POLITECNICO DI BARI

DEPARTMENT OF ELECTRICAL AND INFORMATION ENGINEERING
Master's Degree in Computer Science Engineering

Final Group Project in
Software Architecture and Pattern Design

EarlyCare Gateway: A Microservices Architecture for Clinical Decision Support with Explainable AI

Supervisor
Prof.ssa Marina Mongiello

Students
Roberto Carriero
Luca Cianci
Luca Serio

Academic Year 2025 - 2026

Abstract

The EarlyCare Gateway project addresses critical challenges in the contemporary healthcare landscape, specifically targeting the issues of preliminary analysis. This work presents a Clinical Decision Support System (CDSS) designed to assist doctors through a multimodal AI analysis approach.

The system integrates a set of Artificial Intelligence models to process clinical data. It employs a fine-tuned ClinicalBERT and Gemini 2.5 Flash-Lite for the analysis of unstructured textual symptoms and ECG signals, CheXNet and EfficientNet for the classification of chest X-rays and skin lesions, and XGBoost for cardiovascular risk prediction based on structured tabular data. The solution is built upon a containerized microservices architecture, ensuring high cohesion and low coupling, as well as traceability and data protection.

Experimental validation demonstrates the system's reliability across all strategies. The textual analysis module achieved an accuracy of 88%, while the computer vision components demonstrated a Mean AUC of 0.837 for chest X-rays and an accuracy of 88% for skin lesion diagnosis. Furthermore, the cardiovascular risk prediction model reached a recall of 85.3%.

A central focus of the research is the implementation of Explainable AI (XAI) techniques, including SHAP, Grad-CAM, and Chain-of-Thought reasoning, that mitigate the black-box problem of AI models in critical fields, such as clinical applications.

Contents

Chapter 1

Introduction

The contemporary healthcare field is facing significant challenges, driven by the phenomena of overcrowding in emergency departments, medical shortages, and increasing pressure on triage systems. At present, these prioritization tools often rely on manual protocols and subjective opinions. These particular conditions create a fragile environment that can lead to potential diagnostic errors, substantial variability in decision-making processes, and high rates of burnout among healthcare professionals.

To address these critical issues, the proposed EarlyCare Gateway project aims to develop a Clinical Decision Support System (CDSS), designed to assist medical staff in the preliminary diagnosis and triage optimization process. It is important to emphasize that this system is designed only for professional use, as it is intended to assist doctors in order to prevent the risks associated with self-diagnosis and ensuring that the final decision always depends on a human operator.

1.1 Context and Motivation

While the clinical objective is clear, the engineering challenge lies in creating a solution that is robust, scalable, and capable of adapting to the rapid evolution of Artificial Intelligence.

The adoption of automated decision-making techniques in medicine is often hindered by skepticism regarding the black-box nature of algorithms. Currently, the adoption of automatic decision-making techniques in the medical field is still limited, often slowed down by ethical considerations and strict regulations. The EarlyCare Gateway aims to explore a technological solution that addresses these challenges.

The primary goal is to support the preliminary diagnosis of patient pathologies to optimize the triage process. From an architectural perspective, the solution is based on a containerized microservices architecture. This approach ensures that components are independent yet cooperative, improving system maintainability and scalability.

1.2 Document Structure

This document details the design, development, and implementation of the system, organized as follows:

The State of the Art chapter analyzes the current healthcare context, focusing in particular on the limitations of manual triage protocols, such as the Manchester Triage System. It explores the application of Artificial Intelligence in medicine, comparing the effectiveness of traditional Machine Learning approaches with the emerging capabilities of Large Language Models and Deep Learning. Special attention is given to Explainable AI techniques, which are crucial for transparency in clinical decision support systems.

The Requirements and Architecture chapter provides a description of Functional Requirements and Non-Functional Requirements. This chapter presents the containerized microservices architecture designed to meet these constraints, illustrating its decomposition into isolated components. Furthermore, the implemented Design Patterns are analyzed, and the corresponding UML diagrams and RESTful API contracts are described to clarify the interaction between services.

The AI Solutions and Explainability chapter examines the core of the project. It describes the Swappable AI logic and the specific models integrated to analyse text, signals, images, and structured data.

In the Development and Deployment chapter the technology stack and development tools are presented. Agile Kanban principles are applied during development, while containerization practices are used for deployment. This chapter also covers code organization and the implementation of the main classes that realize the architectural patterns defined in the previous sections.

The Testing and Results chapter reports the analysis of the integration tests performed and the evaluation metrics of the implemented AI models.

Finally, the Conclusion chapter summarizes the milestones achieved with the development of EarlyCare Gateway, highlighting how the solution addresses modern triage challenges by reducing the cognitive load on medical staff. The chapter also outlines potential future developments.

Chapter 2

State of the Art

This chapter provides a comprehensive overview of the current methodologies employed in clinical triage and preliminary diagnosis. It analyses the evolution from standardized manual protocols to advanced data-driven solutions, exploring the potential for the adoption of Artificial Intelligence in clinical context.

2.1 Current Triage Protocols

Among the technologies and protocols currently used to manage patient flow, the Manchester Triage System [1] is one of the most globally adopted frameworks. Functionally, it gives the nurses the possibility to assign a clinical priority to patients based on presenting symptoms, allocating them to one out of five categories depending on the urgency, with the aim of reducing queues in emergency facilities.

However, according to a recent review by Andika et al. [2], despite the system being widely adopted, it presents some limitations. This study states that measurement errors are strictly linked to personal factors of the healthcare staff, leading to variability in decision-making. Moreover, this article states that the issues of subtriage (underestimating severity) and super-triage (overestimating severity) persist, and for these reasons there's the need to implement an automated system that aims to mitigate human error and improve classification accuracy.

2.2 The Role of Artificial Intelligence

One of the possible tools to address limitations of manual protocols is Artificial Intelligence, mainly divided into Traditional Machine Learning and Large Language Models. Historically, AI models such as Random Forest and Gradient Boosting have proven highly effective when applied in this field, allowing to reach clinical benefits through structured data analysis. A systematic review by Sanchez et al. [3] analysed many machine learning methods applied to triage, concluding that these algorithms demonstrate consistency of accuracy, sensitivity and specificity when compared to traditional tools.

However, the study highlights a critical limitation. Infact, while these algorithms perform really well with structured data, they struggle with unstructured data.

To address this problem, the studies has shifted towards Large Language Models, commonly known as LLMs. While traditional algorithms rely on numerical inputs, these new implementations have the capability to process and interpret unstructured data, such as clinical notes and patient narratives.

As described by Thirunavukarasu et al. [4], these models are really helpful. Unlike traditional classifiers trained on specific tasks, LLMs are able to synthesize and simulate clinical reasoning. Another really significative strength of these models is their multimodal architecture, allowing them to integrate text, images and signals.

2.3 Explainability and Trust in Clinical AI

Despite the technical potential, the adoption of LLMs is not trusted by most medical professionals as they avoid to trust algorithms that provide diagnoses without an explanation. To overcome this problem, the integration of Explainable AI techniques has become a must have for modern CDSS. According to a review by Loh et al. [5], implementing explainability methods is essential to give trust and transparency to medical staff. Techniques such as SHAP (SHapley Additive exPlanations) and LIME (Local Interpretable Model-agnostic Explanations) allow the system to highlight which features influenced the model's decision, providing the doctors with the rationale behind the AI's output.

In conclusion, the current landscape needs a system that combines reliability of established medical protocols with the reasoning power of modern AI algorithms. So, the project is designed to cover all the aspect treated in this section within a secure and explainable architecture.

Chapter 3

Requirements and Architecture

This chapter focuses on translating the project's objectives into concrete system specifications and architectural solutions. It emphasizes how the identified requirements, both functional and non-functional. The discussion highlights the rationale behind adopting a microservices architecture, explaining how the decomposition into independent services supports modularity, scalability, and clear separation of responsibilities.

Furthermore, the section on service interactions demonstrates how RESTful APIs and structured communication patterns coordinate workflows across components.

Finally, an overview about the integration of design patterns is presented.

3.1 Requirements Analysis

The requirements analysis phase is critical to defining the operational boundaries of the project. The system is designed to operate in clinical environment, where precision, reliability, and usability are crucial. The analysis identifies the primary stakeholders as authorized doctors, excluding patient access to mitigate the risks associated with self-diagnosis and misinterpretation of clinical data.

System requirements are typically categorized into functional and non-functional requirements, each serving a distinct purpose in defining the system's behavior and quality attributes. In this project, non-functional requirements are informed by the FURPS+ model, which encompasses Functionality, Usability, Reliability, Performance, and Supportability, along with additional concerns such as security, scalability, and maintainability.

3.1.1 Functional Requirements (FR)

Functional Requirements describe what the system must do in terms of the core features, actions, and interactions that enable users to achieve their objectives.

For the project, the following main functional requirements have been identified.

Authentication

The system must provide a mechanisms for managing the lifecycle of medical user accounts. This includes a registration process where new doctors can create an account using a valid email and secure password. The system must handle authentication securely, allowing registered professionals to log in and obtain an access token (JWT, JSON Web Token) for subsequent sessions. The security requirement mandates the use of industry-standard authentication protocols and all data in transit between the client and the server must be encrypted to prevent eavesdropping, satisfying the strict security standards required for handling health-related data.

Intelligent Analysis

The core function of the platform is to perform preliminary diagnostic analyses on varied clinical data. The system must accept different types of input data uploaded by the doctor, specifically medical images (such as X-rays or skin lesion photos), textual clinical notes, or signal data like ECGs. Upon receiving this data, the system is required to trigger the appropriate artificial intelligence model to classify the pathology. A mandatory aspect of this function is that the output must not be a simple label. It must generate a detailed report that includes the predicted diagnosis, and specific explanation of the reasoning, ensuring the doctor understands the basis of the AI's suggestion.

Report Persistence

To support ongoing patient care, the system must ensure the persistence and retrievability of all generated diagnostic reports. Authorized doctors must be able to access a dedicated interface to view the history of their analyses. This requirement implies the need for filtering capabilities, allowing the user to search for reports associated with a specific fiscal code.

3.1.2 Non-Functional Requirements (NFR)

Non-Functional Requirements describe how the system should operate. Rather than focusing on specific functions, they define the quality attributes and constraints that shape the system's performance, reliability, security, scalability, and maintainability.

The main non-functional requirements for the projects are described in the following paragraphs.

Privacy

Protecting patient identity is a foundational constraint. The system must ensure that no personally identifiable information is stored or processed in clear text within the analysis modules. To achieve this, the system requires an anonymization mechanism that transforms sensitive identifiers, such as the fiscal code, into a unique hash string before any processing occurs. This ensures that even if data is intercepted during the analysis phase, it cannot be linked back to a specific individual.

Traceability

Traceability imposes that the system provides a history of usage. It necessitates an asynchronous recording mechanism that captures the timestamp, the type of event, and other information.

Explainability

To address the black-box problem common in AI adoption, the system must satisfy the requirement of explainability. It is not sufficient for the system to be accurate. It must be transparent. For every diagnostic prediction, the system is required to provide an interpretability layer, using techniques that highlights which clinical features influenced the decision.

Configurable Response Time

The system must be adaptable to different domains and pathologies. It is needed to have a Swappable AI capability, which allows the runtime configuration of the analysis strategy. The system must support both traditional Machine Learning and Deep Learning models, as well as LLMs, for high-accuracy and reasoning-rich analysis.

3.1.3 Use-Cases

Use-cases are a fundamental tool in system analysis and design, providing a visual description of the interactions between the system and its users. These interactions are modeled using the Unified Modeling Language (UML).

The use-cases diagram shown in Figure 3.1 illustrates the interactions between the system and its actors. The main actor, the Doctor, performs essential operations such as registration, login, diagnostic requests, and report consultation. Core system functionalities, including JWT validation and event logging, are included in multiple use cases to ensure secure and traceable operations. The diagnostic workflow incorporates an extendable AI analysis component.

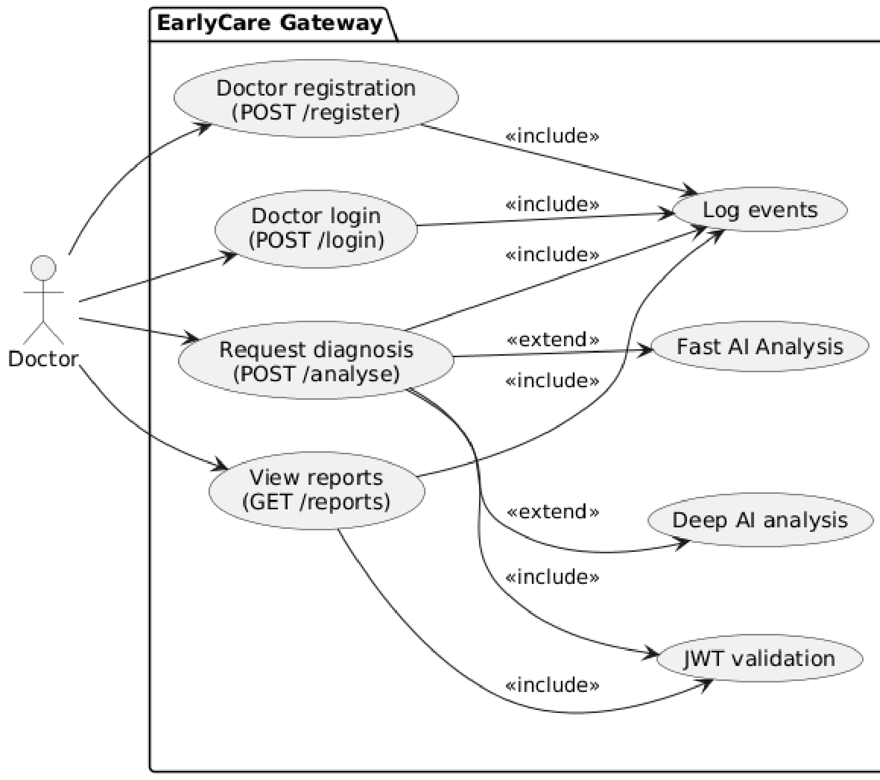


Figure 3.1: Use-Cases Diagram (UML) of the EarlyCare Gateway System.

3.2 Microservices Architecture

The EarlyCare Gateway system is designed following a microservices architecture, that emphasizes modularity, scalability, and maintainability. In this architecture, the system is decomposed into independent services, each responsible for a specific domain of functionality. This separation allows each service to be developed, deployed, and maintained independently, enabling rapid evolution of the system while minimizing the risk of disruptions to other components [6].

Adopting a microservices approach offers benefits in a clinical environment, where reliability, performance, and data security are critical. By isolating functionalities into distinct services, the system can handle concurrent operations, scale resources according to demand, and quickly integrate new technologies or AI models without affecting the overall workflow.

3.2.1 Main Components

The system is organized as a collection of loosely coupled microservices, maintaining high cohesion while minimizing dependencies between them.

The following UML component diagram provides an overview of these architectural elements and their interactions.

