965			
947	947			
948	964			
965	949			
507	507			

According to the data obtained in the 5 simulators test (Table 5) it was possible to see, once again, that the prototype received data efficiently, with no significant differences between production and consumption. The number of messages stored was higher, reaching 3141 with an average of 4.8 messages per second.

Table 5: Performance tests with 5 ICU simulator

Production (bytes/sec avg over 1 min)	**Consumption (bytes/sec avg over 1 min)**	**Storage**	**Messages**	**Messages/sec**
427	441	853.62 kb	3141	4.8
1.33k	1.32k			
1.31k	1.33k			
1.33k	1.32k			
1.32k	1.32k			
1.33k	1.33k			
1.32k	1.32k			
1.32k	1.33k			
1.33k	1.32k			
1.32k	1.31k			
1.32k	1.33k			
595	586			

Observing Table 6, which represents data from 20 simultaneous ICU simulators, it is also possible to conclude that the developed prototype can easily respond to production and consumption values. The number of messages stored in the database is about 13381, which averages to 20.3 messages stored per second. This is a very considerable value for a ICU in Portugal.

Table 6: Performance tests with 20 ICU simulator

Production (bytes/sec avg over 1 min)	Consumption (bytes/sec avg over 1 min)	Storage	Messages	Messages/sec
5.97k	5.99k	4.19 mb	13381	20.3
6.22k	6.22k			
6.23k	6.29k			
6.28k	6.22k			
6.23k	6.28k			
6.29k	6.24k			
6.22k	6.23k			
6.24k	6.27k			
6.28k	6.24k			
6.22k	6.27k			
6.27k	6.25k			
3.12k	3.08k			

The last performance test was the most demanding and involved 40 ICU simulators (Table 7) running simultaneously. Despite the high number of messages stored, 25563, which gives an average of 38.7 messages stored per second, the prototype had no difficulty regarding the difference of bytes in production and in consumption. Table 7 represents the values collected during this experiment.

Table 7: Performance tests with 40 ICU simulator

Production (bytes/sec avg over 1 min)	Consumption (bytes/sec avg over 1 min)	Storage	Messages	Messages/sec
4.01k	4.1k	8.01 mb	25563	38.7
12.5k	12.44k			
12.49k	12.51k			
12.5k	12.51k			
12.5k	12.48k			
12.48k	12.5k			
12.52k	12.5k			
12.51k	12.52k			
12.5k	12.46k			
12.5k	12.49k			
12.47k	12.49k			
7.76k	7.72k			

Briefly, and with the data collected and analyzed during the various experiments performed, it is possible to conclude that the prototype can easily meet the requirement of 40 ICU simulators. The producer and consumer values during the various test scenarios are always similar, i.e. there is no difficulty regarding the data being produced and consumed. This makes it possible to always have the most up-to-date data in real-time. The storage values shown are related to Apache Kafka's internal storage, which gives the prototype extra storage security.

In the event that more than 40 ICU simulators were needed simultaneously and the consumer could not handle the data demand, it would only be necessary to add one more consumer and create a consumer group in order to split the reading of the received data. The strategy of coordinating the consumers would be done simply and automatically by Apache Kafka using Zookeeper.

7.1.2 *Message Format*

As in other areas, healthcare can receive data in different formats, so it was necessary to check whether Apache Kafka can support them. To do this, messages were sent in several formats, such as: Txt, JSON, XML, and HL7 v2.0. HL7 is an international community of health information experts that collaborate to develop standards for the exchange of health information and health systems interoperability. (Bender and Sartipi, 2013)

The following table (8) shows the result of the tests when sending messages with different formats.

Table 8: Message Format tests

Message Format	Evaluation
Txt	Acceptable
JSON	Acceptable
XML	Acceptable
HL7	Acceptable

Although Apache Kafka accepts the different types of text format, it is necessary to configure the consumers to be able to interpret the data later.

