ITMB
COC252
F011321

EHR System Modernization in the Republic of Moldova

by

Calin Corcimaru

Supervisor: Dr. Georgina Cosma

Department of Computer Science
Loughborough University

May 2025

Abstract

Abstract to be added

Acknowledgements

Acknowledgements to be added

List of Figures

Contents

Chapter 1

Introduction

1.1 Background

The Republic of Moldova is a small country in Eastern Europe that borders Romania and Ukraine, with a current population of 2.4 million people (**mdpop**). Since its independence in 1991, Moldova has faced a number of challenges, including political instability, corruption, and economic difficulties which have left Moldova as one of the poorest countries in Europe (**mdpoverty**).

Despite these challenges, Moldova has made significant progress in its digital transformation efforts, with the government launching a number of initiatives to modernize its public services and improve the quality of life for its citizens (**mdega**). An example is the Citizen's Government Portal (MCabinet), which allows citizens to access personal information such as 'valid identity documents, social contributions and benefits, own properties, information about the family doctor and the health institution where the person is registered, tax payments and other information about the citizen-government relationship' (**mdcabinet**).

To continue supporting the existing transformation initiatives, Moldova's Cabinet of Ministers has recently approved the 'Digital Transformation Strategy of the Republic of Moldova for 2023–2030', which aims to transform the country into a digital society by 2030, with the ultimate goal of having 'all public services available in a digitalized format' (**mdstrategy**).

1.2 Problem Statement

The healthcare sector in Moldova has also seen some transformations, with the introduction of a new electronic health record system (EHR) in 15 hospitals across the country in 2017, called 'Sistemul informațional automatizat „Asistența Medicală Spitalicească" (SIA AMS)' (**mdehr**). While the system has been successful in helping doctors access patient information more efficiently such as medical history, examinations, test results, and prescriptions, the system hasn't been updated since its inception in 2017 and there are still challenges that need to be addressed in 2024.

The main challenge with the current system lies in the user experience (UX) — SIA AMS feels old and isn't user-friendly, with a clunky interface that is difficult to navigate, not adhering to modern accesibility standards and only accessible via Internet Explorer or legacy version of Microsoft Edge, with no support for other browsers or devices (**mdehr**).

Another big challenge with the system is its lack of interoperability within public and private medical institutions due to a lack of a nationally-wide integrated system — each hospital and clinic have their own, siloed, information system that contains the patient information, with no communication being made between systems in different hospitals (**mdehr**).

Finally, due to the current economic situation in Moldova, the government has not allocated any funds to upgrade the current or develop new systems, and the hospitals and clinics that use the system do not have the resources to update it themselves.

1.3 The Client

The client, 'Nicolae Testemiteanu' State University of Medicine and Pharmacy in Moldova (USMF), is a public university in Chisinau, Moldova, that offers a range of medical programs, including medicine, dentistry and pharmacy (**mduni**). Many of the faculty at USMF are also practicing doctors at hospitals and clinics across Moldova, and have first-hand experience with the current IT systems used in both public and private medical institutions. The USMF faculty members that the student will be interacting with during the project are part of an innovation team that researches potential opportunities to improve the healthcare sector in Moldova through the use of technology. As such, the client has expressed a need for a prototype that can act as a proof of concept for a modern system that could either replace or augment the current system in Moldova.

1.4 Project Objectives

This project aims to initially conduct some research on the current situation of the IT systems used in the healthcare sector in Moldova by interviewing several stakeholders from various healthcare-related institutions. Afterwards, the project will conduct a literature review on the most appropriate technologies and methodologies for developing a modernized EHR system, and an analysis of existing EHR systems to identify their existing functionality. Finally, based on the information gathered, the project will focus on designing and developing a working prototype, based on the requirements gathered and the feasibility of the chosen solution for the Moldovan healthcare sector. The student's hope is that the solution can then be used as a proof of concept to secure funding for a full-scale implementation of the new system in Moldova by the relevant authorities, such as the Ministry of Health.

As such, the objectives of the project are as follows:

1. Identify 2 to 4 stakeholders from various perspectives, such as healthcare institutions in Moldova and patients, that can provide insights into the current IT systems used in the healthcare sector in Moldova.
2. Conduct interviews with the identified stakeholders to gather information on the current IT systems used in the healthcare sector in Moldova.
3. Carry out a literature review to research the most appropriate technologies (frontend, backend and database) and project management methodologies for developing a modernized EHR system.
4. Explore at least 2 existing EHR systems and identify their strengths and weaknesses.
5. Design and develop a working web or mobile app for an EHR system based on the requirements gathered and the feasibility of the chosen solution for the Moldovan healthcare sector.
6. Offer the client the prototype to be used as a proof of concept to secure funding for a full-scale implementation of the new system in Moldova.

Chapter 2

Research and Requirements

To gain a more complete understanding of the current situation in Moldova, the existing problems and possible needs of the people involved, it is important to start with an analysis to identify the possible key stakeholders for this project. As previously mentioned in the literature review, a diverse group of stakeholders is essential to ensure that the currrent situation is reviewed from multiple perspectives.

Afterwards, the next step is to utilise the chosen stakeholders to gather as much information as possible from various perspective to ensure that the project is aligned with the needs of both patients and healthcare professionals in Moldova and solves the existing problems.

2.1 Stakeholder Analysis

The student has identified the following possible stakeholders for the project:

- Doctors and other medical staff working in hospitals
- Department head in a hospital
- IT staff members in hospitals
- Staff members at CNAM (National Health Insurance Company)
- Staff members at the Ministry of Health
- Patients

These stakeholders have been identified so that they can provide a wider picture on the needs and requirements of the project, and to ensure that the project is aligned with the

expectations of workers within the healthcare industry in Moldova from multiple perspectives.

The stakeholders have also been placed in the stakeholder influence-interest grid (which will be discussed in section 3.2.2) to help the student understand the level of influence and interest that each stakeholder has in the project:

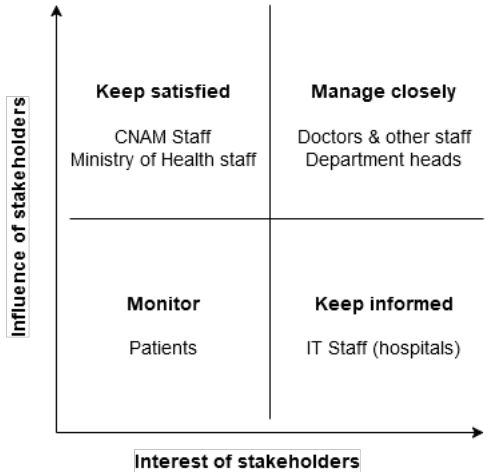


Figure 2.1: Chosen stakeholders in the stakeholder influence-interest grid

2.1.1 Current situation analysis

Following the analysis, the student has conducted several exploratory interviews with the chosen stakeholders to gather insights into current issues with the IT systems used in Moldova's healthcare sector. The student was able to reach out to every stakeholder, except for staff members at CNAM and the Ministry of Health.

After the conclusion of the interviews, three main issues and potential solutions have been identified:

1. Current EHR system is outdated and only accessible via Internet Explorer or legacy version of Microsoft Edge. A potential solution is to develop a new, modernized version of the existing system (retaining the core functionality) that is accessible via modern browsers, is more user friendly and has future upgrade capabilities.

2. Lack of interoperability between medical institutions due to a lack of a nationally-wide integrated system. A potential solution is to create a new system that acts as a patient history archive, where patients can upload their own medical records (such as lab tests, previous medical history, etc) and share them with any medical practitioner,

regardless of the institution they work at.

3. Lack of digitalizion for some systems that still rely on paper-based records or very rudimentary data structures, such as the national transplant registry. A potential solution is digitalized of said system, as is in the case of the transplant registry, that can be accessed by any medical practitioner in Moldova.

Analysing the current issues and potential solutions, the student has determined that the solutions for issues #1 and #3 are too complex, as they require a complete overhaul and integration with existing systems. As such, the student has decided to focus on issue #2, as it is the most feasible and an MVP can be develop within the timeframe of the project.

Consequently, the stakeholder list has been updated to reflect the changed focus of the project:

- Doctors working in hospitals and clinics
- Staff members at the Ministry of Health
- Staff members at CNAM
- Patients that are using both public and private healthcare institutions
- Other medical staff members (nurses, pharmacists, etc)
- Staff members at CNPDCP (National Center for Personal Data Protection)

At the same time, the stakeholder influence-interest grid has been updated to reflect the changes in the project focus:

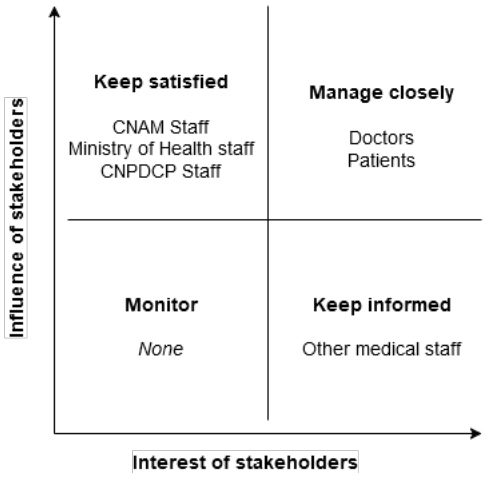


Figure 2.2: Updated stakeholders in the stakeholder influence-interest grid

2.2 Requirements

After the new stakeholders were identified, additional interviews were conducted to focus on the requirements for the chosen solution and enough information was gathered from the other stakeholders to identify the main requirements for the project. The student was unable to reach out to the staff members at CNAM, the Ministry of Health and CNDCP — instead legislation and regulations on their websites were reviewed (**CNAM**; **CNPDCP**; **ministry**).

All of the gathered requirements can be found in the appendix (section A), but the most important requirements have been summarized in the table below:

ID	Category	Requirement
1	Non-functional	The system must be accessible on all modern desktop and mobile-based browsers.
2	Non-functional	The system must store the data in a secure manner, ensuring that only the patient and the doctor can access the data.
3	Document upload	The system must allow patients to upload their own medical records in a variety of formats (PDF, DOC, etc).
4	Personal cabinet	The system must display the patient's history in a chronological order in the form of a timeline.
5	Personal cabinet	The patient personal cabinet must provide an overview of the patient's history through 3 main sections: personal information, lab tests, and doctor consultations.
6	Personal Cabinet	The system must allow patients to add their own personal information, such as name, date of birth, or address.
7	Personal Cabinet	The system must allow patients to add their own allergies and vaccinations.
8	Shareable link	The system must allow the patient to generate a shareable link to provide access to their medical records.
9	Doctor view	When shared with the doctor, the system must allow the doctor to only view the patient's history, not edit it.
10	Patient medication	The system must allow patients to enter their current medication including details such as the name of the drug, dosage, frequency and start/end date.

Table 2.1: Summary of the Most Important Requirements

Chapter 3

Literature Review

This chapter will provide a review of the existing literature, which will be used guide the student in their planning and development efforts of the project.

As such, it will be covering the following areas:

- Software development methodologies
- Requirement and stakeholder management
- Tech stack (backend and frontend)
- Large Language Models (LLMs)
- PHR Systems

3.1 Software development methodologies

3.1.1 Software Development Life Cycle

The Software Development Life Cycle (SDLC) is a process used to guide the development of software applications or systems (**sdlc1**). The SDLC consists of multiple phases, each with its own set of activities and deliverables.‘ ’**sdlc2** outline the phases of the SDLC as following:

1. **Requirement gathering and analysis phase** — the requirements are gathered saved in a document. Based on the requirements gathered, a development plan is created.

2. **Design phase** — requirements are written in a more technical manner and system desings are created.
3. **Implementation phase** — Actual development of the software occurs in this phase. Additionally, some smaller unit tests may be done during this phase.
4. **Testing phase** — may involve multiple types of testing, such as unit testing, integration testing, and system testing.‘ ’**testing** describes the different types of tests:
    - Unit testing — testing individual units or components of the software.
    - Integration testing — performed on two or more units combined together, focusing on the interfaces between these components.
    - System testing — focuses on the ‘end-to-end quality of the entire system’, testing it as a whole.
5. **Maintenance phase** — involves the deployment and maintenance of the software. Additionally, this phase may include end-user acceptance testing, to ensure that it meets their needs (**testing**).

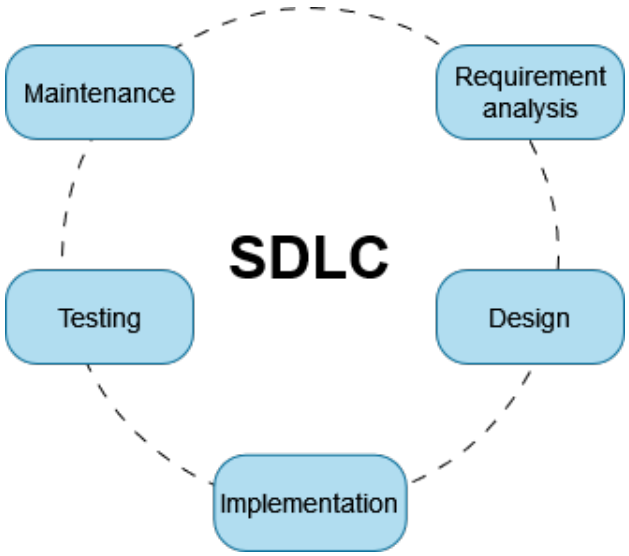


Figure 3.1: Software Development Life Cycle

3.1.2 SDLC Models

The literature describes several SDLC models that have been used in the development of software applications.‘ ’**sdlc1**; **sdlc2** highlight the most common ones: Waterfall model, V model, Spiral model, Interative model, and Agile model.

Waterfall Model

The Waterfall Model is probably the most well-known SDLC model. It is a linear model, where the development process is divided into distinct, sequential phases (which can be seen in figure 3.2).

The Waterfall Model's strengths lie in its simplicty of use, ease of understanding and a clear, structured approach (**waterfall**). An additional stregth that the authors note is its extensive documentation and planning, emphasizing quality and adherence to regulations.

On the other hand, one of Waterfall's main weaknesses is its lack of flexibility in regards to change (**waterfallno**). Thus, this model is not suitable for projects where the requirements are not well understood or are likely to change. Additionally, the project deliverable is not avaialable until the end of the project, any changed or feedback cannot be done during its development (**waterfallno**).

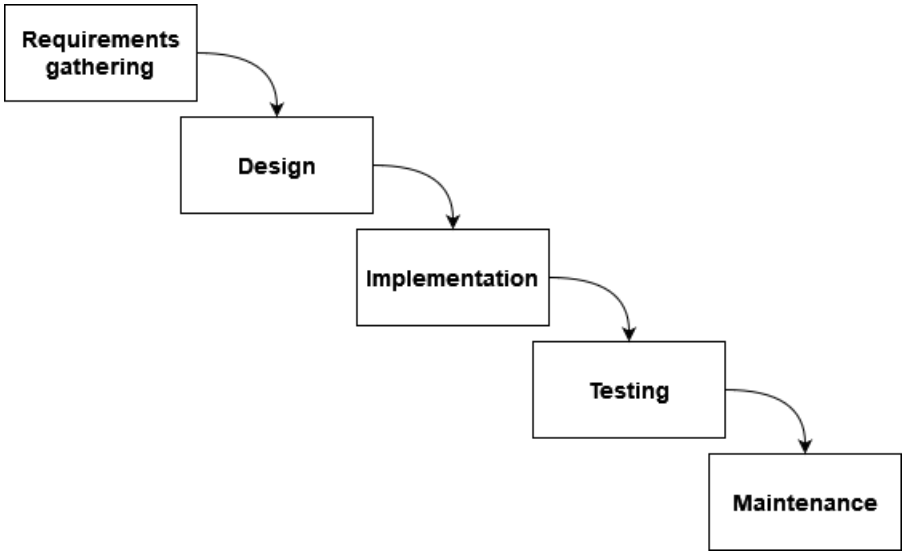


Figure 3.2: Waterfall model

Agile Model

Another well-known model is Agile. It has multiple frameworks, with Scrum and Kanban being the most popular ones. Scrum is a framework where the project is divided into sprints, each lasting between 2–4 weeks, that aim on delivering value to the customer through incremental software features (**scrumban**; **agile**). Kanban focuses on visualizing the project workflow by using a visual board with columns, cards and swimlanes. It uses column limits and a pull system to make the flow of work through the system more efficient (**agile**).