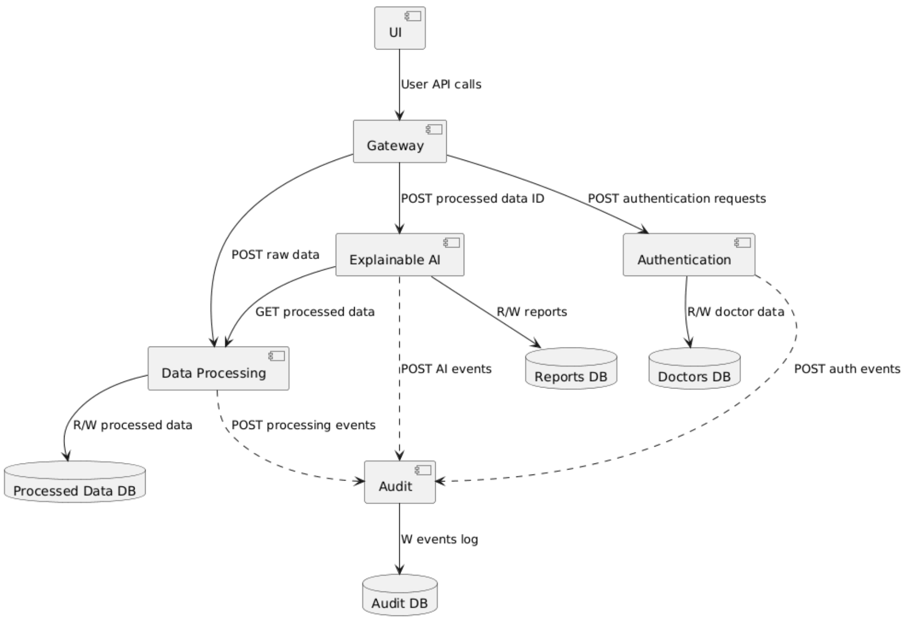


Figure 3.2: Components Diagram (UML) of the EarlyCare Gateway System.

The component diagram in Figure 3.2 illustrates is described below.

User Interface

The User Interface (UI) serves as the entry point for authorized doctors, implemented as a web-based single-page application that allows users to submit clinical data, request diagnostics, and consult reports. All user actions are routed through the Gateway, which acts as a centralized API entry point responsible for directing requests to the appropriate backend services.

Authentication Service

The Authentication Service manages user registration and login, interfacing with the Doctors Database to securely store credentials and access rights. It also communicates asynchronously with the Audit Service to log authentication-related events, ensuring traceability of all access operations.

Data Processing Service

The Data Processing Service is responsible for anonymizing and enriching clinical data, storing intermediate results in the Processed Data Database. This component supports handling of multiple data types, including images, textual notes, and numerical or signal-based clinical measurements. It also reports processing events to the Audit Service.

Explainable AI Service

The Explainable AI Service implements the system's core diagnostic logic. It retrieves preprocessed data from the Data Processing Service, performs analysis, and stores diagnostic reports in the Reports Database. This service is designed to be swappable, enabling runtime selection of the appropriate AI strategy according to data type and needs. All AI operations are logged via the Audit Service to ensure transparency and accountability.

Audit Service

Finally, the Audit Service consolidates logging and monitoring across all services. It writes all events to the Audit Database, providing a complete, asynchronous record of system activity.

3.2.2 Application Program Interface Communication

The microservices paradigm introduces complexity in communication and orchestration. Ensuring consistent and secure interactions among services requires robust API design and clear contract definitions. In the system, all user interactions are routed through the Gateway, which acts as a central API entry point, orchestrating requests to the appropriate backend services.

Sequence diagrams are employed to visually model the flow of information between actors and services, providing clarity on how requests, validations, and responses occur within the system. So, the diagrams serve as a bridge between the conceptual design and the actual implementation, providing a reference during development phase.

An overview of the system's API endpoints is provided below, focusing on the operations performed at each endpoint and illustrating how user requests are processed, routed, and handled by the various services. For technical specifications, including detailed request and response formats, authentication mechanisms, and parameter definitions, refer to the technical documentation.

POST /register

The `POST /register` endpoint in Figure 3.3 allows the registration of a new doctor in the system. This route is publicly accessible and does not require authentication. When a doctor submits a registration request through the UI, the Gateway receives the email and password in the request body and forwards it to the Authentication Service. The Authentication Service checks the existence of the email in the Doctors Database. If the email is already in use, an error is returned to the Gateway, which forwards it to the UI. If the email is available, the service hashes the password, and stores the credentials in the Database. An asynchronous notification of the registration event is sent to the Audit Service, and the Gateway responds to the UI with a success or failure message.

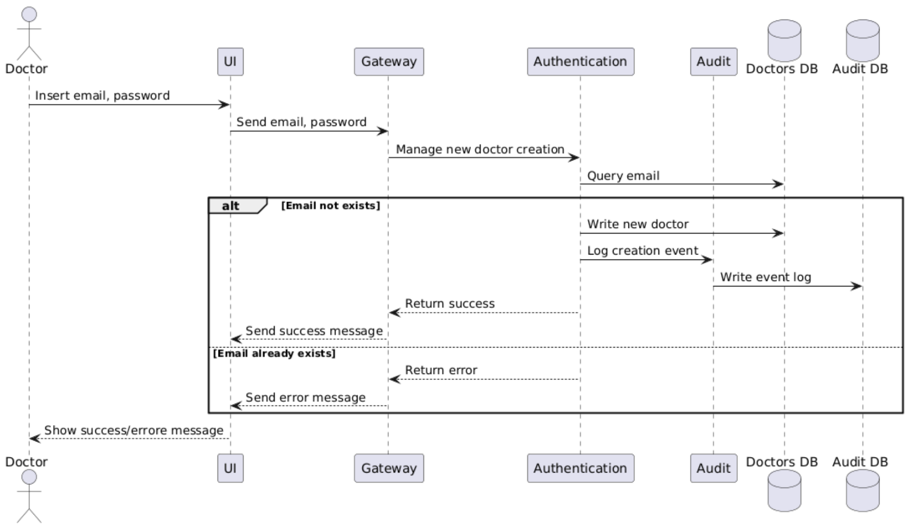


Figure 3.3: UML Sequence Diagram for Registration.

POST /login

The `POST /login` endpoint in Figure 3.4 handles doctor authentication and returns a JWT for subsequent authenticated requests. The UI submits the login credentials to the Gateway, which forwards the request to the Authentication Service. The service verifies the email and password and checks the hashed password. If the credentials are invalid, an error is returned. On success, the service generates a JWT containing the unique doctor ID, name and surname, signs it with a secret key, and asynchronously logs the event in the Audit Service. The Gateway then returns the JWT to the UI for storage and use in subsequent requests.

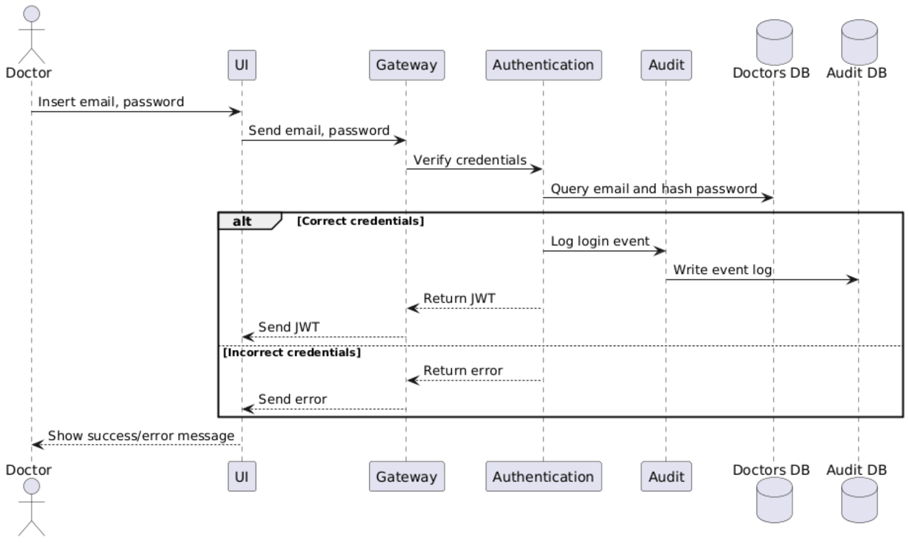


Figure 3.4: UML Sequence Diagram for Login.

POST /analyse

The `POST /analyse` endpoint in Figure 3.5 executes a diagnostic analysis. The request must include a valid JWT in the header. The UI submits the clinical data to the Gateway, which first validates the token via the Authentication Service. If the token is invalid or expired, the request is rejected. Otherwise, the Gateway forwards the raw data to the Data Processing Service. The service performs data handling, stores processed results in the Database, and logs the operation asynchronously in the Audit Service. The processed data ID is then sent to the Explainable AI Service, via Gateway. Explainable AI Service retrieves the data, applies the appropriate AI strategy, generates a diagnostic report, stores it in the Reports Database, and logs the completion in the Audit Service. Finally, the Gateway returns the report to the UI.

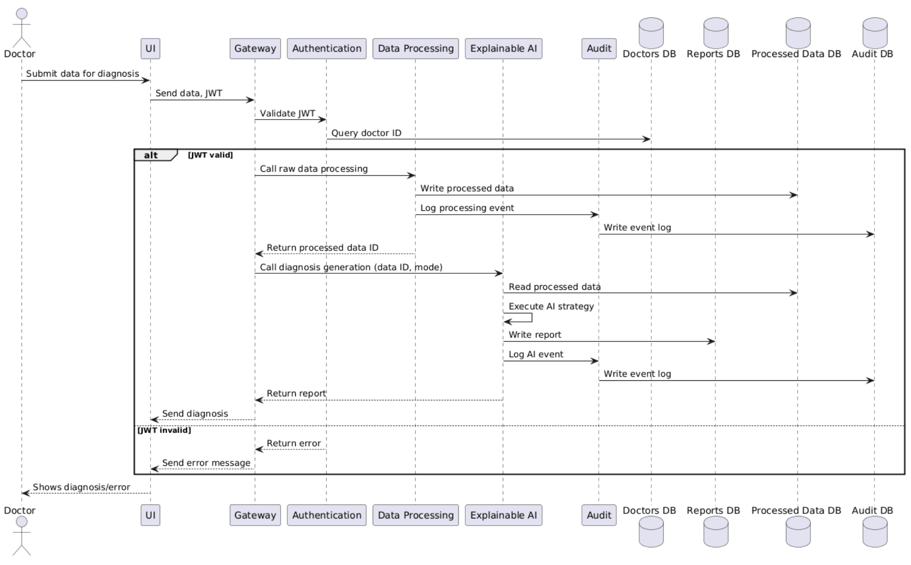


Figure 3.5: UML Sequence Diagram for Analysis.

GET /reports

The `GET /reports` endpoint in Figure 3.6 allows the retrieval of diagnostic reports. A valid JWT is required. The UI requests all reports or those filtered by a specific patient's fiscal code. The Gateway validates the JWT via the Authentication Service and extracts the doctor's ID. The request is forwarded to the Explainable AI Service, which queries the Reports Database using the Repository Pattern. The resulting list of reports is returned to the Gateway, which forwards it to the UI in JSON format.

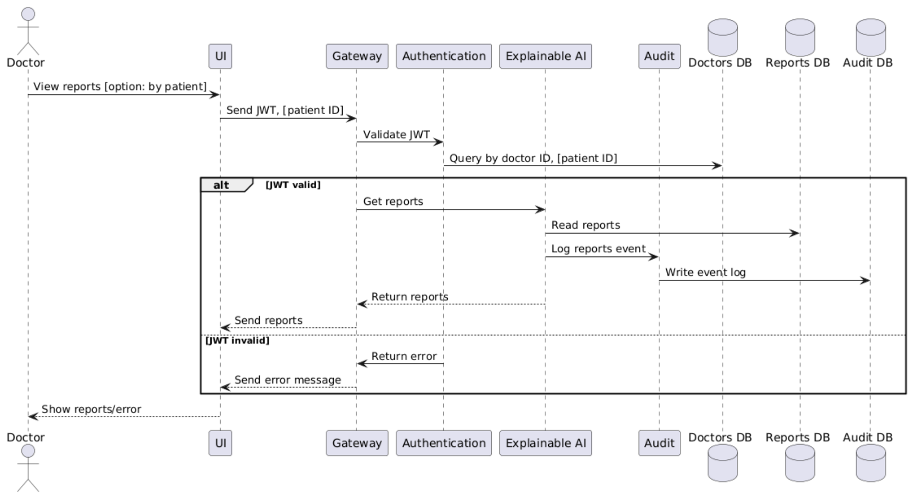


Figure 3.6: UML Sequence Diagram for Report Retrieval.

3.3 Design Patterns

The system leverages software design patterns to ensure maintainability, modularity, and scalability across its microservices architecture. By applying appropriate patterns, the project achieves a clear separation of concerns [7].

The Table 3.1 summarizes the mapping between microservices components and the corresponding design patterns, described below.

3.3.1 Structural Patterns

Structural patterns are employed to define how the components of the system are organized and interact at a higher level. In the project, the following structural patterns are used.

Facade Pattern

The Gateway component functions as a Facade, providing a unified interface to the underlying microservices. It abstracts the internal complexity of the system and simplifies the interaction for external clients, such as the web-based UI. By centralizing API routing and request orchestration, the Facade ensures that clients do not need to manage direct interactions with multiple services.

Repository Pattern

The Repository Pattern is widely used across the Authentication, Data Processing, and Audit services. Each service interacts with its underlying database through a repository interface, encapsulating data access logic. This separation allows for easier database management, and potential replacement of storage technologies without affecting the business logic of the services.

3.3.2 Behavioural Patterns

Behavioural patterns define the interaction strategies among objects and services. They are important for coordinating actions and managing workflows. The behavioural patterns presented in the system are described below.

Strategy Pattern

The Explainable AI service applies the Strategy Pattern to implement swappable AI models. Depending on the type of clinical data submitted, the service selects the appropriate AI model to perform the analysis. This pattern ensures new AI strategies to be integrated without modifying existing code.

Observer Pattern

The Observer Pattern is used to notify the Audit service of operations, such as user registration, login, or diagnostic analysis. Services emit events asynchronously, and the Audit service subscribes to these events to log them persistently. This decouples the main business logic from audit logging while maintaining complete traceability of actions.

Chain of Responsibility Pattern

The Data Processing service employs the Chain of Responsibility Pattern to handle data validation, anonymization, and enrichment in a sequential manner. Each processing step is encapsulated in a handler, which can pass the data to the next handler in the chain, depending on the analysis strategy required.

Table 3.1: Overview of Microservice Components and Design Patterns.

Component	**Design Pattern**
Gateway	Facade
Authentication	Repository, Observer
Data Processing	Repository, Chain of Responsibility, Observer
Explainable AI	Strategy, Observer
Audit	Repository

Chapter 4

AI Solutions and Explainability

This chapter details the Artificial Intelligence architecture implemented in the system. The solution adopts a multimodal diagnostic approach, leveraging specialized models for textual, imaging, signal-based, and structured data analysis.

4.1 Textual Analysis Strategy

The analysis of unstructured textual data, including patient symptoms and clinical transcriptions, is managed by the `TextAnalysisStrategy`. This module implements a two-stage pipeline: a deterministic classification stage based on a fine-tuned Transformer model, followed by a generative reasoning stage powered by a Large Language Model (LLM).

4.1.1 ClinicalBERT Architecture and Fine-Tuning

The core classifier relies on ClinicalBERT (`emilyalsentzer/Bio_ClinicalBERT`) [8]. Unlike standard BERT models trained on general corpora like Wikipedia, this model was pre-trained on the MIMIC-III database, comprising over 2 million electronic health records from intensive care units. This domain-specific pre-training enables the model to capture complex medical semantics and terminology effectively.

Dataset and Preprocessing Fine-tuned has been performed on MTSamples dataset [9], a comprehensive collection of transcribed medical reports. The dataset was filtered to retain eight primary medical specialties: *Cardiovascular/Pulmonary, Orthopedic, Gastroenterology, Neurology, Obstetrics/Gynecology, Urology, ENT, and Hematology-Oncology*. The input text (derived from the *transcription* and *description* fields) undergoes a cleaning process to remove non-printable characters, followed by tokenization using the specific ClinicalBERT tokenizer with a maximum sequence length of $L = 256$ tokens.

Training Configuration The fine-tuning process was executed using the Hugging Face `Trainer` API with the following hyperparameters, optimized for stability and convergence:

- **Epochs**: 4
- **Batch Size**: 8
- **Learning Rate**: $3e^{-5}$
- **Optimizer**: AdamW with weight decay (0.01)

This configuration yielded an accuracy of approximately 88% on the validation set, demonstrating high reliability in categorizing clinical narratives.