7.2 Strengths, Weaknesses, Opportunities and Threats (SWOT) analysis

SWOT analysis is a tool that aims to identify the strengths, opportunities, weaknesses, and threats of the object of study, through a rigorous analysis.

As **strengths** one can identify:

- **Scalable**

One of the key factors that decide the success of an architecture is the ability with which it handles the increased load/users. This increase in load/users must be handled in a simple, sustainable, and continuous

way. With the use of Microservices and Container technology, scaling the application/architecture to meet the increased load is possible by duplicating microservices and using a Load Balancer.

Another important factor is the eventuality that new features need to be added to the application. Solving this problem in the developed architecture is simple. It can be applied to an existing microservice or, if justified, opt to create a new microservice.

Regarding the growth of new IoT devices (Bedside Monitors, etc) or the addition of new external services, the developed architecture can respond to this growth by using Apache Kafka. If the existing Apache Kafka cluster is having trouble receiving and delivering data, it is only necessary to add a new Kafka Broker to the Cluster. Eventually, to be able to consume the data faster it may be useful to create consumer clusters.

- **Reliable**

Considering healthcare a critical market when it comes to data loss, this architecture becomes reliable due to the fact that it uses Apache Kafka as a data intermediary. Apache Kafka technology provides several ways to ensure that data can be received and made available. When it comes to receiving data, coming from IoT devices, it is possible to use Acks to prevent possible failures. Acks allow the Kafka Cluster to resend a message if it fails to receive it, until it succeeds. For the provision of messages, Apache Kafka provides three ways to perform the read commits. It is also possible to choose when the commit is performed, whether at the beginning or at the end of the provision of the message.

In addition to the security methods of writing and reading messages, Apache Kafka allows messages to be stored in a topic for a period defined by the administrator. This way, if an error happens, it is possible to re-consume the messages.

Another important aspect regarding reliability, is that Apache Kafka stores the same message in different partitions of the topic. These partitions are separated among the Kafka Brokers of the Kafka Cluster. That is, if one Kafka Broker is offline, the message is still available in another Kafka Broker for reading.

- **High Availability**

One of the main characteristics of Microservice architectures is that microservices are totally independent from each other, with no dependencies between them. If one microservice is unavailable the others continue to work, thus not putting the application completely offline.

Another factor is that both Apache Kafka and Microservices architectures must be based on Cloud Servers. This way it is possible to have clusters in different geographical locations.

Finally, Apache Kafka duplicates the messages received by different Kafka Brokers. If any Kafka Broker goes offline, there is guaranteed to be no problem due to the message still being present in the Cluster. This requires more than one Kafka Broker in a Kafka Cluster.

- **High Throughput**

In addition to the reliability aspect, healthcare has another important aspect, which is the speed in which data is made available to the end user. It is important that the data received by IoT devices is available quickly, both for

checking anomalies/faults and for visualization. Apache Kafka allows the receival of data from various devices at high speeds as demonstrated in the previous section of this chapter.

- **Diversity of Data**

There are several types of formats in which data can be received. By using Apache Kafka to receive data from IoT devices, it is possible that data comes in several formats, such as TXT, JSON, HL7 v2, among others. It is only necessary to, subsequently, make changes at source code level in the part related to the Consumers, either through external libraries or own converters.

However, the architecture naturally has some weaknesses associated with it. **Weaknesses** include:

- **Complexity**

Regardless of the numerous advantages associated with microservices architectures and Apache Kafka, it is possible to realize that it is a complex concept and technology. The cost of learning and development is higher than for example in a Monolithic architecture because it involves the use of more concepts such as: the communication processes between Microservices; CI and CD processes for DevOps automation; and the use of Cloud to put the application in production. In addition, and especially in Apache Kafka, because it is a complex technology with a large ecosystem, sometimes finding useful solutions to problems that may occur becomes complicated.

- **High Costs**

Since it is a distributed architecture and must be developed based on Cloud, it is natural that the costs related to the DevOps component are higher. Additionally, it is useful to use a tool to support the Apache Kafka cluster, as is the case of Confluent's tool, which allows us to simplify some of the existing complexity but has additional costs, which depending on the number of Brokers, could be high.

