Telemedicine: A Comprehensive Solution for Traditional, Complementary and Alternative Medicine

Mariana Freitas Silva da Cruz Pires - a45287

Project presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Prof. Rui Pedro Lopes

Bragança

October, 2025

Telemedicine: A Comprehensive Solution for Traditional, Complementary and Alternative Medicine

Mariana Freitas Silva da Cruz Pires - a45287

Project presented to the School of Technology and Management in the scope of the Master in Informatics.

Supervisors:

Prof. Rui Pedro Lopes

Bragança

October, 2025

Dedication

The conclusion of this work marks the end of a journey that has been both an extraordinary academic and personal experience. Throughout these years, I have faced challenges, gained knowledge and achieved accomplishments that have exceeded all my expectations. Each stage of this process has represented an opportunity for growth, maturity and self-discovery.

First, I would like to express my gratitude to Professor Rui Pedro Sanches de Castro Lopes, my supervisor, for the support, availability and guidance provided during this phase of writing the thesis.

To my family, especially my mother and my uncle, for their love, encouragement and understanding, for believing in me and giving me courage during the most challenging moments of this journey. And, without a doubt, to my father, the star who accompanies me in every moment.

To my friends, for always being there, for their support, words of encouragement and the moments of relaxation that made this journey lighter. A special thanks to Inês and Francisca for their genuine friendship, constant support in the most demanding times and for always be there for me. I would also like to express my deep gratitude to my colleague and friend, Nelson de Freitas, who has been a pillar throughout these years, without his guidance and shared knowledge, I wouldn't have achieved the level of understanding and experience I have today.

Finally, I would like to thank all the people I met at the Polytechnic Institute of Bragança for their kindness, friendship and companionship throughout these years, which

made this experience truly unique and full of learning, especially to Gonçalo and Alexandra, who throughout the master's degree proved to be present, helpful friends and always there to lend a hand.

Acknowledgment

I would like to express my sincere gratitude to my supervisor for his continuous guidance, support and valuable feedback throughout the development of this document.

Abstract

This project presents the development of a telemedicine platform that integrates traditional, complementary and alternative medicine into a digital ecosystem. The main goal was to increase the accessibility to healthcare services while promoting an inclusive approach to patient care by combining technological innovation with diverse medical practices.

The application was developed using a microservices architecture, with a Java Spring Boot backend, Angular frontend and deployment through Docker and Kubernetes on Google Cloud Platform (GCP). The Continuous Integration and Continuous Deployment process was automated using Jenkins, which orchestrates the compilation, Docker image creation and deployment in development environment. This architecture ensures scalability, maintainability and efficient resource management, while providing secure communication services between patients and healthcare professionals.

In addition, the platform was structured following a modular architecture composed of several independent services interconnected through an API Gateway, responsible for centralizing and managing system requests. This approach enhances modularity, resilience and horizontal scalability, optimizing performance and ensuring service continuity.

In conclusion, this telemedicine solution represents a relevant contribution to digital health by accessibility, interoperability and integration between modern and traditional healthcare models.

Keywords: Telemedicine, Microservices, , Cloud Computing.

Resumo

Este projeto apresenta o desenvolvimento de uma plataforma de telemedicina que integra a medicina tradicional, complementar e alternativa em um ecossistema digital unificado. O principal objetivo foi aumentar a acessibilidade aos serviços de saúde, promovendo uma abordagem inclusiva no atendimento ao paciente por meio da combinação entre inovação tecnológica e práticas médicas diversificadas.

A aplicação foi desenvolvida com base numa arquitetura de microsserviços, utilizando Java Spring Boot no backend, Angular no frontend e implementação através do Docker e Kubernetes na . O processo de integração e entrega contínua foi automatizado através do Jenkins, que coordena a compilação, criação de imagens Docker e implementação no ambiente de desenvolvimento. Essa arquitetura garante escalabilidade, facilidade de manutenção e gestão eficiente de recursos, além de proporcionar serviços de comunicação seguros entre pacientes e profissionais de saúde.

Além disso, a plataforma foi estruturada segundo uma arquitetura modular composta por diversos serviços independentes e interligados por meio de um API Gateway, responsável por centralizar e gerir as requisições do sistema. Esta abordagem reforça a modularidade, a resiliência e a capacidade de escalar horizontalmente, otimizando o desempenho e a continuidade do serviço.

Em conclusão, esta solução de telemedicina representa uma contribuição relevante para a saúde digital, ao promover a acessibilidade, a interoperabilidade e a integração entre modelos de cuidados modernos e tradicionais.

Palavras-chave: Telemedicina, Microserviços, , Computação em Nuvem.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This project presents the development of an application designed to integrate medicine with modern technologies (microservices). In recent decades, significant advances in technology have enabled the creation of solutions that bring these two worlds together, with the purpose of transcending geographic limitations and making healthcare services accessible to everyone.

Given this context, the project proposes to combine knowledge from the fields of healthcare and technology, offering a practical and accessible solution that improve access to medical services. The system was developed based on microservices in a cloud environment, which ensures efficient, secure and easy maintenance.

1.1 Context

The work is part of technology in healthcare, particularly in telemedicine and integration of medical practices. Telemedicine has been recognized as an effective tool in promoting healthcare capable of providing access to quality medical care. Although technological advances help overcome geographical barriers and facilitate access to healthcare services, there are still significant challenges when attempting to combine these different medical approaches. As a result, there are still limitations in implementing more inclusive care from the patient's perspective, in this case, physical, emotional and social. Therefore,

it is essential to create technological solutions that facilitate the integration of these approaches, which provide more comprehensive care that is accessible to all [1].

The application implement microservices architecture, in which each service is independent and responsible for a specific functionality, such as user authentication and management, appointment scheduling, payment processing, healthcare professional management, video calls and medical specialty organization. This approach allows each service to be developed, tested and updated independently, without affecting the operation of the others. This architecture provides greater flexibility, performance and reliability, as well as facilitating Continuous Integration (CI) and optimization of cloud computing, such as GCP. This model differs from the monolithic model, in which all functionalities are integrated into a single code, so this makes maintenance, scalability and the introduction of new functionalities.

1.2 Objectives

The main objective of this work is to design and develop a telemedicine application that integrates multiple healthcare functionalities through a microservices architecture.

This architectural model, based on microservices, supports CI/CD pipelines and leverages the scalability and flexibility of cloud computing platforms, particularly GCP. This approach ensures the system's robustness, optimizes performance and enhances resource efficiency, enabling each service to be deployed, monitored and scaled independently according to demand.

The objectives of this research are:

1. To design and develop a telemedicine application based on a **microservices architecture**, enabling modular and scalable integration of healthcare functionalities such as user management, appointment scheduling, payments, video consultations and medical specialty organization developed in **Spring Boot**. The system's frontend is developed using **Angular**, providing a dynamic, responsive and user-friendly interface for patients and healthcare professionals.

2. To implement and optimize the proposed system through containerization with Docker, integrated into CI/CD pipelines and deployed on Google Cloud Platform (GCP).

1.3 Document Structure

This report is organized into six chapters, each focusing on a different stage of the project's development.

- **Chapter 2** explores the state of the art, outlining the main concepts, technologies and related work that form the foundation of this study.
- **Chapter 3** describes the system architecture, presenting the architecture and requirements.
- **Chapter 4** focuses on the implementation phase, detailing how the proposed solution was developed and integrated.
- **Chapter 5** presents the tests carried out to evaluate the system's performance and validate its functionality.
- **Chapter 6** concludes the report with the main findings, reflections and suggestions for future improvements.

Chapter 2

Evolution of Telemedicine

Advances in information and communication technologies have profoundly transformed the healthcare sector, enabling the emergence of new forms of medical care. Among these, telemedicine stands out, which consists of providing clinical services remotely, using digital platforms to consultations, diagnoses and patient monitoring in real time.

The first experiments with telemedicine date back to the early 20th century, when heart rhythms were transmitted via telephone in the *Netherlands*, marking the beginning of the use of communication technologies in a clinical context. Over the following decades, innovation continued to expand. In the 1920s, radio medical consultations emerged in several European countries and in the 1940s, the first transmissions of radiographic images between cities in the United States (namely in Pennsylvania) were carried out via dedicated telephone lines [2]. These initiatives demonstrated the potential of telecommunications for remote patient monitoring, showing us what would become modern telemedicine.

With the advancement of the internet and the exponential increase in processing power, storage capacity and bandwidth, telemedicine systems have developed significantly, moving from remote communication solutions to complex integrated digital platforms. This technological transformation has allowed telemedicine to move being an experimental and become a central tool in healthcare delivery, especially in contexts where geographical or logistical accessibility represents a barrier [3].

The development of digital health infrastructures and the consolidation of electronic

health record (Electronic Health Records (EHR)) systems have marked a profound change in the way medical information is stored, shared and used. These systems enable the integration of large volumes of clinical data from different institutions and professionals, ensuring continuity of care and promoting more efficient coordination between teams. According to Shen et al. [4], the integration between EHR systems and telemedicine platforms is currently one of the fundamental pillars for ensuring clinical effectiveness, data security and operational efficiency in digital health services.

According to Rush et al. [5], consultations conducted via videoconferencing show higher levels of satisfaction and a better therapeutic relationship when compared to consultations exclusively by telephone, demonstrating the importance of direct visual interaction between doctor and patient. Video calls have been shown to result in fewer medication errors, greater diagnostic accuracy and more accurate clinical decisions compared to consultations conducted by telephone.

The technology achieved at this stage demonstrate that is possible to integrate telemedicine with cloud systems, large-scale data analysis (Big Data) and Artificial Intelligence (AI). Consequently, telemedicine has developed from a simple remote communication tool to a digital health ecosystem, supported by principles of interoperability, security and clinical efficiency [6].

In summary, the evolution of telemedicine closely follows the technological progress of recent decades, moving from simple clinical data transmissions to complex and secure digital platforms capable of integrating clinical data, audiovisual and communication, consolidating itself as an essential element of modern medicine and the digital transformation of healthcare systems.

2.1 Impact of the COVID-19 pandemic on Telemedicine

The COVID-19 pandemic, declared by the World Health Organization (WHO) in March 2020, represented a decisive milestone in the consolidation of telemedicine globally. Mobility restrictions, the need for physical distancing and the overload of hospital systems

forced a rapid reconfiguration of health services, driving the adoption of digital platforms as the primary means of communication between doctors and patients [7].

In summary, telemedicine has expanded dramatically around the world, becoming an essential tool for ensuring access to medical care during the crisis. This advancement hasn't only increased accessibility to health services, but has also changed the way care is delivered, marking the beginning of a new era of digital health.

2.2 Monolithic architecture

Monolithic architecture represents the traditional approach to software development, where all components of an application are integrated into a single unified system. In this model, the entire application is built, deployed and scaled as one unit. This architectures featured an all-in-one structure in which every function and component was tightly incorporated into a single codebase [8]. However, as applications grow in size and complexity, monolithic systems often become rigid, difficult to maintain and challenging to scale.

The limitations in monolithic systems have encouraged the changes towards more modular and distributed architectures. According to the study from Tapia et al. [9], this transition is driven by the need to improve system responsiveness, fault tolerance and continuous delivery capabilities. The monolithic applications offer simplicity in their early stages, their tightly coupled structure becomes a restriction for deployments. In contrast, Service-Oriented Architecture (SOA) and later, Microservices Architecture (MSA), decompose applications into smaller, independent services that can be deployed and scaled individually. This modularization not only improves maintainability and scalability but also supports incremental system evolution, allowing organizations to adopt new technologies or frameworks without disrupting the entire system.

2.3 Microservices Architecture

The microservices architecture has become one of the most significant architectural paradigms for building scalable and maintainable distributed systems. It represents an evolution of the SOA paradigm, emphasizing smaller, autonomous and independently deployable components that collectively form a cohesive system, unlike monolithic architectures, where the entire application operates as a single unit, microservices divide the system into small, self-contained services, each responsible for a specific business capability and communicating through lightweight interfaces such as Representational State Transfer (REST) or messaging protocols.

According to Velepucha and Flores [10], the evolution of microservices is driven by the need for independent deployment, modular scalability and technological flexibility. Microservices are particularly well suited for domains that require continuous evolution and high availability, as each service can be developed and updated independently without affecting the operation of others. Consequently, the microservices design approach focuses on building applications as a collection of small, autonomous services, where each service can be deployed on a separate platform and use its own independent technology stack [11].

The evolution from monolithic systems to service architectures has influenced how modern software is designed and maintained. In this progression, two paradigms have emerged: Service-Oriented Architecture (SOA) and Microservices Architecture (MSA). Both architectural models share the objective of decomposing large, complex systems into smaller, reusable and compatibility services that communicate through well-defined interfaces. However, despite these conceptual similarities, their scope, granularity and operational principles differ considerably [12].

Service-Oriented Architecture (SOA) was originally developed to address the scalability and maintainability limitations of traditional monolithic systems, which often became rigid and difficult to develop when appears a complex service. By promoting the reusability of software components and facilitating interoperability across heterogeneous

environments, SOA enables applications to communicate through standardized service interfaces, reducing redundancy and improving system integration. Communication in this architecture typically depends on a centralized middleware layer, commonly referred to as the Enterprise Service Bus (ESB), which manages message routing, transformation and orchestration among services. According to recent studies, while this centralized approach consistency and coordination, it also introduces constraints such as coupling, increased latency and the potential for single points of failure [13]. Furthermore, the dependence on complex communication protocols such as Simple Object Access Protocol (SOAP) and Extensible Markup Language (XML) adds extra complexity, often making the system less flexible and slower in dynamic environments. These technical and structural limitations have conduct to the development of Microservices Architecture MSA, which follows similar service-oriented principles but introduces more decentralization, independence and scalability to meet the needs of modern distributed systems. Microservices Architecture emerged as lighter and decentralized evolution of SOA. In this model, each microservice is a self-contained unit responsible for a specific business function and capable of being developed, deployed and scaled independently. Unlike SOA, which depends on the ESB for service orchestration, MSA favors direct communication between services through REST API, message queues or asynchronous event brokers. This design choice removes the dependency on centralized middleware and increase scalability, resilience and deployment autonomy [13]. The result is a system that can evolve continuously, aligning with agile development methodologies and DevOps practices and technologies like Kubernetes or Docker.

2.4 API Gateway

In microservices architectures, communication between services and clients requires an efficient and secure routing layer, instead of allowing clients to interact directly with multiple microservices, the API Gateway acts as an intermediary that routes incoming

requests to the appropriate service while abstracting internal system complexity (Figure 2.1).

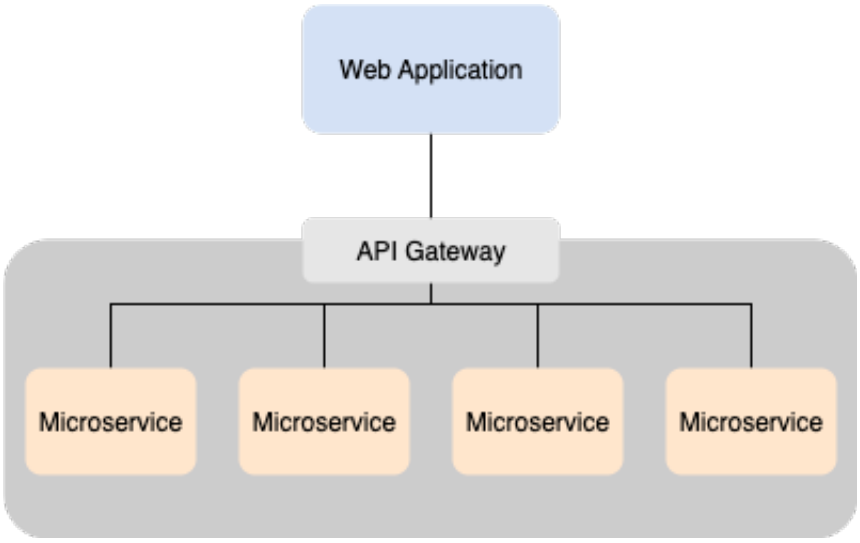

Figure 2.1: API gateway - single interface to manage requests

In the context of MSA, the Application Programming Interface (API) Gateway plays a fundamental role in managing communication between external clients and the internal network of microservices. Since each service in an MSA is designed to operate independently, direct client interaction with multiple endpoints would increase complexity and expose internal system details. The API Gateway resolves this challenge by providing a single entry point to the system, where it centralizes routing, authentication, load and traffic monitoring [14]. This component simplifies client communication while preserving the autonomy of each service, as requests are transparently routed to the appropriate microservice. In this way, the API Gateway enhances system scalability, maintainability and fault isolation, ensuring that the microservices presents a cohesive external interface to users.

2.5 Docker

Containerization technologies have significantly transformed software deployment and scalability by allowing applications to run consistently across different environments.

Docker, introduced in 2013, has become the standard for container-based development due to its simplicity and portability. Unlike traditional virtualization, which relies on full virtual machines, Docker containers share the host operating system's kernel while encapsulating application code, dependencies and configurations into light and portable images.

Docker enables developers to package applications as immutable units that can be moved between development, testing and production environments, ensuring reproducibility and compatibility between different Operating Systems (SO). It also facilitates CI/CD pipelines, as containers can be easily versioned and orchestrated automatically, like Jenkins. More recent research highlights Docker's essential role in microservices architectures, where each service runs in an efficiency, portability, isolation container and scalability [15]. By standardizing the deployment process and simplifying dependency management, Docker provides a well-established platform for modern cloud-native systems.

2.6 Kubernetes

As applications grow in complexity and scale, manually managing multiple containers becomes increasingly impractical. Kubernetes, originally developed by Google and later open-sourced in 2014, emerged as a leading solution for container orchestration. These platforms abstract away the complexity of managing containerized applications at scale, handling tasks such as scheduling, scaling, networking and storage management [15]. Kubernetes abstracts the underlying hardware and provides a uniform interface for defining desired application states, ensuring that containers are automatically deployed and maintained according to those specifications.

Modern studies emphasize Kubernetes as a critical enabler of cloud-native architecture, allowing dynamic scaling through mechanisms like Cluster Autoscaler and Horizontal Pod Autoscaler, which adjust computing resources based on real-time workloads [16]. It also integrates with advanced networking and security features, supporting multi-tenant isolation, secret management and policy enforcement. Furthermore, Kubernetes' ecosystem

supports hybrid and multi-cloud deployments, enabling organizations to achieve portability and independence [17]. Its declarative configuration model, combined with automation and resilience mechanisms, has made Kubernetes a technology for orchestrating microservices and ensuring high availability in distributed systems.

2.7 Technology

During the development of the project, several tools were used to create the application and database. In addition, some applications that had already been developed are also discussed, which served as a reference and guidance for the development of this work.

2.7.1 Java

Java is a multi-platform object-oriented language. It is the basis for applications, mobile phone operating systems, business software and many well-known programs.

For this project, Java was the language chosen to implement and innovate in the area of microservices. The choice was based on the language extensive support, the availability of libraries, easiest of integration with new technologies and the possibility of running on multiple platforms, characteristics that make Java a robust and versatile option for application development.

2.7.2 Spring Boot

Spring Boot is a framework for creating and implementing Java-based code to build microservices and web applications.

It also offers a dependency injection feature, an important principle of Spring Boot that contributes to the organization and management of application components. Instead of each class creating the objects it needs, these are automatically provided by Spring, allowing you to develop modular applications, ideal for microservices and distributed applications.

With Spring Boot, we have the facility to configuration:

- Services;
- Dependency management;
- Libraries;
- Integration with other technologies;
- Database connection;
- Web services, such as, REST API

2.7.3 Angular

Angular is a web development framework maintained by Google, used to create dynamic, scalable and high-performance applications. Based on TypeScript, Angular facilitates the construction of interactive interfaces and efficient application state management.

For this project, Angular was chosen as the frontend technology due to its ability to create responsive and interactive interfaces, direct integration with REST API and compatibility with web development best practices. These features enable a consistent and intuitive user experience, effectively complementing the microservices architecture implemented in the backend.

2.7.4 Intellij IDEA

IntelliJ IDEA is an Integrated Development Environment (IDE) used for programming in Java, Kotlin, Groovy and other Java virtual machine (JVM)-based languages.

IntelliJ also offers testing tools. With JUnit integration, to run unit tests directly and see the results in real time.