4.1.2 Generative Reasoning and Chain of Thought

While ClinicalBERT provides the technical classification (e.g., "Cardiovascular"), the system requires a more granular diagnosis. To achieve this, the output of the classifier is fed into a second stage based on Gemini 2.0 Flash.

The system constructs a prompt using the *Chain of Thought* (CoT) technique, injecting:

1. The patient's raw symptoms.
2. The predicted specialty from ClinicalBERT.
3. The confidence score of the prediction.

The LLM is instructed to act as an expert clinician, reasoning upon the classification to generate a specific pathology (e.g., "Unstable Angina") and a concise explanation (XAI). The response is parsed from JSON format, ensuring structured integration with the backend.

4.2 Imaging Analysis Strategy

The visual diagnostic module is managed by the `ImageAnalysisStrategy` class. This component is designed to handle distinct imaging modalities by dynamically selecting the appropriate neural network architecture based on the input type. The system currently supports two primary workflows: chest X-rays for pulmonary analysis and dermatoscopic images for skin lesion classification.

4.2.1 Data Preprocessing Pipeline

In a web-based microservices architecture, images are transmitted as Base64-encoded strings rather than raw binary files. The preprocessing pipeline, implemented in the `_base64_to_tensor` method, performs the rigorous mathematical transformations required to convert this input into a format suitable for PyTorch models:

1. **Decoding and Resizing**: The Base64 string is decoded into an RGB image and resized to a standard resolution of 224×224 pixels. This ensures compatibility with the fixed input layers of the convolutional networks.

2. **Normalization**: The pixel values are standardized using the mean and standard deviation statistics from the *ImageNet* dataset ($\mu = [0.485, 0.456, 0.406]$, $\sigma = [0.229, 0.224, 0.225]$). This step allows the models to leverage transfer learning effectively by maintaining consistent distribution properties.

3. **Transposition**: The data structure is converted from the standard Height-Width-Channel (HWC) format to the Channel-Height-Width (CHW) format required by the PyTorch inference engine.

4.2.2 Chest X-Ray Analysis (CheXNet)

For thoracic imaging, the system employs the CheXNet architecture, based on a Dense Convolutional Network (DenseNet-121) [10]. The model was trained on the public NIH ChestX-ray14 dataset [11], one of the largest collections of chest X-rays available, comprising over 112,000 frontal images from more than 30,000 unique patients.

The analysis is framed as a *Multi-Label Classification* task, as a single patient may present multiple conditions simultaneously (e.g., Pneumonia and Infiltration). Consequently, the final layer utilizes a Sigmoid activation function rather than Softmax, and the model is optimized using the *Binary Cross Entropy with Logits Loss* (BCE-WithLogitsLoss). The output is a vector of probabilities for 14 different pathologies.

4.2.3 Skin Lesion Analysis (EfficientNet)

For the analysis of skin lesions, the system integrates EfficientNet-B0 [12]. This architecture was selected for its superior parameter efficiency and its ability to extract complex textural features. The model was trained on the HAM10000 (Human Against Machine with 10000 training images) dataset [13], which serves as a standard benchmark for automated dermatoscopic diagnosis.

Unlike the X-ray module, this task is a *Multi-Class Classification* problem aimed at identifying a single, mutually exclusive diagnosis among 7 categories (Melanoma, Nevus, Basal Cell Carcinoma). To address the class imbalance intrinsic to the HAM10000 dataset, the training phase employed a *Weighted Cross-Entropy Loss*. This loss function penalizes errors on rare but critical classes (such as Melanoma) more heavily, thereby improving clinical sensitivity.

4.2.4 Explainability with Grad-CAM

To provide visual interpretability and mitigate the "black box" problem, both imaging strategies implement Grad-CAM (Gradient-weighted Class Activation Mapping) [14]. This technique computes the gradients of the target class score with respect to the feature maps of the final convolutional layer (`features[-1]`). These gradients are used to generate a coarse localization map (heatmap), which highlights the regions of the image that most influenced the model's prediction. The heatmap is overlaid on the original image and returned to the user, allowing clinicians to verify if the AI is focusing on relevant anatomical features.

4.3 Structured Data Analysis Strategy

The analysis of tabular clinical data, such as vital signs and laboratory results, is handled by the `NumericAnalysisStrategy` class. This component specializes in cardiovascular risk assessment, transforming a heterogeneous set of numerical and categorical inputs into probabilistic predictions supported by mathematically grounded explanations.

4.3.1 Dataset and Preprocessing

The model was developed using the Heart Disease UCI Dataset [15], a comprehensive collection aggregating data from four international databases (Cleveland, Hungary, Switzerland, Long Beach VA) totaling 920 patients. This dataset was selected for its greater robustness compared to the Cleveland-only subset, offering a more diverse representation of cardiac pathologies.

The preprocessing pipeline performs the following critical transformations:

- **Data Imputation**: Missing values are handled statistically, using the median for numerical variables and the mode for categorical ones, ensuring model continuity.
- **Feature Engineering**: The 13 clinical input parameters (e.g., `cp` for chest pain, `restecg` for ECG results) are expanded into a vector of 18 features via *One-Hot Encoding*. This allows the model to assign distinct risk weights to each specific sub-category.

4.3.2 XGBoost Classifier

The inference engine is built upon XGBoost (eXtreme Gradient Boosting) [16], an ensemble learning algorithm based on decision trees (Gradient Boosted Decision Trees). XGBoost was chosen for its proven efficiency on structured data, its native handling of sparse values, and its resistance to overfitting. The model operates as a binary classifier, returning the probability of the presence of heart disease. The optimal hyperparameters (e.g., `n_estimators`, `max_depth`) were identified through a *Grid Search* procedure with 5-fold cross-validation.

4.3.3 Local Interpretability with SHAP

To satisfy explainability requirements, the strategy integrates the SHAP (SHapley Additive exPlanations) framework [17]. Using a `TreeExplainer`, the system computes Shapley values for each instance, quantifying the contribution of each feature to the deviation of the prediction from the average. The algorithm selects the top 5 most impactful features and classifies them as *Risk Factors* (positive contribution to disease probability) or *Protective Factors* (negative contribution), providing the physician with a clear picture of the AI's decision rationale.

4.4 Signal Analysis Strategy

The analysis of biomedical signals, particularly electrocardiographic (ECG) tracings, is managed by the `SignalAnalysisStrategy` class. The system adopts an innovative approach that does not rely on traditional 1D convolutional networks, but instead leverages the multimodal reasoning capabilities of Large Language Models (LLMs) to interpret raw numerical data as a semantic sequence.

4.4.1 Preprocessing and Context Optimization

The ECG signal is received by the system as a Base64-encoded string. To enable processing by the LLM without saturating the context window, a data reduction pipeline has been implemented within the `_decode_signal` method:

1. **Deserialization**: The input string is decoded and deserialized (via the `pickle` library) into a NumPy numerical array.
2. **Statistical Extraction**: Global descriptive metrics are calculated on the entire signal, including minimum, maximum, and mean values (`np.min`, `np.max`, `np.mean`). These metadata provide the model with context regarding the amplitude and global voltage of the tracing.
3. **Truncation**: For punctual morphological analysis, the system extracts a significant sample limited to the first 300 data points. This operation reduces computational complexity while maintaining visibility of potential immediate rhythmic anomalies.

4.4.2 Inference via Gemini LLM

The analysis is performed by invoking the Gemini 2.0 Flash model. The system prompt is dynamically engineered by injecting the two previously prepared data blocks: global statistics and the raw data sample. The prompt instructs the model to analyze the data to identify rhythm anomalies and return a valid JSON object containing the diagnosis, confidence level, and a textual explanation [18].

4.5 Explainable AI (XAI) Implementation

The architecture of the EarlyCare Gateway natively integrates explanation mechanisms for each model type, addressing the "Black Box" problem in the clinical domain [19].

4.5.1 SHAP for Structured Data

For the XGBoost predictive model, interpretability is guaranteed by the SHAP (SHapley Additive exPlanations) library. Within the `NumericAnalysisStrategy` class, a `TreeExplainer` is used to calculate Shapley values, which quantify the contribution of each feature to the final prediction. The system filters results to show the user only the 5 most relevant clinical factors (with absolute impact > 0.1), indicating whether they act as risk or protective factors [20].

4.5.2 Grad-CAM for Imaging

The imaging strategies (`ImageAnalysisStrategy`) implement the Grad-CAM technique. The system calculates the gradients of the predicted class with respect to the last convolutional layer of the neural network (e.g., the `features[-1]` block for EfficientNet). These gradients are used to generate a heatmap that highlights the anatomical regions determining the diagnosis. The resulting image is Base64-encoded and sent to the frontend for overlay visualization on the original finding [14].

4.5.3 Chain of Thought (CoT)

For text and signal-based analyses, explainability is intrinsic to the LLM's generative process. Through the *Chain of Thought Prompting* technique, the model is forced to verbalize its logical reasoning. In the code, this is implemented by explicitly requesting an ''`explanation`'' or ''`xai_explanation`'' section in the prompt, which is then parsed and displayed to the doctor as a natural language justification for the suggested diagnosis [21].

Table 4.1: Updated Summary of AI Models and Explainability Techniques

Data Type	**AI Model**	**Explainability**
Textual Data	Bio_ClinicalBERT (Fine-tuned) + Gemini 2.0 Flash	Chain of Thought (CoT)
Chest X-Ray	CheXNet (DenseNet121)	Grad-CAM
Skin Lesions	EfficientNet-B0	Grad-CAM
Physiological Signals (ECG)	Gemini 2.0 Flash (LLM)	Chain of Thought (CoT)
Structured Data	**XGBoost (Gradient Boosting)**	**SHAP (TreeExplainer)**

Chapter 5

Development and Deployment

This chapter describes the development methodology, organizational planning, implementation details, and deployment strategy adopted for the project.

5.1 Planning and Organization

Two main approaches guided the project workflow and organization: Agile methodology and Gantt chart planning. A Gantt chart, designed in Excel, was used to visualize the overall timeline of the project, helping to split tasks among the team members.

Furthermore, the team adopted an Agile Kanban approach to manage tasks and iterations efficiently. The Kanban board, integrated in Microsoft Teams, facilitated task prioritization, progress tracking, and rapid adaptation to evolving requirements.

5.2 Development

The development phase focused on realizing the architecture and functional requirements while adhering to best practices for code quality, modularity, and testability.

5.2.1 Technology Stack and Tools

The project leveraged modern technologies suitable for web-based, AI-driven systems:

- **Backend**: Python with FastAPI for building RESTful API.
- **Frontend**: React for responsive and interactive user interface development.
- **Data**: PostgreSQL database for data persistance.
- **AI**: Pandas, Numpy, Scikit-Learn, XGBoost, Gemini Generative API for adding AI functionality.
- **Testing**: Pytest for integration tests.
- **Versioning**: Git and GitHub for collaboration.

- **Deployment**: Docker and Docker-Compose for containerization and multi-container deployment.

5.2.2 Class Diagrams

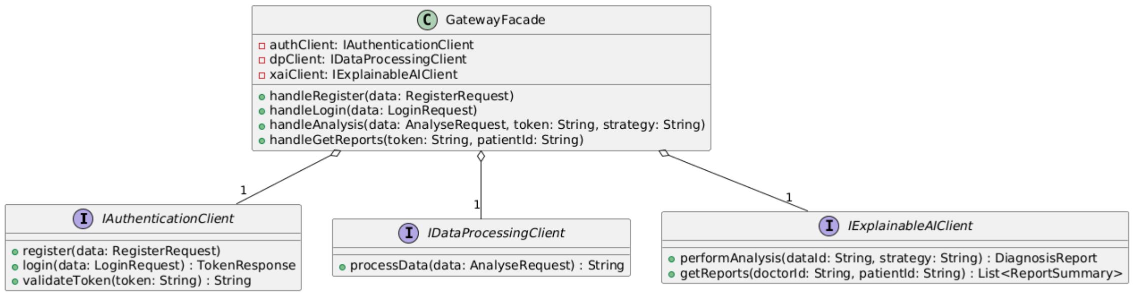
Each microservice was implemented with well-defined classes to encapsulate business logic. Design patterns discussed in Section 3.3 were applied to improve maintainability, flexibility, and testability.

An overview of the UML class diagrams for all the services is reported below.

Gateway

To manage the complexity of interactions between the frontend and the distributed backend services, the Gateway component implements the Facade Design Pattern. The structural design of this component is shown in Figure 5.1.

The core of this implementation is the `GatewayFacade` class, which serves as the single entry point for all client requests. This class decouples the external API consumers from the underlying microservices by aggregating three specialized interfaces.



C GatewayFacade
authClient: IAuthenticationClient
dpClient: IDataProcessingClient
xaiClient: IExplainableAIClient
handleRegister(data: RegisterRequest)
handleLogin(data: LoginRequest)
handleAnalysis(data: AnalyseRequest, token: String, strategy: String)
handleGetReports(token: String, patientId: String)
1
1
1
I IAuthenticationClient
register(data: RegisterRequest)
login(data: LoginRequest) : TokenResponse
validateToken(token: String) : String
I IDataProcessingClient
processData(data: AnalyseRequest) : String
I IExplainableAIClient
performAnalysis(dataId: String, strategy: String) : DiagnosisReport
getReports(doctorId: String, patientId: String) : List<ReportSummary>

Figure 5.1: UML Class Diagram of Gateway Service.

- `IAuthenticationClient`: This interface abstracts the communication with the Authentication Service. It defines methods for user management operations such as `register`,`login`, and `validateToken`.

- `IDataProcessingClient`: This interface encapsulates the interaction with the Data Processing Service. It exposes the `processData` method, which accepts an `AnalyseRequest` and returns the identifier of the processed data, handling the necessary anonymization and formatting steps transparently.

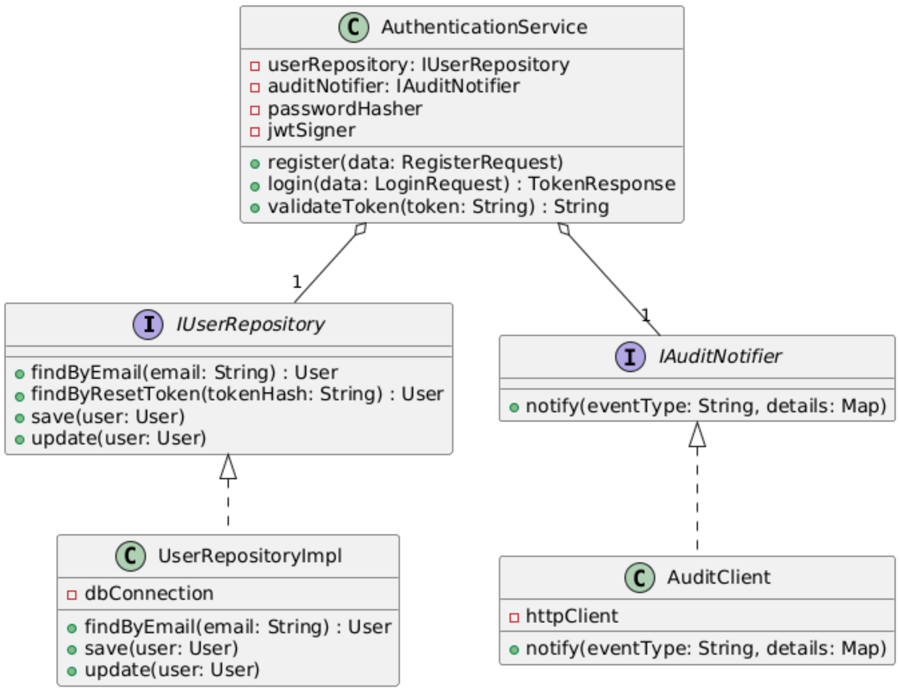
- `IExplainableAIClient`: This interface manages the connection to the Explainable AI Service. It provides the `performAnalysis` method to trigger the diagnostic process (specifying the `dataId` and the desired AI `mode`) and the `getReports` method to retrieve diagnostic history for specific patients.