Agile has some drawbacks — its lack of documentation and formal planning, especially in the early stages of the project, may not be suitable for large scale projects (**agile**; **sdlc2**). Similarly, lack of knowledge on how to use the frameworks may be a barrier for some teams (**waterfallno**; **sdlc2**).

Nowadays, the combined use of Scrum and Kanban is becoming quite popular, with many teams employing both frameworks in their projects, allowing them to adopt the appropriate practices and adapt them accordingly based on their needs (**scrumban**).

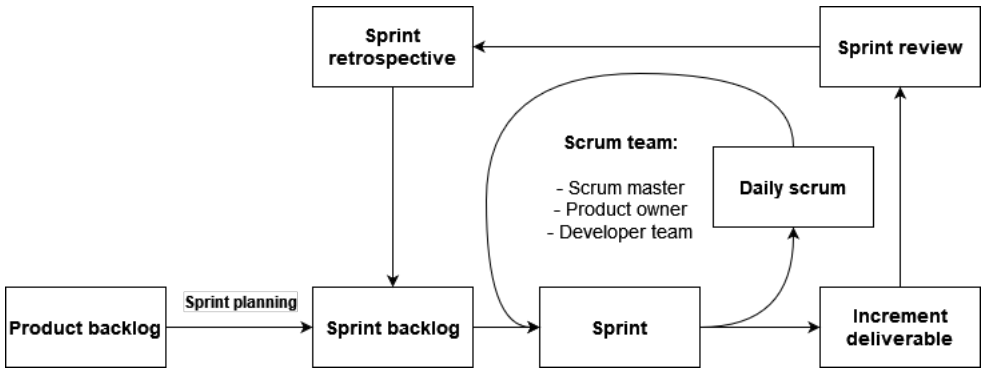


Figure 3.3: Scrum framework

3.1.3 A hybrid approach

A hybrid approach has also been emerging in software development projects. Various surveys report the most common combinations are Scrum, Iterative Development, Kanban, Waterfall and DevOps, with hybrid Waterfall and Scrum being the most popular one (**hybrid1**; **hybrid2**). In this approach, the development part is done in an Agile way, with the rest of the project using Waterfall as a backbone (**hybrid2**).

hybrid1 note that projects using either Agile, traditional or hybrid approach show similar levels of success in terms of budget, time and quality. However, the authors have found

that agile and hybrid approaches perform much better on customer satisfaction than the traditional ones.

3.2 Requirements gathering

Requirements gathering is the first step in any software development process. As described by **reqanalysis2**, a requirement is a 'necessary attribute in a system... that identifies a capability, characteristic, or quality factor of a system in order for it to have value and utility to a user'. Multiple studies mention how proper requirement gathering plays a pivotal role in the project success, with many project failures attributed to poor requirements gathering (**reqanalysis1**; **reqanalysis3**; **reqanalysis5**).

3.2.1 Requirement types

Requirements can be classified into 2 categories: functional and non-functional. Functional requirements describe the system's behavior, while non-functional requirements describe the system's quality attributes, such as performance, security, reliability, etc.' '(**requirements**).

When writing the requirements in a document, it is important to ensure clarity and conciseness to avoid ambiguity. Based on the recommendations of **requirements** and **requirements2**, the following principles and practices can be followed:

- Write in a simple and consistent language.
- Avoid technical jargon, vague terms, and combining multiple requirements in a single statement.
- Ensure that requirements are necessary, appropriate, complete, feasible, and verifiable.
- Include attributes for each requirement, such as identification, owner, priority, risk, rationale, difficulty, and type (functional/non-functional).

Prioritizing requirements is another crucial task, especially in projects with numerous requirements. Some methods mentioned by **moscow** include using a low to high priority, assigning a numerical value within a specific range or MoSCoW, which classifies requirements into four categories:

- Must have — must be implemented in the software before being released
- Should have — important but not necessary for the software to be released
- Could have — desirable but not necessary for the software to be released
- Won't have — requirements that are not included in the current release

Requirements in Agile

In Agile projects, requirements are written in the form of User Stories, which are simple descriptions of a feature desired by the customer, using a specific format: 'As a [user], I want to [action] so that [benefit]' (**requirements**). Components of a User Story include a title, acceptance criteria, priority, story points and description. Epics are requirements that cannot be completed in a single sprint and can be broken down into user stories. Epics and User Stories are part of the Product and Sprint backlogs, which contain the requirements for the whole project and the current sprint, respectively.

3.2.2 Stakeholders

Stakeholders are the individuals who have some interest in the success of the system or project, thus it is important to identify all possible stakeholders in the early stages of the project to avoid missing important requirements or constraints (**requirements**).

Stakeholder analysis can help understand their position within the project. One way of doing it is by using a stakeholder matrix, such as the Influence/Interest grid (see figure 3.4), which classifies stakeholders based on their influence and interest in the project (**stakeholders**; **stakeholders2**).

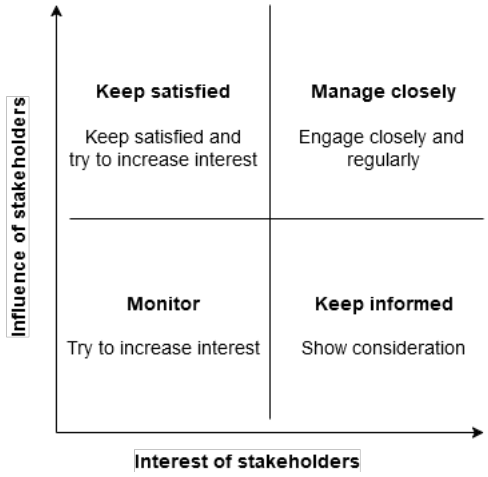


Figure 3.4: Stakeholder Influence/Interest matrix

3.2.3 Requirement gathering techniques

Multiple studies mention the most popular requirement gathering techniques are interviews, workshops, prototyping, modelling, brainstorming, storyboards and observing users

(**reqanalysis1**; **reqanalysis2**; **reqanalysis3**; **reqanalysis4**). In one of them, individuals with multiple years of experience in requirement gathering were interviewed, and the authors found that the most used techniques were collaborative meetings, interviews, ethnography and modelling (**reqanalysis1**).

Multiple research papers recognise interviews as the most common technique for requirement gathering (**interviews5**; **interviews1**; **interviews2**). Some studies have looked at best practices and common mistakes when conducting requirement gathering interviews. The recommended practices, based on **interviews4**; **interviews3**, and the common mistakes, from **interviews1**; **interviews2**, are summarized in Table 3.1 below.

Common Mistakes	**Recommended Practices**
Wrong opening: failing to understand the context before discussing the problem.	State goals at the beginning and allow customer input at the end.‘ ’
Not leveraging ambiguity to reveal knowledge gaps.	Avoid ambiguity by asking clarifying questions.‘ ’
Lack of planning: unstructured sequence of questions.	Plan interviews with a structured sequence of questions.‘ ’
Failing to build rapport with the customer.	Building rapport through small talk or personal questions in the beginning.‘ ’
Implicit goals: failing to ask or clarify stakeholder goals.	Verify alignment and current interpretation with the customer's vision.‘ ’
Question omission: not asking about business processes or doing follow-up questions.	Be flexible by probing into relevant topics.‘ ’
Weak communication: too much technical jargon usage or not listening to the customer.	Use projective techniques like scenarios to encourage deep thinking.‘ ’
Poor question formulation: vague, technical, irrelevant, or too long.	Break down questions or responses into smaller parts and use story telling.‘ ’
Wrong closing sequence: skipping interview summaries or feedback.	Leaving time at the end for the stakeholder to offer any feedback or thoughts.‘ ’

Table 3.1: Comparison of Common Mistakes and Recommended Practices

3.3 Tech stack

3.3.1 Database

There are several types of databases, such as: relational (SQL), NoSQL databases, graph databases or object-oriented databases (**databases2**). The choice of database depends on the project or organisation requirements, such as the amount of data, the complexity of the data, the need for scalability, etc.

Relational databases

Relational databases store structured data in tables, linked through keys to create relationships between entries (**databases**). They use SQL (Structured Query Language) to create queries and schemas to help manage data efficiently. **databases** highlight that relational databases are used, thanks to their high data integrity, for industries like finance and healthcare. Relational databases are widely used, making it easier to find support and resources. However, the rigid schema limits adaptability to rapid data changes or usage of unstructured data. Examples include MySQL, PostgreSQL, Oracle, and Microsoft SQL Server.

NoSQL databases

NoSQL databases manage unstructured or semi-structured data without rigid schemas or relationships (**databases**). As the authors describe, NoSQL databases, such as key-value, document, column-family, and graph databases, excel in flexibility and scalability. NoSQL databases often prioritize performance over strict consistency, making them suitable for large datasets of unstructured data but less ideal for complex transactions. NoSQL systems lack SQL's mature standardization and support. Examples include MongoDB, Cassandra, Couchbase, and Redis.

3.3.2 Backend framework

Choosing the right backend framework is crucial — it is responsible for handling the business logic of the application, such as processing requests, interacting with the database, and returning responses to the client. A good framework also comes with the added benefit of included features such as security and authentication, database support, a big community and documentation.

Based on recent a recent survey by **statista-webframeworks**, the most popular backend frameworks are Express, Flask, Spring Boot, Django and Laravel. This data is also supported by the Stack Overflow Developer Survey 2024, which lists the most popular programming

languages as JavaScript (Express), Python (Flask, Django), Java (Spring Boot) and PHP (Laravel) (**stackoverflow**).

Due to their high popularity among developers, the above frameworks will be compared. Using the information from **spring**; **express**; **django**; **fastapi**, table 3.2 will compare the frameworks based on the following criteria: programming language used, learning curve, community support, security features, database features, and project size suitability.

Framework	**Django**	**FastAPI**	**Spring Boot**	**Express**
Language	Python	Python	Java	JavaScript
Learning curve	Medium	Low	High	Low
Community Support	High	High	High	High
Security features	High	High	High	Medium
Database features	Medium	Low	High	Medium
Project size suitability	Small to medium	Small to medium	Medium to large	Small to medium

Table 3.2: Comparison of backend frameworks

3.3.3 Frontend framework

Similar to the backend frameworks, choosing a suitable frontend framework is equally important — it is responsible for the user interface of the application, such as displaying data, handling user interactions, and making requests to the backend.

Based on the same survey by **statista-webframeworks**, the most popular frontend frameworks are React, Angular, Vue.js, and Svelte. This data is also supported by the Stack Overflow Developer Survey 2024, where those frameworks rank among the highest for desirability and admirability among developers (**stackoverflow**).

Table 3.3 will use the information from **react**; **angular**; **vue**; **svelte** to compare the frameworks based on the following criteria: learning curve, community and documentation, ecosystem and tooling support, performance, state management, and project size suitability.

Framework	**React**	**Angular**	**Vue.js**	**Svelte**
Learning curve	Low	High	Medium	Low
Community and documentation	High	High	High	Medium
Ecosystem and tooling support	High	High	Medium	Low
Performance	High	Medium	Medium	High
State management	High	High	Medium	Low
Project size suitability	Small to large	Medium to large	Small to medium	Small to medium

Table 3.3: Comparison of frontend frameworks

3.4 Large Language Models (LLMs)

Large language models (LLMs) are artificial intelligence systems that are used for natural language processing (NLP) tasks such as text generation, translation, summarization and question answering (**llm2**; **llm_healthcare**). Additionally, LLMs have been found to have emergent capabilities, like reasoning, planning, decision-making and in-contenxt learning (**llm2**). These extraordinary capabilities are achieved through extensive training on large corpus of text data, high parameter count (in the billions) and usage of techniques such as fine-tuning or prompt engineering to improve their performance (**llm2**; **llm_healthcare**).

LLMs are built on the transformer architecture, which allows them to understand text by learning and remembering the relationships between words (**llm**). These models are first pre-trained on large amounts of unlabeled data using, allowing them to excel in a wide variety of tasks (**foundation**; **llm2**). These pre-trained models, known as foundation models such as the GPT or Llama families, can then be fine-tuned for specific tasks, improving their performance and accuracy even further (**gpt4**; **llama3**; **llm2**).

3.4.1 Multimodal LLMs

One advancement in the field of LLMs has been the addition of multimodal abilities, allowing them to process, understand and generate text and images, audio or videos (**mllm**; **mllm2**). These new multimodal LLMs (MLLMs) utilise existing reasoning capabilities of LLMs, which are connected to an encoder that can processes images, audio or videos and a generator that helps with generating multimodal outputs (**mllm**). This integration of new modalities allows MLLMs to become versatile tools, expanding their possible use cases and bridging the gap between human and machine interaction (**llm_healthcare**).

API model providers

Running and hosting LLMs locally can be a challenge, considering their big model sizes and high computational requirements. As such, many platforms offer APIs that allow users to access LLMs through the cloud. A list of some free API providers, the models offered and their rate limits has been compiled by **llmapi** and some are listed in the table below.

Provider	**Model name(s)**	**Free tier limits**
Groq	Llama 3.2 11B Vision	7,000 requests/day, 7,000 tokens/minute
	Llama 3.2 90B Vision	3,500 requests/day, 7,000 tokens/minute
OpenRouter	Llama 3.2 11B Vision Instruct Llama 3.2 90B Vision Instruct Gemini 2.0 Flash Experimental	20 requests/minute, 200 requests/day
Google AI Studio	Gemini 2.0 Flash	4,000,000 tokens/minute, 10 requests/minute
	Gemini 1.5 Flash	1,000,000 tokens/minute, 1,500 requests/day, 15 requests/minute
	Gemini 1.5 Pro	32,000 tokens/minute, 50 requests/day, 2 requests/minute
GitHub Models	OpenAI GPT-4o OpenAI GPT-4o mini	Rate limits dependent on Copilot subscription tier
Cloudflare Workers AI	Llama 3.2 11B Vision Instruct	10,000 tokens/day
glhf.chat	Any model on Hugging Face that fits on an A100 node (640GB VRAM)	480 requests/8 hours

Table 3.4: API providers for LLMs

3.4.2 LLMs in healthcare

One application of MLLMs is in healthcare, where the growing volume and complexity of data creates the need for more advanced tools to process and analyze it. LLMs and MLLMs have found use in various healthcare applications, either by using existing models or by developing new, specialized medical models such as Med-PaLm2, BioMistral or Med-Gemini (**biomistral**; **medgemini**; **medpalm2**). Some of these applications include:

- **Improving medical diagnosis:** By combining patient records, existing symptoms, and medical history, LLMs can use their reasoning capabilities and memory to assist in diagnosing or preventing health conditions (**llm_healthcare**; **llm_healthcare3**; **llm_healthcare4**).
- **Medical Imaging and Multimodal Capabilities:** In diagnostic imaging, multimodal models can assess both text and images (such as X-rays and MRIs) to offer comprehensive analysis. Clinicians can input medical images and contextual information, making MLLMs valuable assistants in the real-time diagnostic processes (**llm_healthcare3**).
- **Virtual Health Assistants:** LLMs can also be deployed as virtual assistants, helping patients with personalised care and general health inquiries (**llm_healthcare**; **llm_healthcare3**). Patients in areas with limited healthcare access can benefit from these assistants, which also supports healthcare providers by lightening their workloads.
- **Administrative Support:** LLMs can assist in generating Electronic Health Records (EHRs), allowing healthcare providers to focus more on patient interaction (**llm_healthcare4**). Additionally, they can also help translate complex medical terms into more simple language, assist in administrative tasks, and more.

3.4.3 Prompt Engineering

The success of LLMs depends not only on the model itself — but also on how it's effectively used by the users, using techniques like prompt engineering, which involves the constant designing and refining of prompts to guide the output of LLMs (**promptmed**; **prompt2**). Prompts represent instructions given to the model to guide its output, such as providing context, examples, or constraints to the model (**prompt**; **prompt1**; **prompt2**).

There are multiple techniques for prompt engineering, ranging from simple to more advanced. The tables and subsections below outlines some of the most common techniques.

Zero-Shot Prompting

Zero-shot prompting are techniques where the LLM is given a prompt without any examples, allowing it to generate an output based on the prompt alone (**prompt1**).

Technique	Description	Source
Role prompting	Assigning a specific role to the LLM in the prompt. The authors note that generally it provides mixed results but may be useful in certain settings.	**role1**
Style prompting	Specifying the desired style or tone in the prompt.	**style**
Emotion prompting	Incorporating phrases of psychological relevance to humans in the prompt.	**emotion**
Re-reading	Adding the phrase 'Read the question again' to the prompt in addition to repeating the question.	**rereading**
Self-Ask	Prompting the LLM to decide if it needs to ask any follow-up questions for a given prompt.	**selfask**

Table 3.5: Zero-Shot Prompt Techniques

Few-Shot Prompting

Few-shot prompting are techniques where the LLM learns how to complete a task based on a few examples given in the input prompt (**prompt1**).