In addition, IntelliJ supports plugins that add new features, such as database connectivity, for example PostgreSQL, facilitating code integration with it.

2.7.5 VS Code

Visual Studio Code (VSC) is a code editor developed by Microsoft. It supports multiple programming languages and offers features such as code autocompletion, Git integration for version control and a huge range of extensions that facilitate web and mobile application development.

2.7.6 Draw.io

Draw.io is an online graphic editor used to create diagrams and visual schemas. It allows users to draw software architecture diagrams, process maps, database diagrams and other types of graphic representations that are essential for project planning and documentation.

For this project, draw.io was used to develop diagrams of the application architecture, including flows, as well as the layout of microservices and their implementation in the Cloud. This tool contributed to a better visualization of the system, facilitating the planning, understanding and communication of the implemented solutions.

2.7.7 PgAdmin

PgAdmin is an open source database administration and management tool developed to work with PostgreSQL. It works as a client application that allows you to access and manage a database server.

It offers features such as SQL query execution, user management, permission control and database performance monitoring.

2.7.8 PostgreSQL

PostgreSQL is an open source Database Management System (DBMS), recognized for its robustness, reliability and support for advanced features such as transactions, referential integrity and complex queries. It is highly scalable and compatible with multiple platforms, allowing it to be used in applications of different sizes and requirements.

PostgreSQL was chosen for its efficiency and compatibility with Spring Boot. Its integration with tools such as pgAdmin facilitated the development, testing and maintenance of the database during the project implementation.

2.7.9 Docker

Docker is a platform that allows you to create, deploy and run applications in containers, ensuring that the software works consistently across different environments. Each container contains the application and all its dependencies, isolating it from the underlying operating system and other applications.

In this project, Docker was used to package the applications microservices, allowing each service to run independently and consistently, facilitating development, testing and deployment through Kubernetes on GCP.

2.7.10 Jenkins

Jenkins is an open-source automation server that facilitates the implementation of CI/CD practices. It automates various stages of the software development lifecycle, such as building, testing and deploying applications, ensuring faster and more reliable delivery. Through pipelines, Jenkins coordinates the execution of tasks in a predefined sequence, providing visibility, repeatability and control over the entire process.

In this project, Jenkins was used as the central automation tool for managing the CI/CD workflow. It orchestrated the compilation of source code, the creation and publishing of `Docker` images to the `Google Artifact Registry` and the deployment of those images to Kubernetes (GKE). This integration enabled continuous delivery of updates, reducing manual intervention and ensuring that each new version of the microservices was deployed efficiently and consistently.

2.7.11 Google Cloud Platform (GCP)

GCP is a cloud hyperscaler, just like Amazon Web Services (AWS) and Microsoft Azure. With GCP and other hyperscalers, customers can access computing resources in data centers distributed around the world, with access being free or pay-as-you-go.

GCP provides public cloud infrastructure for hosting web applications, among other things. Its integration with tools such as Docker and Kubernetes enables efficient management of cloud services, facilitating application development, implementation and maintenance.

2.7.12 Kubernetes

Kubernetes, also known as k8s or kube, is an open-source container orchestration platform for scheduling and automating the deployment, management and scaling of containerized applications, such as those created with Docker. It allows multiple containers to work coordinated, ensuring high availability, resilience and scalability of applications.

Using Kubernetes to manage microservices in the cloud ensures that each service can be scaled, monitored and updated independently, without affecting the operation of the others. Integration with Google Cloud Platform and Docker enables efficient resource management, simplifying application maintenance and continuous implementation CI/CD, creating an environment similar to that currently used in large applications and companies. Kubernetes is currently the leading container orchestration tool, used by 71% of Fortune 100 companies [1].

2.7.13 Stripe

One of the key aspects in the development of telemedicine platforms is the secure and efficient management of electronic payments. Among the most widely used solutions is

[1]The Fortune 100 is a list of the top 100 companies in the United States inside of the Fortune 500, which in turn includes the 500 largest public and private companies in the United Stated, published by Fortune magazine. The ranking is based on annual revenue reports provided by public and private companies to government agencies. The ranking of the top 100 positions is determined by the companies' total revenues in the corresponding fiscal year.

Stripe, a global payment processing platform that allows financial transactions to be integrated directly into web and mobile applications. Stripe offers flexible application programming interfaces (APIs) that facilitate the billing of consultations, subscription plans and other digital health services, ensuring compliance with international security standards such as Payment Card Industry Data Security Standard (PCI-DSS).

In addition this integration offers advanced features such as recurring billing management, automatic receipt issuance and support for multiple currencies. These features make it a particularly relevant tool for telemedicine platforms environments, enabling a secure payment experience for both professionals and patients. The integration of solutions such as Stripe contributes to the automation of administrative processes and the financial sustainability of digital health services.

2.7.14 PayPal

Another essential component for the secure and efficient management of payments is **PayPal**. This service operates as a digital payment platform that enables users to do payment transactions quickly and securely integrated into web applications.

PayPal provides flexible APIs that facilitate the billing of consultations and the processing of payments without requiring patients to directly enter sensitive data into the platform. In addition, the service's infrastructure have a strict security and compliance standards such as PCI-DSS, ensuring robust protection.

2.7.15 Jsonbin.io

Jsonbin.io is a cloud-based platform that allows data storage and management in JavaScript Object Notation (JSON) format through a simple and secure (API).

The main advantage of *Jsonbin.io* is the simplicity of implementation, requiring no complex database configuration or dedicated servers. Through authenticated *HTTP* requests, it is possible to quickly create, update and retrieve information, making it ideal for prototypes, low-cost applications or development environments. In addition, the service

supports data encryption and access control via *API* keys, ensuring the confidentiality and integrity of stored information.

2.7.16 Jitsi

Jitsi allows you to create real-time video and audio sessions without installing external applications, working directly in browsers compatible with the Web Real-Time Communication (WebRTC) protocol.

One of the main advantages of Jitsi is the possibility of implementation on your own servers, ensuring greater control over data privacy and security. The platform supports end-to-end encryption, user authentication and screen sharing.

In addition, Jitsi offers a public API that allows for direct integration into web or mobile platforms, facilitating the creation of customized consultation management.

2.7.17 Terraform

Terraform is an open-source Infrastructure as Code (IaC) tool developed by HashiCorp. It allows you to define, provision and manage infrastructure (servers, databases, networks, etc.) using declarative configuration files, usually written in HashiCorp Configuration Language (HCL).

Terraform makes it possible to describe and automate IaC, allowing the definition of the entire architecture in a consistent, reproducible and scalable way.

Chapter 3

Architecture and Requirements

In this chapter, present a detailed overview of the proposed solution architecture designed to support an integrated platform. The chapter examines the architectural design decisions that guided the system's development, emphasizing the adoption of a microservices architecture to ensure modularity, scalability and service independence. It also describes the requirements that was necessary to implement.

3.1 Architecture Analysis

Relative to the architecture, here presents a comprehensive analysis of the architecture designed for the platform developed in this work. The proposed system combines the principles of , a microservices architecture and cloud-native deployment on . These architectural strategies provide modularity, scalability and resilience essential for healthcare applications that require integration across multiple domains, secure access control and reliable communication.

3.1.1 Domain-Driven Design

The system design be true to the principles of Domain-Driven Design (DDD) to ensure strong alignment between software components and the business domain of telemedicine.

This approach, manage its complexity and maintain a clear separation of responsibilities, the system is divided into five subdomains, each implemented as an independent bounded context following the principles of DDD.

As illustrated in Figure 3.1, the diagram presents the conceptual structure and relationships relative to these bounded contexts. The architecture bound domains such as Health, Clinical, Payment, Consultation and Communication. Each bounded context encapsulates its own responsibilities, managing its data, business logic and interactions independently.

Health Subdomain

The **Health Context** manages the classification of medical information through categories and specialties, forming the structural foundation for organizing healthcare content across the platform. It defines how medical disciplines are grouped and associated with doctors ensuring that information is presented in a coherent and accessible way. This allows the platform to maintain a consistent of medical areas, simplifying navigation for patients and supporting accurate data retrieval for healthcare professionals.

Clinical Subdomain

The **User Context** handles all processes related to user identification, authentication and authorization. It supports both patients and healthcare professionals, managing their personal data, credentials and access permissions in the system. This includes functionalities such as registration, login, logout and profile management, ensuring secure access to system resources. It also serves as a foundational element for other subdomains that require user identity mapping, such as *Appointments*, *Consultations* and *Messaging*, by providing a unified model for user information. Through integration with the backend authentication system, it maintains secure and session-based communication across all microservices.

The **Doctor Context** focuses on maintaining detailed information about healthcare professionals. It includes entities and operations related to medical specialties, academic

qualifications and professional experience. This allows administrators to manage doctors' profiles and availability, ensuring that accurate data are displayed to patients when booking consultations. It interacts directly with the *Appointment Subdomain* to share availability schedules, enabling synchronization between a doctor's timetable and the system's booking workflow.

Consultation Subdomain

The **Appointment Context** governs the workflow of medical scheduling and virtual consultations. It manages all operations related to appointment creation and modification.

The **Payment Context** is responsible for all financial transactions related to teleconsultations. It integrates with the *Stripe* and *Paypal* payment gateway to process secure payments and manage billing details. Each transaction is directly associated with an appointment, ensuring financial operations are linked to the scheduling process. By maintaining its own bounded context, the Payment Context isolates financial logic from healthcare operations, reinforcing both data privacy and system reliability.

The **Video Call Context** is responsible for managing the real-time communication sessions that enable virtual consultations between patients and healthcare professionals. This context handles the creation, initialization and management of video call sessions, linking them to the corresponding appointments and consultations defined in the *Appointment* and *Consultation* Context. Once an appointment is confirmed, the system automatically generates a secure video session using *Jitsi Meet*, providing both participants with a unique access link.

Communication Subdomain

The **Article Context**, it manages articles providing educational content that supports patient knowledge and professional reference.

The **Contact Context**, responsible for messages, feedback and support requests. This bounded context ensures communication and an integrated experience in the application.

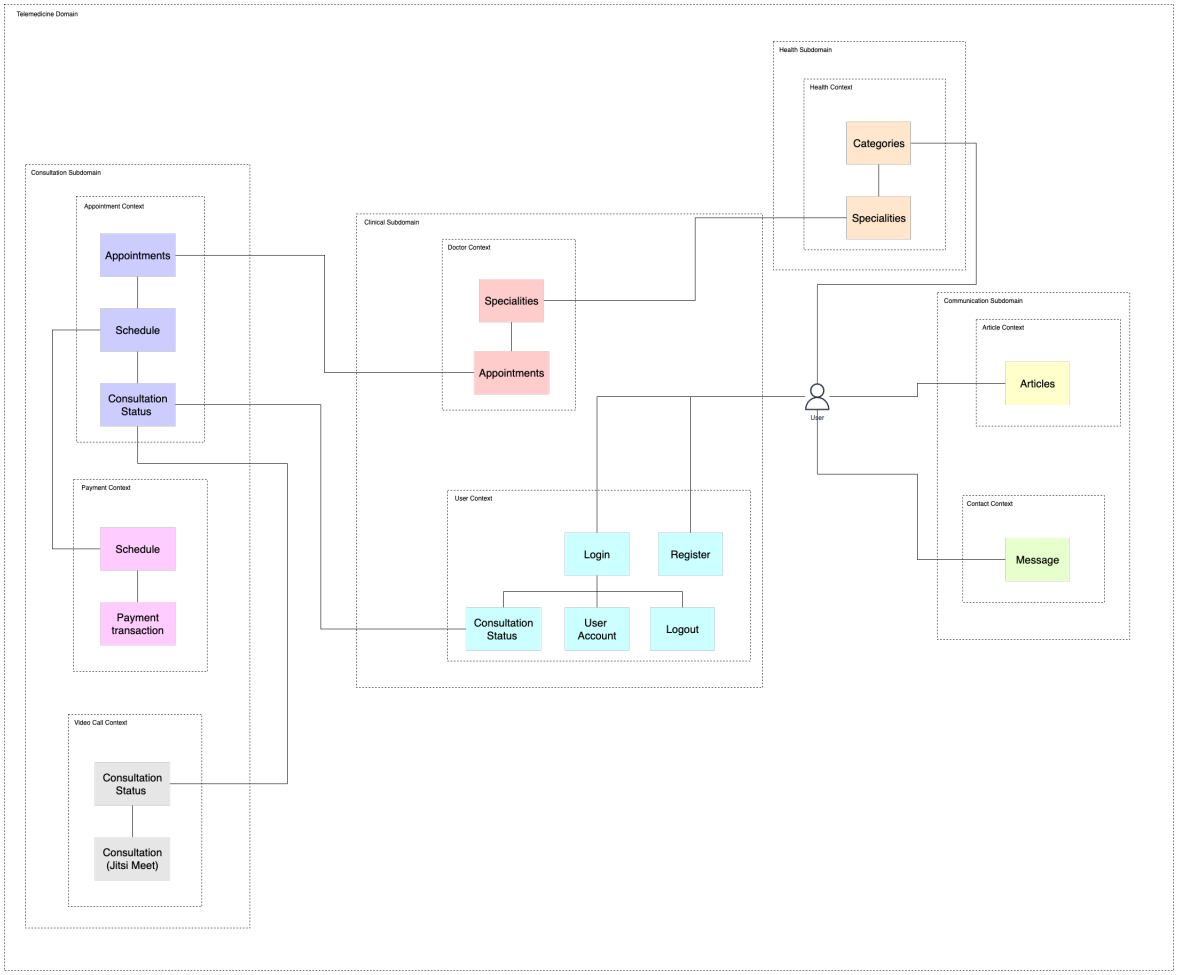

Figure 3.1: Domain-Driven Design Diagram

3.1.2 Microservices Architecture

At the implementation level, the communication model defined in the DDD layer is realized through the microservices architecture (Figure 3.2). This architecture illustrates the logical structure of the platform, which follows a microservices-based architecture. This model decomposes the system into a collection of independent services, each responsible for a specific business capability in Telemedicine Domain.

At the entry point of the system, an api Gateway acts as the intermediary between the client application and the microservices. The gateway manages all incoming Hypertext Transfer Protocol (HTTP) requests, performing routing, authentication before directing the traffic to the appropriate backend service. This centralized layer simplifies

client interaction by exposing a unified API interface while abstracting of the distributed backend.

Each microservice encapsulates its own logic and data persistence layer, ensuring data ownership and isolation. This means that no service directly accesses another service's database, instead, communication occurs exclusively through APIs.

The **Health Service** is responsible for managing the categories and specialties of healthcare professionals. This service integrates with *Jsonbin.io*, which serves as an external JSON-based data storage system.

The **User Service** is responsible for handling authentication, authorization and user profile management, storing its information in a dedicated User Database.

The **Doctor Service** manages data related to healthcare professionals, including their specializations and availability, storing its information in a dedicated Doctor Database.

The **Appointment Service** coordinates scheduling, availability and booking confirmations, storing its information in a dedicated Appointment Database.

The **Payment Service** integrates with third-party payment gateways such as *Stripe* and *Paypal* to process transactions securely and manage billing operations. Similarly, the Video Call Service interacts with **Jitsi Meet** to facilitate real-time video consultations between patients and healthcare professionals.

Additional services, such as the **Article Service** and the **Contact Service**, extend the platform's functionality by complementing aspects of the telemedicine experience. The **Article Service** provides access to educational health content, offering patients and healthcare professionals reliable information on various medical topics. The **Contact Service**, manages user communications, including messages and inquiries exchanged within the platform. Each of these services operates independently and maintains its own dedicated database, ensuring clear separation of concerns and consistent data management across the system.

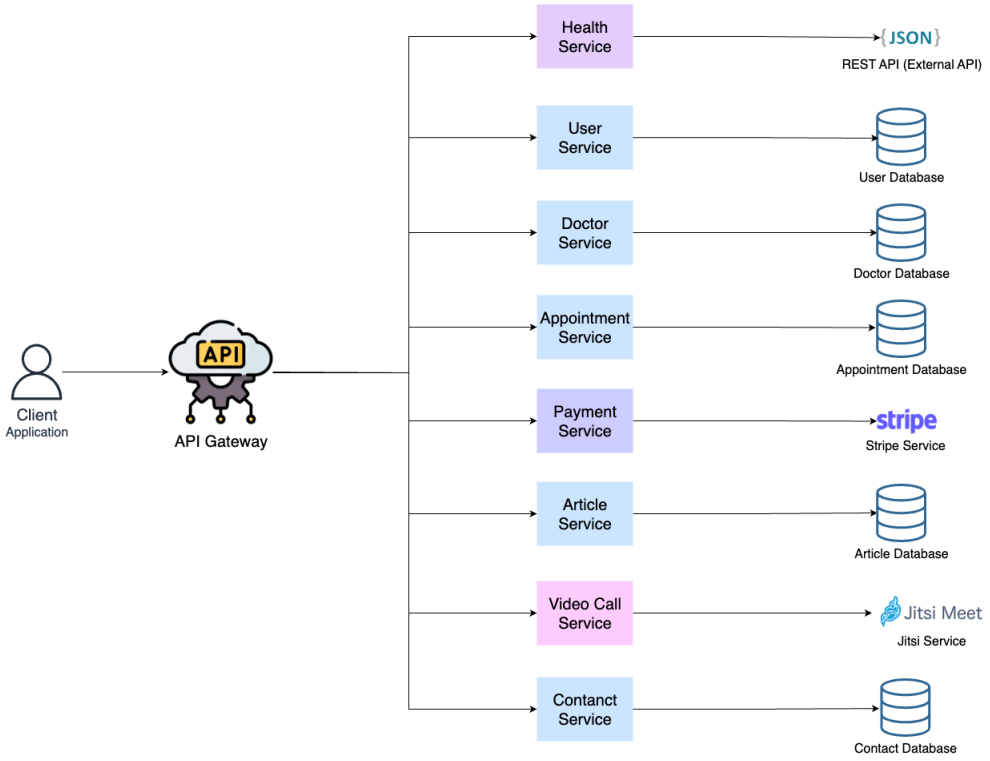

Figure 3.2: Microservices Diagram

3.1.3 Kubernetes Architecture

In the Kubernetes deployment on GCP, Figure 3.3, the communication model between services is managed and facilitated by the platform's networking and routing capabilities.

For synchronous communication, Kubernetes Services of type ClusterIP expose internal Internet Protocol (IP), that allow microservices to communicate over **HTTP** using REST API. The Load Balancer Service manages external communication and routes client requests to the correct backend service. This configuration ensures low-latency, secure and efficient communication both internally and externally.

For asynchronous communication, the Kubernetes environment provides the infrastructure needed to support event-driven messaging. In the current setup, asynchronous communication is simulated through lightweight background processes and HTTP callbacks between services.

Kubernetes enhances fault tolerance by automatically restarting failed pods and maintaining service availability even under transient failures. This ensures that both synchronous api requests and asynchronous event deliveries can recover gracefully from temporary disruptions. Additionally, Kubernetes network and service accounts restrict unauthorized communication between services, maintaining a secure communication in the cluster.