Authentication

The Authentication Service is responsible for managing medical staff identities and securing access to the system. Its internal architecture, shown in Figure 5.2, relies on the Repository and Observer design patterns to ensure separation of concerns and testability.

The core logic is encapsulated within the `AuthenticationService` class. This class orchestrates the registration and login workflows, utilizing internal helpers like `passwordHasher` for credential security and `jwtSigner` for issuing JSON Web Tokens.

To decouple the business logic from data persistence, the service implements the Repository Pattern through the `IUserRepository` interface.



C AuthenticationService
userRepository: IUserRepository
auditNotifier: IAuditNotifier
passwordHasher
jwtSigner
register(data: RegisterRequest)
login(data: LoginRequest) : TokenResponse
validateToken(token: String) : String
1
1
I IUserRepository
findByEmail(email: String) : User
findByResetToken(tokenHash: String) : User
save(user: User)
update(user: User)
I IAuditNotifier
notify(eventType: String, details: Map)
C UserRepositoryImpl
dbConnection
findByEmail(email: String) : User
save(user: User)
update(user: User)
C AuditClient
httpClient
notify(eventType: String, details: Map)

Figure 5.2: UML Class Diagram of Authentication Service.

- `IUserRepository`: Defines the contract for user data access. Methods such as `findByEmail` and `save` are used during registration and login.
- `UserRepositoryImpl`: The concrete implementation that handles the actual database connections and SQL queries.

Furthermore, to satisfy the traceability requirement without coupling the authentication logic with the logging infrastructure, the service adopts the Observer Pattern via the `IAuditNotifier` interface.

- `IAuditNotifier`: An abstraction that allows the service to emit events.
- `AuditClient`: The concrete implementation that asynchronously forwards these events to the Audit Service via REST API call.

Data Processing

The Data Processing Service handles the task of preparing raw clinical data for analysis while ensuring compliance with privacy standards. The internal architecture, illustrated in Figure 5.3, is structured around the Chain of Responsibility Design Pattern to manage the data transformation pipeline efficiently.

The core of this pattern is defined by the `IProcessingStep` interface and the abstract `BaseProcessingStep` class, which allows for the linking of processing stages. The workflow is decomposed into discrete, sequential steps.

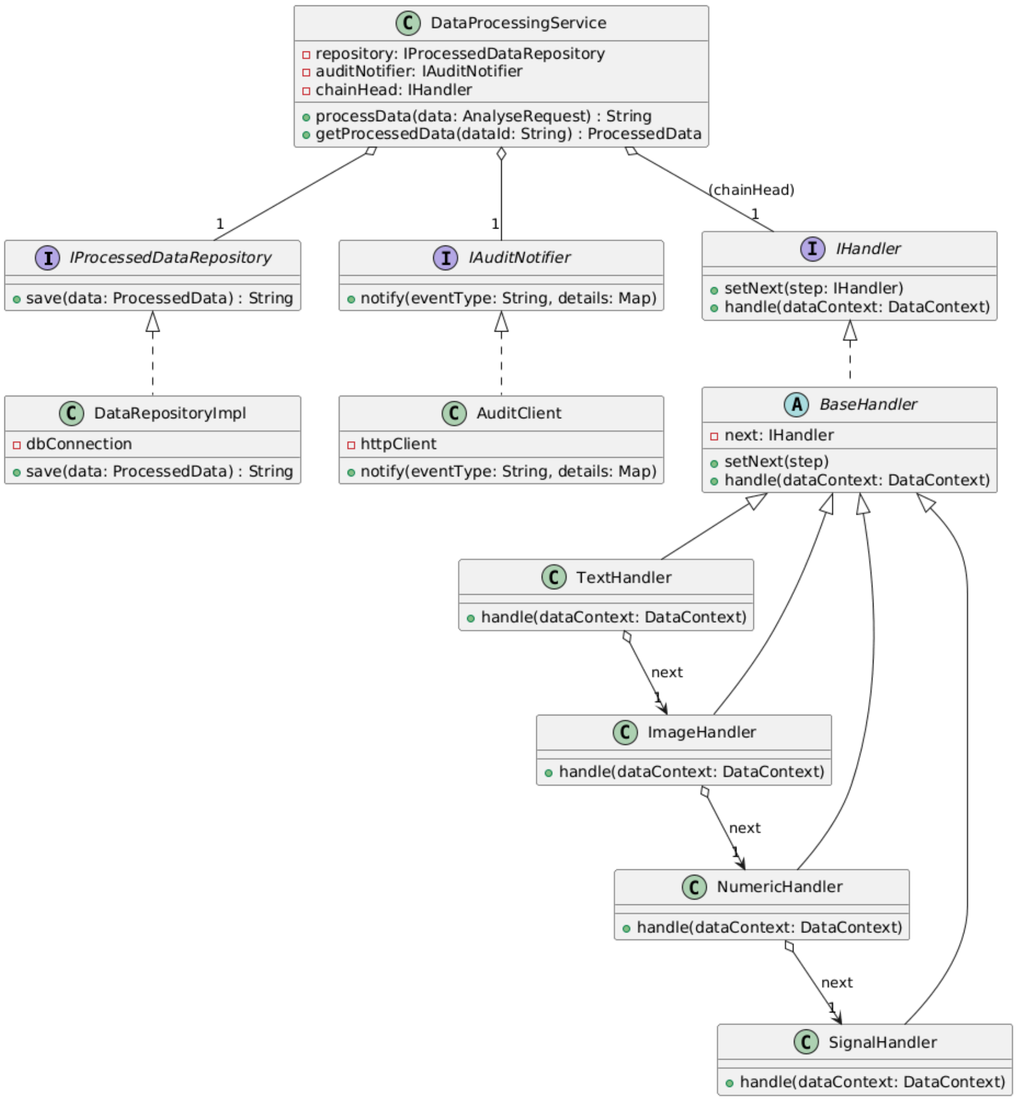

DataProcessingService
repository: IProcessedDataRepository
auditNotifier: IAuditNotifier
chainHead: IHandler
processData(data: AnalyseRequest) : String
getProcessedData(dataId: String) : ProcessedData
1
1
(chainHead)
1
IProcessedDataRepository
save(data: ProcessedData) : String
IAuditNotifier
notify(eventType: String, details: Map)
IHandler
setNext(step: IHandler)
handle(dataContext: DataContext)
DataRepositoryImpl
dbConnection
save(data: ProcessedData) : String
AuditClient
httpClient
notify(eventType: String, details: Map)
BaseHandler
next: IHandler
setNext(step)
handle(dataContext: DataContext)
TextHandler
handle(dataContext: DataContext)
next
1
ImageHandler
handle(dataContext: DataContext)
next
1
NumericHandler
handle(dataContext: DataContext)
next
1
SignalHandler
handle(dataContext: DataContext)

Figure 5.3: UML Class Diagram of Data Processing Service.

- `TextHandler`: The first link in the chain, responsible for textual data.
- `ImageHandler`: This handler is responsible for image processing.
- `NumericHandler`: This handler processes structured numeric data.
- `SignalHandler`: The stage that handles signal samples input.

To handle data persistence and side effects, the service employs two additional abstractions:

- `IProcessedDataRepository`: Implements the Repository Pattern to abstract the saving of anonymized data (`ProcessedData`) to the underlying database.
- `IAuditNotifier`: Allows the service to asynchronously notify the Audit system of processing events.

Explainable AI

The Explainable AI Service represents the diagnostic core of the system. Its design is centered around the Strategy Design Pattern, which is essential to satisfy the swappable AI requirement, allowing the system to select the most appropriate analysis model at runtime. The class structure is illustrated in Figure 5.4.

The orchestrator of this component is the `ExplainableAIService` class. It relies on a set of abstractions to perform its tasks without being coupled to specific implementations.

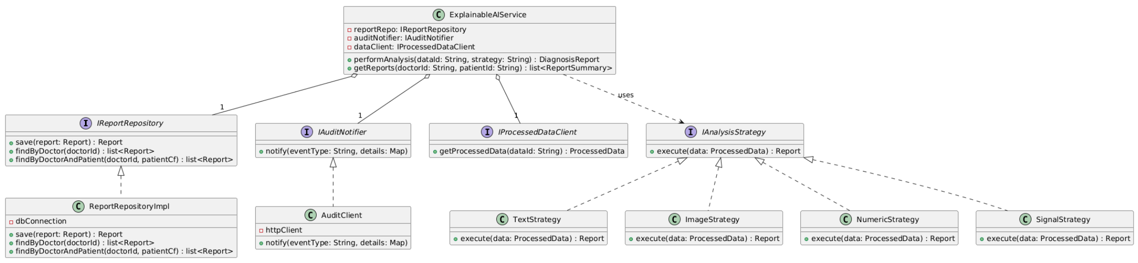

ExplainableAIService
reportRepo: IReportRepository
auditNotifier: IAuditNotifier
dataClient: IProcessedDataClient
performAnalysis(dataId: String, strategy: String) : DiagnosisReport
getReports(doctorId: String, patientId: String) : list<ReportSummary>
1
1
1
uses
IReportRepository
save(report: Report) : Report
findByDoctor(doctorId) : list<Report>
findByDoctorAndPatient(doctorId, patientCf) : list<Report>
IAuditNotifier
notify(eventType: String, details: Map)
IProcessedDataClient
getProcessedData(dataId: String) : ProcessedData
IAnalysisStrategy
execute(data: ProcessedData) : Report
ReportRepositoryImpl
dbConnection
save(report: Report) : Report
findByDoctor(doctorId) : list<Report>
findByDoctorAndPatient(doctorId, patientCf) : list<Report>
AuditClient
httpClient
notify(eventType: String, details: Map)
TextStrategy
execute(data: ProcessedData) : Report
ImageStrategy
execute(data: ProcessedData) : Report
NumericStrategy
execute(data: ProcessedData) : Report
SignalStrategy
execute(data: ProcessedData) : Report

Figure 5.4: UML Class Diagram of Explainable AI Service.

- Strategy Pattern Implementation: The logic for diagnostic analysis is encapsulated within the `IAnalysisStrategy` interface. Two concrete strategies implement this interface:
  - `TextStrategy`: Implements the textual analysis using ClinicalBERT.
  - `SignalStrategy`: Implements the analysis mode using Large Language Models with Chain-of-Thought reasoning.
  - `ImageStrategy`: Implements the imaging analysis mode using CNNs for x-rays and skin lesions.
  - `NumericStrategy`: Implements the structured data analysis mode using XGBoost on cardio risk context.
- Repository Pattern: To manage the persistence of diagnostic reports, the service utilizes the `IReportRepository` interface. This allows the business logic to save and retrieve reports without handling the underlying database connection details.
- External Dependencies: The service interacts with other microservices through specific interfaces:
  - `IProcessedDataClient`: Retrieves the anonymized and pre-processed data required for the analysis.
  - `IAuditNotifier`: Asynchronously notifies the Audit Service of completed analyses, ensuring traceability.

Audit

The Audit Service acts as the centralized logging facility for the entire architecture, satisfying the critical requirement of traceability. Its class design, shown in Figure 5.5, is minimal and optimized for write-intensive operations.

The architecture implements the Repository Design Pattern to abstract the persistence layer.

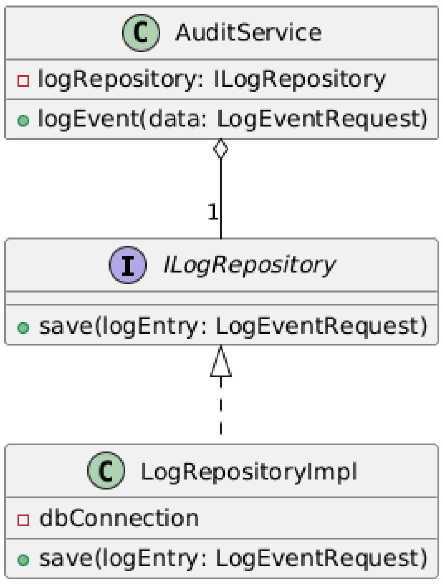

AuditService
logRepository: ILogRepository
logEvent(data: LogEventRequest)
1
ILogRepository
save(logEntry: LogEventRequest)
LogRepositoryImpl
dbConnection
save(logEntry: LogEventRequest)

Figure 5.5: UML Class Diagram of Audit Service.

- `AuditService`: This class serves as the entry point. It exposes the `logEvent` method, which receives `LogEventRequest` objects containing details.
- `ILogRepository`: This interface defines the contract for saving log entries, ensuring that the service logic is not coupled to the specific database technology.
- `LogRepositoryImpl`: The concrete implementation that manages the connection to the underlying SQL database, executing the actual insertion of the log records.

This separation ensures that the logging logic remains consistent even if the storage mechanism changes, maintaining high maintainability.

5.2.3 Database

In the context of the proposed software architecture, the persistence layer is critical for ensuring data integrity, traceability, and secure access to medical records. A strict microservices architecture typically advocates for the *Database-per-Service* pattern to ensure loose coupling. However, to optimize resource usage and simplify the deployment pipeline for this project, a hybrid approach was chosen.

While the system is conceptually designed as a set of distributed microservices, the physical data layer is consolidated into a single PostgreSQL instance. Within this shared physical instance, data are logically separated. Each microservice interacts exclusively with the tables relevant to its context.

The database schema consists of the following entities, designed to support the system's requirements:

- **doctors**: This table stores the profiles of registered medical professionals. It includes a unique identifier, personal details (name, surname, email), and, to ensure security, the password is stored exclusively as a SHA256 hash.
- **reports**: This table persists the outcomes of the AI analysis. It records the unique report ID, the ID of the doctor performing the analysis, and the hashed fiscal code of the patient to preserve anonymity. Additionally, it links to the preprocessed data ID and stores the diagnostic result, the prediction confidence, the specific analysis strategy employed, the textual XAI explanation, and the creation timestamp.
- **processed_data**: This table serves as a storage for input data prepared for the algorithms. It contains a unique identifier, the processing strategy (indicating the data type), the actual preprocessed data payload, and the creation timestamp.
- **logs**: This table maintains the system's audit trail. It records a unique ID, the creation timestamp, the name of the microservice that generated the event, and the event description. Depending on the context of the operation, it optionally populates fields such as the doctor ID, the patient's hashed fiscal code, the processed data ID, or the report ID.

5.2.4 Backend

The codebase was organized into modular packages corresponding to the main microservices, each of one has separate folders for models, schemas, repositories, services, utilities, and API routes.

The following subsections aim to illustrate in detail the implementation choices adopted for the realization of the system.

Authentication

The Authentication Service serves as the centralized authority for identity management and access control within the system. Built using the FastAPI framework, it exposes RESTful endpoints to handle doctor registration, login procedures, and token validation. The internal architecture of this microservice is designed to ensure separation of concerns, asynchronous performance, and rigorous security standards.

To decouple the business logic from the underlying persistence mechanism, the service implements the Repository Pattern. The `DoctorRepository` class acts as an abstraction layer over the database, utilizing SQLAlchemy with `AsyncSession` to perform non-blocking I/O operations. This allows the service to handle high concurrency efficiently. The repository exposes methods such as `find_by_email` and `save`, ensuring that the domain logic in `AuthenticationService` interacts with domain objects (`Doctor` entities) rather than raw SQL queries.

A key architectural feature of this service is the implementation of the Observer Pattern for auditing purposes. The `AuthenticationService` acts as a Subject that notifies attached observers of critical state changes. The `AuditClient`, implementing the `IObserver` interface, listens for events such as registration successes or login failures and asynchronously transmits these logs to the logging microservice via HTTP requests. This design ensures that the authentication logic remains decoupled from the logging infrastructure.