Technique	Description	Source
Self-Generated In-Context Learning	Using the LLM to automatically generate examples when training/example data is not avaialable.	**self-generating**
Prompt Mining	Scanning the training data to discover common formats that can be used as prompt templates.	**mining**

Table 3.6: Few-Shot Prompt Techniques

Thought Generation Prompting

Thought generation are techniques where the prompt encourages the LLM to explain its reasoning process while solving a given problem (**prompt1**).

Technique	Description	Source
Chain-of-Thought (CoT)	Adding a phrase like 'Let's think step by step' at the end of the prompt to encourage the LLM to describe its thought process before offering a final answer.	**cot**
Contrastive CoT	Using CoT and also adding both incorrect and correct examples in order to provide the LLM a more diverse example set.	**contrastive-cot**
Auto-CoT	Using CoT with another LLM to automatically generate CoT examples that can be used to create few-shot CoT prompts for other LLMs.	**auto-cot**
Least-to-Most	Starts with asking the model to break down a problem into sub-problems without solving them. Afterwards, it solves them one by one, appending the result each time, until it arrives at the answer.	**least-most**
Tree-of-Thought (ToT)	Starts with an initial problem and then generates multiple possible steps by using CoT. Then, it evaluates each step, decides which one to take and creates more thoughts until it reaches an answer.	**treeofthought**

Table 3.7: Thought Generation Prompt Techniques

Multimodal and Multilingual Prompting

Multimodal and multilingual prompting are techniques which aim to improve an LLM's performance by leveraging multiple modalities or languages in the prompt (**prompt1**).

Technique	Description	Source
Translate-first prompting	Translating the input prompt into English to leverage LLMs strengths in dealing with English inputs, compared to non-English inputs.	**translate-first**
English prompting	Writing the prompt in English may usually be more effective than using the task language for multilingual tasks. The authors argue it may because of the predominance of the English language in the pretraining data.	**english-prompting**
JSON/XML output formatting	Asking the LLM to format the response in a JSON or XML format and providing the expected schema has been found to improve the accuracy of LLM outputs	**jsonllm**
Multimodal CoT	Similar to the textual CoT, this technique encourages the model to solve a given image-based problem step by step by step.	**multimodal-cot**
Image-as-Text	Generating or writing a textual description of an image that can then be included in a text-based prompt.	**images-as-text**
Chain-of-Images (CoI)	Using the CoT process to generate images as part of its thought process to reason visually.	**coi**

Table 3.8: Multimodal and Multilingual Techniques

Agents

Agents are techniques which encourage LLMs to use external tools or resources to complete a task (**prompt1**).

Technique	Description	Source
Program-aided Language Model (PAL)	Using the LLM to translate a problem into code, which can then be sent to an interpreter to generate an answer.	**pal**
ReAct	Firstly, the model generates a thought based on the input. Then, the model takes an action and observes the result. This process is repeated until the model arrives at an answer.	**react-llm**
Retrieval Augmented Generation (RAG)	This technique involves retrieval of information from an external source and inserting it into the prompt.	**rag**

Table 3.9: LLM Agent Techniques

3.4.4 Challenges and concerns of using LLMs

While LLMs bring many benefits when applied to the healthcare domain, it is important to note that their use does come with several challenges:

- **Data Privacy and Compliance:** Patient data is highly sensitive, thus ensuring compliance with standards is essential, requiring data anonymization and secure handling practices to ensure patient data safety.(**llm_healthcare**; **llm_healthcare2**; **llm_healthcare4**).
- **Transparency and Explainability:** LLMs are often described as 'black boxes', making it difficult to explain their decision-making processes. In healthcare, transparency is crucial — lack of it poses risks and raises ethical concerns about relying on such systems in high-stakes scenarios (**llm_healthcare**; **llm_healthcare2**; **llm_healthcare4**)..
- **Bias and Fairness:** LLMs trained on vast datasets can inherit biases in the data, leading to skewed or unfair outcomes (**llm_healthcare2**).
- **Hallucinations:** LLMs sometimes generate false or fabricated outputs, also known as 'hallucinations'. In healthcare, this poses significant risks, as incorrect or misleading information could jeopardize patient safety and trust in the technology (**llm_healthcare4**; **llm_healthcare**).
- **Accountability:** Responsibility must be clearly communicated and understood by all parties involved in the development and use of the model (**llm_healthcare2**). The author recommends the usage of clear guidelines, policies and code of conducts

to ensure that all parties are aware of their obligations.

- **High Costs and Infrastructure Needs:** Training and operating LLMs requires extensive computational resources, which can be a limiting factor for healthcare institutions (**llm_healthcare4**).

3.5 PHR Systems

A Personal Health Record (PHR) is an electronic resource used by patients to manage their own health information (**phrsecurity**; **phrlist**). PHRs are different from Electronic Health Records (EHRs) and Electronic Medical Records (EMRs) which are inter-organisational or internal systems to organise patient health records (**phrdiff**; **phrlist**). Three different types of PHRs are described by **phrsecurity**: stand-alone, which require manual entry to update the records; instituion-specific, which are connected to a specific healthcare institution; and integrated, which can connect to multiple healthcare systems to aggregate data from multiple sources.

Usage of PHRs can bring many benefits to patients, such as: empowering patients to manage their health, improving patient outcomes, decreasing the cost of healthcare and improving the taking of medication (**phrsecurity**).

PHRs contain highly sensitive health information, so it is important to ensure that the data is secure and private. Based on a survey of health information management and medical informatics experts, **phrsecurity** identified 7 dimensions that need to be addressed when developing a PHR system:

1. Confidentiality
2. Availability
3. Integrity
4. Authentication
5. Authorization
6. Non-repudiation
7. Access rights

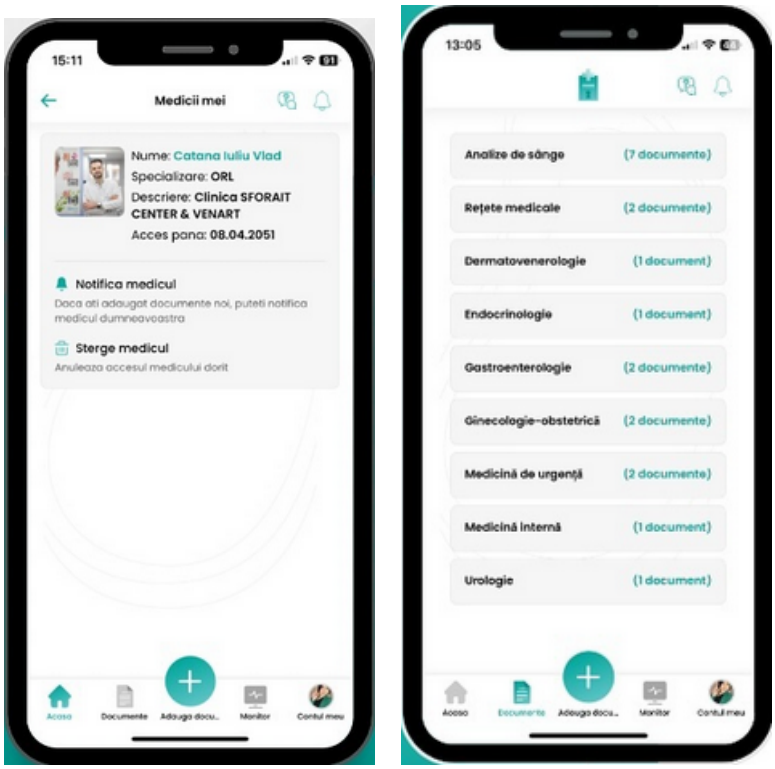
The authors recommend mechanisms to ensure adherence to the above-mentioned dimensions, such as encrypting the data in the database, using backups or defining user access to data and access rights.

3.5.1 Existing Solutions

PHR systems have been implemented nation-wide in many developed countries, such as the NHS App in the UK (**phrlist**). Additionally, there are many private solutions that offer similar features to the one proposed in this project. The next sections will provide a brief overview of 3 existing systems: Medvalet, Andaman7 and Fasten Health.

Medvalet

A mobile app developed in Romania that allows patients to upload their medical history as PDFs or scanned documents (**medvalet**). See Table 3.10 for a summary of its features and limitations and figure 3.5 for a screenshot of the app.



(a) My doctors screen (b) Uploaded documents screen

Figure 3.5: Medvalet screenshots

Key Features/Benefits	Limitations/Drawbacks
• Upload medical history as PDFs or scanned documents. • Categorize documents by type (e.g., prescriptions, lab results). • Graphically track vitals like blood pressure and weight over time. • Patients can input personal details such as name, age, and weight. • Doctors can access patient history directly via the app.	• Requires doctors to create accounts, which may deter use. • Doctors can access patient history without explicit consent, raising privacy concerns. • App acts as document storage, which can be cumbersome to access for lengthy histories. • Lacks data extraction or summarization features from uploaded documents. • Only available as a mobile app, limiting accessibility for desktop-only users.

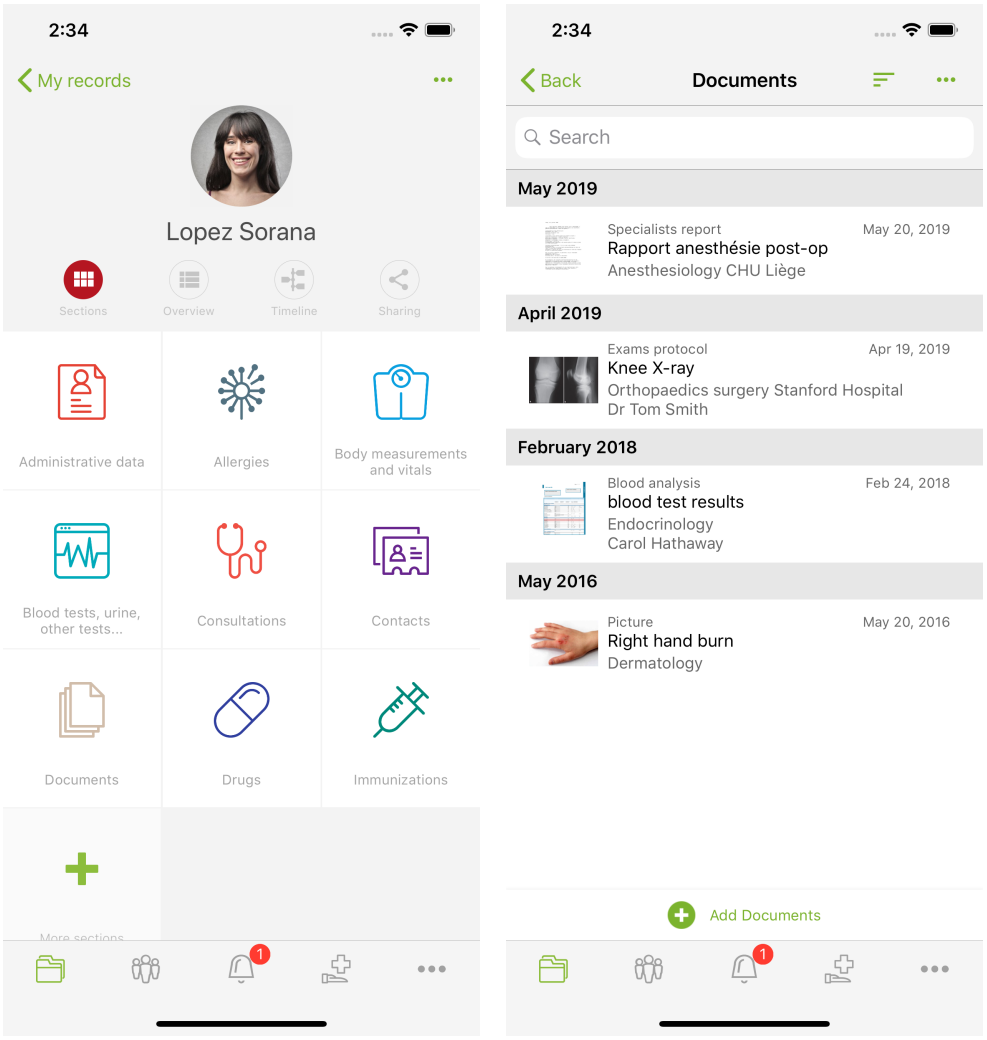
Table 3.10: Medvalet Features and Limitations

Andaman7

A mobile app developed by a Belgian-American eHealth company with the goal to improve doctor-patient communication, compliant with GDPR and HIPAA (**andaman**). See Table 3.11 for a summary of its features and limitations and figure 3.6 for a screenshot of the app.

Key Features/Benefits	Limitations/Drawbacks
• Offers sections for personal information, medical history, allergies, vaccinations, medications, etc. • Automatically collects health data from over 300 hospitals and clinics in the US and Europe. • Supports input from diverse sources like hospitals, labs, smart devices or even manual input. • Stores data locally on patients' devices, ensuring privacy. • Data sharing with QR codes and revokable access. • AI tools for summarization, translation, and simplifying medical jargon.	• Requires patients and doctors to both create accounts. • Does not extract data or values from uploaded documents like lab results. • Limited to mobile platforms, which may limit usability for desktop-only users.

Table 3.11: Andaman7 Features and Limitations



(a) PHR Sections screen

(b) Documents section screen

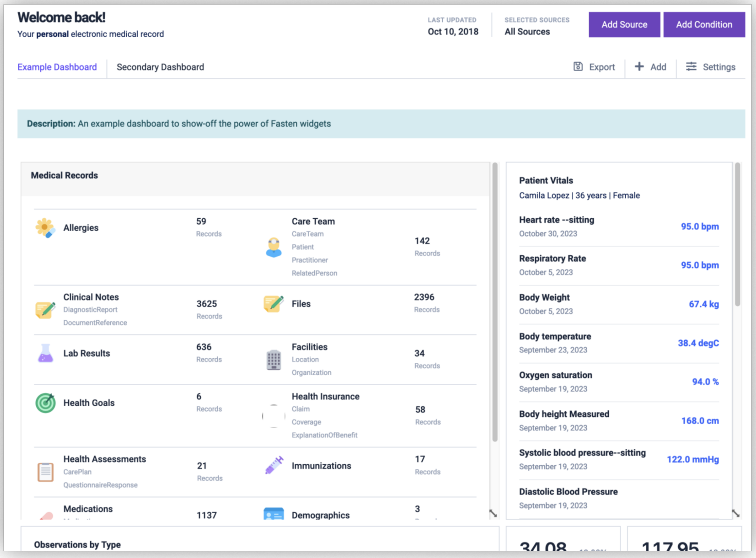
Figure 3.6: Andaman7 screenshots

Fasten Health

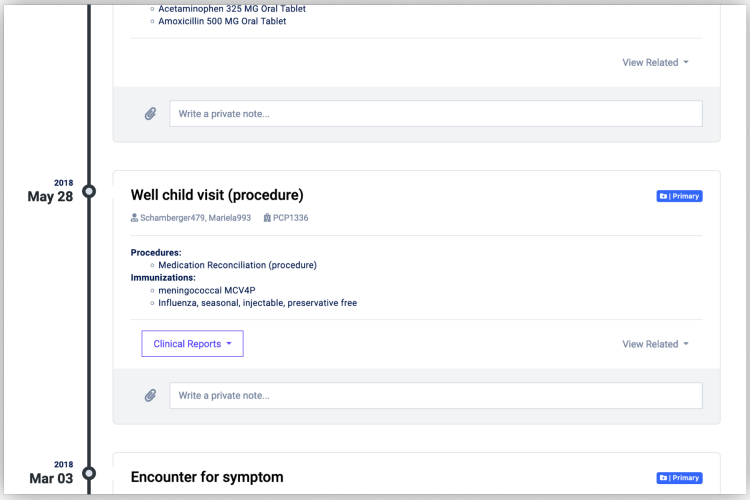
An open-source, self-hosted electronic medical record aggregator with optional paid desktop versions for Windows and Mac (**fasten**). See Table 3.12 for a summary of its features and limitations and figure 3.7 for a screenshot of the app.