From an external perspective, the **Opportunities** identified are:

- **Reuse of microservices**

The use of microservices architecture allows microservices to be developed and used independently. That said, there is the opportunity in the future, and if necessary, to reuse the same microservice for another product that may be developed. This way it avoids the development of code that was already developed.

- **Add new functionalities through the Apache Kafka ecosystem**

Since Apache Kafka, in addition to what is presented in this dissertation, has a vast ecosystem, it is possible to use some of its additional products to assist in troubleshooting. The use of Apache Kafka Connect and Apache Kafka Streams are two examples of Apache Kafka products and together they can serve to assist in the creation of Extract, transform, load (ETL) and Machine Learning (ML) processes.

Among the possible **Threats**, we can consider:

- **Lack of technical resources**

Since microservices architectures are a distributed architecture where microservices are developed and designed independently, it is important to assure that there are several teams developing different microservices of an application. Otherwise, if there is a lack of resources, microservices architectures may not pay off in the end, since it will take more time and more work from a small team.

- **Lack of investment**

As described earlier, one of the weaknesses of using microservices architectures and Apache Kafka is the high cost of development and production. This weakness can later mean lack of investment by government policies and put the development and design of the solution in question.

The points described above represent the SWOT analysis performed on the use of microservices architectures and Apache Kafka. In each point a brief explanation is addressed in order to understand why it is present. The following table represents the SWOT analysis described.

Table 9: Strengths, Weaknesses, Opportunities and Threats (SWOT) Analysis

	Helpful	Harmful
Internal origin	• Scalable • Reliable • High Availability • High Throughput • Diversity of data	• Complexity • High Costs
External origin	• Reuse of microservices • Add new functionality through the Apache Kafka ecosystem	• Lack of technical resources • Lack of investment

8

CONCLUSIONS AND FUTURE WORK

Throughout this dissertation, the studies performed and the project developed were listed and described. The project consisted in the development of a PoC with the ability to simulate the visualization of patient data in a real-time ICU. The main focus of the PoC is to validate the concepts and technologies described in this document, namely Apache Kafka and microservices architectures, applied to healthcare. An analysis of the advantages and disadvantages inherent to its use was duly performed.

Additionally, information from a real dataset, populated with various health metrics and alarms, was used to ensure the trustworthiness of the application. In order to validate the impact of the PoC, several types of load and performance tests were developed, as well as a SWOT analysis. The results obtained during the tests exceeded expectations, since, as the number of ICU simulators representing bedside monitors increased, no difficulty in receiving and providing patient information was verified. If this difficulty is verified, which may occur with a high number of simulators, it is possible, through the resources made available by the concepts and technologies studied, to resolve these problems in a simple and fast manner. However, and despite the results having exceeded expectations, it is important to emphasize that depending on the type of product being developed, specifically for healthcare, it is important to measure if the technical complexity and financial investment justify its use.

The study and development of the PoC allowed for the validation of the employment of the concepts and technologies of this dissertation in healthcare. This validation is promising, opening the way for the development of software in healthcare, specifically in areas where the flow of essential real-time data is critical (namely the ICU that was the object of study of this dissertation).

8.1 ACHIEVED OBJECTIVES

This dissertation had as its main goal to study the applicability of microservices architectures along with Apache Kafka in healthcare, more specifically in the ICU. This study involved understanding the concepts and technologies, the main contexts of use, the performance evaluation, and finally the characterization of the advantages and disadvantages of its use. The objectives initially proposed within the scope of this dissertation were achieved, namely:

- A literature review of the main concepts associated with the theme was performed;
- Understanding Apache Kafka and its main features;

- Designed, planned, and implemented a PoC for real-time data visualization of an ICU, using real data available in a dataset;
- Evaluation of the results obtained in PoC;
- Description of a set of advantages and disadvantages of using the dissertation topics.