The registration process, handled by the `register_doctor` method, adheres to strict security protocols. Upon receiving a `RegisterDoctorRequest`, the service first verifies that the email is not already associated with an existing account. Crucially, user passwords are never stored in plaintext. The service utilizes a `PasswordHasher` utility—leveraging the bcrypt algorithm via the `passlib` library—to generate a secure hash of the password before persistence. Once the `Doctor` entity is successfully saved to the database via the repository, the service triggers a notification to the audit system to log the creation of the new user.

The login workflow is managed via the `/login` endpoint. When a user submits credentials, the service retrieves the corresponding doctor record and verifies the provided password against the stored hash using the `PasswordHasher.verify` method. Upon successful verification, the system employs a `JwtSigner` utility to generate a JSON Web Token (JWT). This token is signed using the HS256 algorithm and contains the user's ID as the subject (`sub`) and an expiration timestamp. This stateless authentication mechanism allows other microservices to verify the user's identity without querying the central database for every request.

To support this distributed verification, the service exposes a `/validate` endpoint. This endpoint accepts a token, verifies its digital signature and expiration, and ensures the associated user still exists in the system. If valid, it returns the user's ID, effectively acting as an introspection endpoint for the rest of the architecture.

Audit

The Audit Service acts as the centralized observability hub for the entire distributed architecture. Implemented with FastAPI, it provides a unified interface for other microservices (such as Authentication and Core Diagnostic) to offload their operational logs. This design ensures that the business logic of individual services remains uncluttered by logging concerns, while simultaneously aggregating a comprehensive audit trail in a single persistence store.

The service exposes its functionality through a RESTful API, specifically via the `/audit/log` endpoint. This `POST` route serves as the entry point for event ingestion. When an external service (e.g., the Authentication Service reporting a login failure) sends a request, the `AuditService` acts as the controller. It leverages FastAPI's Dependency Injection system to instantiate the service layer, ensuring that the request is processed within a valid asynchronous database session scope.

Data integrity is enforced at the application boundary using Pydantic schemas. The `CreateLogRequest` model defines the strict contract for incoming logs. While fields such as `service`, `event`, and `description` are mandatory to ensure basic traceability, the schema allows for flexibility through optional context fields. Attributes like `doctor_id`, `patient_hashed_cf`, `report_id`, and `data_id` are defined as nullable types (e.g., `int | None`). This structural design allows the service to handle heterogeneous events—ranging from a generic system error (which has no patient context) to a specific diagnostic report generation (which requires links to the doctor, patient, and data)—without requiring multiple database tables.

The persistence logic is encapsulated within the `LogRepository` class, which implements the `ILogRepository` interface. This abstraction isolates the database operations from the business logic.

- **Writing:** The `save` method utilizes SQLAlchemy's asynchronous session to commit the `Log` entity to the PostgreSQL database. Notably, the timestamping is handled automatically via the `server_default=func.now()` directive in the model, ensuring temporal consistency across all records.

- **Reading:** The repository also implements a sophisticated retrieval mechanism via the `find_with_filters` method. Instead of multiple specific query methods, it constructs a dynamic SQL query using SQLAlchemy's `select` and `and_` operators. It programmatically appends `WHERE` clauses based on which filters (e.g., filtering by `doctor_id` or `event` type) are present in the request, allowing for granular audit inspections.

Data Processing

The Data Processing Service is responsible for ingesting raw patient data, applying modality-specific preprocessing algorithms, and storing the normalized output for subsequent AI analysis. Given the multimodal nature of the project (handling text, images, and biosignals), this service requires a highly flexible architecture capable of routing requests to the appropriate processing logic dynamically.

To manage the complexity of supporting multiple data types, the service implements the Chain of Responsibility (CoR) design pattern.

The `DataProcessingHandler` abstract base class defines the interface for handling requests and passing them to the next handler in the chain. The service constructs a processing pipeline in the `_build_preprocessing_chain` method, linking specific handlers:

- `TextPreprocessingHandler`
- `ImagePreprocessingHandler`
- `NumericPreprocessingHandler`
- `SignalPreprocessingHandler`

When a `process` request is received, the input flows through this chain. Each handler inspects the `strategy` field of the request. If the handler matches the strategy (e.g., "img_rx"), it processes the data; otherwise, it delegates execution to the `next_handler`. This ensures that adding a new data type in the future (e.g., genetic data) only requires creating a new handler class and adding it to the chain, adhering to the Open/Closed Principle.

Each handler implements specific algorithms to normalize data for AI consumption:

Image Processing The ImagePreprocessingHandler performs rigorous transformation to prepare inputs for Convolutional Neural Networks (CNNs). It decodes the Base64 input, resizes images to a standard 224×224 resolution, and applies normalization (using standard mean and standard deviation values). Finally, it converts the image into a tensor format, serializes it via pickle, and encodes it back to Base64 for storage.

Signal Processing The SignalPreprocessingHandler handles time-series data (e.g., ECGs). It parses the raw JSON input into NumPy arrays and applies Min-Max Normalization:

$$X_{norm} = \frac{X - X_{min}}{X_{max} - X_{min} + \epsilon}$$

This ensures that signal amplitudes are scaled between 0 and 1, preventing numerical instability during model inference.

Text and Numeric Processing The text handler uses Regular Expressions (re) to sanitize inputs by removing excess whitespace, while the numeric handler converts JSON features into structured NumPy arrays, ensuring type consistency (float32).

Once processed, the data is persisted via the ProcessedDataRepository. The service utilizes the same Observer Pattern seen in other microservices to report operations. Upon successful storage, the DataProcessingService notifies the AuditClient, which logs a "data_processed" event linked to the new data_id. This creates a traceable link between the raw data ingestion and the system audit trail.

Explainable AI

The Explainable AI Service represents the cognitive core of the architecture. Its primary responsibility is to retrieve preprocessed data, execute specific diagnostic algorithms based on the data modality, and, crucially, generate human-understandable explanations for its predictions.

To manage the diverse algorithmic requirements of multimodal data without creating a tangled conditional logic, the service implements the Strategy Pattern. The XAiService acts as the context, while the specific logic for each data type is encapsulated in concrete classes implementing the AnalysisStrategy interface.

The service maintains a registry mapping:

- **img_rx:** ImageAnalysisStrategy (for X-Rays)
- **img_skin:** ImageAnalysisStrategy (for Dermoscopy)
- **numeric:** NumericAnalysisStrategy (for Heart Disease tabular data)
- **text:** TextAnalysisStrategy (for Clinical Notes)

- **signal:** `SignalAnalysisStrategy` (for ECGs)

When an analysis request is received, the service inspects the `strategy` field, dynamically instantiates the correct class, and delegates the execution. This allows the system to be extended with new AI models (e.g., MRI analysis) simply by adding a new strategy class, adhering to the Open/Closed Principle.

Each strategy implements a distinct combination of inference models and explainability techniques.

Image Analysis The `ImageAnalysisStrategy` leverages PyTorch and Convolutional Neural Networks (CNNs). It utilizes a *DenseNet* (CheXNet) for chest X-rays and an *EfficientNet* for skin lesions. To provide explainability, the service implements Grad-CAM (Gradient-weighted Class Activation Mapping). By computing the gradients of the target class score with respect to the feature maps of the final convolutional layer, the system generates a heatmap. This heatmap is overlaid on the original image to visually highlight the specific regions (e.g., a lung nodule or skin border) that influenced the model's decision, providing immediate visual verification for the doctor.

Numeric Analysis The `NumericAnalysisStrategy` utilizes an XGBoost classifier to assess heart disease risk based on patient vitals. Unlike opaque decision trees, this strategy integrates SHAP (Shapley Additive Explanations). SHAP values are based on game theory and quantify the contribution of each feature (e.g., cholesterol level, age) to the prediction. The service returns a sorted list of features that pushed the risk score up or down, offering a precise local explanation for each specific patient.

Textual and Signal Analysis For unstructured data, the system employs a hybrid approach:

- **Text:** The `TextAnalysisStrategy` uses a ClinicalBERT transformer (via HuggingFace) for high-precision classification of medical notes into macro-categories (e.g., Cardiology, Neurology). It then chains this result into a Google Gemini (Generative AI) prompt to synthesize a natural language explanation of *why* the symptoms match that category.

- **Signals:** The `SignalAnalysisStrategy` processes ECG data by calculating statistical descriptors (min, max, mean) and passing the normalized signal array to the Gemini 2.5 Flash model. The LLM acts as an anomaly detector, analyzing the waveform morphology to identify arrhythmias and returning a structured JSON diagnosis.

The analysis workflow is asynchronous and distributed. Upon receiving a request, the `XAiService` performs an HTTP `GET` request to the Data Processing Service to retrieve the prepared data payload. Once the strategy returns the diagnosis, confidence score, and explanation, the result is encapsulated in a `Report` entity. This report is persisted via the `ReportRepository`, linking the AI output to the Doctor and the Patient's hashed ID. Finally, the service notifies the `AuditService` via the Observer pattern, ensuring that every diagnostic event is immutably logged.

Gateway

The Gateway Service functions as the single entry point (API Gateway) for the entire distributed system. Implemented using FastAPI, it abstracts the underlying microservices architecture from the client applications (Frontend), acting as a reverse proxy and a service orchestrator. This design implements the Facade Pattern, simplifying client interactions while enforcing security policies at the perimeter.

Unlike a simple pass-through proxy, the Gateway actively coordinates workflows that span multiple internal services. The most critical example of this orchestration is the `/analyse` endpoint, which implements a synchronous saga pattern to generate a diagnostic report.

The workflow executed by the `Gateway.analyse` method is as follows:

1. **Authentication:** Upon receiving a request with a `Bearer` JWT, the Gateway first calls the Authentication Service (`/validate`). It retrieves the verified `doctor_id`, ensuring the requester is authorized before any processing resources are consumed.

2. **Data Retrieval:** Next, it extracts the `raw_data` and `strategy` from the request and forwards them to the Data Processing Service (`/process`). It waits for the response containing the unique `processed_data_id`.

3. **Diagnostic Execution:** Finally, it combines the `doctor_id` (from step 1) and the `processed_data_id` (from step 2) into a new payload. This is sent to the Explainable AI Service (`/analyse`) to trigger the actual inference and explanation generation.

This approach ensures that the frontend only needs to make a single HTTP request, while the Gateway handles the complexity of chaining the internal services.

The Gateway is responsible for enforcing network security policies:

- **CORS Management:** It utilizes `CORSMiddleware` to handle Cross-Origin Resource Sharing, allowing the frontend application to communicate with the backend API securely, regardless of the hosting domain.

- **Token Extraction:** It implements a dependency (`get_jwt`) that strictly parses the `Authorization` header. If the header is missing or malformed, the Gateway rejects the request immediately with a 401 Unauthorized error, preventing invalid traffic from reaching the internal microservices.

To maintain high throughput, the service utilizes HTTPX as its internal HTTP client. The `HttpClient` wrapper ensures that all upstream requests to internal microservices are non-blocking (asynchronous). This allows the Gateway to handle concurrent incoming requests efficiently without being blocked while waiting for the AI or Database operations to complete in other containers.

5.2.5 Frontend

The frontend of the Early Care Gateway platform is implemented as a single-page application (SPA) using React, with the primary goal of providing an intuitive, responsive, and secure user interface for clinical users. The application is designed to guide authorized doctors through the authentication process, the execution of AI-powered clinical analyses, and the consultation of previously generated reports.

At the architectural level, the application is structured around React Router, which enables client-side routing and a clear separation between public and protected areas of the system. The entry point of the application initializes the React rendering process and mounts the main application component, ensuring global styles and configuration are applied consistently across all views.

Routing and Access Control The routing logic defines a public authentication route and a protected dashboard area. Access to the dashboard and its internal pages is regulated through a dedicated route protection mechanism based on JSON Web Tokens (JWT). Upon successful authentication, the JWT token is stored in the browser local storage and automatically attached to subsequent API requests. If no valid token is found, the user is transparently redirected to the authentication page, preventing unauthorized access to clinical functionalities.

Authentication Page The authentication page provides both login and registration functionalities within a single, adaptive interface. Input validation is performed client-side to ensure correctness of credentials and to provide immediate feedback to the user. The interface is designed with a clear visual separation between branding elements and form controls, improving usability and reinforcing the identity of the platform. Figure 5.6 and Figure 5.7 illustrate the registration and login pages.

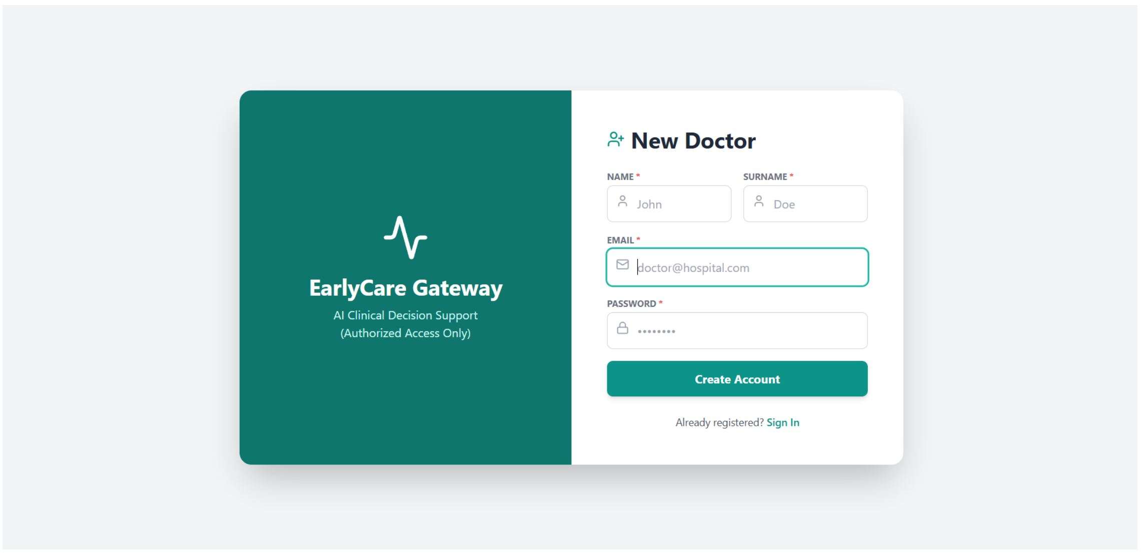


Figure 5.6: User Interface of Registration Page.

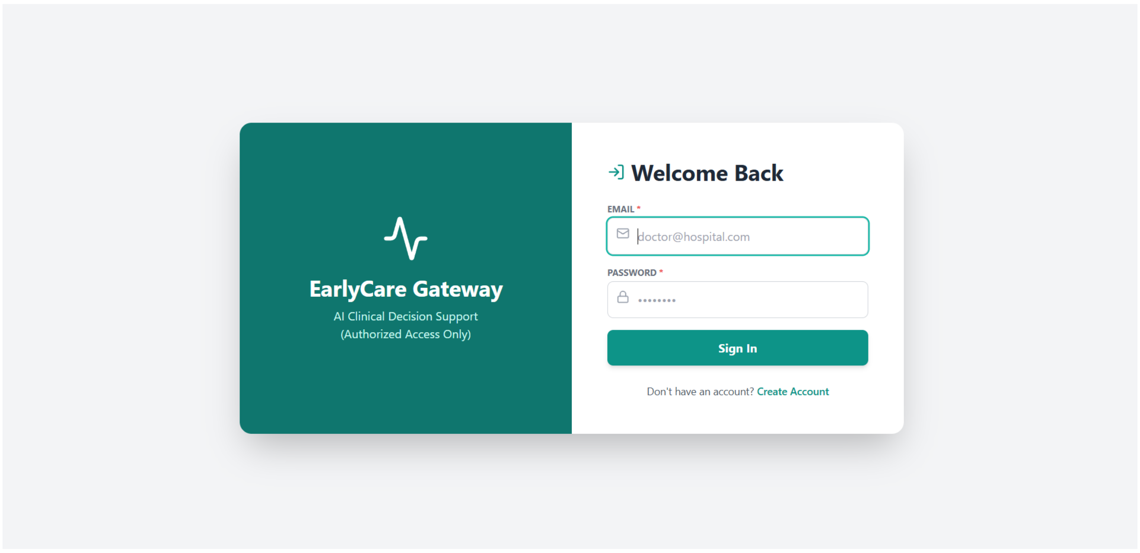


Figure 5.7: User Interface of Login Page.