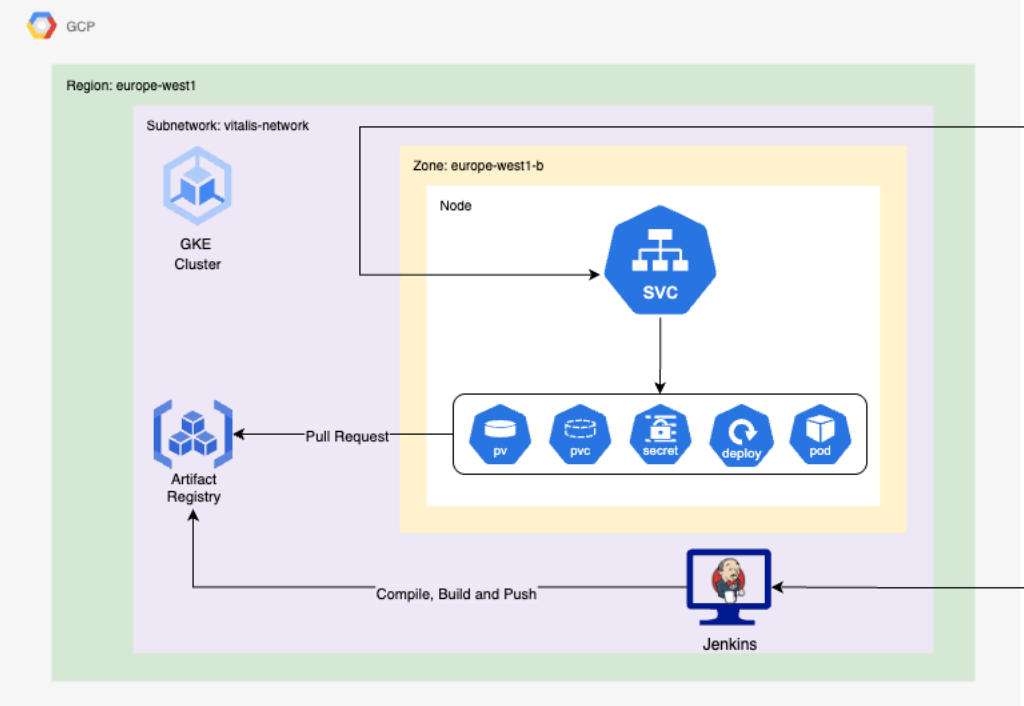

Figure 3.3: Kubernetes Diagram

3.2 Requirements Analysis

In this section, the analysis of requirements for application shouldn't focus exclusively on the functionalities, but should also consider the conditions that guarantee security, data privacy and confidentiality of information. Recent studies indicate that factors such as internet access, users digital literacy, concerns about data privacy and geographical inequalities are significant barriers to the effective use of telemedicine [18].

The requirements were divided into two main categories: Functional requirements (FR), which describe the operations that the system must perform and Non-Functional requirements (NFR), which define the qualities and constraints associated with the application's performance, security and usability.

3.2.1 Functional Requirements

Functional Requirements refer to the functionalities that the system must implement and provide to enable users to perform their tasks properly [19]. With this in mind, the following Functional Requirements are presented:

User Management

The system must allow users to register, authenticate and manage their personal profiles securely. It must differentiate between patients healthcare professionals and administrators, granting appropriate permissions and access levels based on user roles.

Doctor and Specialty Management

The platform enable the registration and management of healthcare professionals, including their qualifications, areas of specialization and availability. This information is maintained by the Health Service, which integrates with Jsonbin.io to store and organize categories and medical specialties.

Appointment Scheduling

The system support the creation and modification of medical appointments. It should allow patients to select doctors, check their availability and book time slots in real time. The scheduling logic must ensure that overlapping appointments are prevented and receive confirmation notifications.

Payment Processing

The platform provide secure payment handling through integration with third-party payment gateways such as *Stripe* or *PayPal*. This includes the ability to initiate, authorize and confirm payments, as well as to handle potential transaction failures and notify users accordingly.

Consultation Management

The system facilitate online medical consultations through real-time video sessions using *Jitsi Meet*.

Content and Communication

The platform include complementary services that enhance the user experience, such as access to educational medical content and communication tools. The **Article Service** provides users with health-related articles and information, while the **Contact Service** manages patient international messages about the service.

System Integration and Event Handling

The system coordinate its operations through both synchronous REST communication and *asynchronous* domain events, ensuring that actions are properly propagated across different services while maintaining system consistency and reliability.

3.2.2 Non-Functional Requirements

The NFR of the system have been defined in alignment with its architectural design, technological stack and operational objectives. Given that the platform is based on a microservices architecture and deployed in a Google Kubernetes Engine (GKE) environment, these requirements address aspects such as scalability, performance, reliability, security and usability.

The main non-functional requirements are:

Scalability

The system was capable of scaling horizontally to accommodate increased workloads, particularly during periods of high demand. The microservices architecture enables each service to scale independently without affecting others. Kubernetes supports this requirement through its native orchestration capabilities, which allow additional pods to be deployed dynamically as needed.

Performance and Responsiveness

The platform provide low-latency responses for real-time operations. The use of REST APIs for *synchronous* communication ensures direct and efficient interactions between services, while *asynchronous* event handling improves throughput for background and non-blocking operations.

Performance optimization is further supported by containerization with Docker, which minimizes resource overhead and load balancing via GKE Ingress, which distributes incoming requests evenly across service instances. These measures collectively ensure that the system maintains a high level of responsiveness, even under concurrent usage.

Reliability and Availability

Reliability is a critical requirement for telemedicine systems, where service interruptions can directly impact patient care. The platform ensures reliability through the independent deployment of microservices, where failures in one service do not compromise the availability of others. Additionally, GKE provides automatic pod restart and node repair mechanisms, minimizing downtime and maintaining service continuity.

Maintainability and Extensibility

The modular nature of the microservices architecture promotes maintainability, as each service encapsulates its own logic and can be updated, tested or replaced independently. The use of CI/CD ensures continuous integration and automated deployment, reducing

manual intervention and the risk of configuration errors.

Extensibility is supported by the event-driven communication model, which allows new microservices or features to be integrated without modifying existing ones.

Usability and Accessibility

The frontend of the platform, developed in Angular, provide a user-friendly and responsive interface that adapts to different devices, including desktops, tablets and smartphones. The design emphasizes intuitive navigation and clarity, ensuring that both patients and healthcare professionals can access the necessary functionalities.

Cost Efficiency

Since the platform operates in a development environment, the deployment model is optimized for cost efficiency. The GKE cluster uses a single node pool with lightweight machine types and shared resources managed through Kubernetes. This setup minimizes operational costs while maintaining a realistic simulation of production conditions. Despite the reduced infrastructure, the system preserves architectural consistency, ensuring an easy transition to a scalable production environment when required.

3.2.3 Application Actors

The system actors represent the various entities that interact with the platform. Each actor performs specific actions and engages with the system through defined interfaces, contributing to the overall delivery of healthcare services. Identifying and modeling these actors is essential to understanding how the system supports user interactions, dataflow and service coordination with the environment. In Figure 3.4, provides a visual and simplified representation of the interactions between the actors and the system.

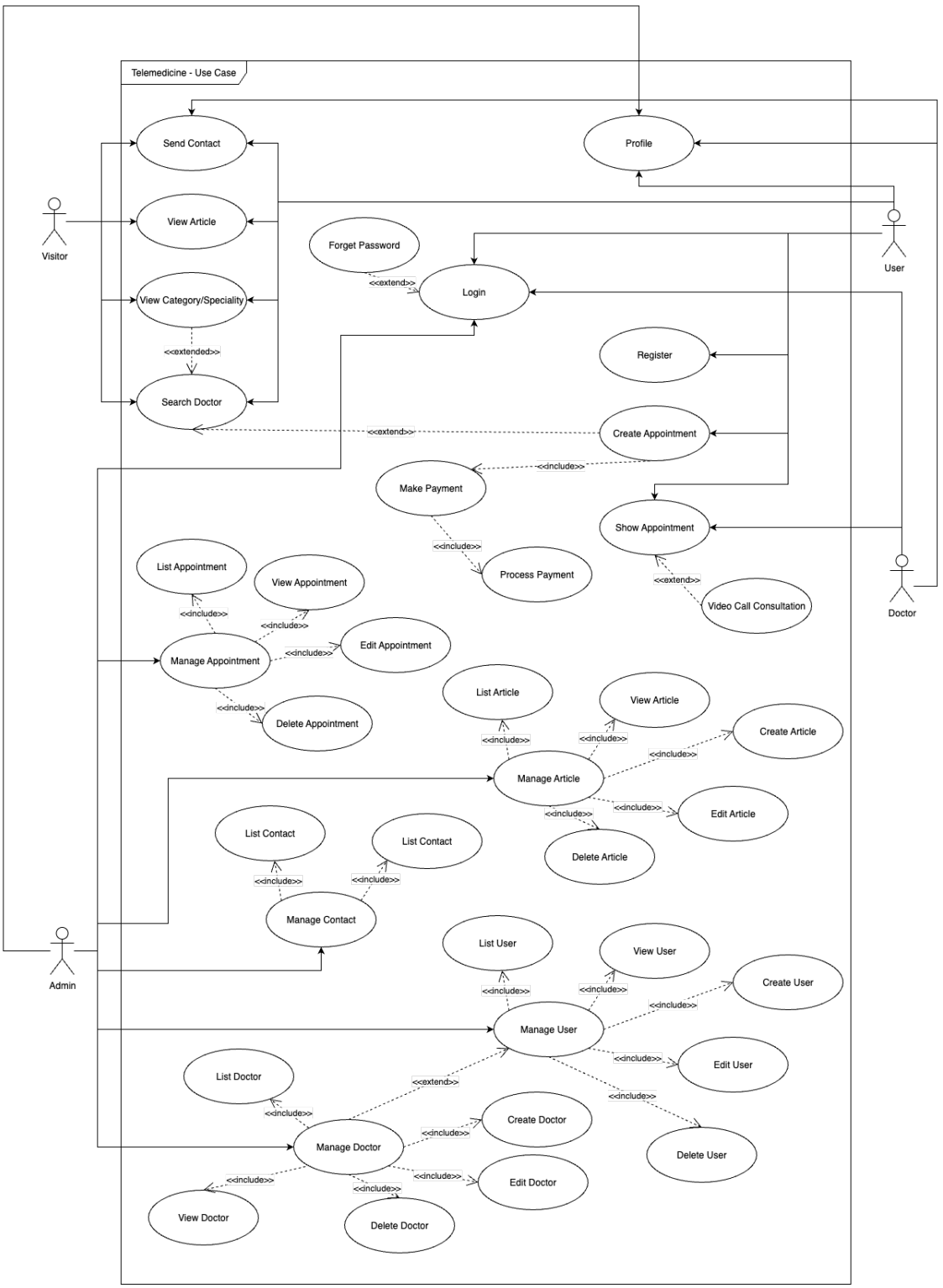

Figure 3.4: Use Case Diagram

Visitor

The Visitor represents any user who accesses the platform without an account. This actor typically explore general information such as available medical services, categories or educational content published. Visitors may browse public pages and view doctor specialties but can't perform authenticated actions such as booking appointments or participating in consultations.

This actor is important for user acquisition, as the platform provides an accessible entry point that encourages visitors to register and become active users.

User (Patient)

The User is the primary end-user of the platform. This actor represents individuals seeking medical consultations, health information and online communication with healthcare professionals.

The patient's main interactions include:

- Registering and authenticating in the platform.
- Viewing available doctors, categories and specialities.
- Scheduling and modifying aapointments.
- Making payments through integrated payment gateways such as *Stripe* or *PayPal*.
- Participating in online consultations via the *Jitsi Meet* integration.
- Accessing educational content and contacting healthcare professionals through messaging or inquiry forms.

The patient's perspective is central to the platform's design, as usability, accessibility and privacy are critical to ensuring trust and adoption of digital solutions.

Doctor

The Doctor actor represents practitioners who provide medical consultations, manage appointments and deliver healthcare services through the platform.

The main responsibilities of this actor include:

- Managing online consultations with patients via the integrated video call service.

Administrator

The Administrator is responsible for managing and supervising the platform's operational integrity. This actor ensures that all components of the system function properly and that both patient and professional data are handled securely and in compliance with regulatory standards.

Administrative tasks include:

- Managing user accounts and permissions.
- Managing publication of medical articles.
- Managing contacts.
- Managing scheduler and profissional information about doctors.

In Figure 3.5 presents the class diagram of the platform, illustrating the structure of the system and the relationships between the entities. The diagram is composed of the classes User, Doctor, Appointment, Article, Contact, Category and Speciality, which represent the elements of the application's domain.

The **Category class** is associated with several *specialities* through a `one-to-many (1:N)` relationship, while each Speciality can be linked to multiple Doctors through a `one-to-many (1:N)` relationship. These two entities are retrieved from an external data source, *jsonbin.io*, in JSON format, allowing dynamic updates of medical areas without modifying the primary database.

The **User class** represents all profiles in the system, including patients, doctors and administrators. A *User* may optionally be associated with a *Doctor* through a `zero-to-one (0:1)` relationship, indicating that not every user is a doctor, but each doctor must correspond to exactly one registered user. Additionally, a *user* can be linked to *multiple appointments* through a `one-to-many (1:N)` relationship, since a patient may schedule several consultations.

The **Doctor class** extends the user profile with professional and medical information. It is also related to *multiple appointments* in a `one-to-many (1:N)` relationship, as a doctor can have several scheduled consultations.

The **Article** and **Contact classes** operate independently from the other identities.

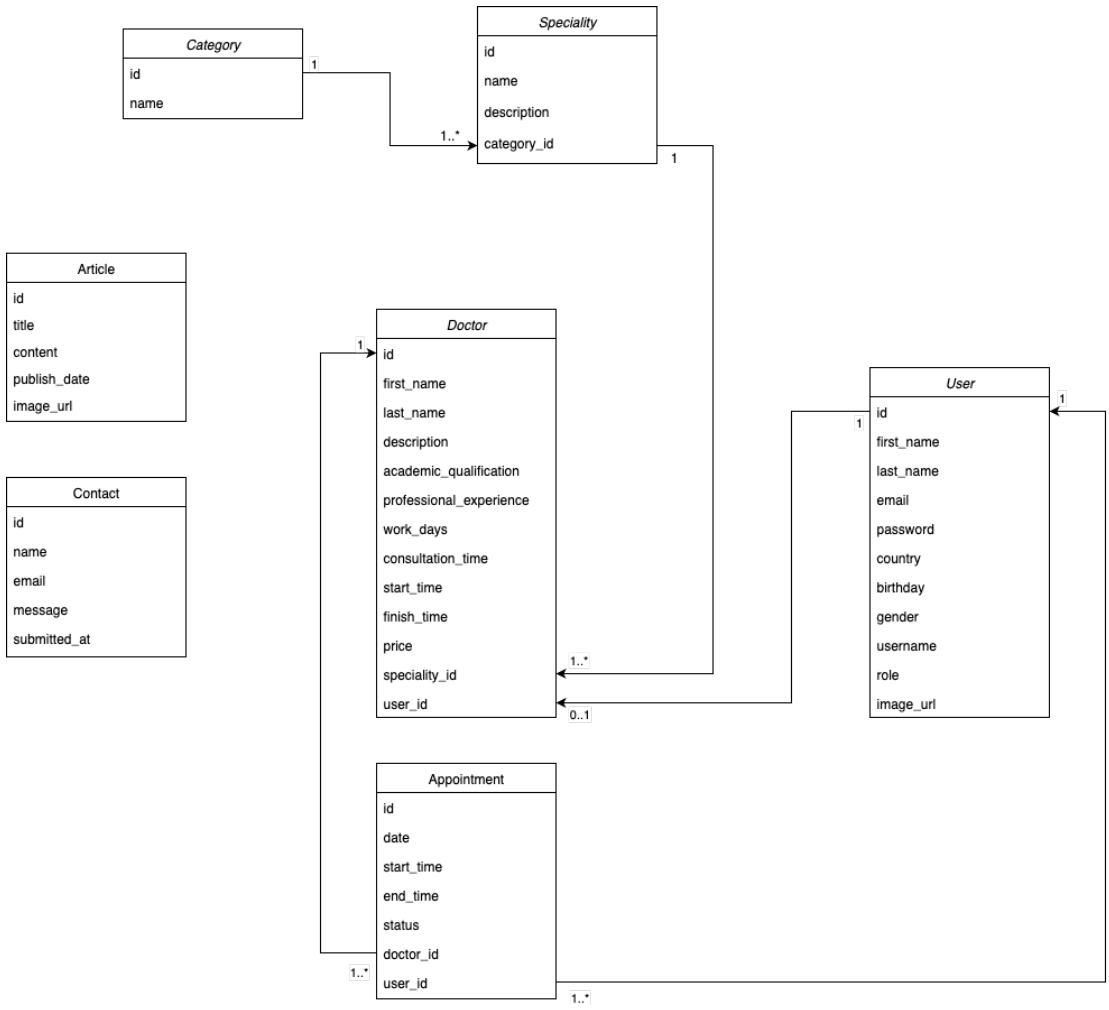

Figure 3.5: Class Diagram

3.3 Web User Interface

The Web User Interface (UI) represents the presentation layer of the telemedicine platform. It provides an accessible, responsive and user-friendly environment for both patients and healthcare professionals. Through this interface, users can perform essential operations such as account registration, authentication, appointment scheduling, consultation access and contact.

The UI is designed following modern web development principles, ensuring cross-platform compatibility and an optimal user experience across devices. It communicates exclusively with the REST API, allowing the presentation layer to remain independent from the business and domain logic. This separation of concerns promotes maintainability, scalability and flexibility for future interface enhancements or technology upgrades.

3.3.1 Mockups

Mockups are detailed graphical representations of the software requirements. They are especially useful for illustrating practical scenarios, using real data to exemplify use cases rather than abstract descriptions [20]. This approach facilitates understanding by both *developers* and *end users*.

Home

The **Home Page** (Figure 3.6) serves as the introductory interface of the platform. It was designed to immediately communicate the platform purpose and core values. The slogan *Caring for those who cared for us* reinforces the platform human centered approach, emphasizing empathy and respect in the provision of telemedicine.

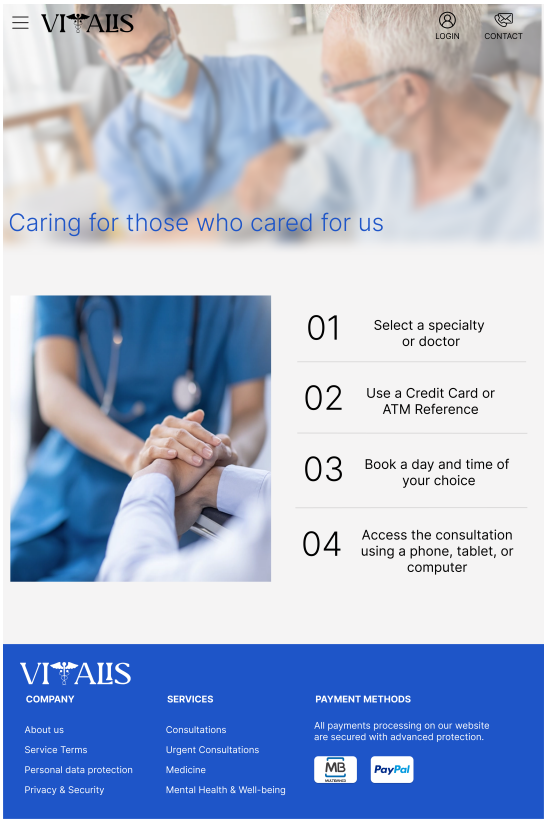

Figure 3.6: Home Page

Login and Register

The **Login Page** (Figure 3.7) and **Register Page** (Figure 3.8) were developed to allow users to access their accounts or create new ones.