From this investigation, it became clear that microservices architectures and Apache Kafka are a viable alternative for the processing of messages in healthcare, specifically in the ICU. This conclusion derives from the tests results presented above (section 7.1). From these tests it was possible to verify an array of characteristics, namely, security, performance and scalability.

8.2 DIFFICULTIES AND LIMITATIONS ALONG THE WAY

Throughout this dissertation several difficulties were experienced, starting with the scarcity of articles related to the dissertation topics, Microservices Architectures and Apache Kafka, specifically in healthcare.

For the development of the PoC, it was necessary to use a real dataset with data related to an ICU. Since healthcare is an area where confidential patient data is present, there are few public datasets, so in most cases it is necessary to acquire a license or make a request to the authors to obtain access. This difficulty was overcome by using an older dataset.

Finally, the main limitation and difficulty is associated with the infrastructure to perform the tests. Due to the complexity of Apache Kafka, the creation of a cloud cluster using the main providers (Google Cloud, Amazon Web Services, etc) became a limitation so it was necessary to resort to platforms such as Confluent for the use of Apache Kafka in the cloud. The Confluent platform provides a monetary stipend for testing, but it had to be managed to not exceed the initial value.

8.3 FUTURE WORK

Despite having achieved all the objectives initially proposed, there are aspects that may be considered in the future as a way to complement the work developed.

One of the topics addressed during the dissertation, Apache Kafka, is a recent tool that has been adopted by several companies in different ways. That said, it is important to identify several issues to be addressed in the future, namely:

- **Creating ETL pipelines in the cloud**: Studying the use of Apache Kafka in creating ETL pipelines, using tools from the Apache Kafka ecosystem such as Apache Kafka Connect and Apache Kafka Stream. Through these two tools it is possible to extract, transform and load data in a simple way;
- **Online machine learning**: Using Apache Kafka, it allows easy incorporation with other tools, so it is important to understand the advantages associated with this. The study of Apache Kafka along with

Apache Spark and Spark Streaming is critical to understanding the opportunities of performing Online Prediction (Machine Learning) in healthcare.

Regarding the other theme of the dissertation, Microservices Architectures, there is also future work that can be explored, such as:

- **Migration from Monolithic Architectures to Microservices in production**: In a production environment, study the real impact, exposing the problems and difficulties that exist in the migration of a Software based on monolithic to a Microservices architecture. Furthermore, understand which are the best ways to make this migration;
- **Applying tools like kubernetes**: For the DevOps area there are several challenges, so it is crucial to understand the advantage of using Docker together with Kubernetes. Kubernetes aims to automate the deployment and management of containerized applications.
- **Authentication (Single sign-on)**: Study the advantages of using single sign-on in microservices architectures.

BIBLIOGRAPHY

Apache Kafka. Apache Kafka. URL https://kafka.apache.org/.

Babylon Health. About, 2013. URL https://www.babylonhealth.com/about/.

D. Bender and K. Sartipi. Hl7 fhir: An agile and restful approach to healthcare information exchange, 06 2013. URL https://ieeexplore.ieee.org/abstract/document/6627810.

NEJM Catalyst. Healthcare big data and the promise of value-based care, 01 2018. URL https://catalyst.nejm.org/doi/full/10.1056/CAT.18.0290.

Veronica Combs. Humana uses azure and kafka to make healthcare less frustrating for doctors and patients, 11 2020. URL https://www.techrepublic.com/article/humana-uses-azure-and-kafka-to-make-healthcare-less-frustrating-for-doctors-and-patients/.

Confluent. Humana adopts event streaming and interoperability using confluent, 2020. URL https://www.confluent.io/customers/humana/.

Sabyasachi Dash, Sushil Kumar Shakyawar, Mohit Sharma, and Sandeep Kaushik. Big data in healthcare: management, analysis and future prospects. *Journal of Big Data*, 6, 06 2019. doi: 10.1186/s40537-019-0217-0.

Cesar de la Torre, Bill Wagner, and Mike Rousos. .net microservices: Architecture for containerized .net applications, 06 2019. URL https://docs.microsoft.com/en-us/dotnet/architecture/microservices/.

Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: Yesterday, today, and tomorrow. *Present and Ulterior Software Engineering*, pages 195–216, 2017. doi: 10.1007/978-3-319-67425-4_12. URL https://link.springer.com/chapter/10.1007%2F978-3-319-67425-4_12.

Forbes. Humana hum, 2021. URL https://www.forbes.com/companies/humana/?sh=5fce4cf04390.

Navdeep Singh Gill. Service-oriented architecture vs. microservices | the comparison, 01 2020. URL https://www.xenonstack.com/insights/service-oriented-architecture-vs-microservices.

Romana Gnatyk. Microservices vs monolith: Which architecture is the best choice for your business?, 10 2018. URL https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/.

Ary L. Goldberger, Luis A. N. Amaral, Leon Glass, Jeffrey M. Hausdorff, Plamen Ch. Ivanov, Roger G. Mark, Joseph E. Mietus, George B. Moody, Chung-Kang Peng, and H. Eugene Stanley. Physiobank, physiotoolkit, and physionet: Components of a new research resource for complex physiologic signals. *Circulation [Online]*, 101:e215–e220, 06 2000. doi: 10.1161/01.cir.101.23.e215. URL https://physionet.org/content/mimicdb/1.0.0/.

Dana Groce. How mongodb helped a healthcare firm scale horizontally - dzone big data, 09 2014. URL https://dzone.com/articles/leaf-in-the-wild-doctoralia-scales-patient-service.

Fawzya Hassan, Masoud E., and Radhya Sahal. Real-time healthcare monitoring system using online machine learning and spark streaming. *International Journal of Advanced Computer Science and Applications*, 11, 2020. doi: 10.14569/ijacsa.2020.0110977.

Matt Heusser. Orchestration vs. choreography in microservices architecture, 08 2020. URL https://searchapparchitecture.techtarget.com/tip/Orchestration-vs-choreography-in-microservices-architecture.

Marilena Ianculescu and Adriana Alexandru. Microservices - a catalyzer for better managing healthcare data empowerment. *Studies in Informatics and Control*, 29:231–242, 07 2020. doi: 10.24846/v29i2y202008.

Java. Java. URL https://www.oracle.com/pt/java/.

Java Spring. Java Spring Framework. URL https://spring.io/.

Brendan Lawlor, Richard Lynch, Micheál Mac Aogáin, and Paul Walsh. Field of genes: using apache kafka as a bioinformatic data repository. *GigaScience*, 7, 04 2018. doi: 10.1093/gigascience/giy036.

James Lewis and Martin Fowler. Microservices, 03 2014. URL https://martinfowler.com/articles/microservices.html.

Stéphane Maarek. The kafka api battle: Producer vs consumer vs kafka connect vs kafka streams vs ksql !, 10 2019. URL https://medium.com/@stephane.maarek/the-kafka-api-battle-producer-vs-consumer-vs-kafka-connect-vs-kafka-streams-vs-ksql-ef584274c1e.

Zoran Maksimovic. Mongodb 3 succinctly, 06 2017. URL https://s3.amazonaws.com/ebooks.syncfusion.com/downloads/MongoDB_3_Succinctly/MongoDB_3_Succinctly.pdf.

Akhlaq Malik. Etl with kafka, 03 2018. URL https://blog.codecentric.de/en/2018/03/etl-kafka/.

Microsoft SQL Server. Microsoft SQL Server. URL https://www.microsoft.com/en-us/sql-server/.

MongoDB - JSON and BSON. JSON and BSON. URL https://www.mongodb.com/json-and-bson.

MongoDB - TTL Indexes. TTL Indexes. URL https://docs.mongodb.com/manual/core/index-ttl/.

MongoDB for Healthcare. MongoDB for Healthcare. URL https://www.mongodb.com/industries/healthcare.

George B Moody and Roger G Mark. The mimic database, 1992. URL https://physionet.org/content/mimicdb/.

Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka : the definitive guide : real-time data and stream processing at scale*. O'reilly Media, 1 edition, 2017a.

Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka: The definitive guide: Real-time data and stream processing at scale*. O'reilly Media, 2017b.

Andre Newman. Is your microservice a distributed monolith?, 09 2020. URL https://www.gremlin.com/blog/is-your-microservice-a-distributed-monolith/.

Richard Noble and Francesco Nobilia. One key to rule them all, 05 2019. URL https://www.confluent.io/kafka-summit-lon19/one-key-to-rule-them-all/.

Ajay Ohri. Types of big data: Simplified (2021), 03 2021. URL https://www.jigsawacademy.com/blogs/big-data-analytics/types-of-big-data#Semi-Structured-Data.

Mike P. Papazoglou and Willem-Jan van den Heuvel. Service oriented architectures: Approaches, technologies and research issues. *The VLDB Journal*, 16:389–415, 03 2007. doi: 10.1007/s00778-007-0044-3.

Akash Paul. Microservices vs soa: What's the difference?, 02 2020. URL https://www.tiempodev.com/blog/microservices-vs-soa/.

Hugo Peixoto, Tiago Guimarães, and Manuel Filipe Santos. A new architecture for intelligent clinical decision support for intensive medicine. *Procedia Computer Science*, 170:1035–1040, 2020. doi: 10.1016/j.procs.2020.03.077.

Kousik Rajendran. Mongo db and its application (a case study of mongodb in healthcare), 06 2012. URL https://kousikraj.me/2012/06/05/mongo-db-and-its-application-a-case-study-of-mongodb-in-healthcare/.

Chris Richardson and Floyd Smith. Microservices from design to deployment, 2016. URL https://www.nginx.com/resources/library/designing-deploying-microservices/.

Hengky Sanjaya. Monolith vs microservices, 03 2020. URL https://medium.com/hengky-sanjaya-blog/monolith-vs-microservices-b3953650dfd.

SAS. What is big data and why it matters, 2018. URL https://www.sas.com/en_us/insights/big-data/what-is-big-data.html.

Matthias J. Sax. Apache kafka. *Encyclopedia of Big Data Technologies*, pages 1–8, 2018. doi: 10.1007/978-3-319-63962-8_196-1.

Yasmine Ska and Jamine P. A study and analysis of continuous delivery, continuous integration in software development environment. *International Journal of Emerging Technologies and Innovative Research*, 6:96–107, 09 2019. URL http://www.jetir.org/papers/JETIRDD06019.pdf.

Andrew Stevenson and Jeremy Frenay. Big data ldn 2019: Freeing up engineering and infrastructure resources to scale with dataops, 12 2019. URL https://www.youtube.com/watch?v=J7bEunZXkxc.

Diego Sucaria. Microservices architecture - orchestrator, choreography, hybrid... which approach to use?, 04 2021. URL https://diegosucaria.info/microservices-architecture-orchestrator-choreography-hybrid-which-approach-to-use/.

TutoialPoint. MongoDB - Overview. URL https://www.tutorialspoint.com/mongodb/mongodb_overview.htm.

Jacob Vasquez. Demystifying apache kafka message delivery semantics, 11 2020. URL https://keen.io/blog/demystifying-apache-kafka-message-delivery-semantics-at-most-once-at-least-once-exactly-once/.

Elin Vinka. What is zookeeper and why is it needed for apache kafka? - cloudkarafka, apache kafka message streaming as a service, 07 2018. URL https://www.cloudkarafka.com/blog/cloudkarafka-what-is-zookeeper.html.

VueJS. VueJS. URL https://vuejs.org/.

What is a Document Database? What is a document database? URL https://www.mongodb.com/document-databases.