Key Features/Benefits	**Limitations/Drawbacks**
• Automatically aggregates records from multiple providers, like hospitals and labs. • Supports self-hosting for complete control over data, stored locally. • Compatible with protocols such as DICOM, FHIR, and OAuth2. • Allows manual entry for allergies, vaccinations, and medications. • Offers multiple dashboards with graphs to visualize health data. • Supports multi-user functionality for families.	• Paid desktop versions may deter users. • Manual data entry limited to new or existing encounters, complicating usage. • Does not support OCR or automatic data extraction from documents. • Lacks data-sharing capabilities with doctors. • Requires technical expertise for self-hosting. • Restricted to healthcare providers in the United States.

Table 3.12: Fasten Health Features and Limitations



(a) Dashboard screen



(b) Visit history screen

Figure 3.7: Fasten Health screenshots

Chapter 4

Project Planning and Design

4.1 UML Diagrams

UML, or Unified Modeling Language, is a standardized modeling language that consists of a set of diagrams used for modeling business processes and documenting software systems, helping better communicating potential designs and architectural decisions (**uml**).

The most common UML diagrams include:

- **Use Case Diagram** — Illustrates the system's intended functionality in terms of actors, use cases, and their relationships, showing how the system delivers value to users. The diagram can also be accompanied by a use case specifications document, which provides a detailed description of each use case.
- **Class Diagram** — Depicts the structure of the system by showing classes, attributes, operations, and static relationships between classes.
- **Sequence Diagram** — Demonstrates how objects interact in a particular, timed sequence scenario, focusing on the messages passed between objects.
- **Activity Diagram** — Represents the workflow of a target use case or business process through a series of activities, emphasizing steps, choices, iterations, and concurrency.

The student has used UML diagrams to present the stakeholders with a visual representation of the system's design and functionalities. The diagrams can be found below, under their respective sections.

4.1.1 Use Case Diagram

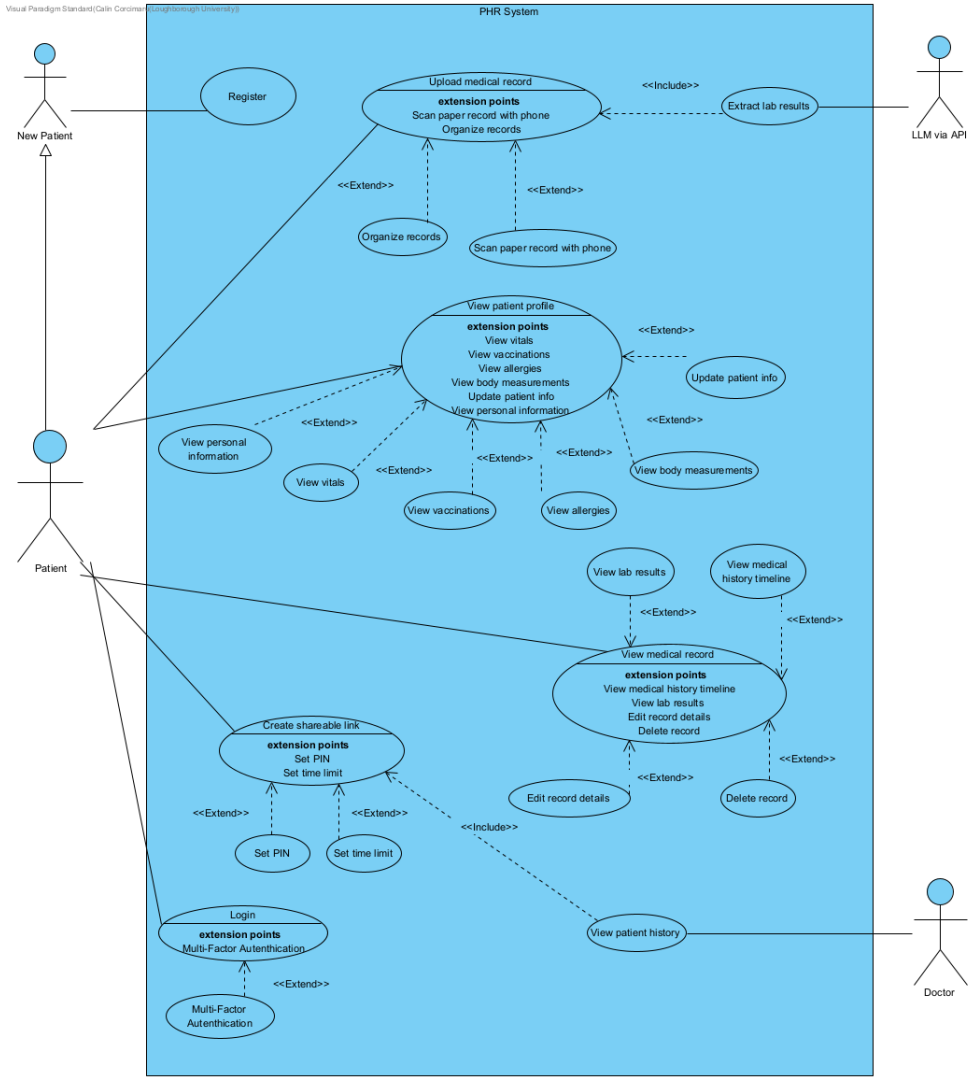


Figure 4.1: UML Use Case Diagram

4.1.2 Use Case Specifications

Login

Description	The Login use case allows the app user to log in to their existing account via their credentials, with an optional use of MFA.
Actors	Patient
Preconditions	Must have an existing account
Steps	1. User enters user and password 2. User clicks on the login button 3. App validates user credentials 4. Patient enters MFA code if enabled 5. App logs user in

Register

Description	The Register use case allows the app user to create a new account.
Actors	New Patient
Preconditions	No existing account with the email used to register
Steps	1. User enters user and password 2. User clicks on the register button 3. App validates user credentials 4. App logs user in 5. App sends verification email 6. User verifies email

Upload Record

Description	The Upload Record use case allows the patient to upload their medical records to the app.
Actors	Patient, LLM
Preconditions	Must be logged in
Steps	1. User selects file upload (or camera scan) 2. User selects the record to upload 3. User clicks on the upload button 4. App validates the record 5. User selects the appropriate record type 6. If record type is lab result, app sends the record to LLM via API for processing into JSON format 7. App adds the record to the database 8. App shows confirmation to user

Share Records

Description	The Share Records use case allows the patient to share their medical records with doctors.
Actors	Patient, Doctor
Preconditions	Must be logged in and have records uploaded
Steps	1. User selects option to create a share link 2. OPTIONAL: User selects the records to share 3. OPTIONAL: User adds a PIN to the share link 4. User selects time limit for the share link 5. App generates the share link 6. App sends the share link to the doctor via email 7. App shows confirmation to user 8. Doctor clicks on the share link 9. Doctor enters the PIN (if required) 10. App validates the PIN 11. App shows the records to the doctor

View Records

Description	The View Records use case allows the patient to view and edit their uploaded medical records.
Actors	Patient
Preconditions	Must be logged in and have records uploaded
Steps	1. App provides a list of records or medical history 2. User selects record to view from list or medical history 3. App retrieves the record from the database 4. App shows the record to the user 5. OPTIONAL: User can view the record in a graphical format if lab result 6. OPTIONAL: User edits the record 7. OPTIONAL: User deletes the record

View Patient Profile

Description	The View Patient Profile use case allows the patient to view and edit their profile information, including health information such as allergies, medications, vaccinations and recorded health data like blood pressure, glucose levels, etc.
Actors	Patient
Preconditions	Must be logged in
Steps	1. User selects the profile section 2. App retrieves the profile information from the database 3. App shows the profile information to the user — vaccinations, allergies, medications, health data 4. OPTIONAL: User edits the profile information 5. OPTIONAL: User adds new health data 6. OPTIONAL: User deletes health data 7. OPTIONAL: User can view health data in a graphical format