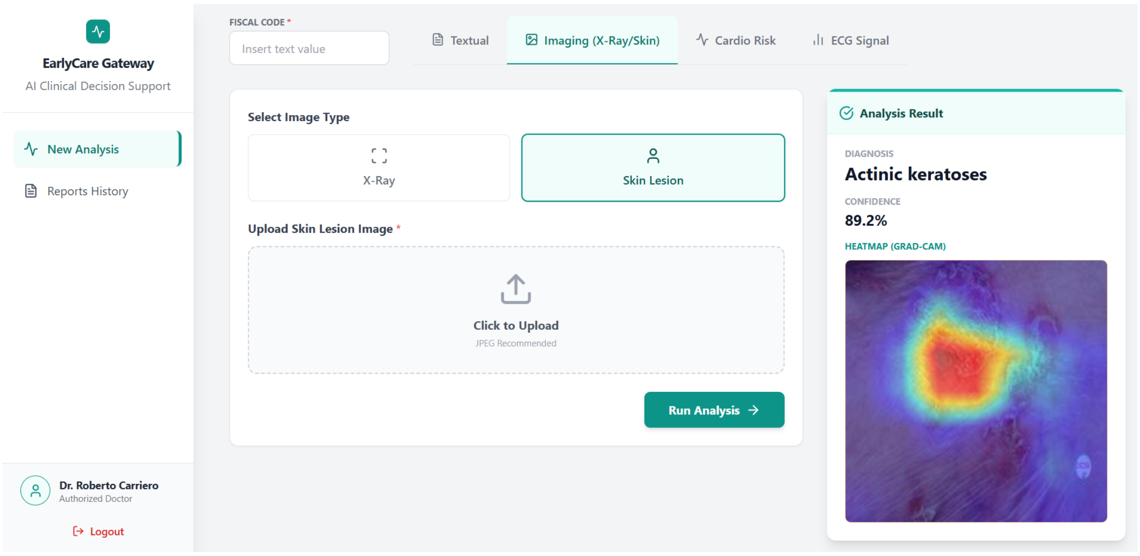
Dashboard Layout and Navigation Once authenticated, users are redirected to the main dashboard, which is built around a reusable layout component. This layout includes a persistent sidebar containing navigation controls, user information, and logout functionality, while the main content area dynamically renders the selected view. The sidebar clearly separates the core functionalities of the platform, namely the execution of new analyses and the consultation of report history, enabling fast and intuitive navigation.

User information, such as the doctor's name and surname, is extracted from the decoded JWT token and displayed in the interface, reinforcing transparency and accountability within the clinical environment.

Updated Diagnostic Hub The Diagnostic Hub, shown in Figure 5.8 allows users to perform analyses based on textual clinical notes, medical images (X-ray and skin lesion), structured cardiovascular risk parameters, and raw ECG signal samples. The component enforces mandatory patient identification through a fiscal code, which is validated on the client side before being transmitted to the backend for hashing and audit purposes.

Depending on the selected analysis strategy, the frontend dynamically adapts the user interface, validates inputs, and performs preliminary preprocessing, including Base64 encoding of images, numerical parsing of ECG signals, and client-side feature engineering for cardiovascular risk assessment through one-hot encoding of categorical variables and normalization of continuous features. All processed data are sent to the AI Gateway using a unified request format, enabling backend services to remain strategy-agnostic.

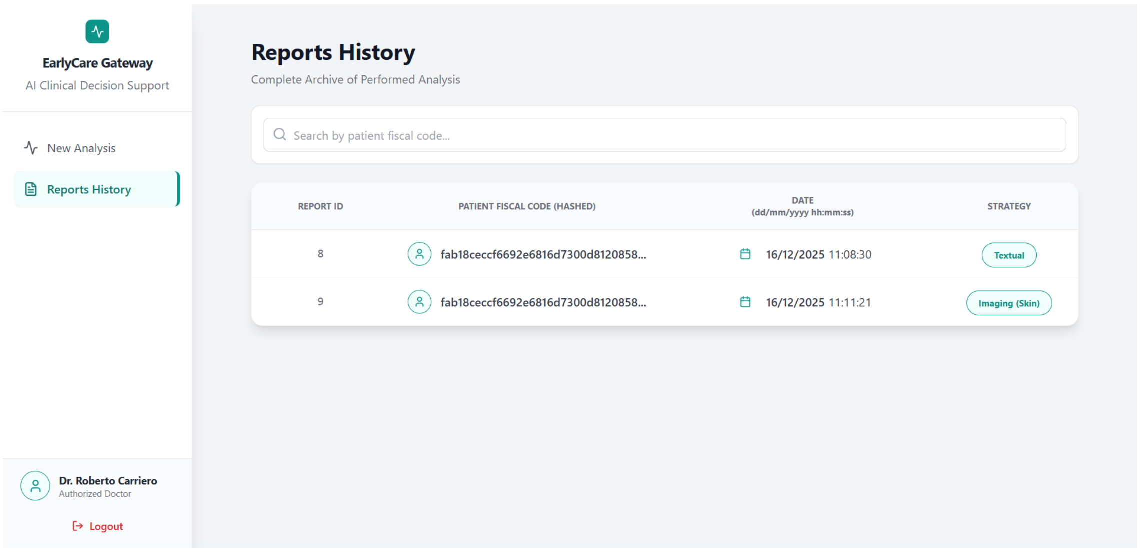
The Diagnostic Hub also handles the visualization of AI outputs, displaying diagnoses, confidence scores, and explainability artifacts such as Grad-CAM heatmaps for imaging tasks, feature impact tables for cardiovascular predictions, and textual explanations for text- and signal-based analyses, thereby improving transparency and interpretability of the overall system.



DIAGNOSIS
Actinic keratoses
CONFIDENCE
89.2%

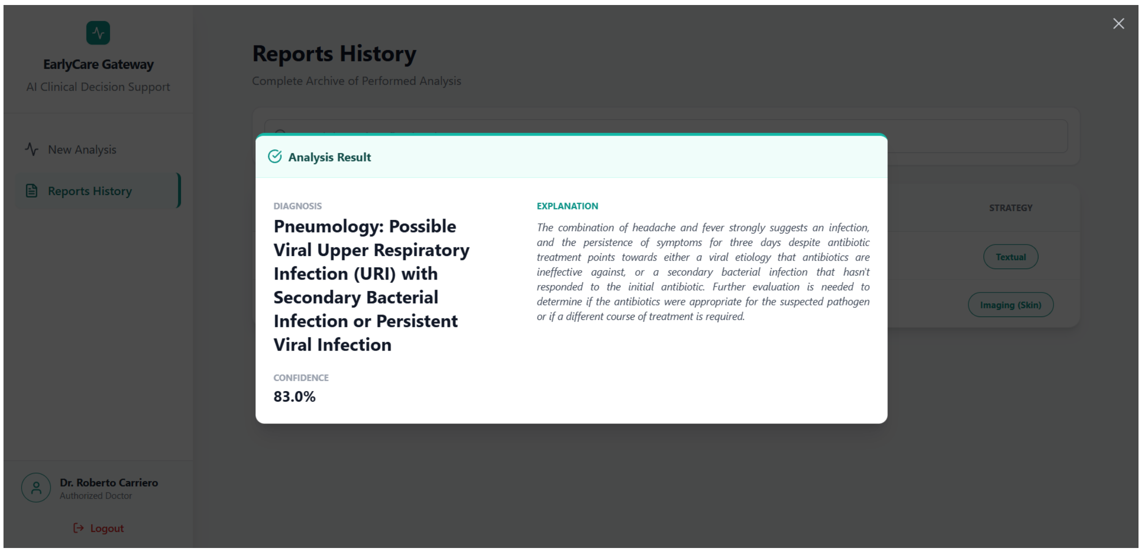
Figure 5.8: User Interface of Diagnostic Hub Page.

Reports History Page In addition to real-time analysis, the frontend provides a dedicated view for consulting previously generated reports. This section allows doctors to retrieve historical analyses, optionally filtering them by patient fiscal code, enabling evaluation and clinical follow-up. The reports history interface is shown in Figure 5.9. By clicking on a report, a detailed modal is displayed, providing additional information about the diagnosis, as illustrated in Figure 5.10.



REPORT ID	PATIENT FISCAL CODE (HASHED)	DATE (dd/mm/yyyy hh:mm:ss)	STRATEGY
8	fab18ceccf6692e6816d7300d8120858...	16/12/2025 11:08:30	Textual
9	fab18ceccf6692e6816d7300d8120858...	16/12/2025 11:11:21	Imaging (Skin)

Figure 5.9: User Interface of Reports History Page.



Analysis Result

DIAGNOSIS

Pneumology: Possible Viral Upper Respiratory Infection (URI) with Secondary Bacterial Infection or Persistent Viral Infection

CONFIDENCE

83.0%

EXPLANATION

The combination of headache and fever strongly suggests an infection, and the persistence of symptoms for three days despite antibiotic treatment points towards either a viral etiology that antibiotics are ineffective against, or a secondary bacterial infection that hasn't responded to the initial antibiotic. Further evaluation is needed to determine if the antibiotics were appropriate for the suspected pathogen or if a different course of treatment is required.

Figure 5.10: User Interface of Reports History Details Modal.

Styling and User Experience The entire frontend leverages a utility-first styling approach through Tailwind CSS, ensuring visual consistency, responsiveness across devices, and rapid interface iteration. Icons and visual cues are used extensively to guide user interaction, reduce cognitive load, and clearly distinguish between different analysis modalities and system states.

Overall, the frontend architecture combines modular React components, secure routing mechanisms, and a user-centered design philosophy to provide a robust and clinically oriented interface for interacting with the Early Care Gateway platform.

5.3 Deployment

Deployment strategies focused on containerization and scalable hosting to ensure reproducibility and system availability. Docker was employed to encapsulate each microservice in independent containers, isolating dependencies and simplifying deployment.

The entire system is designed to be infrastructure-agnostic through the use of Docker containerization. This approach ensures consistency between development, testing, and production environments.

The `Dockerfile` for each microservice has been optimized to minimize the final image size and accelerate build times. By separating the dependency installation phase (copying `requirements.txt` first) from the source code copy, the build process leverages Docker's layer caching mechanism. If the application code changes but the dependencies do not, Docker reuses the cached dependency layer, significantly reducing deployment time.

All sensitive parameters, such as database credentials, API keys, and internal service URLs, are injected via environment variables defined in a `.env` file.

Service orchestration is defined in the `docker-compose.yaml` file, which describes a complete environment composed of eight interconnected services.

The backend services are built from their respective contexts. Crucially for the development phase, these services mount the local source code as a volume (`./backend/...:/app`) and run with the `-reload` flag. This enables Hot Reloading, allowing developers to see code changes immediately without rebuilding the containers.

The services communicate via an internal Docker bridge network using service discovery. For instance, the Gateway communicates with the Authentication Service using the hostname `http://auth_service:8000`, resolving internal IP addresses automatically.

Furthermore, execution scripts are provided to simplify the project startup process, enabling a one-click deployment while automatically performing bootstrap integration tests.

Chapter 6

Testing and Results

This chapter presents the tests executed on the systems and the experimental results obtained from the validation of the AI models.

6.1 Microservices Integration Test

Given the asynchronous nature of the microservices (built with async FastAPI), the testing framework required tools capable of handling non-blocking I/O operations effectively.

The testing suite relies on the following key libraries:

- **pytest**: The primary testing framework, chosen for its simplicity and powerful fixture system.
- **pytest-asyncio** and anyio: Essential plugins that allow pytest to execute coroutines and manage asynchronous event loops, enabling the testing of async def endpoints.
- **httpx**: A modern, async-capable HTTP client used to send requests to the microservices. It mirrors the production behavior of the Gateway service, ensuring that tests simulate inter-service communication.

The tests are designed as Integration Tests, verifying the interaction between the application logic, the database, and the external API endpoints. The following subsections detail the scenarios covered.

The `test_register_login_validate_flow` validates the complete user lifecycle. It ensures that a doctor can register with unique credentials, login to receive a valid JWT, and that duplicate email registrations are correctly rejected with a 400 Bad Request status.

The data processing tests verify that raw data (text, signals, or images) is accepted, processed, and retrieved. The `test_process_and_retrieve_flow` confirms that the system returns a valid `processed_data_id`, which is required for the subsequent analysis steps.

The most complex test scenario, `test_analyse_and_get_reports_flow`, simulates the core of the system. It orchestrates a sequence of actions across multiple services:

1. POST raw data to the Data Service.
2. Use the returned ID to request an analysis from the XAI Service.
3. Verify that the XAI Service returns a diagnosis and an explanation.
4. Query the `/reports` endpoint to ensure the result was persisted and is retrievable by the doctor.

Finally, the `test_create_log` ensures that operational events are captured. It validates that logs can be filtered by specific query parameters.

The entire testing process is automated via a custom script. This script orchestrates the full lifecycle of the test session, minimizing manual intervention and handling the synchronization between the container startup and the test runner.

The execution pipeline performs the following steps sequentially:

1. **Start the system** It initializes the environment using `Docker-Compose`, ensuring that the latest version of the code is containerized and running.
2. **Check the services:** The script implements a polling mechanism to verify that each microservice is fully operational. It systematically pings the Swagger UI endpoint (`/docs`) of every service (Audit, Auth, Data, XAI, Gateway) and waits for an HTTP 200 OK response. This eliminates race conditions where the test suite might attempt to connect to a service that is still initializing.
3. **Manage the dependencies:** It automatically creates a dedicated Python virtual environment (venv) and installs the specific test dependencies defined in `requirements.txt`.
4. **Run the tests:** Once the infrastructure is healthy and dependencies are ready, it launches the `pytest` suite.

6.2 Experimental Setup and Metrics

The performance evaluation focuses on measuring the system's ability to correctly classify clinical data across different modalities: text, images, and structured data.

The validation process was conducted using a hold-out strategy. The datasets were split into training (80%) and validation/test sets (20%) to ensure that the evaluation metrics reflect the model's generalization capabilities on unseen data. The training was performed on NVIDIA Tesla T4 GPUs to accelerate the fine-tuning of deep learning models (ClinicalBERT, CheXNet, EfficientNet).

To comprehensively assess the performance, the following metrics were employed:

- **Accuracy:** The ratio of correctly predicted observations to the total observations.
- **Precision:** The ratio of correctly predicted positive observations to the total predicted positives. High precision relates to a low false positive rate.

- **Recall (Sensitivity):** The ratio of correctly predicted positive observations to the all observations in the actual class. In a medical context, high recall is crucial to minimize missed diagnoses (False Negatives).
- **F1-Score:** The harmonic mean of Precision and Recall. It is particularly useful for evaluating performance on imbalanced datasets, as it seeks a balance between precision and sensitivity.
- **Confusion Matrix:** A tabular layout that visualizes the performance of the classifier, showing the discrepancies between predicted and actual labels to identify specific classes that are prone to misclassification.

6.3 Textual Analysis Results (ClinicalBERT)

The textual analysis module, based on the fine-tuned Bio_ClinicalBERT model, was evaluated on the validation subset of the MTSamples dataset ($N = 338$ samples). The model was tasked with classifying clinical notes into 8 distinct medical specialties.

6.3.1 Global Performance

The model achieved an overall Accuracy of 88% on the validation set. The weighted average F1-Score was also 0.88, indicating consistent performance across the majority of classes despite some imbalance in the dataset distribution.

Table 6.1: Detailed Classification Report for ClinicalBERT on the Validation Set.