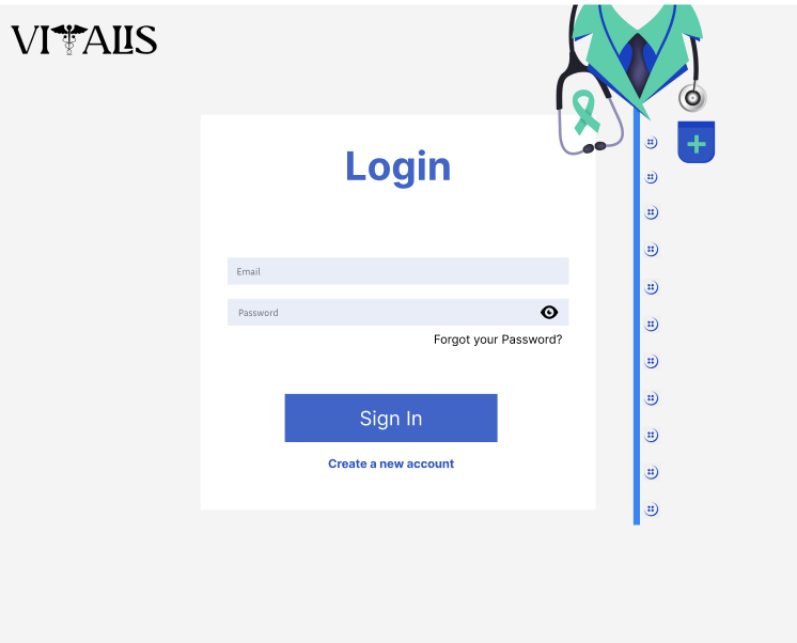

Figure 3.7: Login Page

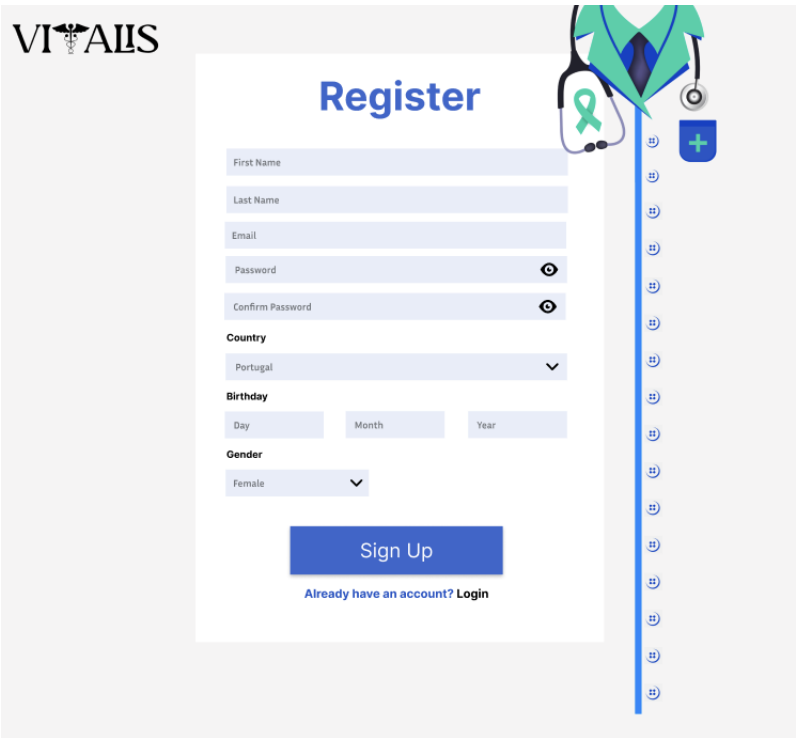

Figure 3.8: Register page

Specialities

The **Specialities page** (Figure 3.9) serves as the entry point for medical exploration in the platform. It displays a set of medical fields such as Cardiology, Neurology, Pediatrics, Psychiatry and Nutrition, each represented visually by an image.

This layout encourages intuitive navigation, allowing users to quickly identify and select the medical area relevant to their needs. The design promotes accessibility for both new and returning users.

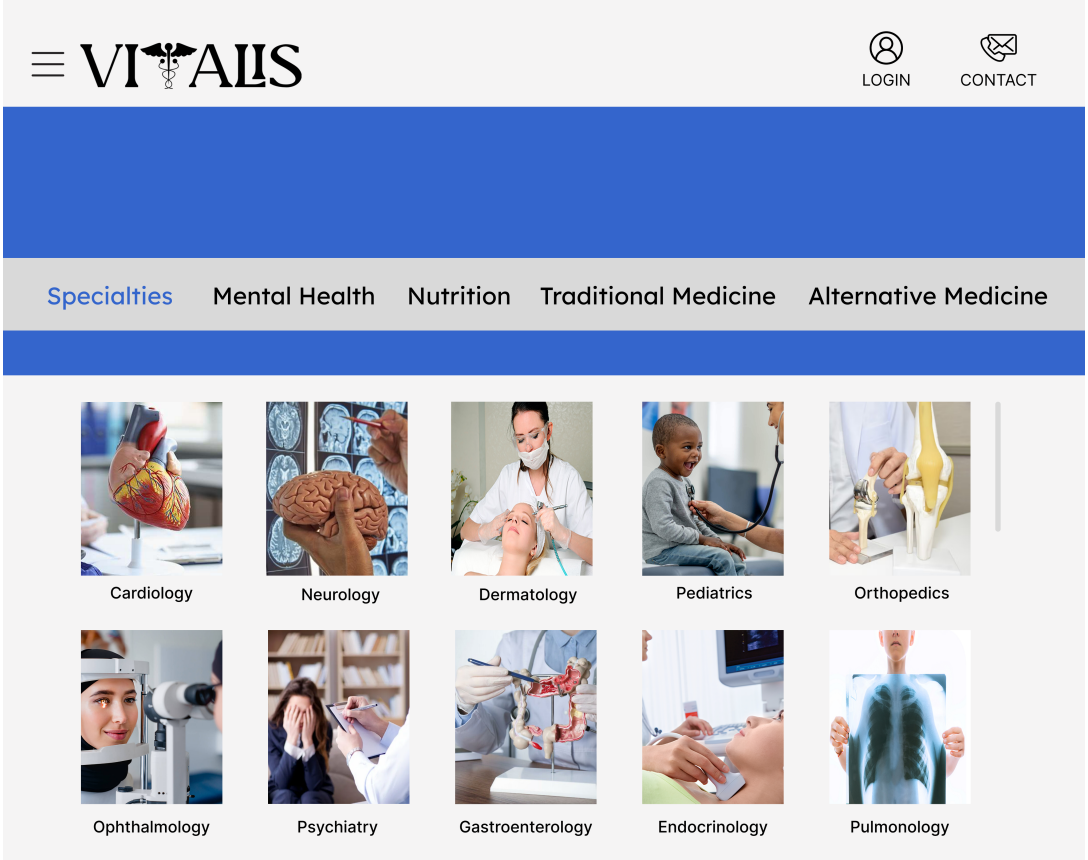

Figure 3.9: Specialty page

Doctor

Once a specialty is selected, the **Doctor page** (Figure 3.10) presents a list of available professionals relative that category. Each doctor is displayed with a profile image and name.

A search bar enables users to filter doctors according to specific needs (Name), facilitating efficient access.

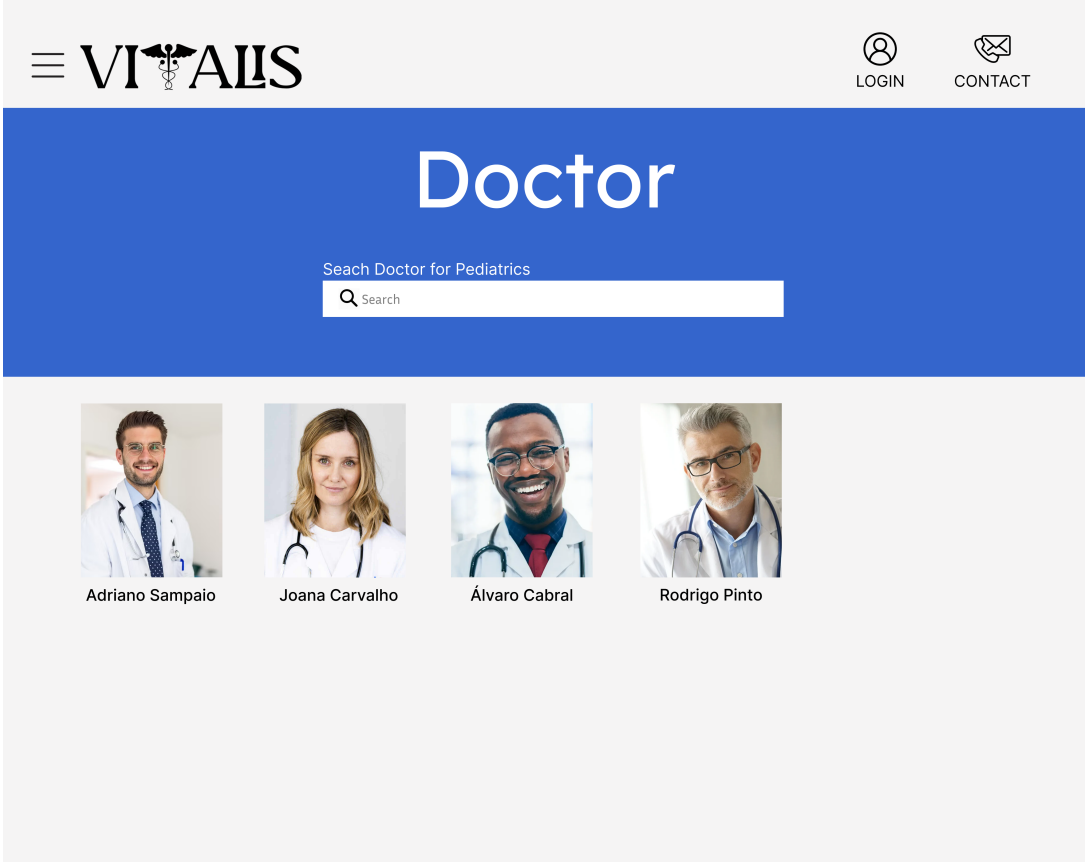

Figure 3.10: Doctor Page

Doctor Profile and Scheduling

The **Doctor Profile and Scheduling page** (Figure 3.11) provides a comprehensive overview of each professional information, including academic qualifications, medical experience and area of specialization.

The scheduling section allows users to select a preferred date and time for consultation based on the doctor's availability.

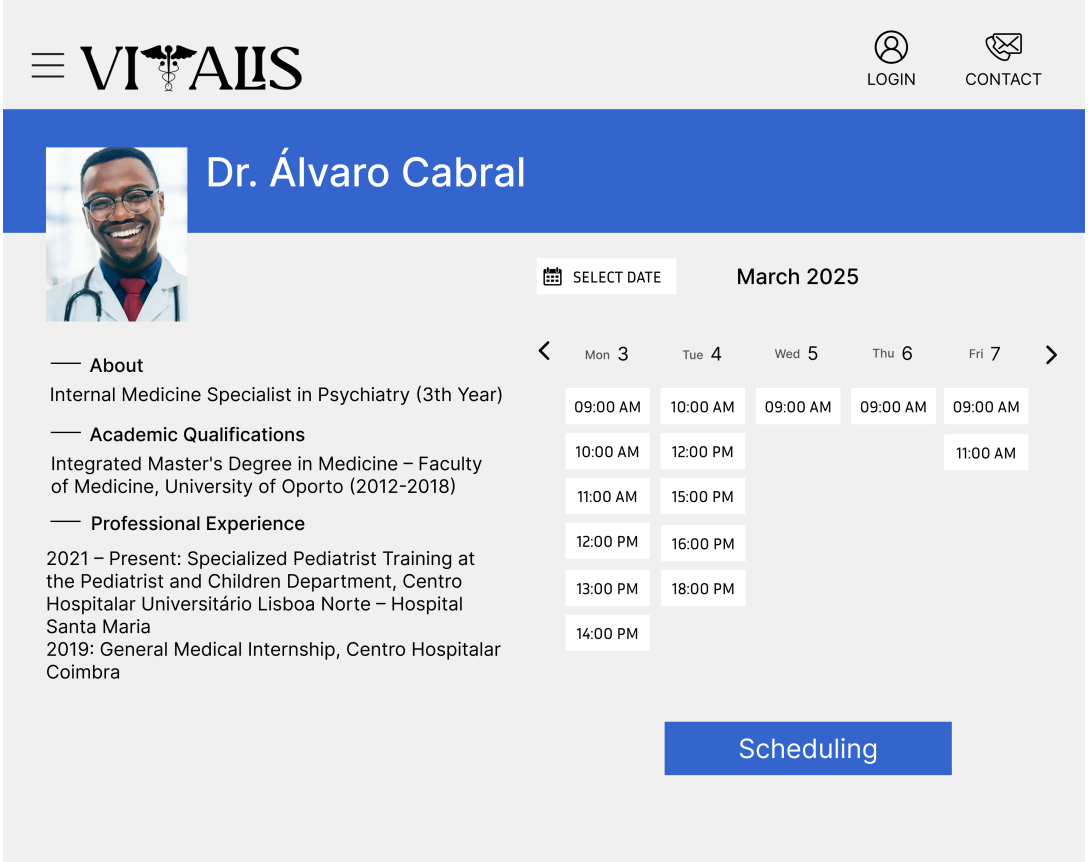

Figure 3.11: Doctor Profile and Scheduling

Payment

After selecting a consultation slot, the user is redirected to the **Payment Page** (Figure 3.12), where they can finalize the appointment. The platform supports multiple secure payment options, including Credit Card and PayPal.

A detailed summary of the consultation, including doctor's name, specialty, date and price.

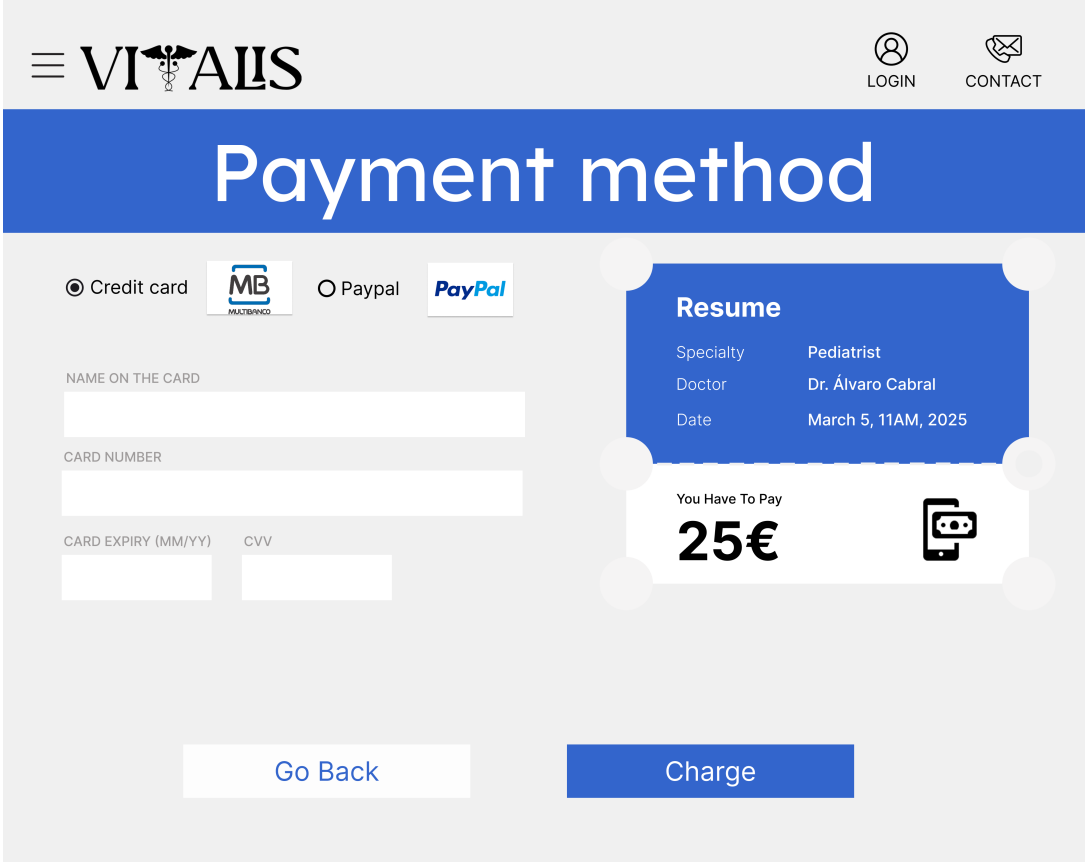

Figure 3.12: Payment Page

Article

The **Articles page** (Figure 3.13) was designed to promote health education and care. It features a collection of informative articles accompanied by images and short text. Each article includes a *View* button that redirects users to the full text.

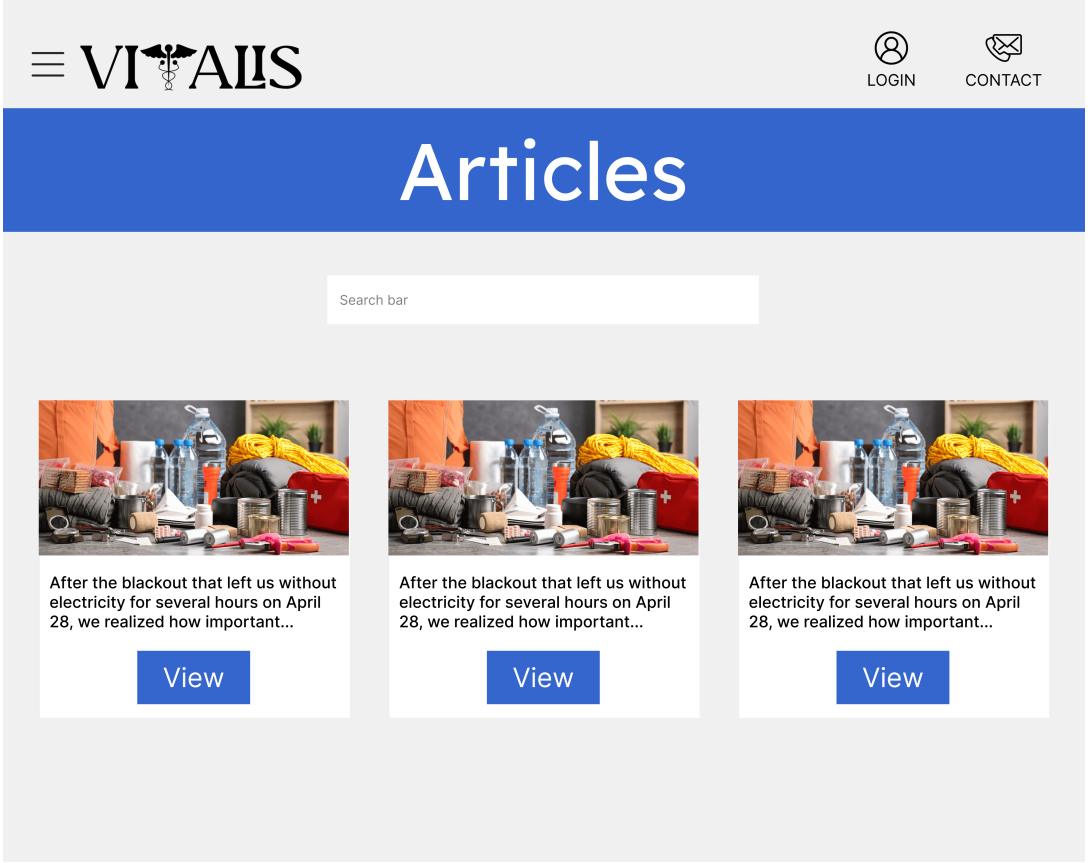

Figure 3.13: Article Page

Contact

The **Contact page** (Figure 3.14) provides a direct communication channel between users and visitors and administrators. It includes a form with fields for name, email and message, allowing to ask questions, request assistance or provide feedback.

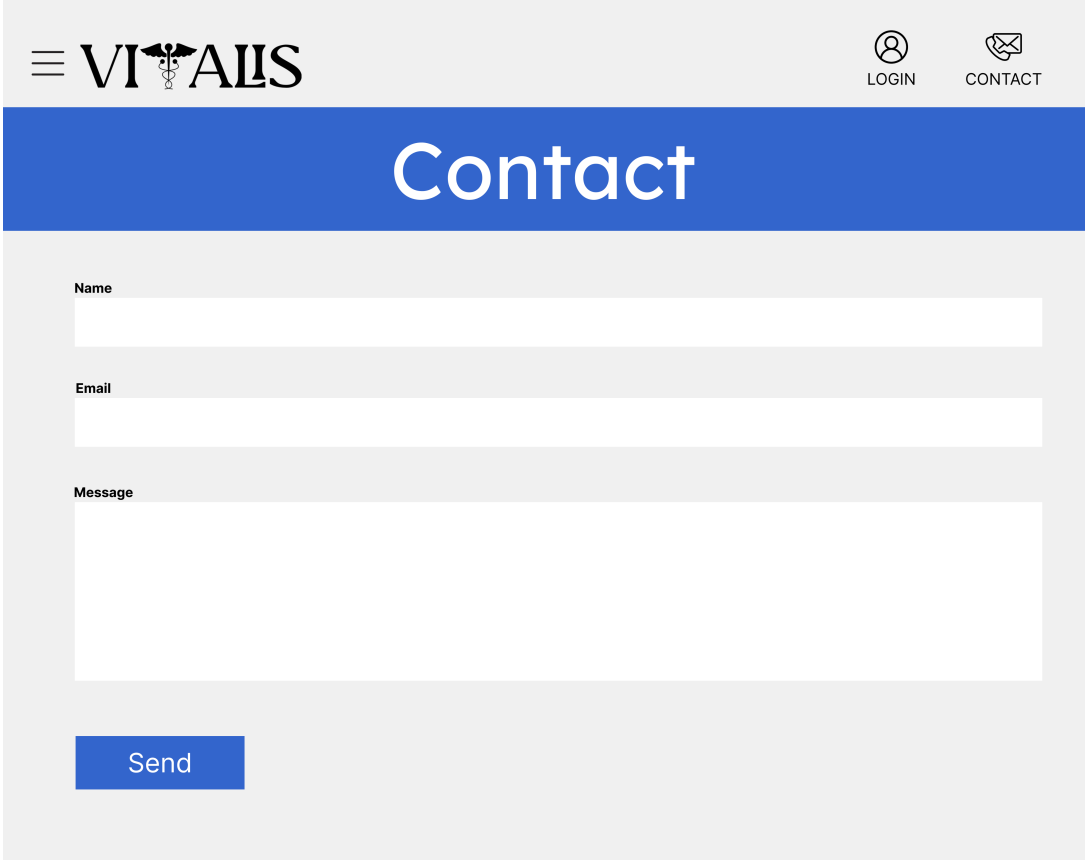

Figure 3.14: Contact Page

Profile

The **Profile page** (Figure 3.15) enables users to view and update their personal data, including name, email, birthday and country. It also supports photo upload for personalization and includes options to update or delete the account, ensuring compliance with data privacy and user autonomy principles.