A

MOCKUPS

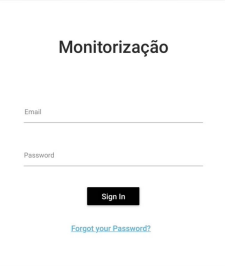

Figure A1: Mockup: Login Page

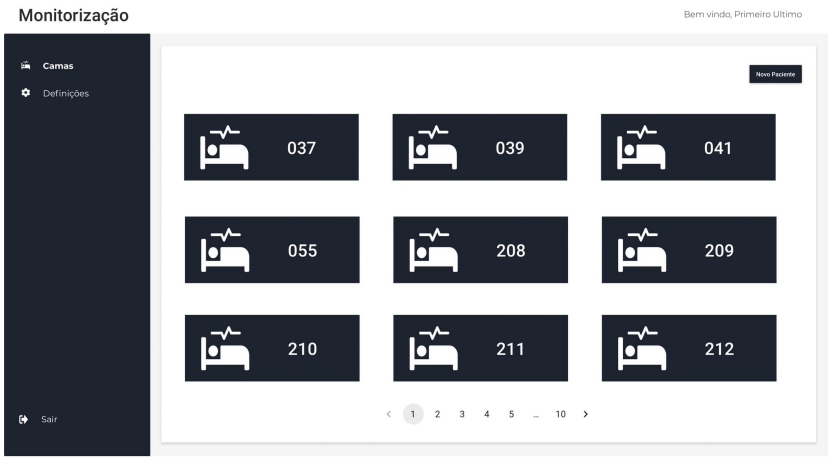

Figure A2: Mockup: Main page of the application

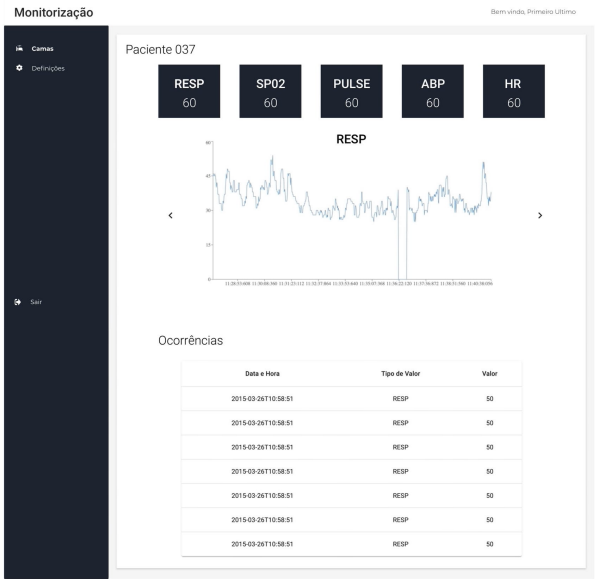

Figure A3: Mockup: Page responsible for a patient's information with occurrences

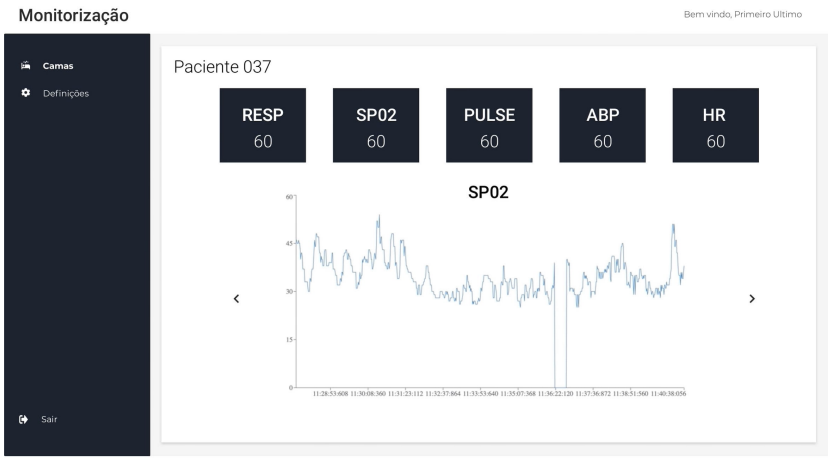

Figure A4: Mockup: Page responsible for a patient's information without occurrences