Medical Specialty	**Precision**	**Recall**	**F1-Score**	**Support**
Cardiovascular / Pulmonary	0.92	0.96	0.94	74
Neurology	0.77	0.76	0.76	45
Urology	0.93	0.88	0.90	32
Orthopedic	0.83	0.87	0.85	71
Obstetrics / Gynecology	0.94	0.97	0.95	32
Hematology - Oncology	0.80	0.67	0.73	18
Gastroenterology	0.96	0.93	0.95	46
ENT - Otolaryngology	0.89	0.85	0.87	20
Accuracy			0.88	338
Macro Avg	0.88	0.86	0.87	338
Weighted Avg	0.88	0.88	0.88	338

6.3.2 Class-wise Analysis

As detailed in Table 6.1 and visualized in the histograms in Figure 6.1, the model demonstrates varying degrees of effectiveness depending on the specialty:

- **High Performance:** The model performed well in *Obstetrics / Gynecology* (F1: 0.95) and *Gastroenterology* (F1: 0.95). Notably, *Cardiovascular / Pulmonary* achieved the highest Recall (0.96), meaning the system identified 96% of all true cardiovascular cases, a critical factor for triaging life-threatening conditions.
- **Challenges:** The lowest performance was observed in *Hematology - Oncology* (F1: 0.73, Recall: 0.67). This drop can be attributed to the low support (only 18 samples in validation) and the fact that oncological symptoms often manifest systematically, overlapping with other organ-specific specialties.

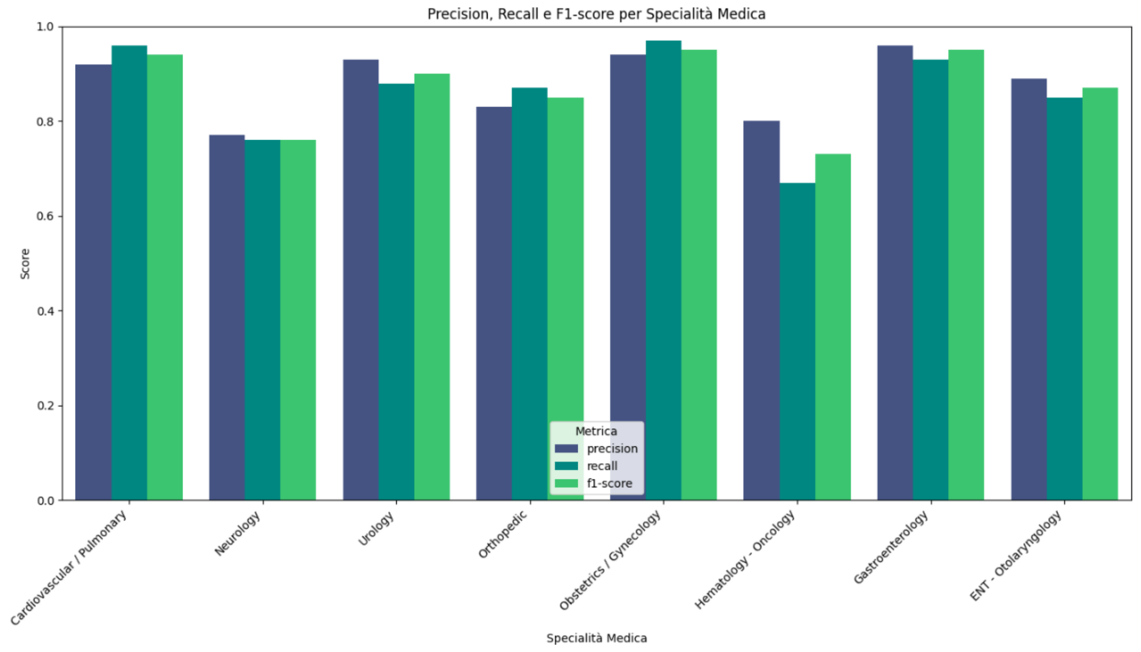
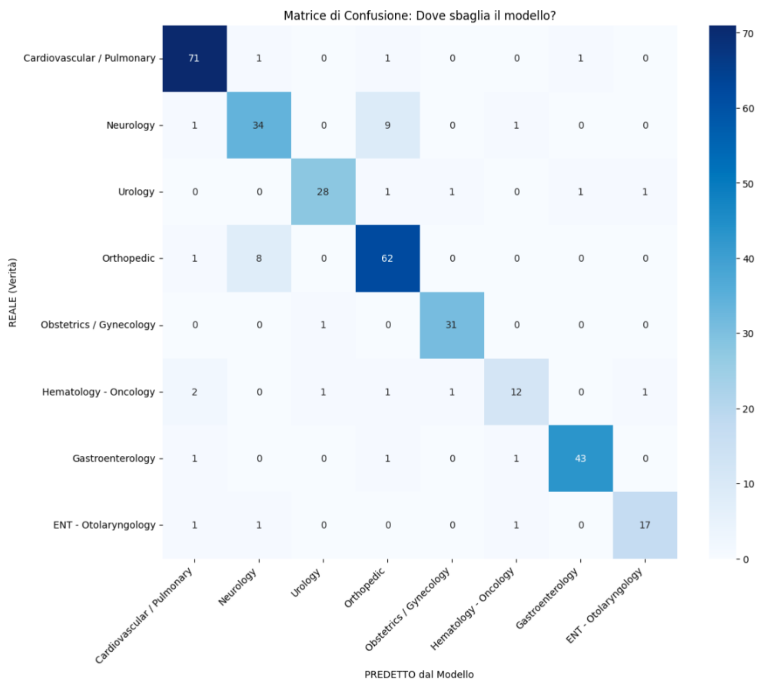


Figure 6.1: Bar Chart comparing Precision, Recall, and F1-Score across Medical Specialties.

6.3.3 Error Analysis

The Confusion Matrix (Figure 6.2) provides deeper insight into the misclassifications.



	Cardiovascular / Pulmonary	Neurology	Urology	Orthopedic	Obstetrics / Gynecology	Hematology - Oncology	Gastroenterology	ENT - Otolaryngology
Cardiovascular / Pulmonary	71	1	0	1	0	0	1	0
Neurology	1	34	0	9	0	1	0	0
Urology	0	0	28	1	1	0	1	1
Orthopedic	1	8	0	62	0	0	0	0
Obstetrics / Gynecology	0	0	1	0	31	0	0	0
Hematology - Oncology	2	0	1	1	1	12	0	1
Gastroenterology	1	0	0	1	0	1	43	0
ENT - Otolaryngology	1	1	0	0	0	1	0	17

Figure 6.2: Confusion Matrix for the Textual Classification task.

The most significant confusion occurs between Neurology and Orthopedics.

- 9 Neurology cases were misclassified as Orthopedic.
- 8 Orthopedic cases were misclassified as Neurology.

This confusion is explainable, as both specialties often deal with overlapping symptomatology such as pain, numbness, and motor dysfunction (e.g., back pain could be spinal/orthopedic or neurological). Conversely, classes with distinct lexicons, such as *Urology* and *Gastroenterology*, show very high diagonal values with minimal leakage into other categories.

6.4 Medical Imaging Results

This section evaluates the performance of the computer vision models. Due to the distinct nature of the tasks—Multi-Label classification for chest X-rays and Multi-Class classification for skin lesions—specific evaluation metrics and visualization techniques were adopted for each domain.

6.4.1 Chest X-Ray Analysis (CheXNet)

The performance of the DenseNet121-based model was assessed using the Area Under the Receiver Operating Characteristic Curve (AUC-ROC) for each of the 14 target pathologies. The AUC is the standard metric for multi-label medical diagnostics, as it measures the model's discriminative ability across all possible classification thresholds, which is critical for imbalanced datasets.

Quantitative Performance

The model achieved a Mean AUC of 0.8371, demonstrating strong generalization capabilities. Table 6.2 provides the detailed AUC scores for each pathology, sorted by performance.

Table 6.2: AUC-ROC Performance of the CheXNet model for the 14 Thoracic Pathologies.

Pathology	**AUC Score**	**Pathology**	**AUC Score**
Emphysema	0.9143	Fibrosis	0.8155
Edema	0.9008	Pleural Thickening	0.8031
Cardiomegaly	0.8971	Consolidation	0.8016
Pneumothorax	0.8888	Nodule	0.7810
Effusion	0.8836	Pneumonia	0.7507
Hernia	0.8705	Infiltration	0.7208
Mass	0.8665		
Atelectasis	0.8246	MEAN AUC	0.8371

As shown in the ROC Curves (Figure 6.3) and the performance chart (Figure 6.4), the system excels in detecting conditions with distinct visual features such as *Emphysema* (0.91) and *Edema* (0.90). The lower performance on *Infiltration* (0.72) is consistent with medical literature, as this pathology presents diffuse and subtle radiological patterns that are challenging even for expert radiologists.

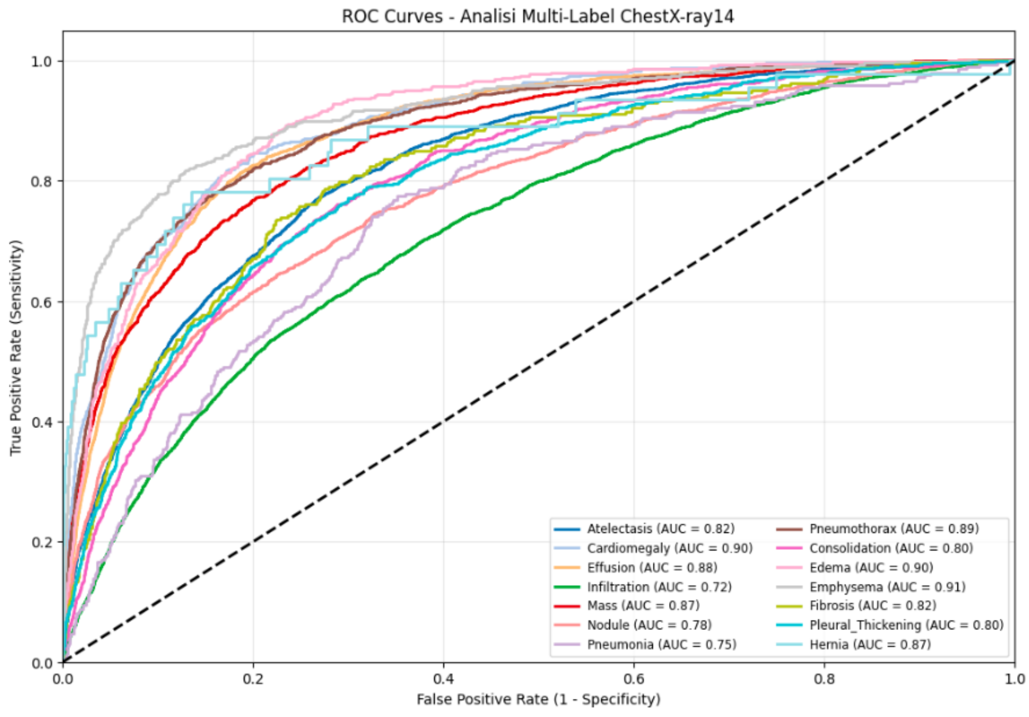


Figure 6.3: ROC Curves for the 14 Findings in the ChestX-ray14 Dataset.

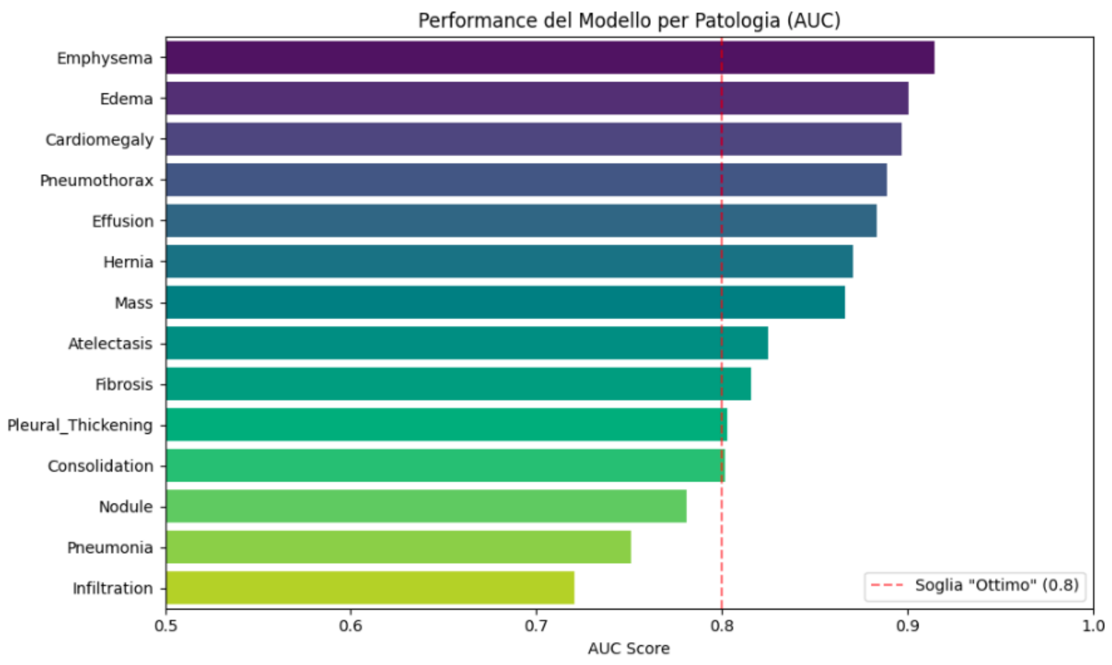


Figure 6.4: Bar Chart Ranking the AUC Scores by Pathology.

6.4.2 Skin Lesion Diagnosis (EfficientNet)

For skin lesion classification, the primary objective was to maximize sensitivity (Recall) for malignant conditions, which are often underrepresented. The EfficientNet-B0 model, trained with a *Weighted Cross-Entropy Loss*, achieved an overall Accuracy of 88% on the test set ($N = 2003$).

Class-wise Analysis

Table 6.3 details the precision, recall, and F1-score for each diagnostic category.

Table 6.3: Classification Report for the Skin Lesion Analysis task (Weighted).

Diagnosis	**Precision**	**Recall**	**F1-Score**	**Support**
Melanocytic nevi	0.95	0.92	0.93	1341
Basal cell carcinoma	0.87	0.84	0.86	103
Vascular lesions	0.77	0.96	0.86	28
Dermatofibroma	0.76	0.96	0.85	23
Benign keratosis	0.75	0.80	0.77	220
Actinic keratoses	0.69	0.74	0.71	65
Melanoma	0.69	0.73	0.71	223
Accuracy			0.88	2003
Weighted Avg	0.88	0.88	0.88	2003

Key Findings

- **Melanoma Detection:** The model achieved a Recall of 73% for Melanoma. In a screening context, this sensitivity is vital to ensure that nearly 3 out of 4 malignant cases are flagged for biopsy, prioritizing patient safety over a slight increase in false positives (Precision 69%).
- **Rare Classes:** The impact of class weighting is evident in the *Dermatofibroma* class (Recall 96%) and *Vascular lesions* (Recall 96%), where the model correctly identified almost all cases despite their scarcity in the dataset.
- **Robustness:** The majority class, *Melanocytic nevi*, maintained excellent performance (F1 0.93), confirming that the model did not overfit to the minority classes.