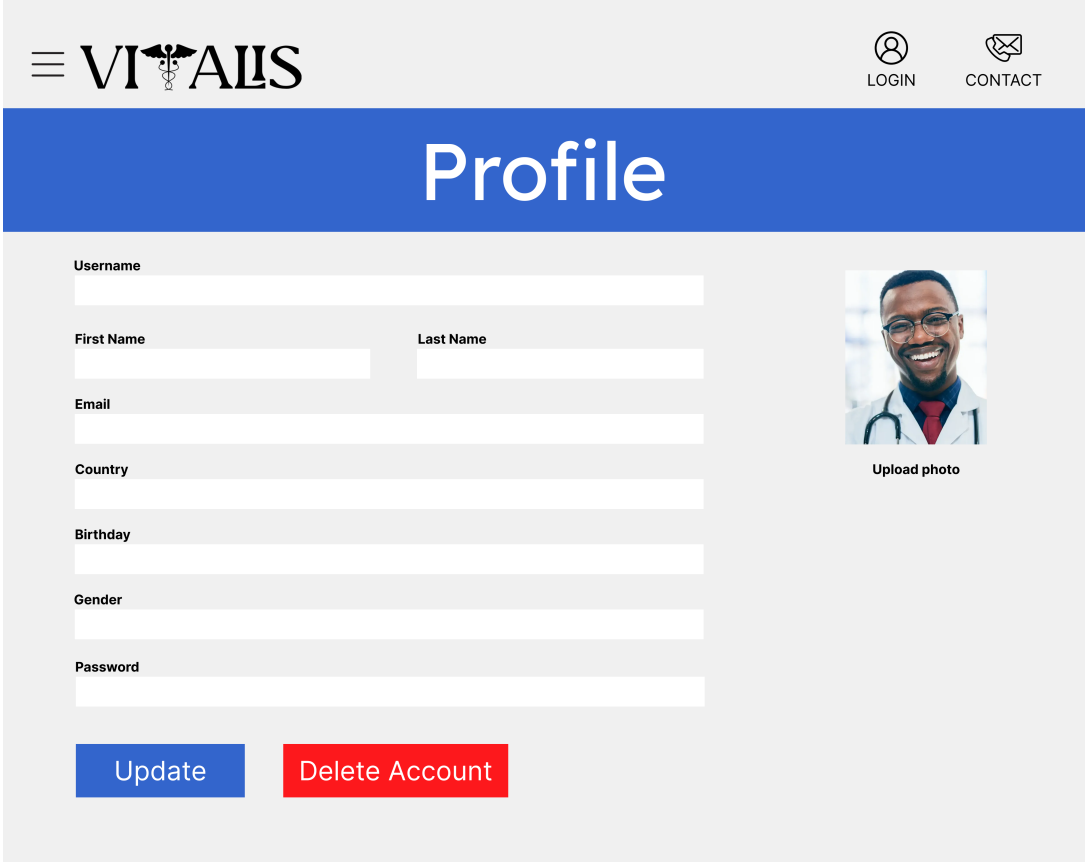

Figure 3.15: Profile Page

3.3.2 REST API

The REST API serves as the communication gateway between the user interface and the business logic defined in each bounded context. It exposes a set of well-structured endpoints that correspond to the system's main functionalities, providing secure and standardized access to application services.

Each endpoint represent requests to the appropriate domain service in its bounded context, ensuring that all business rules are enforced consistently. This design supports scalability and modularity, as each context can evolve independently while maintaining a integration access for clients.

The REST API also facilitates integration with external systems, such as third-party payment gateways (*stripe* and *paypal*) and videoconference service (*Jitsi*), without exposing internal implementation details.

The **Health API** (Figure 3.16) provides access to general healthcare data such as medical categories and specialties. It includes endpoints for listing all categories (`GET /category`), retrieving specialties by category (`GET /category/id/specialities`) and listing all available specialties (`GET /category/specialities`). This service supports the categorization of doctors and helps users navigate medical disciplines more efficiently.

Figure 3.16: Category and Speciality Requests (endpoints)

The **User API** (Figure 3.17) manages all operations related to user accounts, including registration, authentication and profile management. It provides endpoints for user creation (`POST /auth/register`), login (`GET /auth/login`), forget password (`POST /auth/forgetPassword`) and standard CRUD operations on user data (`GET, PUT, DELETE /users/id`). This service is fundamental for authentication and authorization across the platform, ensuring secure and consistent user management.

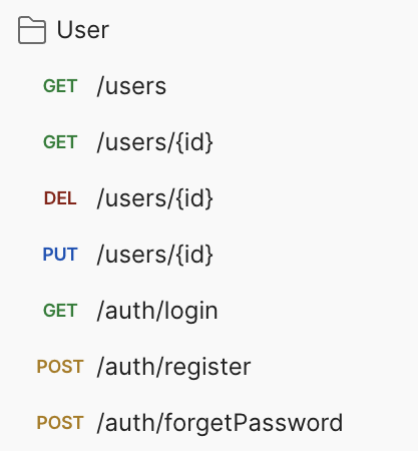

Figure 3.17: User Requests (endpoints)

The **Doctor API** (Figure 3.18) handles the creation, retrieval and maintenance of doctor profiles. It includes CRUD operations on doctor data (`GET, POST, PUT, DELETE /doctor/id`) and manage specific information such as specialties (`GET /speciality /specialityId`). This api connects with the User api to associate each doctor with a corresponding user account.

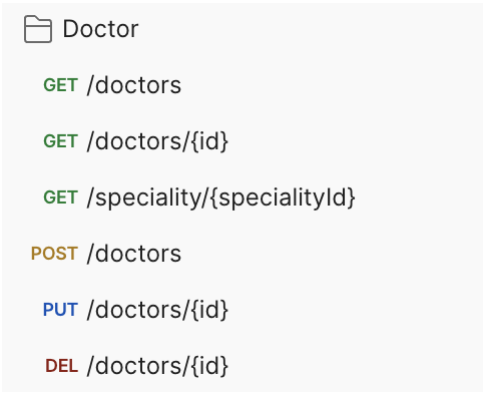

Figure 3.18: Doctor Requests (endpoints)

The **Appointment API** (Figure 3.19) coordinates the scheduling and management of medical appointments. It supports creating new appointments (`POST /appointments/ createAppointment`), retrieving appointments by doctor and user (`GET /appointments/ doctors/id, GET /appointments/users/userId`) and deleting them (`DELETE`

`/appointments/id`). This api ensures proper relationship between patients and doctors while managing time availability and consultation status.

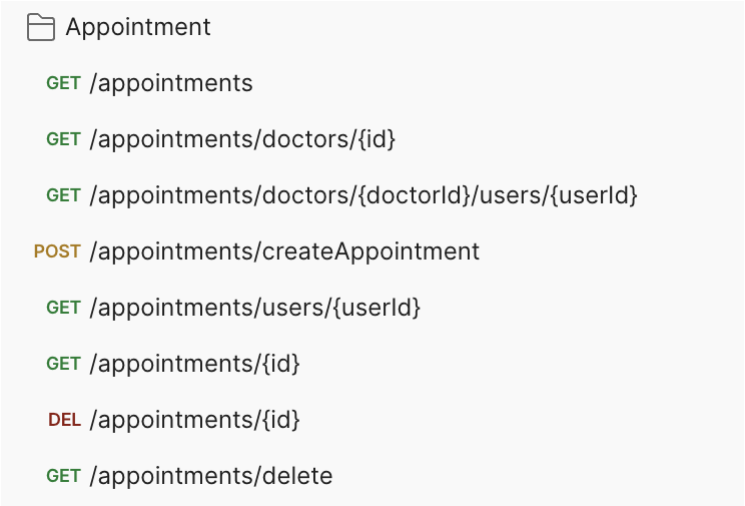

Figure 3.19: Appointment Requests (endpoints)

The **Article API** (Figure 3.20) manages the publication and maintenance of medical content on the platform. It includes endpoints for retrieving `(GET /articles, GET /articles/ id)`, creating `(POST /articles)`, updating `(PUT /articles/id)` and deleting `(DELETE /articles/id)` articles. This service allows administrators to publish verified health information, contributing to patient education and the improve of medical knowledge across the system.

Figure 3.20: Article Requests (endpoints)

The **Contact API** (Figure 3.21) facilitates user communication with the platform's support. It allows message creation `(POST /contact/createContact)`, retrieval `(GET`

`/contact/id)` and management `(PUT, DELETE /contact/id)`. This api enhances user engagement by maintaining an organized and asynchronous communication channel.

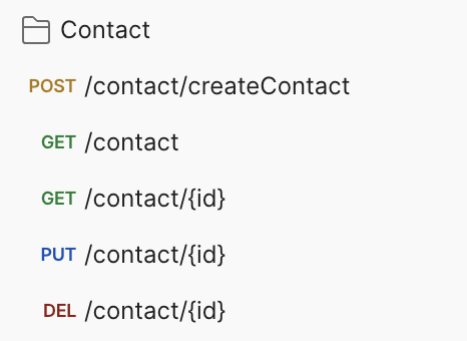

Figure 3.21: Contact Requests (endpoints)

The **Video Call API** (Figure 3.22) enables real-time communication between patients and doctors during remote consultations. Through the endpoint `(GET /videocall/id)`, users can access session information linked to a specific appointment. This api interacts with the Appointment Context, integrating third-party video conferencing tools such as Jitsi Meet to provide a secure and reliable teleconsultation environment.

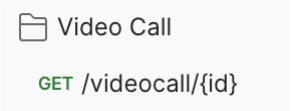

Figure 3.22: Video Call Requests (endpoints)

The **Payment API** (Figure 3.23) manages all payments related to appointments. It provides endpoints for processing payments `(POST /payment/createPayment)` and `(POST /payment/createPayPalPayment)` and supports the initiation of new payment requests. This api is integrated with external payment gateways, such as Stripe and Paypal, to ensure secure handling of transactions between patients and healthcare providers.

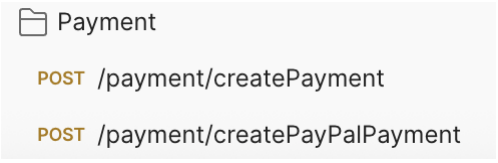

Figure 3.23: Payment Requests (endpoints)

Chapter 4

Development

This chapter describes the development and integration of the system designed based on Chapter 3. The project was designed based on a modern and modular architecture (Table 4.1).

Table 4.1: Technologies used in application development.

Frontend	**Backend**	**External Serv.**	**Database**	**CI/CD**	**Containers**
Angular	Java	Jsonbin.io	SQL	Jenkins	Docker
HTML	Spring Boot	Jitsi	-	-	Kubernetes
CSS	REST API	Stripe	-	-	-
TypeScript	-	PayPal	-	-	-

4.1 User interface implementation

The implementation of the UI relies on a modern and modular web technology that supports the architectural principles of scalability, maintainability and clear separation of concerns. The application is developed using **Angular**, the frontend framework that facilitates the creation of reusable and organized modules.

The interface is implemented using TypeScript as the primary programming language. The visual structure of the application is defined with HTML and CSS, ensuring semantic organization and responsive design across various devices and screen resolutions.

Communication between the client and the backend services occurs through a REST API, using HTTP and JSON as the standard protocols for data exchange. This architecture guarantees that the UI remains lightweight, with the business logic isolated in the backend.

4.1.1 Authentication and Users Workflow

The user experience begins with the authentication system, which serves as the primary gateway to the platform. The authentication interfaces constitute the first point of interaction between users and the system, allowing them to verify their identity and securely access the available features. These interfaces implement well-defined authentication patterns, including form validation and error handling, which guide users through the authentication process while establishing the security foundation for all subsequent interactions.

The authentication flow is centrally managed through the stateless username/password authentication system. This context integrates directly with the backend API using secure HTTP requests, applying mechanisms that protect user credentials. This identifier is stored in the browser's *sessionStorage*, ensuring isolation between user sessions and preventing data overlap across browser contexts.

Relative to user registration, is performed through the `/register` endpoint, which accepts multipart form data containing both textual and optional image inputs. During registration, the system validates user data, encrypts the password using a secure hashing algorithm (`BCryptPasswordEncoder`) and stores the resulting user entity in the database. If an image is provided, it is uploaded to a dedicated storage directory and associated with the user profile. To prevent duplication, the system verifies that no existing account is registered with the same email address.

Authentication is handled through the `/login` endpoint, where user credentials are validated against the stored, encrypted password. Upon successful verification, the corresponding user data are returned, allowing the client-side application to manage access to the system's features according to the user's role and permissions. Although the system

currently performs stateless authentication, it ensures secure password comparison and proper handling of invalid login attempts.

The module also provides mechanisms for password recovery through the `/forgetPassword` endpoint. When a user requests password reset, the system generates a new random password using cryptographically secure random values, encodes it, updates the user record and sends the new credentials via email.

4.1.2 User Management

After authentication, administrators are directed to the administrative dashboard, which serves as the central workspace for managing users. This interface provides a comprehensive overview of all registered accounts, including patients, doctors and other administrators. From this dashboard, administrators can review user information, validate new registrations and manage account status.

At the top of the interface, a search and filtering panel allows administrators to quickly locate specific users by name, email or role. Additionally, the interface provides options to create new accounts manually, update user details or unauthorized profiles. This design facilitates efficient supervision and enables immediate intervention when inconsistencies or security concerns are detected. Through this structured administrative workflow, the platform achieves effective user governance, ensuring that only authorized individuals can manage or modify critical user data.

As illustrated in Figure 4.1, the available administrative actions, such as editing, deleting or revalidating user accounts.

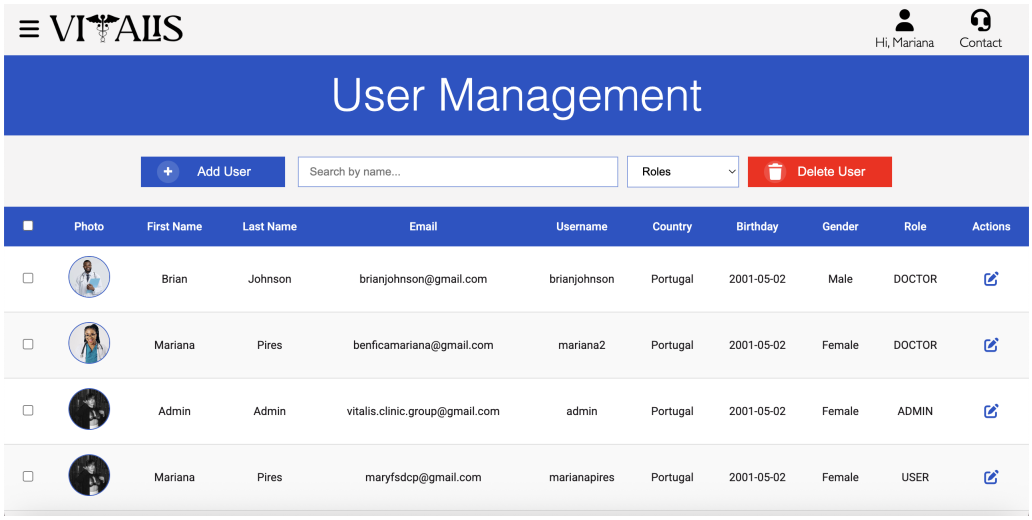

Figure 4.1: Dashboard view of the Admin relative to user data

4.1.3 Doctor Management

After authentication, administrators can access the **Doctor Management Dashboard**, shown in Figure 4.2, which serves as the central interface for organizing and supervising healthcare professionals registered on the platform. This module provides a clear and structured view of all doctors, allowing administrators to review, validate and maintain professional records efficiently.

At the top of the interface, administrators can find an action `toolbar` that includes a `button` to add new doctors, a search field for filtering results by name and a `dropdown menu` to filter by medical specialty. Additionally, a `delete button` is available to remove one or multiple selected profiles when necessary. This layout ensures that administrative actions remain intuitive and easily accessible within a single workspace.

On the right side of each row, an `edit icon` provides quick access to profile management features, allowing administrators to update personal information, modify consultation details or approve pending registrations. The interface enforces permission checks so that only users with administrative privileges can modify or delete doctor profiles, ensuring consistency with the system's role-based access control model.

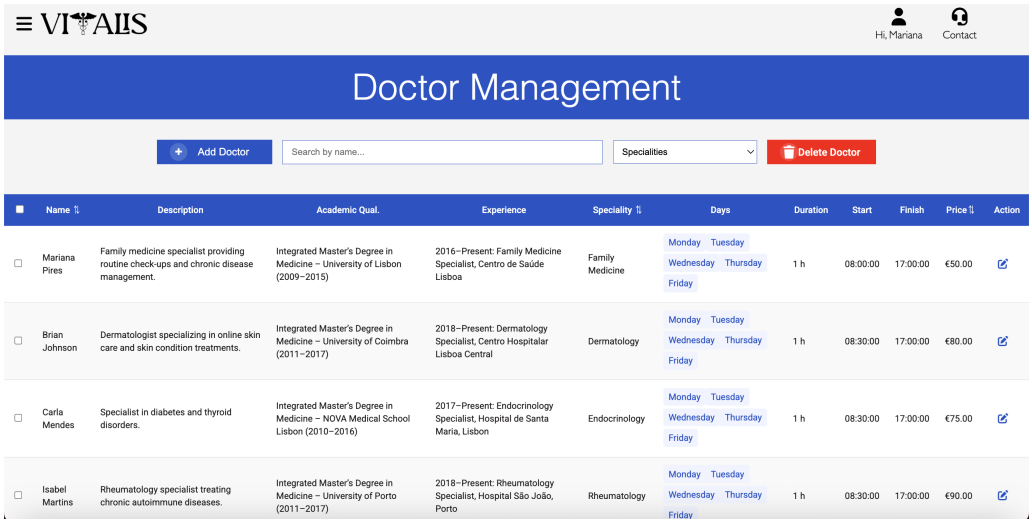

Figure 4.2: Dashboard view of the Admin relative to doctor data

4.1.4 Appointment Management

After authentication, administrators can access the **Appointment Management Dashboard**, which serves as the central interface for supervising and coordinating all scheduled consultations in the platform. As illustrated in Figure 4.3, this workspace provides a comprehensive overview of appointments created by patients and assigned to healthcare professionals, enabling administrators to monitor ongoing activities and ensure the consistency of scheduling operations.

At the top of the interface, administrators have access to a filter and search panel that allows quick retrieval of appointments by doctor or patient. Additional controls enable administrators to update existing ones or remove invalid entries when necessary. This centralization of functionality supports efficient management and promotes transparency across the scheduling process.

The appointment table displays key details such as the patient name, assigned doctor, appointment date and time, duration and status. Each row provides quick access to view, edit or cancel appointments. This structure ensures that administrators can quickly assess the operational state of the system and respond to scheduling issues in real time.

The design of the appointment management interface prioritizes clarity and efficiency,

enabling administrators to maintain control over scheduling activities while ensuring service reliability.