4.1.3 Sequence Diagrams

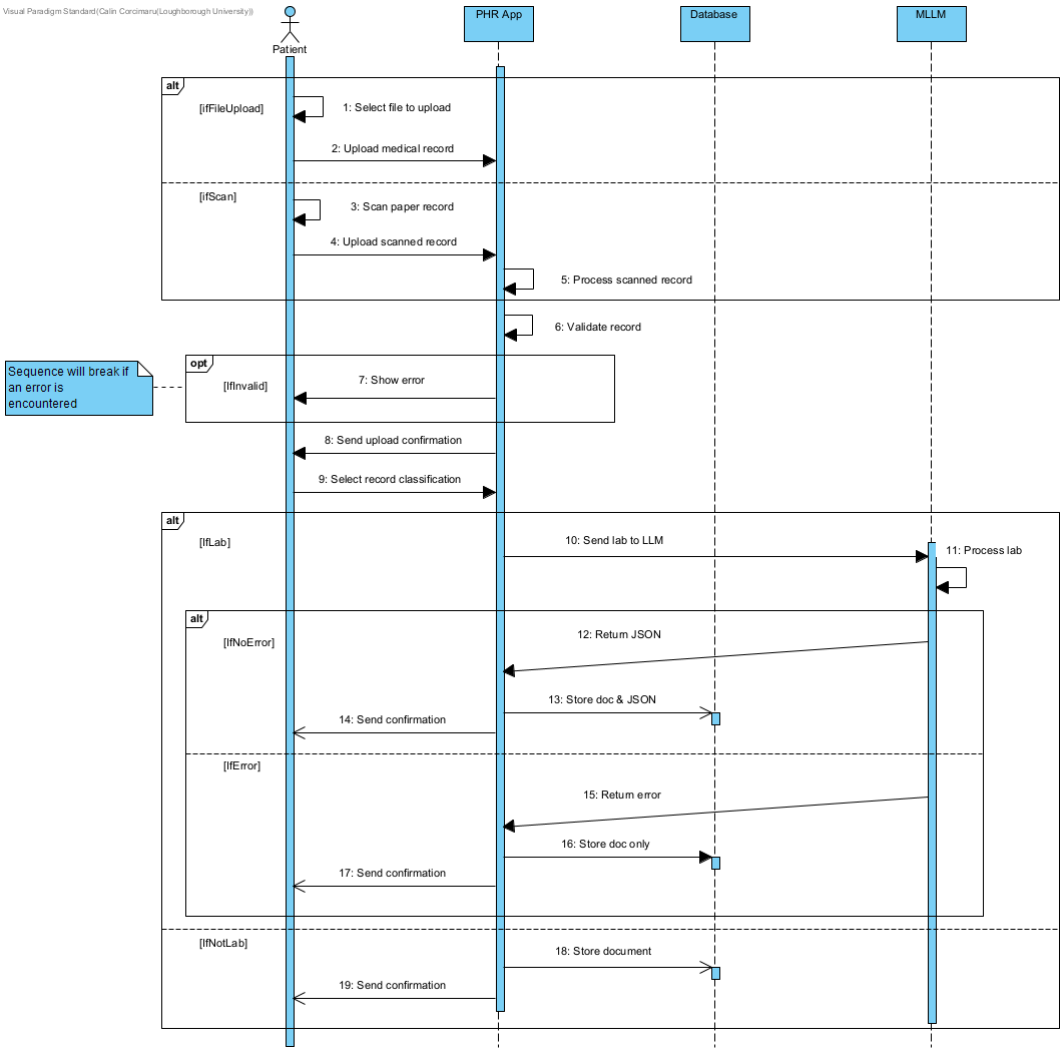


Figure 4.2: UML Sequence Diagram — Upload Record Use Case

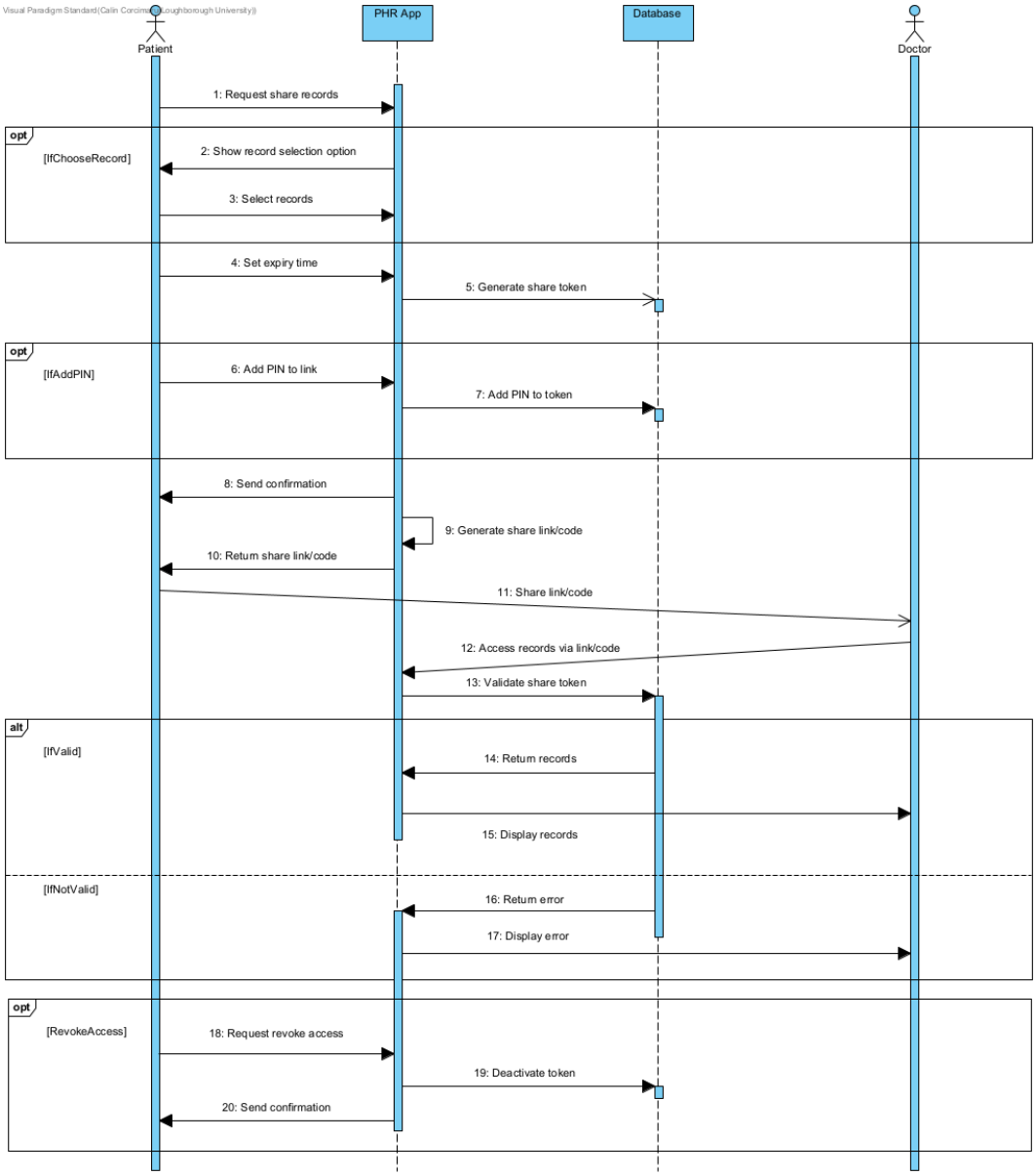


Figure 4.3: UML Sequence Diagram — Share Records Use Case

4.1.4 Activity Diagrams

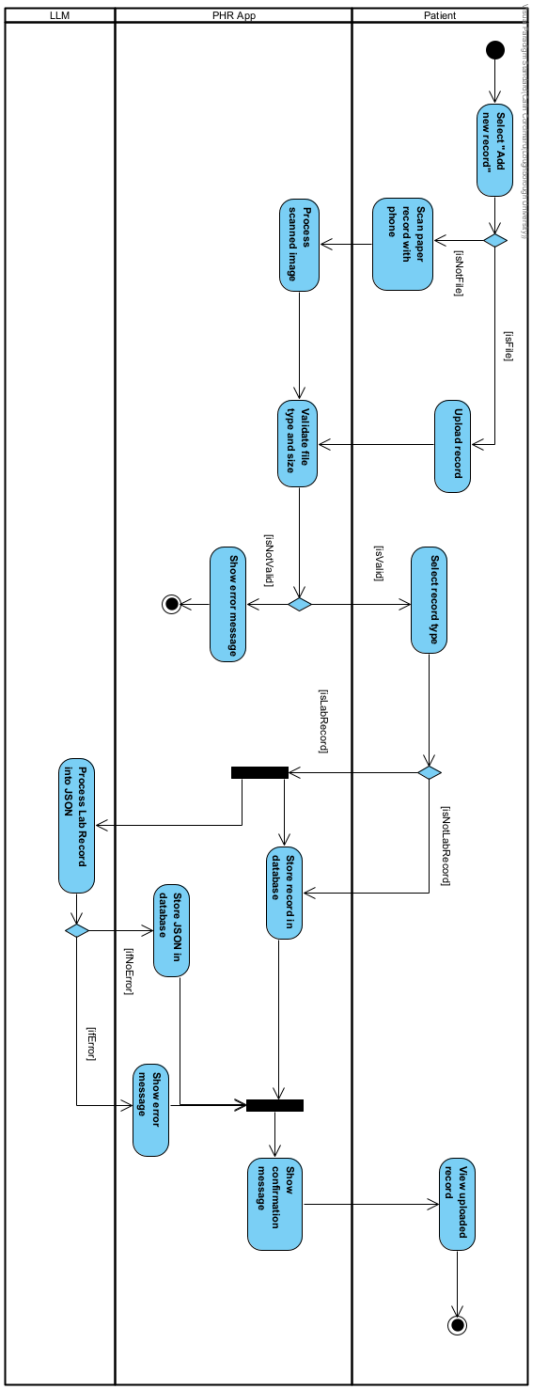


Figure 4.4: UML Activity Diagram - Upload Record Use Case

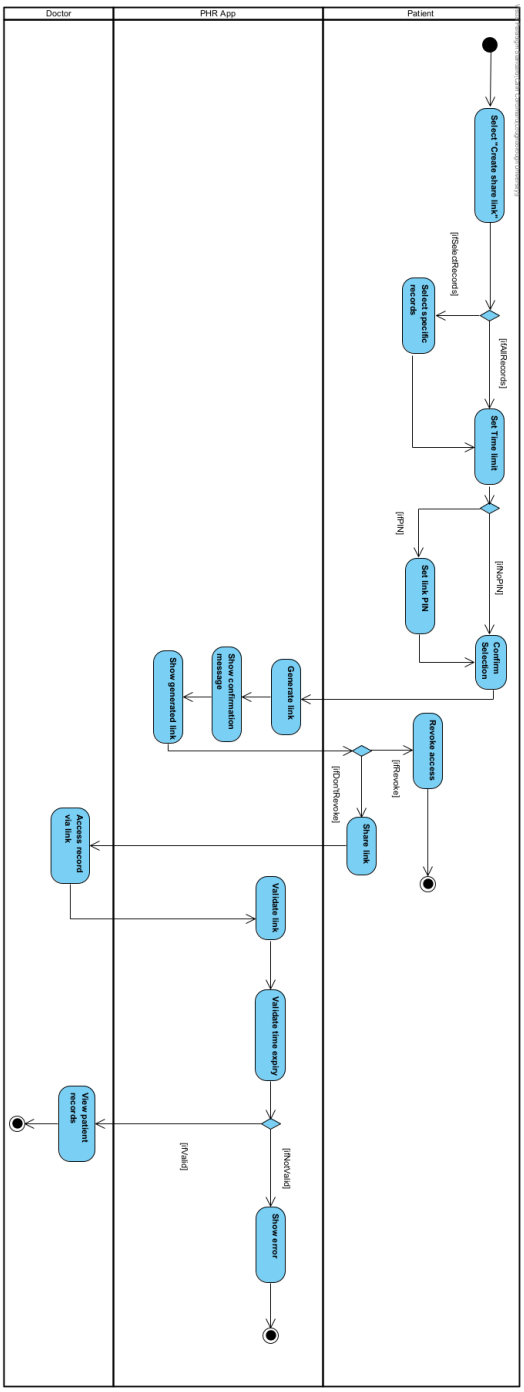


Figure 4.5: UML Activity Diagram - Share Records Use Case

4.2 Database Design

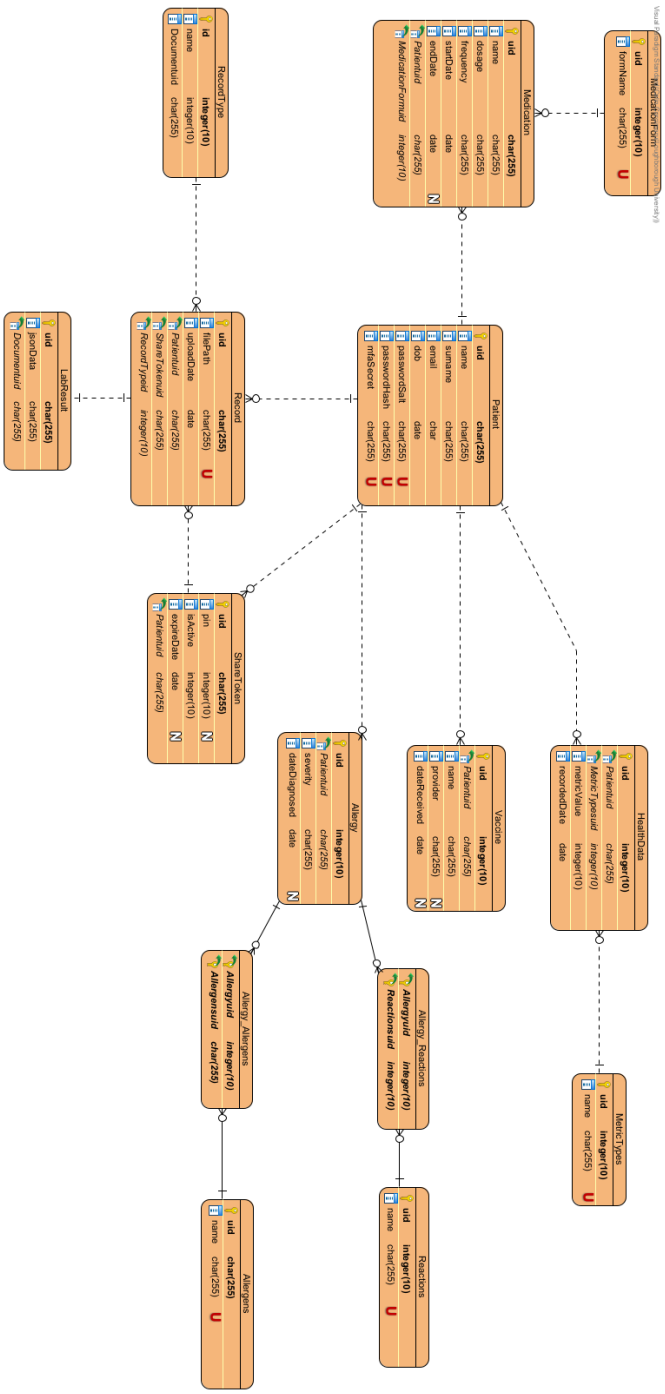


Figure 4.6: Entity Relationship Diagram

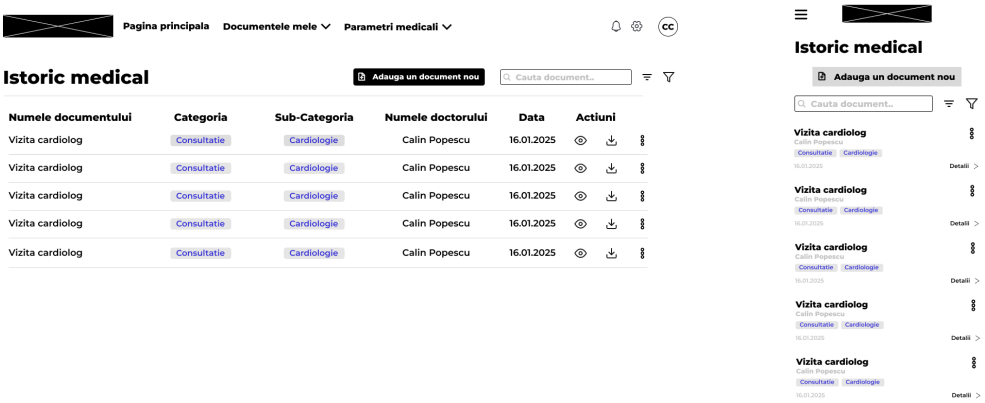
4.3 Wireframes

Wireframes have been used to provide a high-level overview of how the web application would look like and present some of its key functionalities. The wireframes have been created using Figma and then shown to some of the stakeholders to gather feedback. Based on the feedback received, the wireframes have been adjusted and some examples can be seen below in figures 4.7, 4.8 and 4.9 while the full set of wireframes can be found in the appendix B.

(a) Desktop version

(b) Mobile version

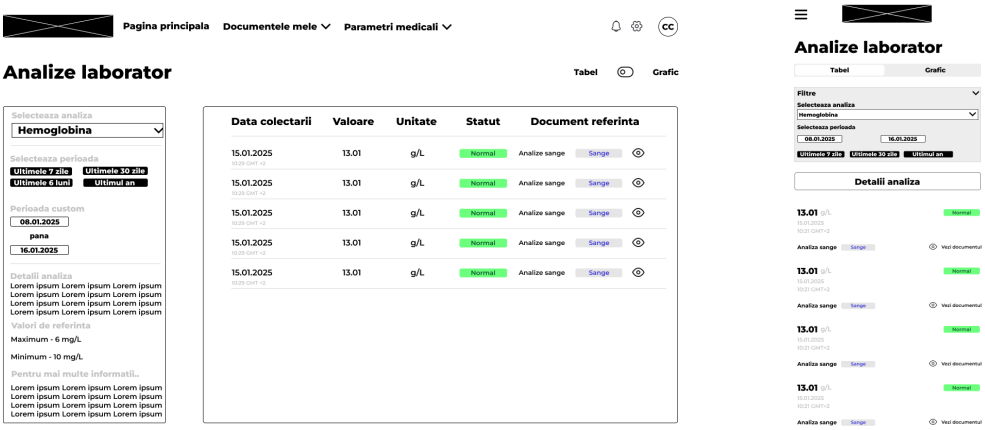
Figure 4.7: Desktop and Mobile version of the Dashboard screen



(a) Desktop version

(b) Mobile version

Figure 4.8: Desktop and Mobile version of the Medical History screen



(a) Desktop version

(b) Mobile version

Figure 4.9: Desktop and Mobile version of the Lab Test Results screen

4.4 Project Tech Stack

Based on the research conducted in sections 3.3.1, 3.3.2 and 3.3.3, the student has decided to use the following technologies for the project, which are shown in the diagram 4.10 below.

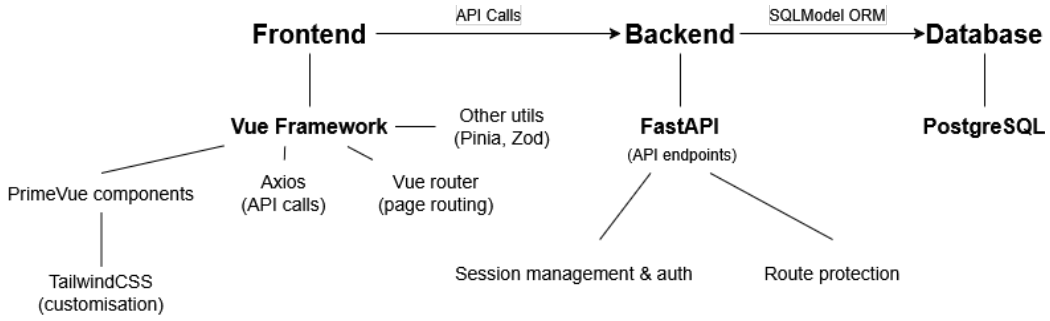


Figure 4.10: Proposed System Architecture

In the next subsections, each part of the system will be discussed in more detail.

4.4.1 Frontend

For the frontend part of the project, the student has decided to use Vue, which is a JavaScript framework for building single-page applications (SPAs) and UIs (**vue**). This framework's strengths lie in its ease of use and flexibility, with features like component and view-based architecture, element reactivity and Single File Components (SFCs) that allows HTML, CSS and JavaScript to be combined in a single `.vue` file. Vue also boasts a large and active community, with many libaries and guides available for developers. Finally, the student has had previous experience working with Vue in their Team Project module, making it an ideal choice for this project.

Diagram 4.11 below showcases how different components and libraries of the frontend will interact with each other. The next subsections will discuss each part in more detail.

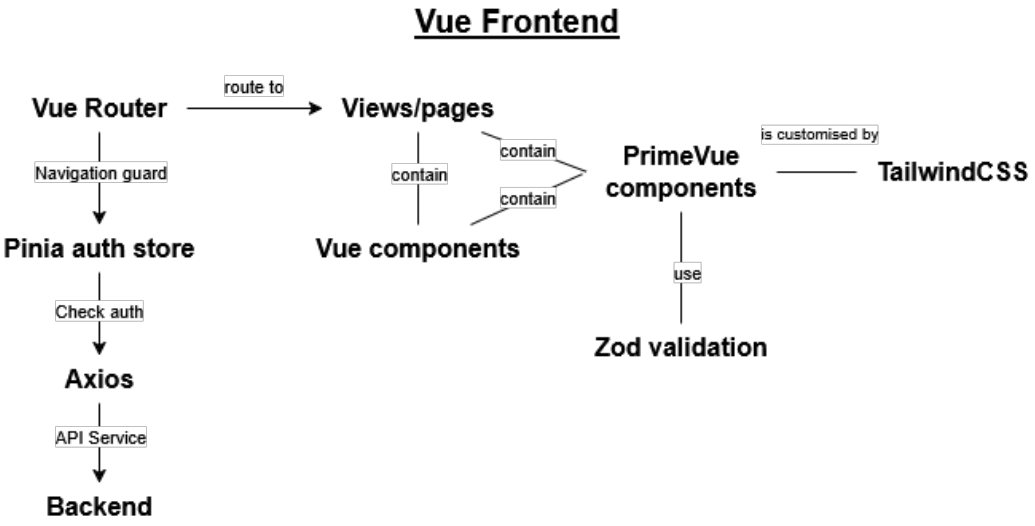


Figure 4.11: Interaction of Frontend Components

Vue Router

One of the key elements of the frontend is the Vue Router. It is the official client-side router for Vue.js, which allows the user to navigate between different pages of the application without a full page reload (**vuerouter**). This makes the full use of the SPA capabilities of Vue, providing a seamless and fast user experience. Similarly, Vue Router can be used to create navigation guards to protect certain routes from unauthorized access, which is crucial for the security of the application.

Pinia

The next element is Pinia, which is a store library for Vue made by the same team that created the frontend framework (**pinia**). Its main functionality is to provide a way to share states across Vue components and pages. In this project, Pinia can be used to store the user's authentication status and other global states that need to be shared across the application.

Axios

The next element is Axios, which is a promise-based HTTP client for the browser and Node.js (**axios**). It is used to make HTTP requests to the backend API, which is crucial for the frontend to interact with the backend. Similarly, it allows for the automatic interaction with cookies in the requests sent and responses received, which is important for the authentication process.

Vue views and components

Vue components and views (or pages) represent the foundation of the frontend for the application. Components allow to break down the UI into smaller, independent and reusable elements, which can be later combined to create the frontend of the application (**vuecomponents**). Each component can be stored in a separate `.vue` file and then imported within a view/-page or another component. Views, on the other hand, represent the different pages of the application, which can be navigated to using the Vue Router.

PrimeVue and TailwindCSS

The final two elements of the frontend are PrimeVue and TailwindCSS. PrimeVue is a component library for Vue, which provides a set of pre-built components that can be used to more easily create the frontend of the application (**primevue**). TailwindCSS, on the other hand, is a CSS framework that provides a set of utility classes that can be used to style the different elements of the application (**tailwind**). In this application, PrimeVue provides the basic components like buttons, forms, and tables, while TailwindCSS is used to style these components.

4.4.2 Backend

For the backend part of the application, the decision was made to use FastAPI as the main framework. It is a modern and high performance web framework used in building APIs with Python (**fastapi**). Based on the student's research, FastAPI seemed to be a good choice due to its ease of use, simplicity and good documentation available. The student also had previous experience with Python, making a Python-based framework an easy choice for this project. Finally, the reason for choosing a different, Python-based framework for the backend allows for more flexibility in the future, as the backend can be developed independently from the frontend and be re-used for different platforms, such as mobile or desktop applications.

The diagram 4.12 below shows how different components of the backend will interact with each other.