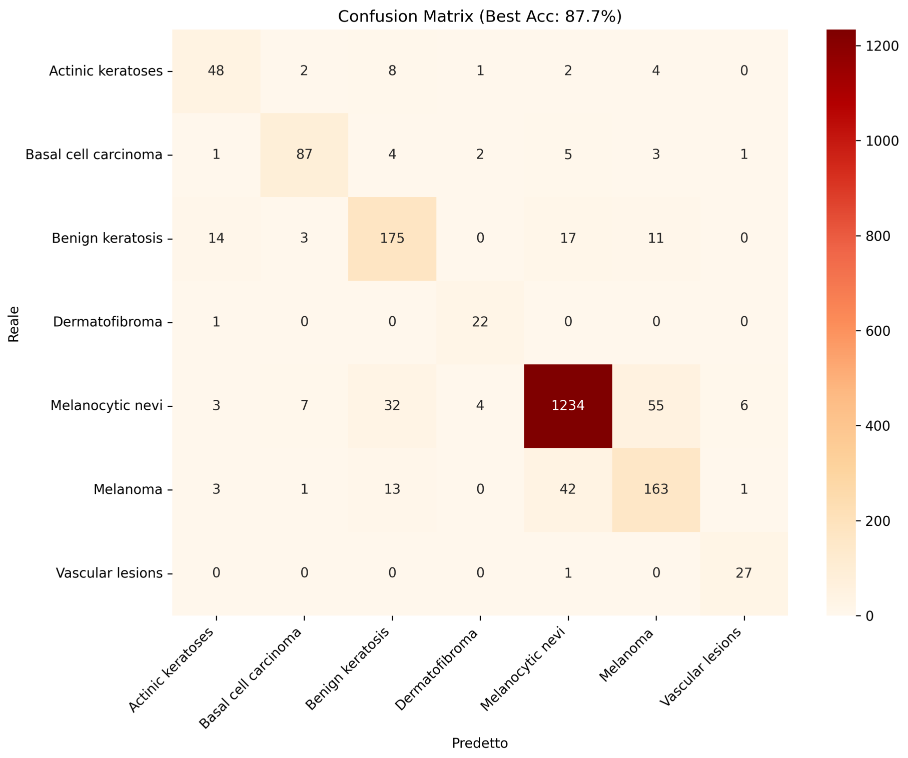


Figure 6.5: Confusion Matrix for Skin Lesion Classification. Note the High Diagonal Values for Nevi and the Effective Retrieval of Melanomas.

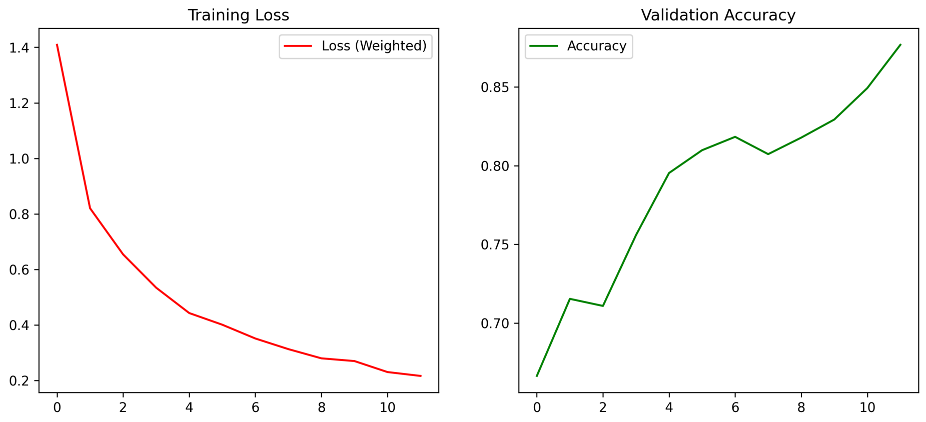


Figure 6.6: Training Loss (Weighted) and Validation Accuracy curves, showing Stable Convergence over 12 Epochs.

6.5 Cardiovascular Risk Prediction (XGBoost)

The assessment of cardiovascular risk using structured clinical data represents a critical component of the triage process. The XGBoost model, optimized via Grid Search, was evaluated on a test set of 184 patients derived from the UCI Heart Disease dataset.

6.5.1 Quantitative Performance

The model achieved an overall Accuracy of 82.1%, a good result for a tabular medical dataset. The detailed classification report is presented in Table 6.4.

Table 6.4: Classification Metrics for the Cardiovascular Risk Prediction task.

Class	**Precision**	**Recall**	**F1-Score**	**Support**
Healthy (0)	0.810	0.780	0.795	82
Heart Disease (1)	0.829	0.853	0.841	102
Accuracy			0.821	184
Macro Avg	0.819	0.817	0.818	184
Weighted Avg	0.820	0.821	0.820	184

A critical metric for this domain is the Recall for the Heart Disease class, which stands at 85.3%. This indicates that the system successfully identifies the majority of patients at risk, minimizing false negatives (missed diagnoses), which could have clinical consequences.

6.5.2 Error Analysis

The Confusion Matrix (Figure 6.7) further illustrates the model's behavior:

- **True Positives:** 87 patients correctly identified as at risk.
- **False Negatives:** Only 15 cases were missed.
- **False Positives:** 18 healthy patients were flagged as at risk. In a triage setting, this slight tendency to over-diagnose is acceptable as it prioritizes patient safety.

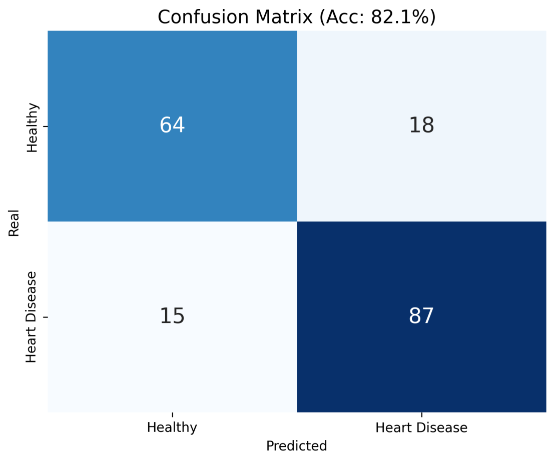


Figure 6.7: Confusion Matrix for Cardiovascular Risk prediction. The Model shows a Balanced Performance with a Bias towards Sensitivity.

6.5.3 Model Confidence and Interpretability

The probability distribution histogram (Figure 6.8) reveals a bimodal distribution, meaning the model is confident in its predictions (most probabilities are close to 0 or 1), with few ambiguous cases around the 0.5 threshold.

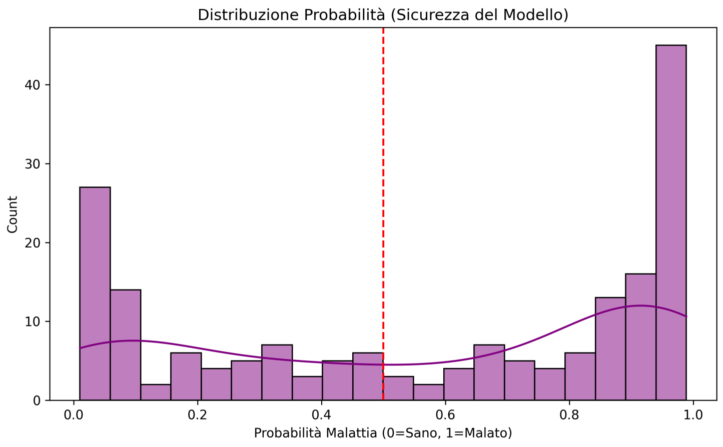


Figure 6.8: Distribution of Predicted Probabilities. The clear separation indicates Strong Model Confidence.

Finally, the Feature Importance analysis (Figure 6.9) confirms the clinical validity of the model. The most influential factors driving the predictions include:

1. **Exercise Induced Angina:** The strongest predictor of heart disease.
2. **Chest Pain Type (`cp`):** Specifically atypical and non-anginal pain.
3. **ST Slope (`slope`):** An essential ECG marker during stress tests.

These findings align perfectly with established cardiological knowledge, reinforcing trust in the AI's decision-making process.

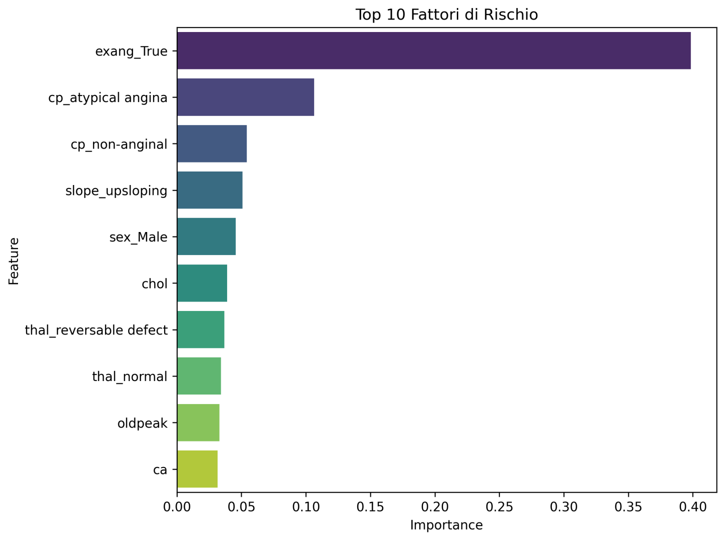


Figure 6.9: Top 10 Most Important Features identified by XGBoost.

Chapter 7

Conclusion

The development of the project has demonstrated the effectiveness of a Clinical Decision Support System designed to address the pressing challenges of modern healthcare, such as increasing cognitive load on medical professionals.

The results obtained throughout the experimental phase confirm the validity of the architectural and methodological choices adopted. From a software engineering perspective, the implementation of a containerized microservices architecture successfully achieved the goals of modularity and maintainability. By decoupling critical components such as authentication, data processing, and analysis logic, the system ensures high cohesion and low coupling, while satisfying strict non-functional requirements regarding data privacy, anonymization, and audit traceability.

On the artificial intelligence front, the multimodal strategy proved to be a decisive factor. The system's ability to adapt to different data types leveraging ClinicalBERT for textual notes, CheXNet and EfficientNet for medical imaging, and XGBoost for structured data, llowed it to achieve high performance metrics, as evidenced by the accuracy rates and AUC scores presented in the results chapter. Furthermore, a defining characteristic of this project was the integration of Explainable AI (XAI) techniques. By implementing SHAP, Grad-CAM, and Chain-of-Thought reasoning, the predictions are turned into transparent insights that assist human doctors.

In summary, EarlyCare Gateway stands as a reliable prototype that balances AI innovation and software engineering standards for data-driven optimization of clinical processes.

7.1 Future Developments

While the current iteration of the system has met its design objectives, the transition from a prototype to a fully operational deployment within hospital ecosystems requires specific evolutions. Future development will focus on enhancing system interoperability.

7.1.1 Clinical Interoperability

The system should communicate with existing Hospital Information Systems (HIS) and Electronic Health Records (EHR). It requires the integration of standard exchange protocols such as HL7 (Health Level Seven) and FHIR (Fast Healthcare Interoperability Resources). This standardization will enable the direct and automated ingestion of patient data, eliminating the need for manual entry and ensuring that diagnostic reports are automatically pushed to the patient's clinical history.

7.1.2 AI Evolution

In the domain of physiological signal analysis, while the current implementation leveraging Large Language Models for ECG interpretation has demonstrated semantic reasoning capabilities, future iterations will adopt 1D-Convolutional Neural Networks (1D-CNNs). These Deep Learning architectures are engineered for time-series data and are expected to offer better performance in detecting morphological features and rhythmic anomalies with greater computational efficiency compared to generative models.

7.1.3 Medical Feedback

To ensure the system continuously adapts to real-world medical complexities it is needed the development a Q&A (Question and Answer) mechanism that allows clinicians to validate and correct AI's suggestions. This expert feedback will generate a domain-specific dataset that will be used for the continuous re-training and fine-tuning of the models, allowing the algorithms to learn from their errors and adapt to local trends.

7.1.4 Infrastructure Scalability

The current deployment, based on Docker Compose, is suitable for development but will be migrated to Kubernetes. This orchestration platform will enable auto-scaling capabilities, allowing the system to dynamically allocate resources during peak hours. Concurrently, the inter-service communication architecture will transition towards a fully event-driven model using a Message Broker (such as RabbitMQ or Apache Kafka). This shift will further decouple the microservices, ensuring that resource-intensive AI inference tasks are handled asynchronously without impacting the responsiveness of the user interface.

Bibliography

[1] J. M. Zachariasse, N. Seiger, P. P. Rood, C. F. Alves, P. Freitas, F. J. Smit, and H. A. Moll, “Validity of the manchester triage system in emergency care: A prospective observational study,” *PLOS ONE*, vol. 12, no. 2, p. e0170811, 2017.

[2] I. P. J. Andika, S. Safaruddin, T. Y. Christina, Y. S. Baso, and S. Utami, “Effectiveness of the manchester triage system in the emergency department: A literature review,” in *BIO Web of Conferences*, vol. 152, p. 01004, EDP Sciences, 2025.

[3] R. Sánchez-Salmerón, J. L. Gómez-Urquiza, L. Albendín-García, M. Correa-Rodríguez, M. B. Martos-Cabrera, A. Velando-Soriano, and N. Suleiman-Martos, “Machine learning methods applied to triage in emergency services: A systematic review,” *International Emergency Nursing*, vol. 60, p. 101109, 2022.

[4] A. J. Thirunavukarasu, D. S. J. Ting, K. Elangovan, L. Gutierrez, T. F. Tan, and D. S. W. Ting, “Large language models in medicine,” *Nature medicine*, vol. 29, no. 8, pp. 1930–1940, 2023.

[5] H. W. Loh, C. P. Ooi, S. Seoni, P. D. Barua, F. Molinari, and U. R. Acharya, “Application of explainable artificial intelligence for healthcare: A systematic review of the last decade (2011–2022),” *Computer methods and programs in biomedicine*, vol. 226, p. 107161, 2022.

[6] N. Alshuqayran, N. Ali, and R. Evans, “A systematic mapping study in microservice architecture,” in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, pp. 44–51, 2016.

[7] J. Hunt, *Gang of Four Design Patterns*, pp. 135–136. Cham: Springer International Publishing, 2013.

[8] E. Alsentzer, J. Murphy, W. Boag, W.-H. Weng, D. Jin, T. Naumann, and M. McDermott, “Publicly available clinical bert embeddings,” in *Proceedings of the 2nd Clinical Natural Language Processing Workshop*, pp. 72–78, Association for Computational Linguistics, 2019.

[9] T. Boyle, “MTSamples: Transcribed medical samples dataset.” `https://www.kaggle.com/datasets/tboyle10/medicaltranscriptions`, 2018. Accessed: 2025-11-28.

[10] P. Rajpurkar, J. Irvin, K. Zhu, B. Yang, H. Mehta, T. Duan, D. Y. Ding, A. Bagul, C. P. Langlotz, K. S. Shpanskaya, M. P. Lungren, and A. Y. Ng, "Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning." arXiv preprint arXiv:1711.05225, 2017.

[11] X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri, and R. M. Summers, "Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2097–2106, 2017.

[12] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*, pp. 6105–6114, PMLR, 2019.

[13] P. Tschandl, C. Rosendahl, and H. Kittler, "The ham10000 dataset, a large collection of multi-source dermatoscopic images of common pigmented skin lesions," *Scientific data*, vol. 5, no. 1, pp. 1–9, 2018.

[14] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," *International Journal of Computer Vision*, vol. 128, no. 2, pp. 336–359, 2020.

[15] A. Janosi, W. Steinbrunn, M. Pfisterer, and R. Detrano, "Heart disease data set." UCI Machine Learning Repository, 1988. `https://archive.ics.uci.edu/ml/datasets/heart+disease`.

[16] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 785–794, 2016.

[17] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[18] G. DeepMind, "Gemini: A family of highly capable multimodal models." `https://deepmind.google/technologies/gemini/`, 2023.

[19] E. Tjoa and C. Guan, "Explainable artificial intelligence (xai) in healthcare," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 3, pp. 1217–1235, 2020.

[20] S. M. Lundberg and S.-I. Lee, "A unified approach to interpreting model predictions," *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[21] J. Wei *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, pp. 24824–24837, 2022.