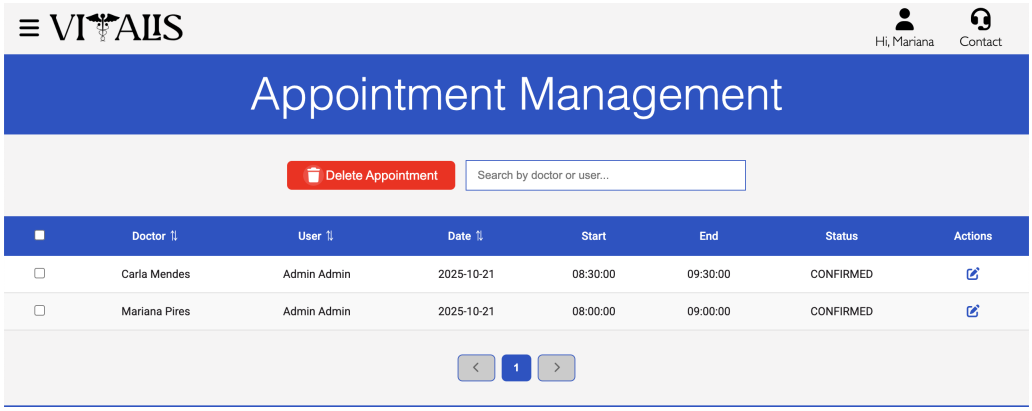

Figure 4.3: Dashboard view of the Admin relative to appointment data

4.1.5 Contact Management

The **Contact Management Dashboard**, shown in Figure 4.4, provides administrators with an organized interface for managing communication between users and the platform's support team. This module plays a key role in maintaining transparency and responsiveness by centralizing all messages and feedback submitted through the system's contact forms.

Administrators can access the dashboard, which displays a list of contact messages sent by patients, healthcare professionals or visitors. At the top of the interface, a `search` and `filter bar` enables retrieval of messages based on sender name and subject.

Each table entry presents essential details, including the sender's name and contact information, the subject of the inquiry, the message body, date and time of submission. Administrators can open individual messages to view their full content and associated details. When necessary, messages may be deleted to maintain a clear and organized communication history.

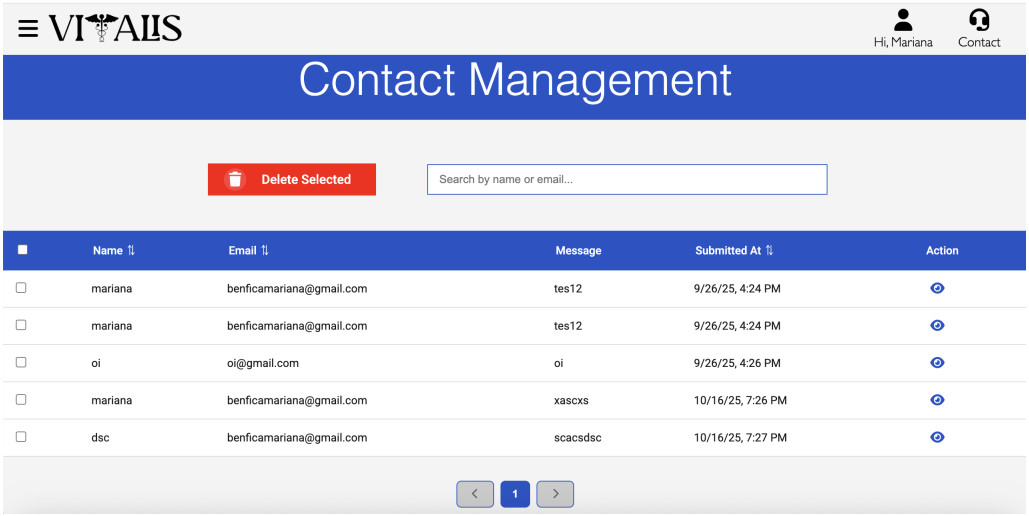

Figure 4.4: Dashboard view of the Admin relative to contact data

4.1.6 Article Management

The **Article Management Dashboard**, shown in Figure 4.5, provides administrators with a centralized environment for creating, editing and organizing informational content in the platform. This module supports the publication of medical articles, healthcare guidelines and educational resources that contribute to patient awareness and professional collaboration.

After authentication, administrators can access the dashboard to manage all articles published on the platform. The interface presents a structured table displaying each article's title, author, category and publication date. At the top of the page, a search and filter bar allows administrators to locate specific articles by title, while action buttons provide options to create new articles or delete.

Administrators can open an article to view or edit its full content, update data and attach relevant images or media. This functionality ensures that the platform's content remains accurate, current and aligned with verified medical information.

Access to the article management module is restricted to users with administrative privileges, preserving data integrity and preventing unauthorized modifications to published materials.

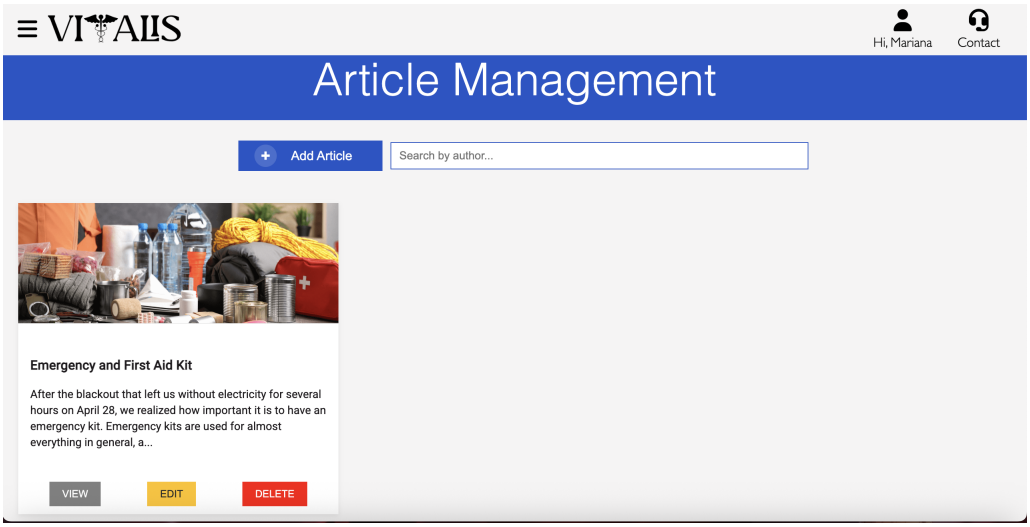

Figure 4.5: Dashboard view of the Admin relative to article data

4.1.7 Appointment

Before accessing the appointment scheduling module, users have two main ways to find the healthcare professional they are looking for. They can either browse through the Clinic Team (Figure 4.6), which displays all doctors registered on the platform or explore the list of medical specialties (Figure 4.7), which groups professionals according to their area of expertise. These two navigation paths provide flexibility for users who either want to select a specific doctor they already know or discover available professionals by specialty.

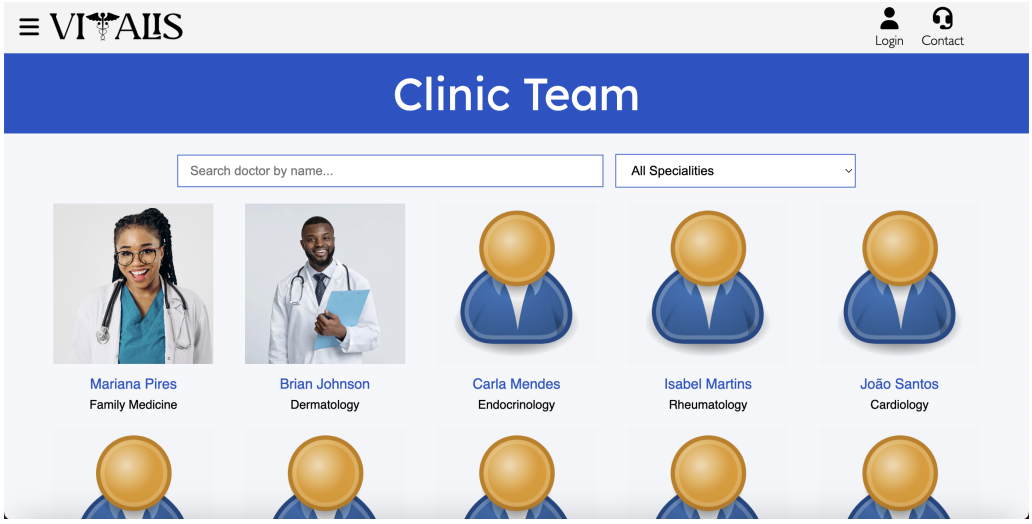

Figure 4.6: Clinic Team

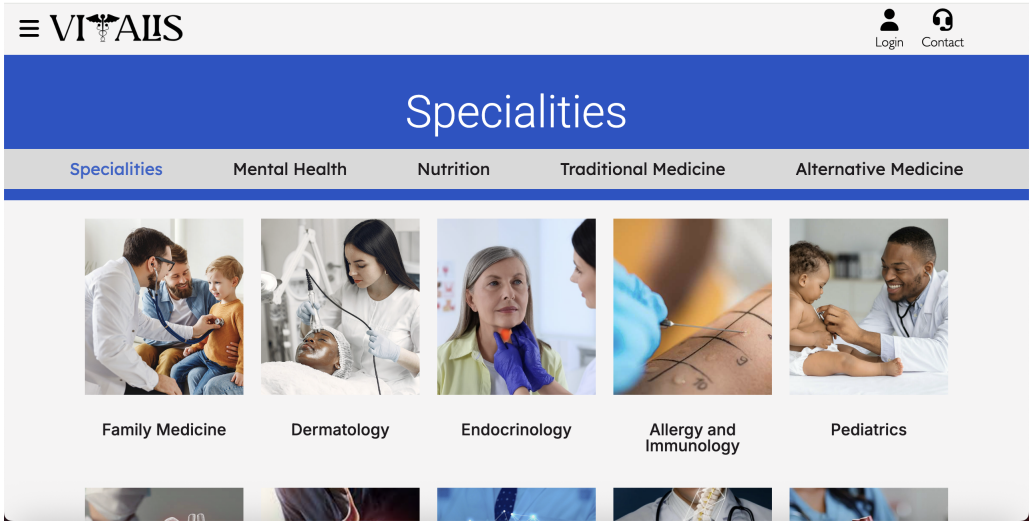

Figure 4.7: Specialities

After selecting a doctor, users are directed to the Appointment Dashboard, which serves as the main interface for managing medical consultations. This module enables patients to schedule, view and modify their appointments with the professionals, while doctors can use the same interface to monitor their own schedules. As illustrated in Figure 4.8, the interface is designed to be intuitive and responsive, ensuring efficient coordination between patients and doctors. Once a suitable time slot is selected, the system validates the doctor's availability and automatically creates a new appointment record. During this process, the interface provides real-time form validation and confirmation messages, ensuring that users are guided clearly (Figure 4.9).

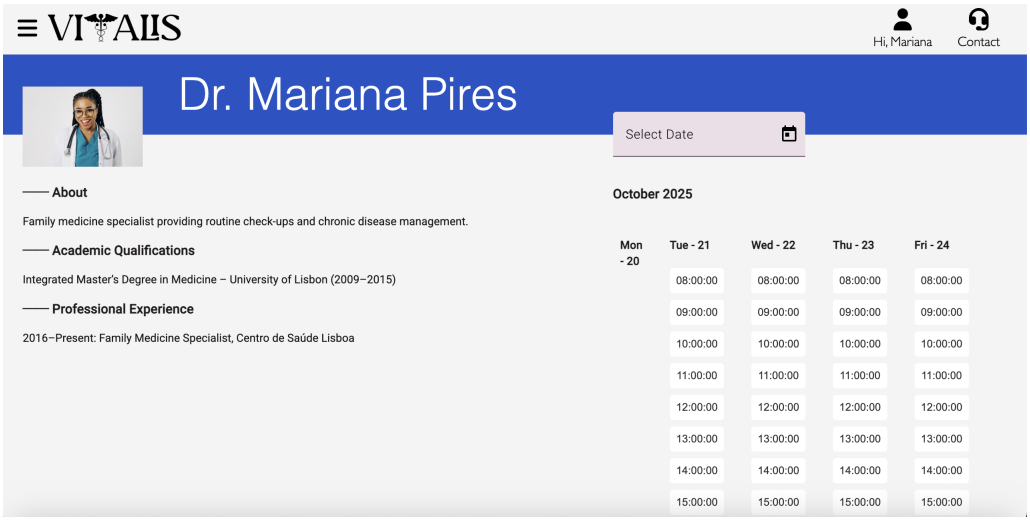

Figure 4.8: Appointment consultation for a specific doctor

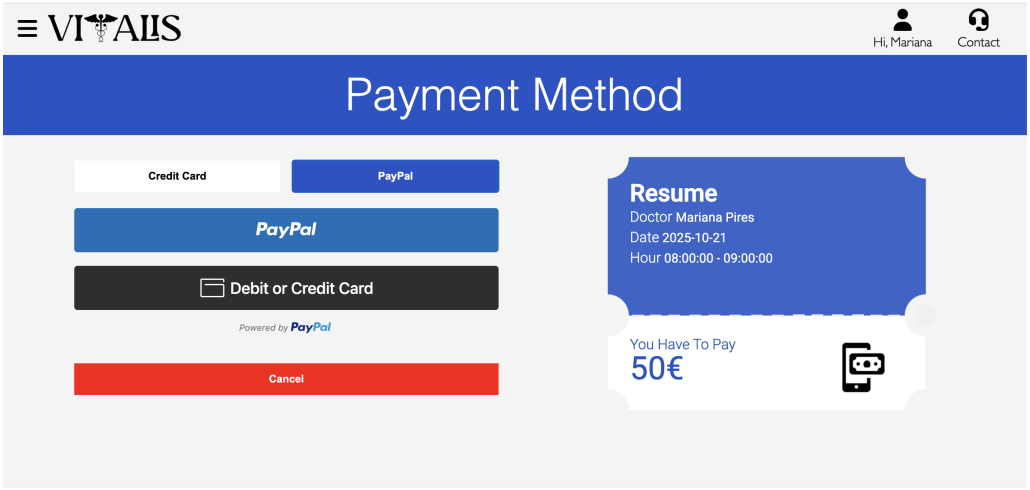

Figure 4.9: Payment appointment consultation for a specific doctor

After the appointment is successfully created, the platform automatically sends an email notification to the patient, confirming the scheduled consultation details such as price, date and time (Figure 4.10). This confirmation email reinforces reliability and provides users with a permanent reference for their upcoming appointment.

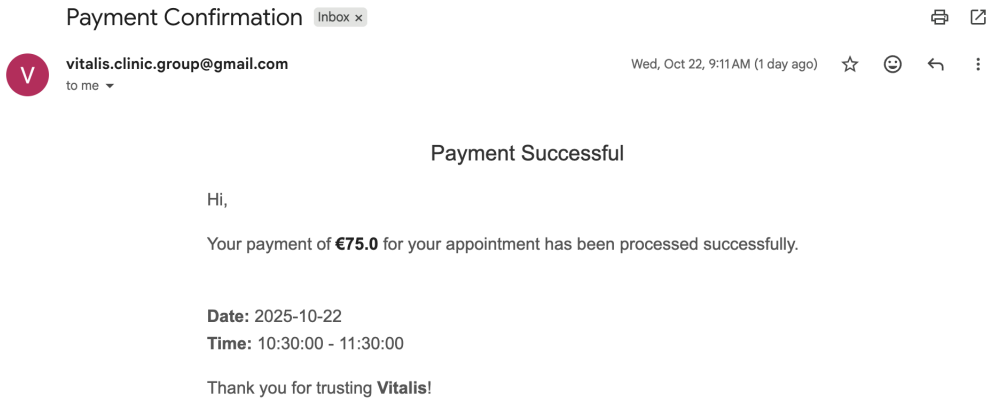

Figure 4.10: Email

The appointment list displays key information such as the doctor's name, specialty, appointment date and time and duration (Figure 4.11). Doctors, in turn, have access to a similar interface tailored to their professional workflow, listing all upcoming consultations along with the respective patient details. Each appointment can also be accessed through a Jitsi Meet video call (Figure 4.12).

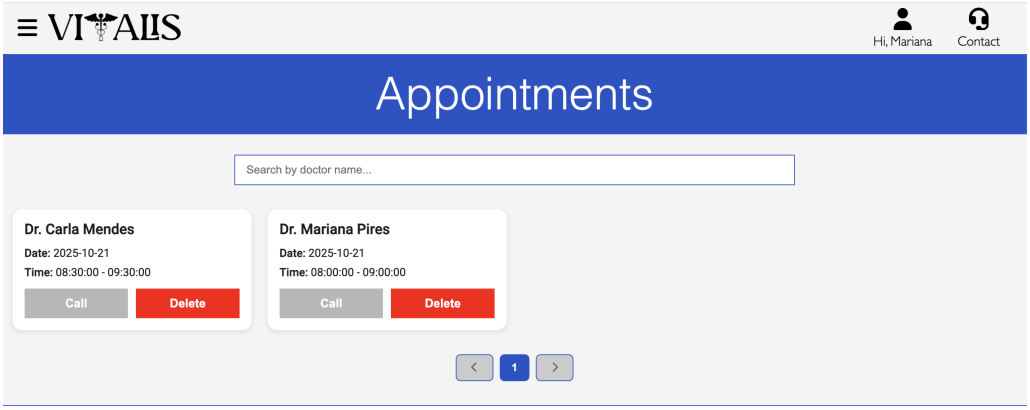

Figure 4.11: Appointment list

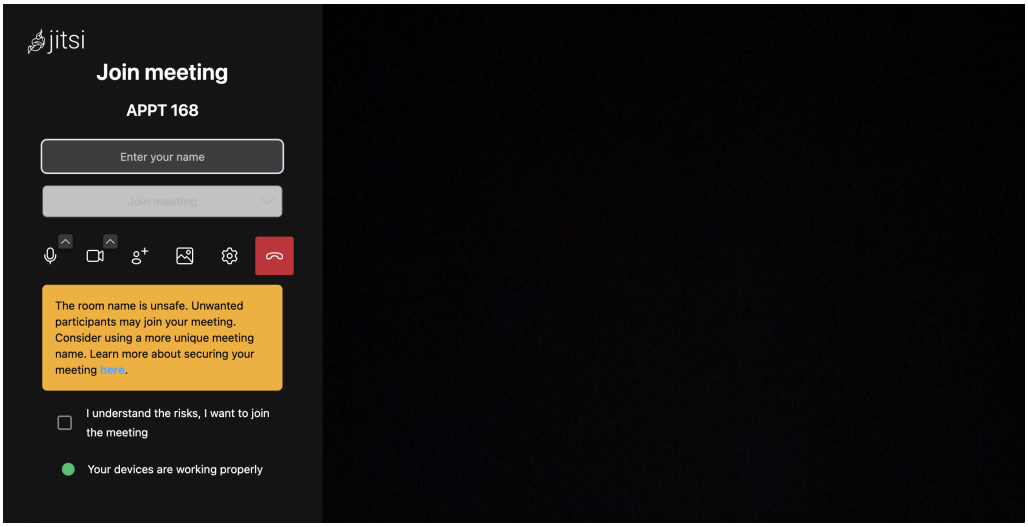

Figure 4.12: Video Call

Beyond scheduling, the dashboard allows direct navigation to related modules such as consultation access and payment confirmation, offering a end-to-end experience from booking to service. Notifications and visual indicators keep users informed of updates, upcoming consultations and status changes, enhancing coordination and minimizing missed sessions.

Through this module, the platform provides a streamlined and user-centered scheduling experience by integrating intuitive navigation, validation mechanisms and real-time synchronization between patients and doctors.

4.2 Database

The database constitutes a critical component for serving as the foundation for data storage and real-time storage. It ensures the integrity, consistency and availability of information across all system modules.

For this project, PostgreSQL was selected as the DBMS due to its robustness, scalability and strong alignment with the architectural principles adopted in the system. PostgreSQL offers advanced features such as transactional integrity, foreign key constraints, indexing and JSON data handling, making it suitable for managing heterogeneous datasets.

The database is responsible for storing and managing all key entities in the system, including users, doctors, appointments, articles and contact messages. Each of these entities corresponds to a specific bounded context, ensuring clear separation of concerns and preserving the modularity defined in the architectural design.