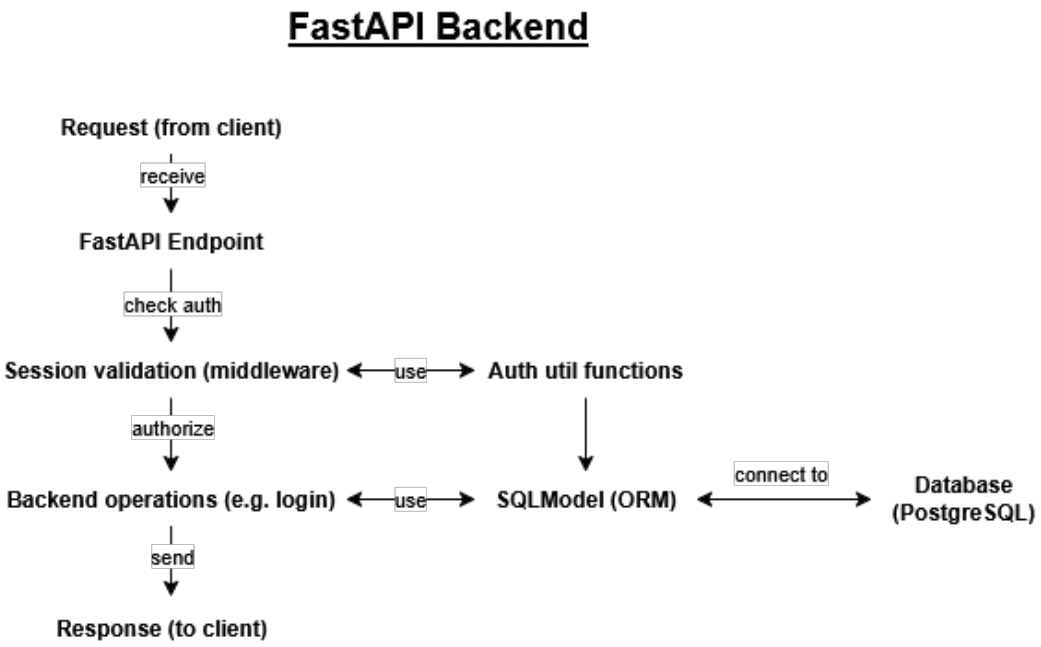


Figure 4.12: Interaction of Backend Components

4.4.3 Database

For the database, the student has chosen PostgreSQL, an open source relational database system that boasts a good reputation for its reliability, wide feature set and good performance and scalability (**postgres**). PostgreSQL supports a wide range of data types, including JSON and UUID, which are important for this project. Similarly, PostgreSQL offers robust security features like an access-control system, column or row-level security and secure connections with SSL, among others.

SQLModel, a Python library built on top of SQLAlchemy and Pydantic, will be used to interact with the database. SQLModel provides a way to define database tables using classes, which are then used to interact with the database, making it easier and safer to work with databases in the backend (**sqlmodel**).

4.5 Project Management

4.5.1 Methodology and Tools

Based on the research in section 3.1, the student has decided that he will be using a hybrid approach, with Waterfall as the main methodology for planning and managing the initial

part of the project, such as requirements gathering and the design of the system. The development part of the project will be done using ScrumBan, so that the student will be able to utilise elements from both frameworks. There are several reasons for this choice:

1. The nature of the project — the student is working on a project that has a limited timeframe (about 6–7 months) and is of a smaller scale.
2. Documentation requirements — the student is required to document the progress during the project in this report, including the requirements gathered, design considerations and implementation decisions and outcomes.
3. Regulatory requirements — the student is required to adhere to the regulations and standards of the healthcare industry, which may require extensive documentation and planning.
4. Customer involvement — the student will be working closely with the project stakeholder, who will be providing feedback and guidance throughout the project.
5. Familiarity with both Agile and Waterfall — the student has experience with both Agile (specifically Scrum and Kanban) and Waterfall methodologies, and has worked on projects that have used both approaches.

The student will use Notion as their project management tool, which is a comprehensive workspace that allows for task management, documentation, collaboration and knowledge sharing. Notion provides a wide range of features, including templates for Agile project management, creation of Agile elements like user stories, epics, backlogs, sprints and boards, and maitaining of documentation if necessary (**notion**).

4.5.2 Sprint planning

Based on the time remaining after the completion of the literature review, requirements gathering and system design, it was decided that only 6 sprints will be able to be completed before the project deadline. The sprints will be planned as follows:

- **Sprint 1** — Will focus on choosing the tech stack, installing the necessary tools and frameworks and setting up initial project files. Additionally, a basic frontend and backend will be created, with a focus on the basic functionalities like user authentication and registration and connection with the API/database.
- **Sprint 2** — Will focus on the patient profile, managing vaccines, allergies, medications and health data. The dashboard will also be created, with a basic layout and design.

- **Sprint 3** — Will focus on finishing the main dashboard and the upload and viewing of medical records and medical history.
- **Sprint 4** — Will focus on processing lab records with the LLM API and displaying them in tabular or graphical format.
- **Sprint 5** — Will focus on sharing records with doctors through various methods like email, share links or QR codes.
- **Sprint 6** — Will address any additional features or functionalities that need to be added, as well as any bugs or issues that need to be fixed.

Chapter 5

Development

This chapter will discuss the development of the project. Rather than going into detail how each sprint has been done, it will focus on the main features implemented, how this was achieved and will talk about successes or challenged that were encountered in the development process.

5.1 Sprint #1

The first sprint of this project was focused in laying the groundwork for the project, which mainly involved setting up the schemas for the ORM that would create the tables for the database, the initial setup of the API endpoints, such as the login and register endpoints, authentication and other security measures, but also setting up the frontend of the project.

5.1.1 SQLModel Schemas & Database creation

SQLModel played a pivotal role in helping quickly build the database tables. As previously mentioned in 4.4, SQLModel is built on top of SQLAlchemy and Pydantic, both being very powerful Python libraries. Pydantic was used to create the schemas that were used both within the backend but also by SQLAlchemy to create the tables in the database.

Below is an example of a schema that was used to create the User table in the database.

```
# Base model that contains the field serialiser for date formatting to dd-mm-yyyy and the date field itself
class DateFormattingModel(SQLModel):
    dob: date | None = None
```

```
4
5           @field_serialiser('dob')
6           def serialise_dob(self, value: date) -> str:
7               return value.strftime("%d-%m-%Y")
8
9       class User(DateFormattingModel, table=True):
10          id: uuid.UUID = Field(default_factory=uuid.uuid4, primary_key=
    True)
11          name: str
12          email: EmailStr = Field(index=True, unique=True)
13          hashed_password: str
```

Listing 5.1: SQLModel User Schema

Similarly, SQLModel (and more specifically Pydantic schemas) could be used to create response/request models that would be used by the endpoints to validate the data that was being sent to or by the API. These schemas did not create tables in the database, but were used to control or validate the data being sent to or from the API.

Below is an example of a SQLModel/Pydantic schema that was used to validate the data that was being sent to the register endpoint.

```
1       # User Data model used for login and registration
2       class UserAuth(SQLModel):
3           email: EmailStr
4           password: str
5           name: str | None = None # Will only be used for registration
6
7       # User Data model used for most API responses
8       class UserPublic(SQLModel):
9           id: uuid.UUID
```

Listing 5.2: SQLModel Auth Schema

5.1.2 Authentication & APIs

As the project was centered around building a PHR system that would deal with sensitive health data, one of the main concerns was the security of the system and its users.

One of the important choices to make was how to handle the authentication. Based on research on modern authentication methods, the student has identified 4 different ways to handle authentication in the project:

- JWT Tokens
- OAuth2
- Session-based authentication
- Using 3rd party authentication services, like Auth0

Each method has its own disadvantages and advantages, which will be briefly discussed below. JWT (JSON Web Tokens) authentication involves generating tokens containing user information for client-side storage (**auth1**). While it requires minimal server-side management, tokens can be vulnerable to XSS and CSRF attacks and cannot be invalidated once created (**auth1**; **auth2**). The authors recommend using methods like pairs of access and refresh tokens to mitigate these risks, and also to store the tokens in cookies.

Session-based authentication stores session information server-side, using cookies to maintain client sessions (**auth1**; **auth2**). While cookies can face similar security risks as JWT, they can be secured through flags like HttpOnly and SameSite (**mozilla**). Sessions can be invalidated server-side, providing better security control.

Third-party authentication options include OAuth2 through providers like Google (**auth2**; **auth3**) or services like Auth0 (**auth0**). While these offer quick implementation, they may introduce dependencies or security concerns due to third-party data storage.

Moldova's electronic government iniatives have created MPass, which is a national Single Sign On (SSO) authentication system that offers a secure and easy way to access electronic government services (**mpass**). While it may not be possible to currently integrate MPass into the system, it will be considered as a possible future integration, given the system's Moldovan user focus, which will be discussed in the 7.2 section.

In the end, the student decided to go with a session-based authentication system. One of the main reasons was its ease of implementation that still provided a good level of security. The ability to control the session on the server side was an additional, important factor that contributed to this decision. This allows the system to safely log out users, invalidate sessions due to inactivity or suspicious activity, and also to control the session's lifetime, all without relying on storing tokens on the client side.

Sessions were created in the backend by storing them in a Session table in the database. The table contained a Session ID, which was passed to the user as a cookie. Upon a succesful login, the backend would set the cookie in the user's browser, which would then be sent with every request to the server.

To ensure that the cookies were safe from common attacks like Cross-Site Scripting (XSS)

and Cross-Site Request Forgery (CSRF), the student security used guidance from **owasp**; **mozilla** to set the following flags on the cookies:

```
    # Set the session cookie in the response and send it to the client
    response.set_cookie(
        "session_id",
        str(session_id), # Using str() to convert the UUID to a string
        httponly=True,
        max_age=3600, # 1 hour
        samesite="strict",
        secure=True)
```

Listing 5.3: Session Cookie Flags

To further secure the system, some of the endpoints that were created in this sprint were protected by using session validation checks to ensure that the user was authenticated before accessing the endpoint. This was achieved thanks to FastAPI's dependency injection system, which allowed the student to create a dependency that would check if the user was authenticated before allowing the request to continue. An example of this can be seen below:

```
@app.post("/logout")
async def logout(
  response: Response,
  request: Request,
  user_id: uuid.UUID = Depends(validate_session),
  session: Session = Depends(get_session)
):
  session_id = request.cookies.get("session_id")
  cookie_user_id = session.get(AuthSession, uuid.UUID(session_id)).
    user_id

  if cookie_user_id != user_id:
    raise HTTPException(
      status_code=403,
      detail="You do not have permission to log out this user"
    )

  # Code that deletes session and the cookie
  return {"status": status.HTTP_200_OK, "message": "Logout successful"
    }

```

```python
async def validate_session(request: Request, session: Session =
    Depends(get_session)):
  session_id = request.cookies.get("session_id")
  if not session_id:
    raise HTTPException(
      status_code=status.HTTP_401_UNAUTHORIZED,
      detail="Session cookie not found"
    )

  existingAuthSession = session.exec(
    select(AuthSession)
    .where(AuthSession.id == uuid.UUID(session_id))
    .where(AuthSession.expires_at > datetime.now())
  ).first()

  if not existingAuthSession:
    raise HTTPException(
      status_code=status.HTTP_401_UNAUTHORIZED,
      detail="Session not found"
    )

  return existingAuthSession.user_id
```

Listing 5.4: FastAPI Dependency for Session Validation

Finally, to ensure that the system was secure, the system also used a hashing algorithm to hash the passwords before storing them in the database. The student identified multiple hashing algorithms that could be used, with the main contenders being bcrypt, scrypt and Argon2.

Bcrypt is a popular and secure hashing algorithm that is widely used in the industry (**hash3**; **hash2**). It is considered to offer an optimal balance between security and speed, making it a popular choice for real-world implementation (**hash1**). In 2 of the previously mentioned papers, bcrypt was ranked on the mid-to-high end of the security scale, with algorithms such as scrypt and Argon2id being scored higher in the dimension of security (**hash1**; **hash3**). The authors of these papers concluded that Argon2id was the best choice for hashing passwords, followed closely by scrypt and bcrypt. However, while scrypt and Argon2id are more secure, they are also more computationally expensive, and may require more configuration to be used effectively (**hash1**).

In the end, the decision was made to use bcrypt as the hashing algorithm. The main reasons

for the use of this algorithm was its simplicity of use, wide adoption and its balance between security and perfomance. The passlib library was used to hash the passwords before storing them in the database. The library allowed for automatic salt generation and storage within the hash and by default, it used a work factor of 12, which is considered to be a good balance between security and performance. An example of this can be seen below:

```
    # Util function to verify the password hash against the plaintext
    password
    def verify_hash(plaintext_password: str, hashed_password: str) ->
    bool:
        return bcrypt.verify(plaintext_password, hashed_password)

    # Util function to create a password hash from the plaintext
    password
    def create_hash(plaintext_password: str) -> str:
        return bcrypt.hash(plaintext_password)
```

Listing 5.5: Hashing passwords with bcrypt

A final security measure that could've been implemented was a rate limiter to all of the created endpoints. Similarly, Starlette, the framework FastAPI was built on top of, also offered other middleware such as HTTPSRedirectMiddleware, TrustedHostMiddleware, and others that could be used to further secure the system. However, in the interest of time, it was decided to leave these for a future sprint.

5.1.3 Frontend Setup

While most of the work done in this sprint was done in the backend, some work was also done in the frontend, mainly focusing on the login and register pages. Here, the student used the PrimeVue component library to quickly and easily create the forms that would be used to log in and register users.

An example of PrimeVue components being used can be seen below in the Register page. Some of the components taken from PrimeVue include InputText, DatePicker, Password, Button and Message.

```
          <DatePicker
            name="dob"
            dateFormat="dd/mm/yy"
            placeholder="Data nasterii"
            showIcon
            fluid
```

```
7               :maxDate="maxDate" />
```

Listing 5.6: PrimeVue components in the Register page

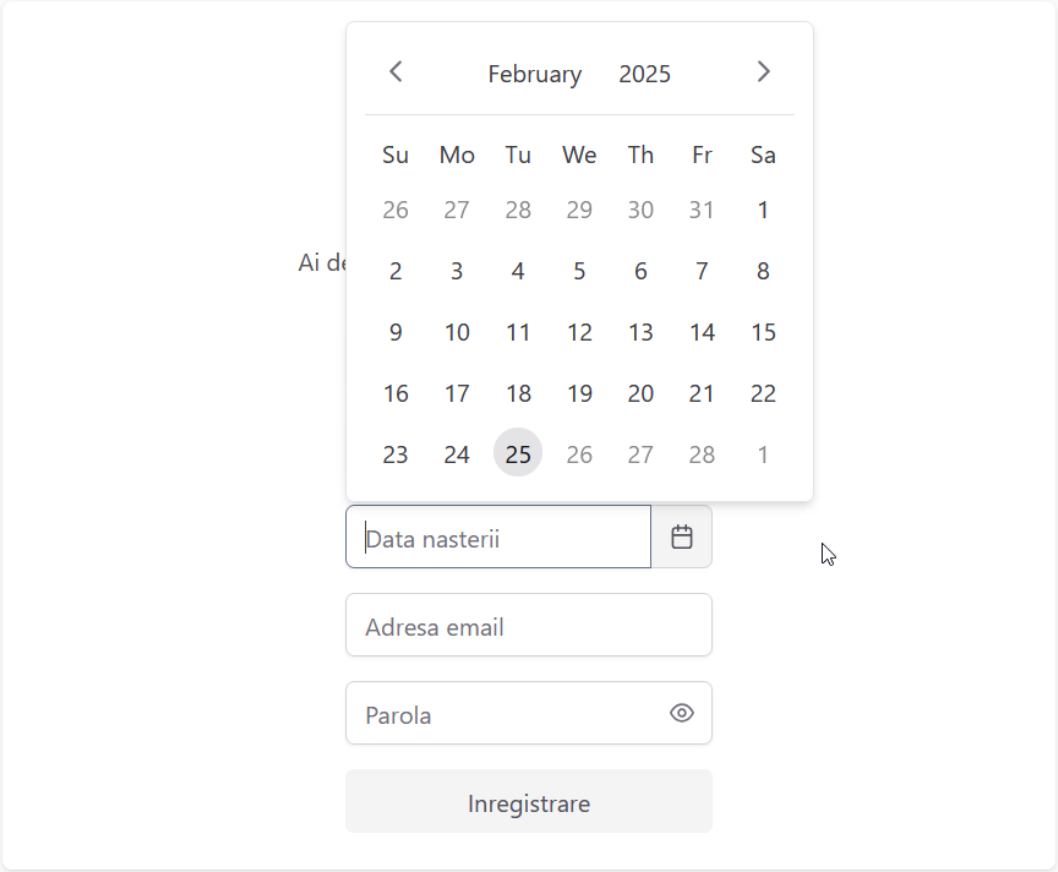


Figure 5.1: PrimeVue DatePicker component

As previously mentioned in 4.4, Vue also comes with other in-built tools that help create a well-working frontend. One of these tools is Vue Router, which was used to create the routes for the frontend. The student created a simple router that would allow the user to navigate between different pages. A navigation guard was used to protect the routes that required the user to be authenticated. An example of this can be seen below:

```
1 router.beforeEach(async (to) => {
2   const authStore = useAuthStore()
3
4   if (to.meta.requiresAuth) {
5     await authStore.checkAuth() // Check if user is authenticated for
    protected routes
```

```
    if (!authStore.isAuthenticated) {
      return { name: 'Login' } // Redirect to login if not
    authenticated
    }
  }
  // More code to handle other cases
})
export default router
```

Listing 5.7: Vue Router Navigation Guard

Similarly, Pinia store was used to manage the authentication state of the user in the frontend. The store was then used by elements like the Navigation Guard to check if the user was authenticated before allowing them to access certain routes. An example of the store can be seen below:

```
export const useAuthStore = defineStore('auth', () => {
  const isAuthenticated = ref(false)
  const user = ref(null)

  async function checkAuth() { // check if user is authenticated
    try {
      const response = await api.get('/me') // return user object if
    user is authenticated
      isAuthenticated.value = true // user is authenticated
      user.value = response.data.user
    }
    // More code to handle errors and no authentication
  }
  return { isAuthenticated, user, checkAuth }})
```

Listing 5.8: Pinia Store for Authentication

5.1.4 Challenges

Lack of experience

One of the biggest challenges encountered in this sprint was the lack of experience using some of the frameworks and libraries used in the project. As such, this sprint's progress moved slower due to the learning curve that needed to be overcome first by reading the documentation and finding any guides/help online.

System-related decisions

Another challenge was making decisions regarding different sections of the system, such as authentication, hashing algorithms, and others. Additional time was spent researching the best practices for each respective functionality analysing their advantages, disadvantages and ways to implement them in the current system.

5.1.5 Requirements completed

As this was the first sprint, the main focus was on setting up the groundwork for the project. The main requirements completed in this sprint were:

- The system must provide a secure login mechanism for patients by using a combination of login and password.
- The database must store the user credentials in a secure manner.
- The system must be accessible on all modern desktop and mobile-based browsers.
- The system must allow patients to add their own personal information, such as name or date of birth.

5.2 Sprint #2

The second sprint of this project was focused on building some of the smaller elements of the system, such as the ability to add vaccines, allergies and medications. The decision to start with the smaller elements was made to allow the student to get a better understanding of how the frameworks used in the project worked and to get a better understanding of how to structure the project.

5.2.1 Adding Vaccines, Allergies and Medications functionality

The student started by creating the schemas for the Vaccines, Allergies and Medications tables in the database. These tables were created in a similar way to the User table, using SQLModel schemas. Afterwards, the student created the endpoints that would allow the user to add, update, delete and view the data in these tables.

On the frontend side, the system used Vue's main strengths to enable a smooth developer and future user experience. Some of the strengths used were using Vue components to create reusable elemenets, such as a Vaccine Card that would be used to display the vaccines that the user had added. The card also used other Vue elements such as props and emits to pass

data between parent and child elements and pass events, respectively. An example of the Vaccine Card can be seen below:

```vue
<Card class="w-full" :pt="cardStyles">
  <template #title>
    <span class="font-bold text-2xl">{{ name }}</span>
  </template>
  <template #subtitle>
    <div class="flex items-center justify-between">
      <div>
        <span class="font-bold">{{ provider }}</span>
        <span>{{ date_received }}</span>
      </div>
      <div>
        <Button icon="pi pi-eye" class="p-button-rounded p-button-text
    "
          @click="emit('showFile', props.id)" v-if="hasCertificate"/>
        <Button icon="pi pi-ellipsis-h" class="p-button-rounded p-
    button-text"
          @click="toggle"/>
        <Menu ref="menu" :model="items" :popup="true" />
      </div>
    </div>
  </template>
</Card>
```

Listing 5.9: Vue Vaccine Card Component Example

The VaccineCard component can then be easily used in the main view:

```vue
<VaccineCard
  v-for="vaccine in vaccines"
  :key="vaccine.id"
  v-bind="vaccine"
  :has-certificate="!!vaccine.certificate"
  @delete="deleteVaccine"
  @open-edit="openEditDialog"
  @show-file="showCertificate"
/>
```

Listing 5.10: Using VaccineCard Component

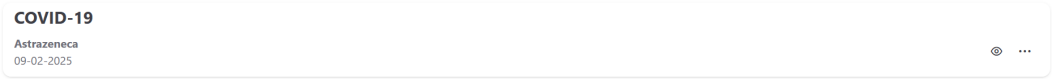


Figure 5.2: Vue Vaccine Card component

This format was used across the system to easily create reusable components that could be used in multiple places, making the system more modular and easier to maintain.

To make sure the data displayed in the frontend was always up to date, the frontend used Vue's reactivity system. This allowed the data to be automatically updated whenever any change was made to the data in the backend or frontend. Examples of the reactivity system in use can be seen below, with functions that add and delete vaccines from the frontend when called:

```
// Will add a vaccine to the vaccines array, which will automatically
// create a new VaccineCard element
const addVaccine = (vaccine) => {
  vaccines.value.push(vaccine)
}

// Will remove a vaccine from the vaccines array, which will
// automatically remove the VaccineCard element
const deleteVaccine = (id) => {
  vaccines.value = vaccines.value.filter((vaccine) => vaccine.id !==
    id)
}
```

Listing 5.11: Vue Reactivity System

5.2.2 File Uploads

Another major feature that was implemented in the 2nd sprint was the ability to upload files. At the time of implementation, the feature was only to be used for uploading vaccine certificates, however it could be easily extended to other parts of the system, such as uploading health records or lab results.

To allow a proper upload of files, a new table was created in the database, called FileUpload. This table contained the file metadata, such as its name, size, type, path and others. Similarly, a new table for Allergy Severity was created to ensure consistency in the user uploads and to allow for easy filtering and searching of the data.

The updated ERD diagram with the new tables can be seen below in figure 5.3:

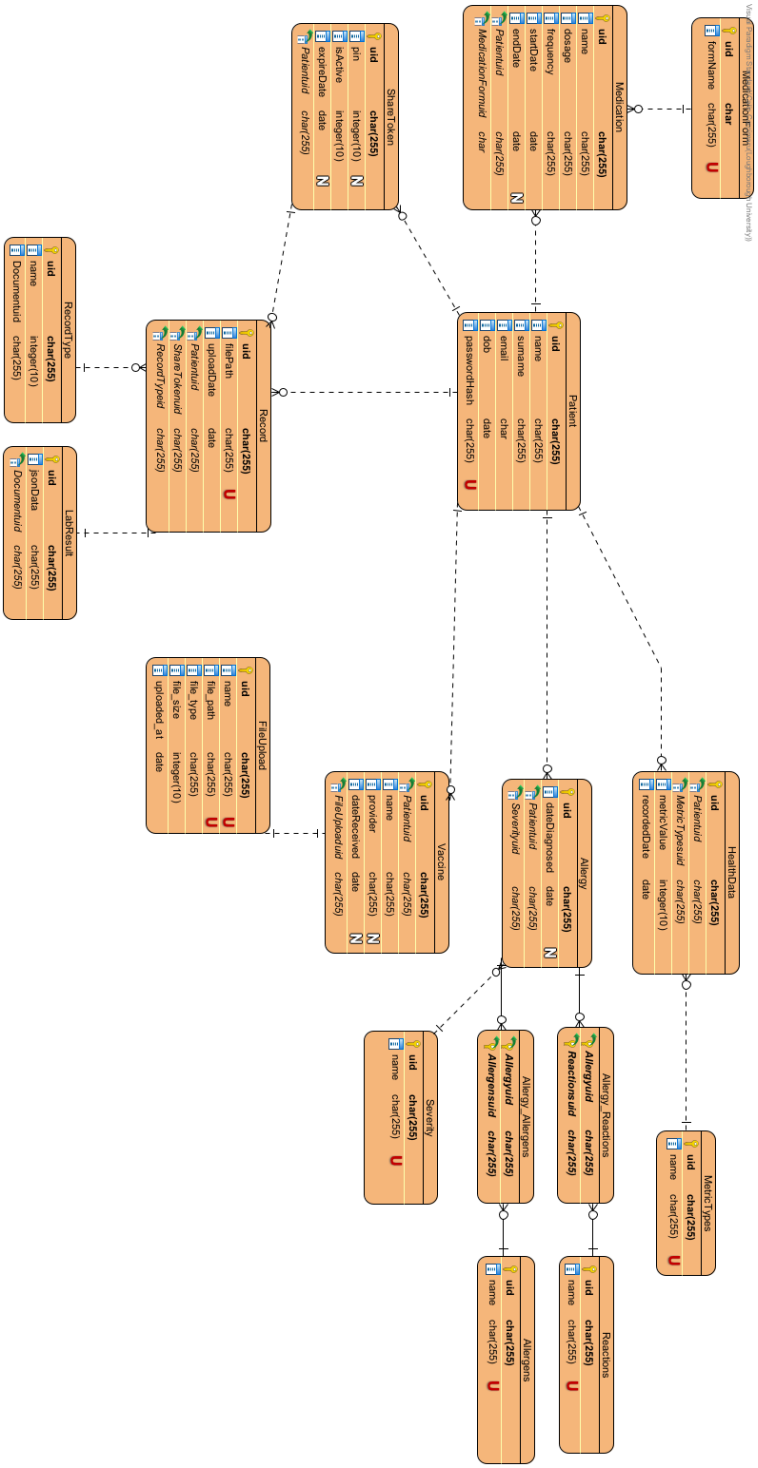


Figure 5.3: Updated Entity Relationship Diagram Sprint # 2

To add more security to the system, the files were further encrypted before being stored in the database. AES was chosen as the encryption algorithm. Based on the articles analysed, it was consistently found to have the best balance between security and performance among other symmetric and asymmetric encryption algorithms (**crypt1**; **crypt2**; **crypt3**). The choice to go with a symmetric algorithm was made due to its simplicity, as the key could be stored on the server-side in an environment variable, allowing the encryption and decryption of the files to be done on the server-side. This also allowed to avoid the issue of users losing access to their files if they lost their private encryption key, in the case of asymmetric encryption.

The encryption was done using the Fernet symmetric encryption algorithm from the cryptography library. Fernet uses AES-128 to encrypt the files. The key used to encrypt the files was stored in the environment variables and was generated when the system was started. In the interest of time, no rotation system for the key was implemented, however this would be a good feature to add in the future.

The file upload process can be seen in figure 5.5 below. The file upload process was done in two steps. The first step was to upload the file to the server by using multipart form data. The next step would have the file be validated by checking its file type and size, then encrypted and stored in the file system and database. To access the files, the system used a new endpoint that would allow the user to download the file. The endpoint would take the file ID as a parameter and would stream the file to the user.

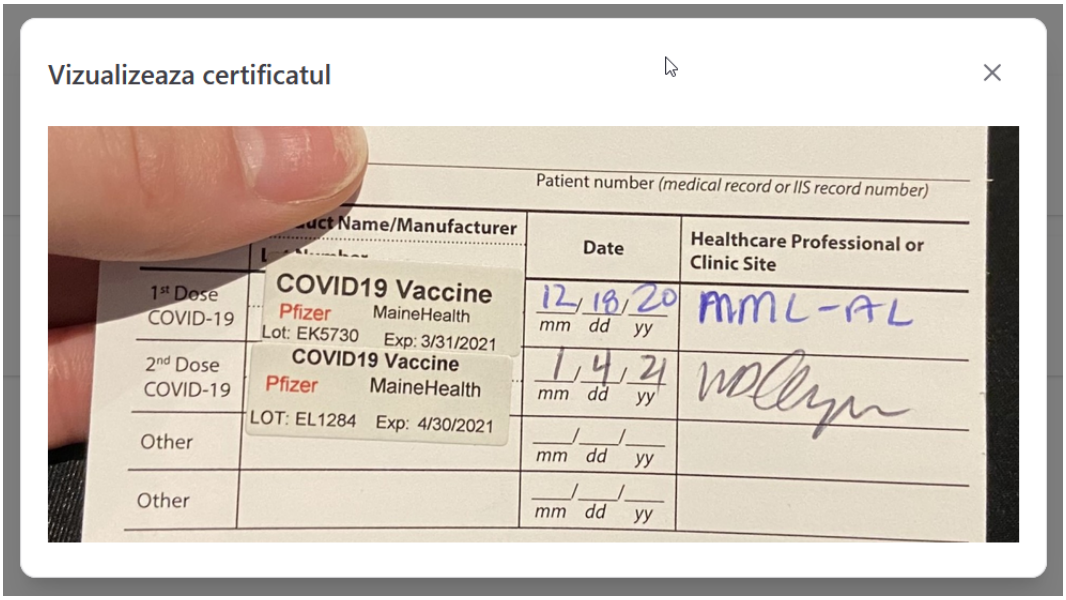
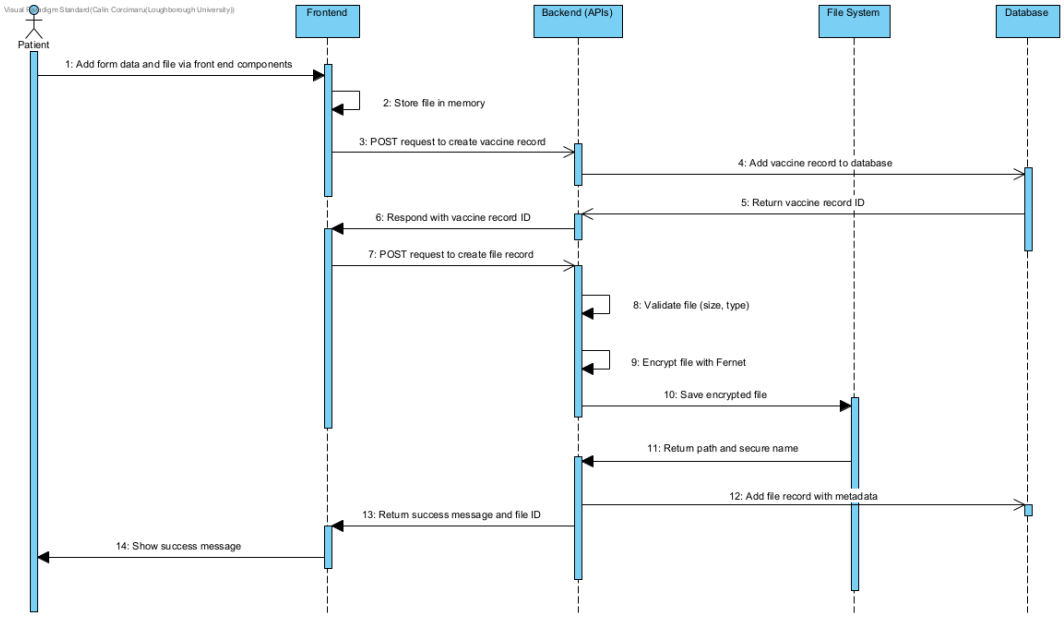
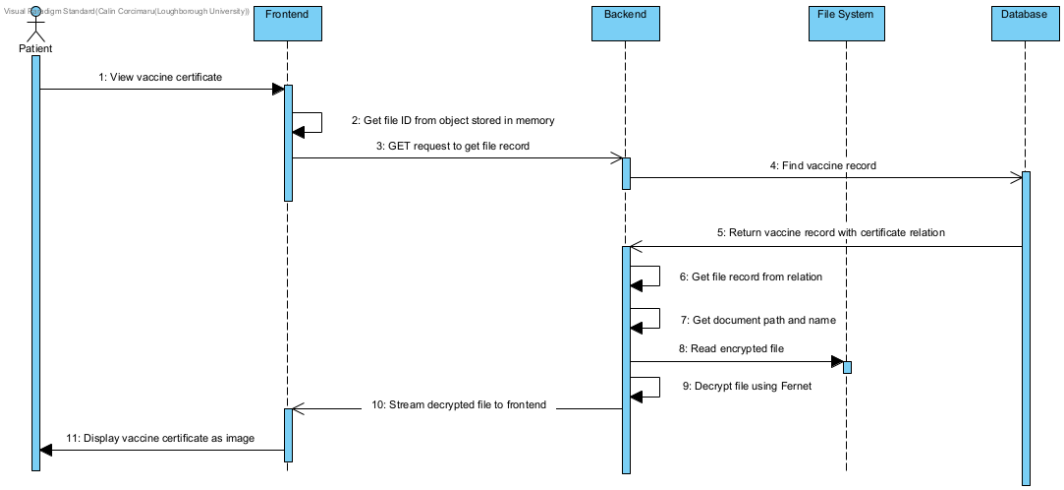