User

The **users** table is integrated into the user management and authentication microservice.

The table has the following attributes:

- **first_name**, **last_name**, **email** and **username**, which identify the user and ensure uniqueness through unique constraints.
- **password**, which is stored in encrypted form to ensure security, using *BCryptPasswordEncoder*.
- **country**, **birthday** and **gender**, that complement the user profile.
- **role**, which defines the type of access within the system (*USER*, *ADMIN*, or *DOCTOR*).
- **image_url**, which references the profile image stored locally on the server.

This table (Figures 4.13) is the focus of the application's authentication and authorization system, responsible for ensuring secure and controlled access to the different pages on the application.

Doctor

The **doctors** table belongs to the microservice dedicated to managing healthcare professionals. It contains detailed information about each registered doctor, from their personal details to their work schedules and consultation prices.

The table has the following attributes:

- **first_name** and **last_name**, which identify the doctor.

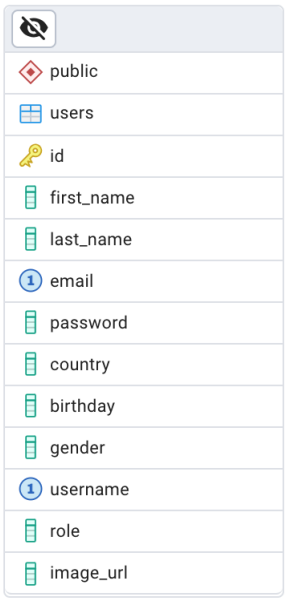

Figure 4.13: Users table

- **description**, **academic_qualification** and **professional_experience**, which describe the professional and academic profile.

- **speciality_id**, which associates the physician with his or her medical specialty.

- **work_days**, **consultation_time**, **start_time** and **finish_time**, which define the schedule and duration of available appointments.

- **price**, which indicates the consultation price.

- **user_id**, which establishes the connection with the corresponding user account in the users table.

The table (Figures 4.14) was designed to support features such as searching for doctors by specialty, displaying professional profiles and direct integration with the appointment scheduling system.

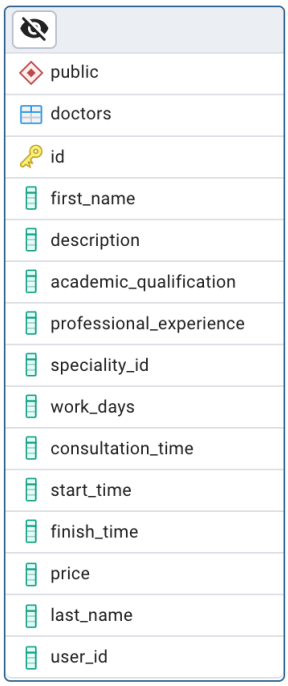

Figure 4.14: Doctors table

Appointment

The **appointments** table belongs to the microservice responsible for managing medical appointments. This database centralizes all information related to appointments between doctors and patients, ensuring control and traceability of each video call session.

The table has the following attributes:

- **doctor_id** and **user_id**, which establish the relationships between physicians and users.
- **date**, **start_time** and **end_time**, which define the time interval of the video call.
- **status**, which indicates the current status of the booking.

This table (Figures 4.15) allows you to effectively manage the appointment lifecycle, supporting features such as listing, scheduling and medical history.

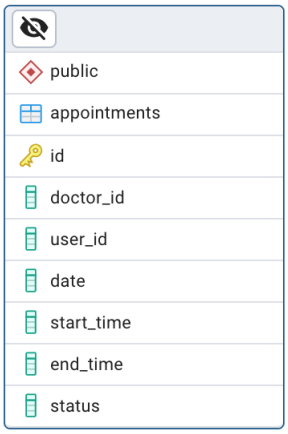

Figure 4.15: Appointments table

Article

The **articles** table is part of the microservice responsible for managing informational articles available on the platform. Each record corresponds to an article created by an administrator or authorized physician, contributing to the promotion of health literacy among users.

The table has the following attributes:

- **title**, **author** and **content**, which store the title, author and body text of the article.
- **publish_date**, which records the date of publication.
- **image_url**, which refers to the associated illustrative image.

Through this structure (Figures 4.16), it is possible to dynamically present updated and relevant content, reinforcing the application's medical information.

Contact

The **contact** table integrates the microservice designed to manage communications between users and the application team. New records are automatically added via the contact form, sending immediately an email to notify the responsible team.

Figure 4.16: Articles table

The table has the following attributes:

- **name**, **email** and **message**, which identify the sender and the content sent.
- **submitted_at**, which records the date and time of submission.

This structure (Figure 4.17) allows for organized monitoring and responding to received messages, ensuring effective and centralized communication between users or visitors and administrators.

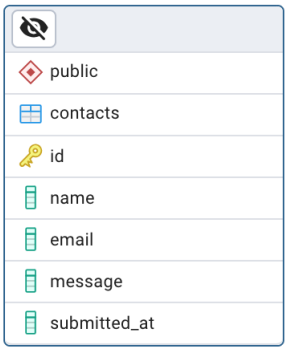

Figure 4.17: Contacts table

4.3 Containerization

To enable uninterrupted integration between all microservices, the platform was fully containerized using Docker and Docker Compose. Containerization ensures that each application component runs in an isolated environment, independent of the host operating system, guaranteeing portability and reproducibility across development, testing and production environments. This approach eliminates dependency conflicts between services and simplifies deployment, aligning with the architectural principles of scalability and modularity established in the system design.

Each microservice is encapsulated with its own Docker container. The Dockerfile defines all necessary configurations and dependencies for each service, ensuring consistent execution environments. For the microservices developed in Java using the Spring Boot framework, a multi-stage build strategy was implemented to optimize image size and runtime performance. This process (Listening 4.1) follows a sequence of well-defined steps, summarized as follows:

- **Base image selection:** the image is used as the foundation for compiling and running the application.
- **Working directory configuration**: the application files are placed inside the `/app` directory within the container, serving as the execution environment.
- **Dependency management and build**: the mvnw tool is used to install dependencies and compile the source code. The command `RUN ./mvnw clean package -DskipTests` generates the final executable package.
- **Image optimization**: the second stage of the Dockerfile copies only the generated .jar file from the build stage, minimizing the final image size and excluding unnecessary build dependencies.
- **Port exposure and entry point**: the container exposes for a specific port, each service have their own, for communication and specifies the application's startup

command using `ENTRYPOINT ["java", "-jar", "/app/app.jar"]`.

```
FROM ubuntu:latest
LABEL authors="marianapires"

FROM openjdk:21-jdk-slim AS builder
WORKDIR /app
COPY mvnw pom.xml ./
COPY .mvn .mvn
COPY src src
RUN ./mvnw clean package -DskipTests


FROM openjdk:21-jdk-slim
WORKDIR /app
COPY --from=builder /app/target/*.jar app.jar
EXPOSE 8087
ENTRYPOINT ["java", "-jar", "/app/app.jar"]
```

Listing 4.1: Exemple of Dockerfile

Once defined, the image can be created using the docker build command. In case of build errors, Docker provides detailed logs identifying the cause of the issue, facilitating debugging and image refinement. The resulting container can then be executed independently and orchestrated using Docker Compose, which manages service communication and network configurations.

This containerization strategy supports CI/CD practices, allowing automatic builds, testing and updates of microservices.

By adopting Docker and Docker Compose, the platform achieves high levels of portability, stability and maintainability. The resulting architecture ensures that all services can be easily deployed, updated or replicated across environments, maintaining consistent behavior and performance regardless of the underlying infrastructure.

4.4 Jenkins Pipeline Implementation

The integration of `Jenkins` in the development workflow played a central role in automating and streamlining the CI/CD process. Jenkins acted as the orchestration layer responsible for managing the compile, build and deployment stages of each microservice, ensuring consistency and reliability in the entire development pipeline.

This automation eliminated the need for manual intervention during the build and deployment phases, significantly reducing potential human error and improving overall delivery speed. Each pipeline stage, from the compilation of source code to the creation of Docker images and their deployment into `Artifact Registry`, was executed in a controlled and reproducible environment.

The pipeline follows a structured sequence of stages, each responsible for a specific task in the CI/CD lifecycle (Listining 4.2):

- **Compile project**: Initiates a clean compilation using Maven or NodeJS, ensuring that the latest version of the codebase is built.
- **Docker build**: Packages the compiled project into a Docker image.
- **Push to Artifact Registry**: Authenticates with the Google Cloud service account, tags the image according to the defined naming convention and uploads it to the `Artifact Registry`, from which it can later be deployed to GKE.

```
stages {
    stage('Compile project') {
        steps {
            dir('/home/vitalis_clinic_group/videocallService') {
                sh 'mvn clean compile'
            }
        }
    }

    stage('Docker Build') {
```

```
        steps {
            dir('/home/vitalis_clinic_group/videocallService') {
                sh '''
                    docker images -af reference="$IMAGE_NAME" -q || true
                    docker build -t "$IMAGE_NAME":latest .
                    docker image prune -f
                    '''
            }
        }
    }

    stage('Push to Artifact Registry') {
        steps {
            withCredentials([file(credentialsId: 'gcloud-creds',
               variable: 'GC_CLOUD_CREDS')]) {
                dir('/home/vitalis_clinic_group/videocallService') {
                    sh '''
                        gcloud auth activate-service-account --key-file
                           ="$GC_CLOUD_CREDS"
                        gcloud auth configure-docker "$LOCALIZATION"-
                           docker.pkg.dev
                        docker tag "$IMAGE_NAME" "$LOCALIZATION"-docker.
                           pkg.dev/"$PROJECT_ID"/"$ARTIFACT_REPOSITORY
                           "/"$IMAGE_NAME"
                        docker push "$LOCALIZATION"-docker.pkg.dev/"
                           $PROJECT_ID"/"$ARTIFACT_REPOSITORY"/"
                           $IMAGE_NAME"
                        '''
                }
            }
        }
    }
}
```

Listing 4.2: Exemple of Pipeline

This automation pipeline enables consistent and repeatable deployments across all microservices, minimizing manual intervention and ensuring efficient coordination between the different stages of software delivery, from code integration to production deployment.

4.5 Kubernete Integration in GCP

The deployment of the platform's microservices was performed using Kubernetes, running on GCP. This setup provides a managed orchestration environment capable of automating the deployment, scaling and monitoring of containerized services. By using GKE, the system achieves greater operational efficiency and consistency across distributed microservices, while ensuring scalability and fault tolerance.

In the Kubernetes architecture (Figure 4.18), the **Control Plane** is the central management layer responsible for maintaining the overall state of the cluster. When deploying the platform through GKE, the control plane is automatically provisioned and managed by GCP, abstracting the complexity of cluster administration. This managed service ensures continuous availability, automated updates and integration with other GCP components such as monitoring, networking and identity management.

The control plane oversees the communication between the user and the Kubernetes cluster. It exposes the Kubernetes API, which serves as the primary interface for cluster operations, including the deployment, scaling and configuration of microservices. Every action, such as creating a Deployment, applying a ConfigMap or updating a Service, is processed by the control plane, which ensures that the desired cluster state defined by the developer matches the actual runtime state.

The control plane is composed of several key components, each responsible for a specific aspect of cluster management:

- **kube-apiserver** – Acts as the communication hub of the cluster, handling all REST API requests. It validates and processes commands from users, CI/CD pipelines or internal services, ensuring that all changes to the cluster are properly authenticated and authorized.

- **etcd** – A distributed key-value store that functions as the system's database. It stores all cluster configuration data and the current state of Kubernetes objects, ensuring consistency and reliability across replicas.

- **kube-scheduler** – Responsible for assigning Pods to nodes. It evaluates resource availability, node conditions and workload constraints to determine the optimal node for each new Pod.

- **kube-controller-manager** – Executes background control processes that regulate the cluster's state. It ensures that the desired number of replicas, network routes and service endpoints are always running according to the defined specifications.

- **cloud-controller-manager** – Integrates Kubernetes with the underlying cloud infrastructure. In GKE, it manages operations such as load balancing, node provisioning and volume attachment through Google Cloud APIs.

By coordinating these components, the control plane continuously monitors the health of the cluster and automatically corrects discrepancies between the desired and actual states.

In GKE, the control plane operates in a fully managed mode, running on dedicated infrastructure separate from user workloads. Through this architecture, the control plane functions as the **brain of the Kubernetes cluster**, orchestrating all operations and maintaining synchronization across distributed microservices.

Each microservice is deployed as an independent container in the Kubernetes cluster. This architecture promotes isolation between services while maintaining communication through internal networking and service discovery mechanisms provided by Kubernetes.

The implementation follows a consistent configuration model using essential Kubernetes resources such as ConfigMaps, Secrets, Deployments, Services and Persistent Volume (PV)/Persistent Volume Claim (PVC). These elements work together to manage configuration data, handle sensitive information securely and ensure that each microservice remains highly available and resilient to failures.

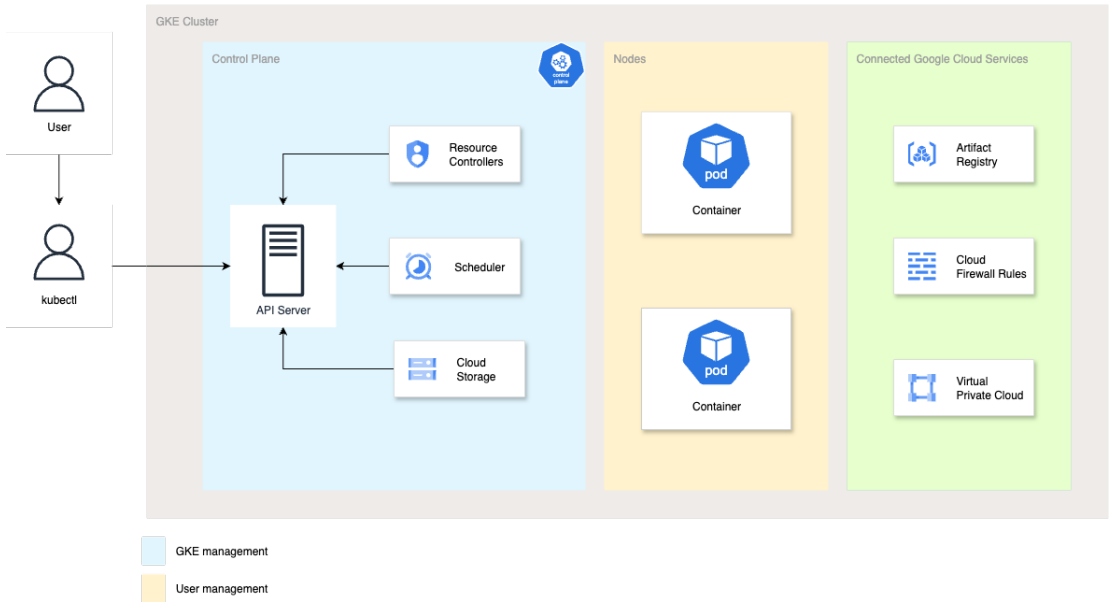

Figure 4.18: GKE cluster architecture based on GCP documentation

ConfigMap

In the Kubernetes environment, a **ConfigMap** was created to manage the configuration data required by the platform's microservices. This object stores environment variables in key–value format, allowing services to access configuration parameters dynamically without embedding them directly into the container images. By centralizing configuration in a ConfigMap, the platform maintains consistency across microservices and simplifies updates, as changes can be applied without rebuilding the images.

PV

A **Persistent Volume** represents a unit of storage provisioned in the Kubernetes cluster. It can be created statically by an administrator or dynamically through a storage class. Persistent Volumes abstract the physical storage details, providing a consistent interface for data persistence and management.

PVC

A **Persistent Volume Claim** is a user request for storage resources in the cluster. It defines specific requirements such as capacity, access mode and performance class. Once created, Kubernetes automatically binds the claim to a suitable **PV**, allowing applications to use persistent storage without direct administrative configuration.

Secret

A **Secret** is an API object designed to store sensitive information such as passwords, tokens and private keys. Secrets are securely injected into **Pods** as environment variables or mounted files, enabling applications to access credentials. This mechanism enhances security and ensures that sensitive data remains isolated from non-secure resources.

Deployment

A **Deployment** defines and manages a set of Pods that run an application workload. It ensures that the specified number of replicas are always running and automatically replaces failed or outdated instances during updates. Deployments provide declarative configuration and support rolling updates, contributing to high availability and simplified maintenance.

Service

A **Service** defines a stable network endpoint that exposes one or more **Pods** running an application. It provides an abstraction layer that allows communication between components regardless of *Pod IP* changes. Services can expose applications externally through a LoadBalancer or internally in the cluster using a ClusterIP, ensuring consistent connectivity and traffic distribution across replicas.

Chapter 5

Results

In this chapter, the tests conducted on the Vitalis platform are presented and discussed in order to evaluate the functionality. The objective of this testing phase was to validate whether all modules, including user management, appointment scheduling, communication and administrative operations, functioned correctly and in accordance with the project's defined requirements.

5.1 Docker Integration

Each component of the platform, including the backend, the frontend, the database and the API Gateway, was encapsulated within its own Docker container (Figure 5.1), each defined through a dedicated Dockerfile.

	Name	Container ID	Image	Port(s)
☐	vitalis	-	-	-
☐	doctor-service AMD64	8bb07c6d9785	vitalis-doctor-service	8081:8081
☐	user-service AMD64	d8f5f71e22d2	vitalis-user-service	8083:8083
☐	appointment-service AMD64	ba4e22951148	vitalis-appointment-service	8082:8082
☐	article-service AMD64	6356d31b6642	vitalis-article-service	8086:8086
☐	contact-service AMD64	3b9214fdb9f2	vitalis-contact-service	8087:8087
☐	video-call-service AMD64	1beac597dfa1	vitalis-video-call-service	8085:8085
☐	postgres AMD64	2beb81f35638	postgres:17	
☐	jitsi-meet AMD64	c7836720ecdc	jitsi/web:stable	8443:443
☐	health-service AMD64	c5355a01296a	vitalis-health-service	8080:8080
☐	payment-service AMD64	9f3d06c40ca6	vitalis-payment-service	8084:8084
☐	frontend AMD64	8a1f56e5815b	vitalis-frontend	4200:4200
☐	api-gateway AMD64	d14d2e372661	vitalis-api-gateway	8088:8088

Figure 5.1: Containers

Subsequently, by orchestrating these containers and exposing the necessary network ports, a local environment was established in which the entire application could be executed. Through the frontend URL automatically generated during the container deployment process, it became possible to access and test the integrated system locally (Figure 5.2).

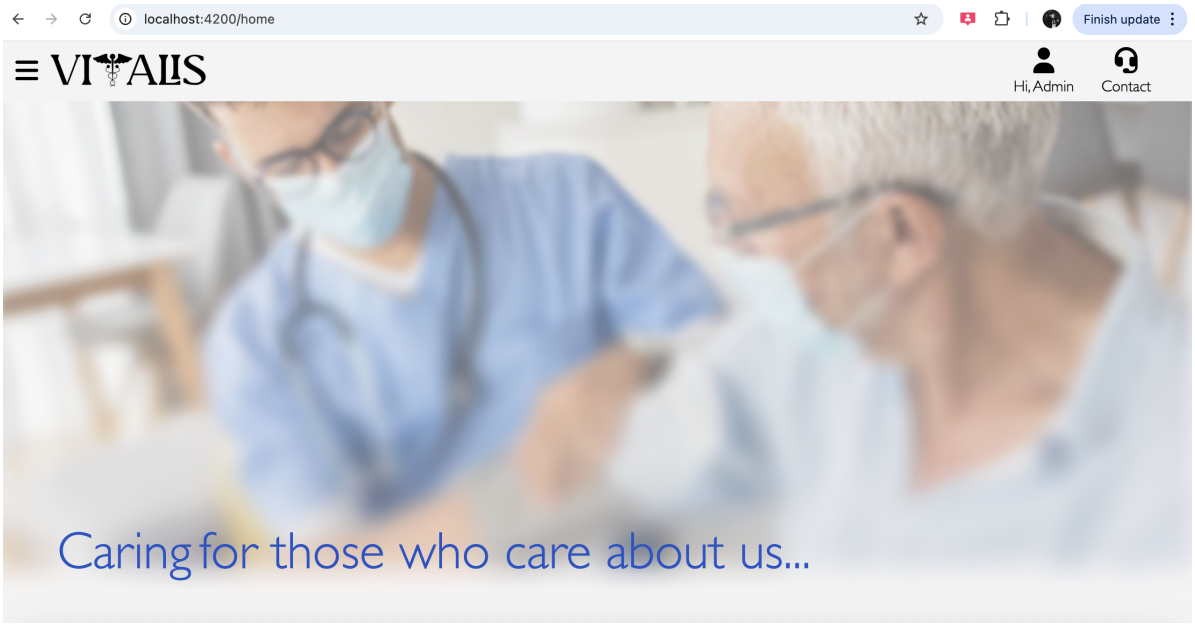

Figure 5.2: Local application through Docker (localhost)

5.2 Kubernetes Integration

In the Kubernetes environment, each deployed microservice is encapsulated in a dedicated pod (Figure 5.3), ensuring isolated execution and resource management.

```
vitalis_clinic_group@cloudshell:~ (ardent-depth-474623-t8)$ kubectl get pods
NAME                           READY   STATUS    RESTARTS        AGE
api-gateway-6b684cfd64-2jqzl   1/1     Running   0               4h50m
appointment-7446887ff9-d8twl   1/1     Running   2 (4h49m ago)   4h50m
contact-5754665df6-sj7jl       1/1     Running   1 (4h49m ago)   4h50m
doctor-575b6bfc8b-nm57w        1/1     Running   2 (4h49m ago)   4h50m
frontend-65b89696f6-5tskr      1/1     Running   0               4h40m
health-77dd876f4b-p8rw2        1/1     Running   0               4h50m
jitsi-meet-59d988b5b6-gnvrb    1/1     Running   0               4h50m
payment-566b486c49-7bs58       1/1     Running   0               4h50m
postgres-75d46fdd4b-7f9c7      1/1     Running   0               4h50m
user-86bccfc985-9cg77          1/1     Running   0               4h50m
video-call-55cc54d54c-7sgpc    1/1     Running   0               4h50m
```

Figure 5.3: Pod for each deployment

This setup not only facilitates scalability and fault tolerance but also provides a clear overview of the system's operational state. As shown in Figure 5.4, the configuration enables visualization of all pods with their corresponding services, which coordinate both internal and external communication between the different components of the platform.

```
vitalis_clinic_group@cloudshell:~ (ardent-depth-474623-t8)$ kubectl get svc
NAME          TYPE           CLUSTER-IP       EXTERNAL-IP     PORT(S)          AGE
api-gateway   LoadBalancer   34.118.231.89    34.38.138.58    8088:31474/TCP   5h20m
appointment   ClusterIP      34.118.235.66    <none>          8082/TCP         5h20m
article       ClusterIP      34.118.238.20    <none>          8086/TCP         5h20m
contact       ClusterIP      34.118.234.249   <none>          8087/TCP         5h20m
doctor        ClusterIP      34.118.239.8     <none>          8081/TCP         5h20m
frontend      LoadBalancer   34.118.229.236   35.189.219.75   80:31956/TCP     5h20m
health        ClusterIP      34.118.229.137   <none>          8080/TCP         5h20m
jitsi-meet    ClusterIP      34.118.230.140   <none>          8443/TCP         5h20m
kubernetes    ClusterIP      34.118.224.1     <none>          443/TCP          5h25m
payment       ClusterIP      34.118.225.84    <none>          8084/TCP         5h20m
postgres      ClusterIP      34.118.233.39    <none>          5432/TCP         5h20m
user          ClusterIP      34.118.225.65    <none>          8083/TCP         5h20m
video-call    ClusterIP      34.118.229.213   <none>          8085/TCP         5h20m
```

Figure 5.4: Load Balancer and ClusterIP Services

Consequently, the Figure 5.5 shows the application running successfully in the Kubernetes environment .

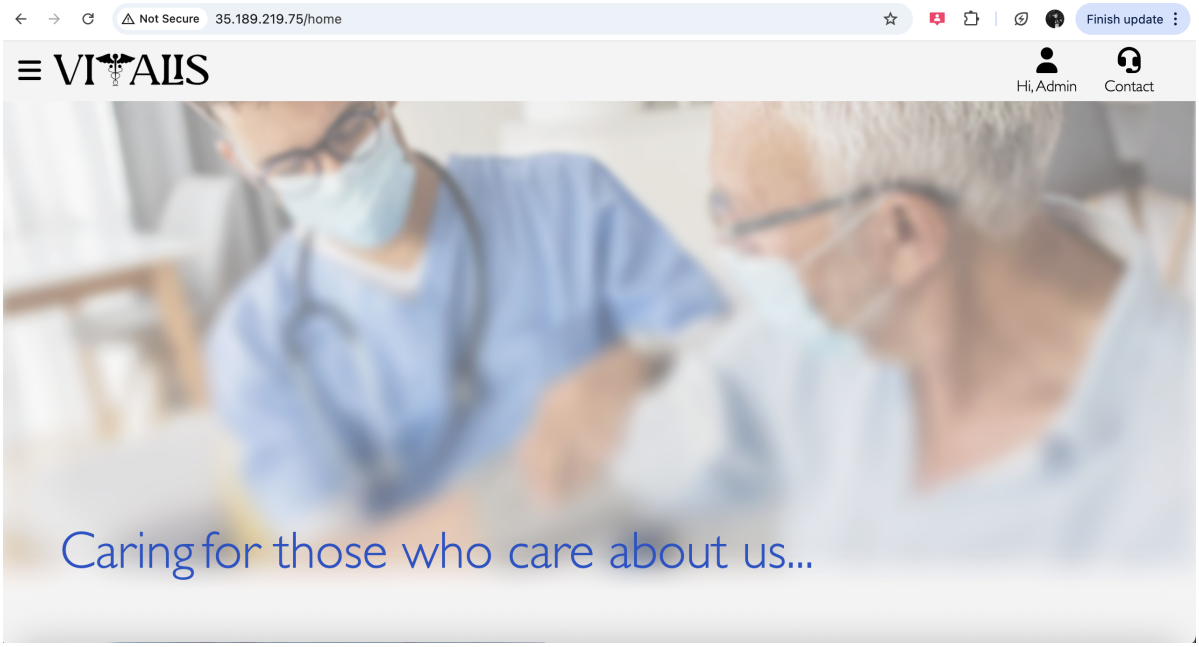

Figure 5.5: Online application through Kubernetes (Frontend LoadBalancer)

5.3 System Performance Evaluation

5.3.1 CI/CD Performance

In this section, a detailed performance analysis of the **Continuous Integration and Continuous Deployment (CI/CD)** process is presented, focusing on the time measurements recorded for each `Docker` stage across all implemented microservices. The objective is to compare the execution times obtained **locally using Docker** with those recorded **in the automated `Jenkins` environment**, in order to evaluate pipeline efficiency and identify potential performance issues. This comparison provides valuable insight into how automation affects deployment time and resource utilization, particularly when operating under limited computational capacity and cost constraints.

The Docker evaluation is divided into three stages:

1. `docker build`

2. `docker tag`

3. `docker push`

This representing, respectively, the image build process, image tagging and the upload to the cloud artifacts *Google Artifact Registry*.

The `docker build` phase corresponds to the local construction of Docker images. This stage is typically the most computationally demanding, as it involves dependency resolution, code compilation and layered image creation. The recorded build times for each service are shown in Table 5.1.

Table 5.1: `docker build` Times

Services	**Total Time**
api-gateway	0m50s984ms
appointment-service	0m2s426ms
article-service	0m1s758ms
contact-service	0m1s588ms
doctor-service	0m1s479ms
frontend	0m41s229ms
health-service	0m1s833ms
payment-service	0m1s714ms
user-service	0m1s519ms
video-call-service	0m34s653ms
Total	**2m57s183ms**

The `docker tag` phase represents the process of labeling the images with the appropriate repository and version information prior to pushing them to the registry. Although it is a superficial step in terms of resource usage, this is critical to maintain consistent traceability and version control across environments. The results are presented in Table 5.2.

Table 5.2: `docker tag` Times

Services	**Total Time**
api-gateway	0m0s95ms
appointment-service	0m0s38ms
article-service	0m0s61ms
contact-service	0m0s41ms
doctor-service	0m0s41ms
frontend	0m0s58ms
health-service	0m0s51ms
payment-service	0m0s43ms
user-service	0m0s43ms
Total	**0m0s421ms**

The `docker push` phase measures the time required to upload the built images to the cloud artifacts (*Google Artifact Registry*). This phase mainly depends on the available network speed and the responsiveness of the *Artifact Registry*, which often results in the highest variability in execution time. The results are presented in Table 5.3.

Table 5.3: `docker push` Times

Services	**Total Time**
api-gateway	5m6s970ms
appointment-service	0m8s787ms
article-service	0m6s896ms
contact-service	0m6s995ms
doctor-service	5m6s740ms
frontend	0m17s300ms
health-service	0m10s17ms
payment-service	0m6s748ms
user-service	5m7s50ms
video-call-service	0m4s687ms
Total	**16m22s190ms**

In the Table 5.4 presents a consolidated summary of the total time spent on each Docker command.

Table 5.4: Total time by Docker

Command	**Total Time**
`docker build`	2m57s183ms
`docker tag`	0m0s421ms
`docker push`	16m22s190ms
Total	**19m19s794ms**

In contrast, the execution of the same process in the Jenkins environment revealed slightly different results, as shown in Table 5.5. The Jenkins pipeline, operating in a controlled cloud environment with the minimum possible allocated resources (CPU and memory), achieved a total execution time of **15m06s**. Although the difference is not substantial, the previously independent Docker operations are consolidated into a single automated workflow, simplifying DevOps management and streamlining the overall image deployment process.

This demonstrates that the adoption of an automated CI/CD pipeline not only ensures greater consistency and scalability but also delivers a more efficient deployment process, with minimal performance compared to local execution.

Table 5.5: Total time by Jenkins

Services	**Total Time**
api-gateway	1m18s
appointment-service	1m57s
article-service	1m17s
contact-service	1m17s
doctor-service	1m24s
frontend	3m38s
health-service	1m22s
payment-service	1m8s
user-service	1m25s
video-call-service	1m
Total	**15m06s**

5.3.2 Kubernetes Performance

To assess the performance and resource efficiency of the deployed microservices, the platform's operational metrics were monitored directly in the Kubernetes cluster. In Table 5.6 presents the CPU and memory utilization for each Pod during normal system operation. These measurements were collected through the Kubernetes monitoring interface to evaluate how the computational resources are distributed across the different services.

The data show that the **frontend** component presents the highest CPU and memory consumption, which is expected due to its responsibility for handling user interactions and rendering dynamic content. **Backend services** also exhibit moderate resource usage, reflecting their central role in request routing and business logic processing. In contrast, auxiliary services such as **PostgreSQL and Jitsi Meet** demonstrate relatively low utilization levels, as they operate mainly in response to specific events, resulting in reduced activity during periods of low demand.

This evaluation confirms that the microservices architecture operates efficiently under the configured resource limits. The balanced distribution of CPU and memory usage among Pods ensures system scalability and stability, while preventing resource saturation.

Table 5.6: Current Usage Pod

Pod	**CPU**	**Memory**
api-gateway	354m	153Mi
appointment-service	353m	145Mi
article-service	356m	131Mi
contact-service	366m	159Mi
doctor-service	357m	142Mi
frontend	793m	327Mi
health-service	184m	80Mi
jitsi-meet	0m	29Mi
payment-service	206m	82Mi
postgres	17m	21Mi
user-service	451m	145Mi
video-call-service	224m	84Mi

Chapter 6

Conclusion and Future Work

The development of the telemedicine platform made possible to design and implement a customized and efficient health management solution. The main objective of this project was to promote accessibility and digital inclusion in healthcare through the use of technological innovation, particularly in the field of telemedicine. The resulting system confirms the potential of modern software technologies to support the delivery of secure, scalable and user-centered medical services.

The platform was built using a set of modern technologies, including Spring Boot, Angular, Docker and Kubernetes, which together provided a robust foundation for scalability and maintainability. The use of containerization and orchestration allowed the system to be easily deployed and managed within a cloud environment, ensuring both reliability and portability. At the architectural level, the application of Domain-Driven Design DDD principles enabled a clear separation of responsibilities through the definition of multiple bounded contexts, representing the main subdomains of the telemedicine business model.

The adoption of a microservices architecture was essential in achieving modularity and independence between components. Each microservice encapsulates its own logic, database and configuration, simplifying development and enabling independent scaling. This modular approach also facilitated maintenance and testing, as updates to one service didn't interfere with the operation of others.

From a user perspective, the platform provides an organized structure that supports four distinct roles, patient, doctor, administrator and visitor. Each profile has permissions and functionalities, ensuring a personalized experience and a logical flow of interactions in the system. The system's usability and its ability to integrate multiple services demonstrate the effectiveness of the architectural model and the success of the project relative to the initial objectives.

6.1 Future Work

Although the platform successfully implements the main functionalities required for a telemedicine environment, several opportunities for improvement and expansion remain. The following topics outline potential directions for future development.

Implementation of JWT for Authentication and Authorization

A key enhancement for future iterations of the system is the adoption of JSON Web Tokens (JWT) for authentication and authorization. Currently, the application uses *session-based* authentication, which is effective but less suitable for large-scale distributed architectures. With JWT, the authentication process becomes stateless, allowing microservices to verify user credentials independently without maintaining a shared session state.

This approach would improve scalability and interoperability across services, as the token can securely store user information and permissions. JWT also aligns with widely adopted standards such as *OAuth 2.0* and *OpenID Connect*, facilitating potential integrations with third-party systems, mobile applications or external healthcare providers. The implementation of JWT will therefore strengthen the platform's security, flexibility and compatibility with cloud hyperscalers.

Enhancement of Message-Driven Communication with ActiveMQ

Another significant improvement involves the integration of *ActiveMQ* as a message broker to manage asynchronous communication between microservices. Currently, the system

relies mainly on communication, which can introduce coupling and latency as services scale.

By introducing ActiveMQ, the platform can adopt an event-driven architecture, allowing services to communicate through message queues rather than direct requests, although the frontend already implements asynchronous functions to handle non-blocking operations and improve user responsiveness, ActiveMQ extends this asynchronous behavior to the backend microservices, enhancing scalability, reliability and communication flexibility between services.

Extension of the Payment Subdomain

The Payment Subdomain can be extended to include invoice generation and subscription management. These additions would improve the system's financial management capabilities, making it suitable for clinics or healthcare organizations that operate under different billing models.

Performance and Scalability Optimization

To ensure that the platform can handle increased workloads, performance testing should be conducted under simulated production conditions. The use of GKE cluster autoscaling in Kubernetes would allow automatic adjustment of computing resources according to demand, ensuring efficient scaling and resource utilization. However, due to budget limitations during the development phase, certain adjustments were made to restrict the number of active nodes and allocated resources, prioritizing cost efficiency while maintaining system stability for testing purposes.

Bibliography

[1] A. R. Soares, "A telemedicina na pandemia: Avanços, desafios e oportunidades para a saúde," *International Integralize Scientific*, Apr. 2025.

[2] J. H. Mahar, G. J. Rosencrance, and P. A. Rasmussen, "Telemedicine: Past, present and future," *Cleveland Clinic Journal of Medicine*, vol. 85, no. 12, pp. 938–942, 2018, ISSN: 0891-1150. DOI: 10.3949/ccjm.85a.17062. eprint: https://www.ccjm.org/content/85/12/938.full.pdf. [Online]. Available: https://www.ccjm.org/content/85/12/938.

[3] A. Haleem, M. Javaid, R. P. Singh, and R. Suman, "Telemedicine for healthcare: Capabilities, features, barriers and applications," *Sensors International*, vol. 2, p. 100 117, 2021, ISSN: 2666-3511. DOI: https://doi.org/10.1016/j.sintl.2021.100117. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2666351121000383.

[4] Y. Shen, J. Yu, J. Zhou, and G. Hu, "Twenty-five years of evolution and hurdles in electronic health records and interoperability in medical research: Comprehensive review," *J Med Internet Res*, vol. 27, e59024, Jan. 2025, ISSN: 1438-8871. DOI: 10.2196/59024. [Online]. Available: https://doi.org/10.2196/59024.

[5] K. L. Rush, L. Howlett, A. Munro, and L. Burton, "Videoconference compared to telephone in healthcare delivery: A systematic review," *International Journal of Medical Informatics*, vol. 118, pp. 44–53, 2018, ISSN: 1386-5056. DOI: https://doi.org/10.1016/j.ijmedinf.2018.07.007. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1386505618300650.

[6] G. Aceto, V. Persico, and A. Pescapé, "Industry 4.0 and health: Internet of things, big data and cloud computing for healthcare 4.0," *Journal of Industrial Information Integration*, vol. 18, p. 100 129, 2020, ISSN: 2452-414X. DOI: https://doi.org/10.1016/j.jii.2020.100129. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2452414X19300135.

[7] E. Monaghesh and A. Hajizadeh, "The role of telehealth during covid-19 outbreak: A systematic review based on current evidence," Apr. 2020. DOI: 10.21203/rs.3.rs-23906/v1.

[8] S. Mooghala, "A comprehensive study of the transition from monolithic to micro services-based software architectures," *Journal of Technology and Systems*, vol. 5, pp. 27–40, Nov. 2023. DOI: 10.47941/jts.1538.

[9] F. Tapia, M. Á. Mora, W. Fuertes, H. Aules, E. Flores, and T. Toulkeridis, "From monolithic systems to microservices: A comparative study of performance," *Applied Sciences*, vol. 10, no. 17, 2020, ISSN: 2076-3417. DOI: 10.3390/app10175797. [Online]. Available: https://www.mdpi.com/2076-3417/10/17/5797.

[10] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges," *IEEE Access*, vol. 11, pp. 88 339–88 358, 2023. DOI: 10.1109/ACCESS.2023.3305687.

[11] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices architecture enables devops: Migration to a cloud-native architecture," *IEEE Software*, vol. 33, no. 3, pp. 42–52, 2016. DOI: 10.1109/MS.2016.64.

[12] V. Velepucha and P. Flores, "A survey on microservices architecture: Principles, patterns and migration challenges," *IEEE Access*, vol. 11, pp. 88 339–88 358, 2023. DOI: 10.1109/ACCESS.2023.3305687.

[13] L. Rushani and F. Halili, "Differences between service-oriented architecture and microservices architecture," *International Journal of Natural Sciences: Current and Future Research Trends*, vol. 13, no. 1, pp. 30–48, Apr. 2022. [Online]. Available:

https://ijnscfrtjournal.isrra.org/Natural_Sciences_Journal/article/view/1089.

[14] M. Marcello, I. W. Widi Pradnyana, and H. K. Prabu, "Microservice availability measurement by performance comparison of load balancing algorithms in api gateway," in *2024 International Conference on Informatics, Multimedia, Cyber and Information System (ICIMCIS)*, 2024, pp. 340–345. DOI: 10.1109/ICIMCIS63449.2024.10957588.

[15] D. Manor, D. Rod, H. Fang, and J. Watkins, "Containerization in cloud computing: A review," Feb. 2025.

[16] R. Molleti, "Kubernetes advanced auto scaling techniques," *Journal of Mathematical Computer Applications*, vol. 1, pp. 1–4, Dec. 2022. DOI: 10.47363/JMCA/2022(1)E126.

[17] G. Aksholak, A. Bedelbayev, and R. Magazov, "Securing kubernetes: An in-depth analysis of vulnerabilities, tools and future directions," *Physico-mathematical series*, Mar. 2025. DOI: 10.32014/2025.2518-1726.325.

[18] A. S. Cunha, A. R. Pedro, and J. V. Cordeiro, "Facilitators of and barriers to accessing hospital medical specialty telemedicine consultations during the covid-19 pandemic: Systematic review," *Journal of Medical Internet Research, 25, e44188–e44188*, NA 2023.

[19] B. M. Aljallabi and A. Mansour, "Enhancement approach for non-functional requirements analysis in agile environment," in *2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE)*, 2015, pp. 428–433. DOI: 10.1109/ICCNEEE.2015.7381407.

[20] M. Urbieta, N. Torres, J. M. Rivero, G. Rossi, and F. J. D. Mayo, "Improving mockup-based requirement specification with end-user annotations," *Lecture Notes in Business Information Processing, 19–34*, May 2018.