Figure 5.4: Uploaded Vaccine Certificate



(a) File Upload Sequence Diagram



(b) File View Sequence Diagram

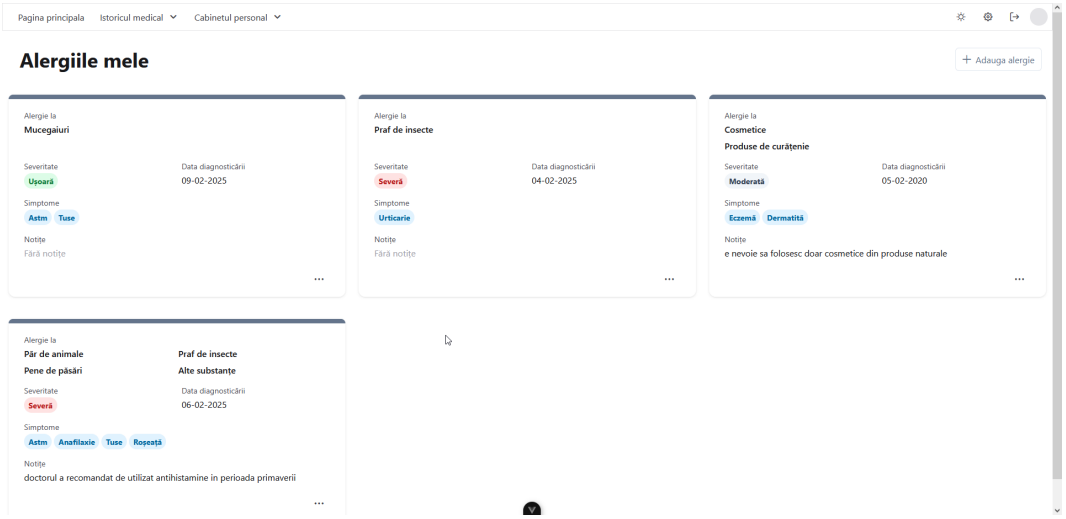
Figure 5.5: Sequence Diagrams for File Upload and File View

The endpoint can be seen below:

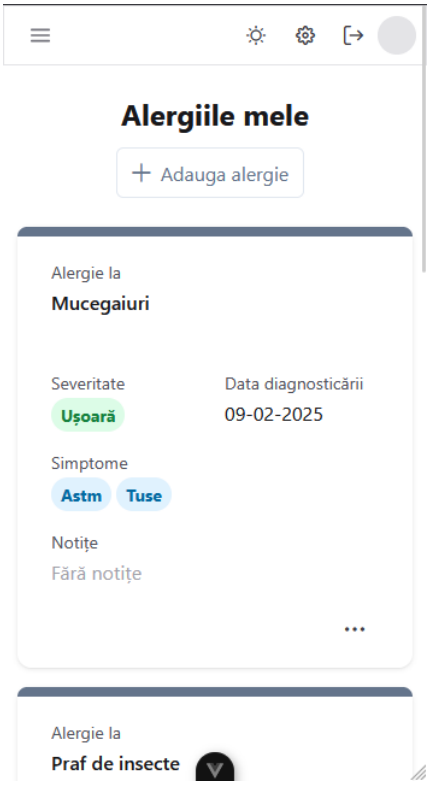
```python
# Get a file by ID
@app.get("/files/{record_type}/{record_id}")
async def get_file(
  record_type: str,
  record_id: uuid.UUID,
  user_id: User = Depends(validate_session),
  session: Session = Depends(get_session)
):
  # Code to get the file record from the database

  async def get_data_from_file():
    with open(file_record.file_path, "rb") as f:
      encrypted_content = f.read()

    decrypted_content = decrypt_file(encrypted_content)

    yield decrypted_content

  return StreamingResponse(
    content=get_data_from_file(),
    media_type=file_record.file_type,
    status_code=status.HTTP_200_OK,
    headers={"Content-Disposition": f"inline; filename={file_record.
    name}"}
  )
```

Listing 5.12: File Download Endpoint

Finally, the frontend was made with both mobile and desktop browsers in mind. This was achieved by using PrimeVue components and styling them with TailwindCSS. The system was also made to be responsive, so that it would look good on any screen size. Examples of the allergies page on both desktop and mobile can be seen below:



(a) Desktop version



(b) Mobile version

Figure 5.6: Desktop and Mobile version of the Allergies page

5.2.3 Challenges

Introducing new functionality

In this sprint, the main challenge was introducing the new features/functionality in the system, while ensuring they worked properly together with both the existing frontend and backend. For some features, there was a need to revise and change some of the existing code, such as the API endpoints and even the database schemas to accommodate the newly implemented changes. This added some additional complexity, as it was crucial to ensure the changes did not break any of the existing functionality, while allowing the new features to work as intended.

File Uploads

Another challenge was understanding how some of the newly implemented features worked, such as the file upload system. It is important to note that the student had no previous experience implementing any file upload functionality, so extra time and work was dedicated to ensure the relevant documentation was studied and then implemented correctly.

System complexity

Finally, due to lack of time and poor initial planning because of the complexity of the system, some of the features intended for this sprint had to be moved to the next one. Some examples include the health data section and also the main dashboard parts for vaccines, allergies, medications and health data.

These changes and challenges show that undertaking a more agile approach to the project was beneficial, as the project embraces change and allows for more flexibility in the development process.

5.2.4 Requirements completed

The main requirements completed in this sprint were:

- The system must store the data in a secure manner, ensuring that only the patient and the doctor can access the data.
- The system must allow patients to upload their own medical records in a variety of formats (PDF, DOC, etc).
- The system should allow to upload files/documents for other types of records (vaccine certificates, etc).

- The system must allow the patient to add their own allergies.
- The system must allow the patient to add their own vaccinations.
- The system must allow patients to enter their current medication including details such as the name of the drug, dosage, frequency and start/end date.
- The system must allow patients to add new medication to their list.

5.3 Sprint #3

The third sprint of the project was focused on building the main dashboard view of the system, which would display an overview of the most recent records added, such as vaccines, allergies, medications, etc. Additionally, another aim for this sprint was to complete the vital signs feature of the system, which would allow patients to add information such as weight or blood pressure in both a tabular and graphical format.

All these features were part of the initial requirements discussed with the stakeholders, and were chosen to be completed in this sprint as their technical implementation was similar to more complex features that would be implemented in the future, which would help build additional proficiency in the frameworks and libraries used in the project.

However, besides the above-mentioned features, additional work had to be done on how the information is displayed in the frontend. This was a change requested by one of the stakeholders, based on their feedback after the sprint 2 features were presented to them. As such, additional time had to be spent on reworking the frontend to better display the information in a more user-friendly and consistent way.

5.3.1 Updated frontend

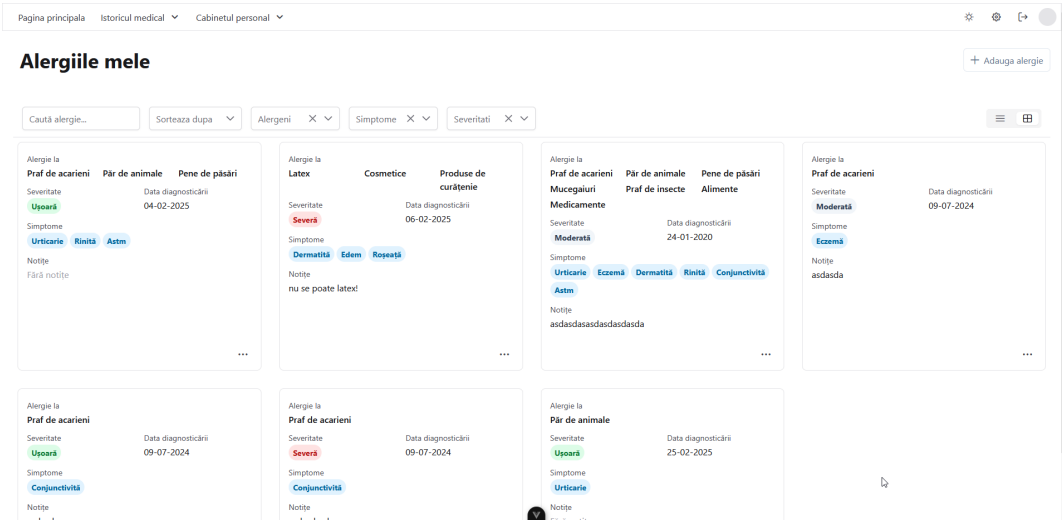
One of the main feedback received from the stakeholders was the inconsistency in how the information was displayed for some of the sections in the system, such as the vaccines, allergies and medications. This was due to the different amount of information each record contained: vaccines contained the least amount of information, followed by allergies and medications. As such, some records were stored in a grid view, while others were stored in a list view. Also, some cards had different ways of displaying the information, such as using tags or colored lines, etc.

The decision was made to keep things simple, by removing unnecessary colors and ensuring that there would be consistency in how the cards were displayed either by having only one

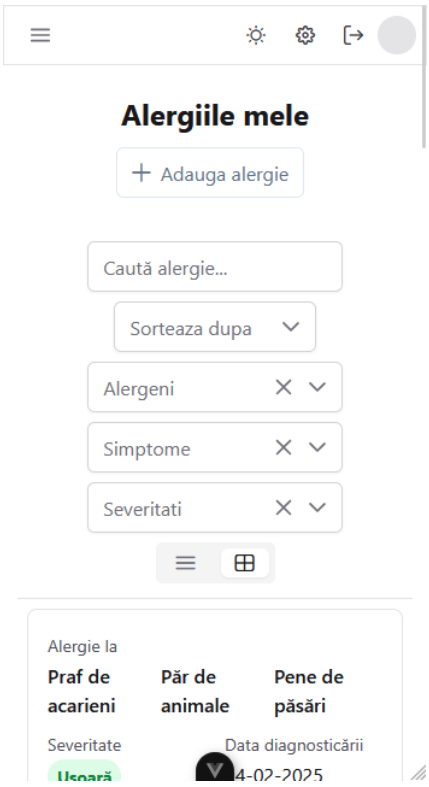
type of view (grid/list) or offering both for the user to choose from. Tags were kept as they allowed for an easy way to display the information in a more readable way.

Thankfully, PrimeVue offered an easy way to re-work the frontend thanks to its Data View component a flexible component that could be used to display data in a grid or list view. The component also offered a way to customize the data displayed, such as adding tags, icons, etc.

An example of the updated allergies page can be seen below in figure 5.7. The page now offers both a grid and list view, which can be toggled by the user. The page also offers a more consistent way of displaying the information, with each card containing the same information, but displayed in a different way depending on the view chosen.



(a) Desktop version



(b) Mobile version

Figure 5.7: Desktop and Mobile version of the Allergies page 2nd version

5.3.2 Filtering and sorting

Another feedback provided by the stakeholders was the ability to filter and sort the information displayed in the system. This was especially important for the medications, where a patient could have multiple medications and would want to quickly find a specific one. Suggestions for this feature was to mainly add a search bar that would allow the user to find a specific record, however the stakeholders added that it would be nice to have some sorting/filtering options like sorting by date added or filtering based on type of medication, allergy severity, etc.

Sorting and filtering was implemented in the frontend by using Vue's reactivity system, powered by the ref and computed functions. The ref function was used to create reactive variables that would be updated whenever the data changed. The computed function was used to create computed properties that would be updated whenever the reactive variables changed. As such the sort query could be stored within a reactive variable, while the computed function would change the data displayed based on the sort query.

An example of the sorting and filtering functionality can be seen below:

```
const filteredMedicine = computed(() => {
  return props.medications.filter((medication) => {
    return medication.name.toLowerCase().includes(searchQuery.value.
  toLowerCase())
  })
})
```

Listing 5.13: Medication Sorting and Filtering

```
const filteredAllergies = computed(() => {

if (selectedAllergens.value.length > 0) {
  return props.allergies.filter((allergy) =>
    allergy.allergens.some((allergen) => selectedAllergens.value.
  includes(allergen))
  )
}

if (selectedReactions.value.length > 0) {
  return props.allergies.filter((allergy) =>
    allergy.reactions.some((reaction) => selectedReactions.value.
  includes(reaction))
  )
```

```
  }

  if (selectedSeverities.value.length > 0) {
    return props.allergies.filter((allergy) =>
      selectedSeverities.value.includes(allergy.severity)
    )
  }

  return props.allergies.filter((allergy) =>
    allergy.allergens.some((allergen) =>
      allergen.toLowerCase().includes(searchQuery.value.toLowerCase())
    )
  )
})
```

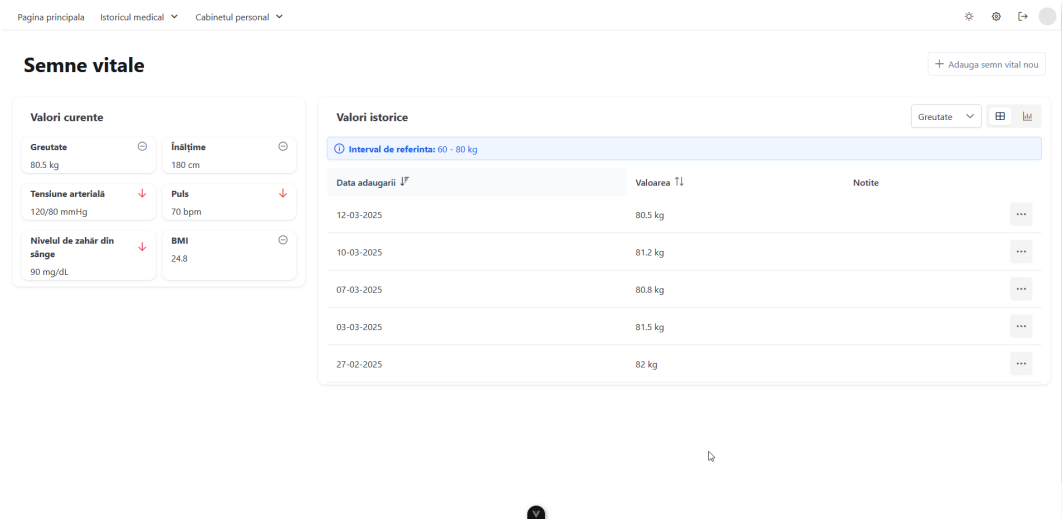
Listing 5.14: Allergies Sorting and Filtering

5.3.3 Vital Signs

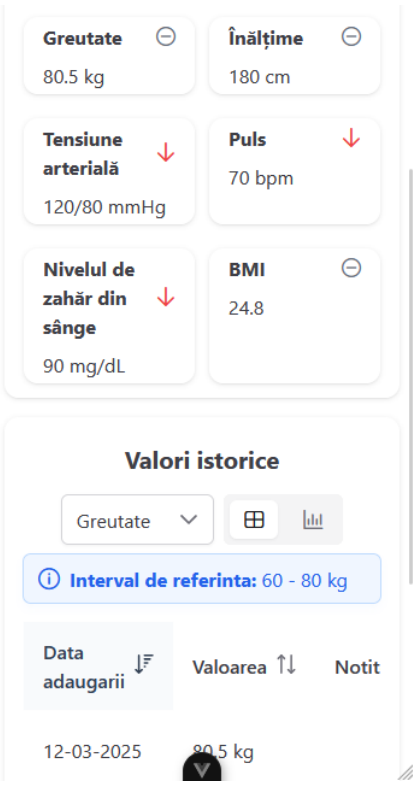
Coming back to the sprint's focus, one of the main features intended to be implemented was the vital signs feature. This feature would allow the patient to add their own vital signs, such as weight, height, blood pressure, etc. The data would be displayed in both a tabular and graphical format, allowing the patient to see their progress over time.

When discussing this requirement with the stakeholders, a suggestion was given to display the most recent/current health data point added in a separate card, so that a patient could view their most recently added information easily. Additionally, a suggestion was given to display an arrow/icon near the value of the vital data point to show whether the value was above or below versus the previous value.

To display the data in a tabular and graphical view, the system uses PrimeVue's Data Table component and Chart.js library respectively. Examples of how the Vitals page looks can be seen below in figure 5.8 and 5.9.

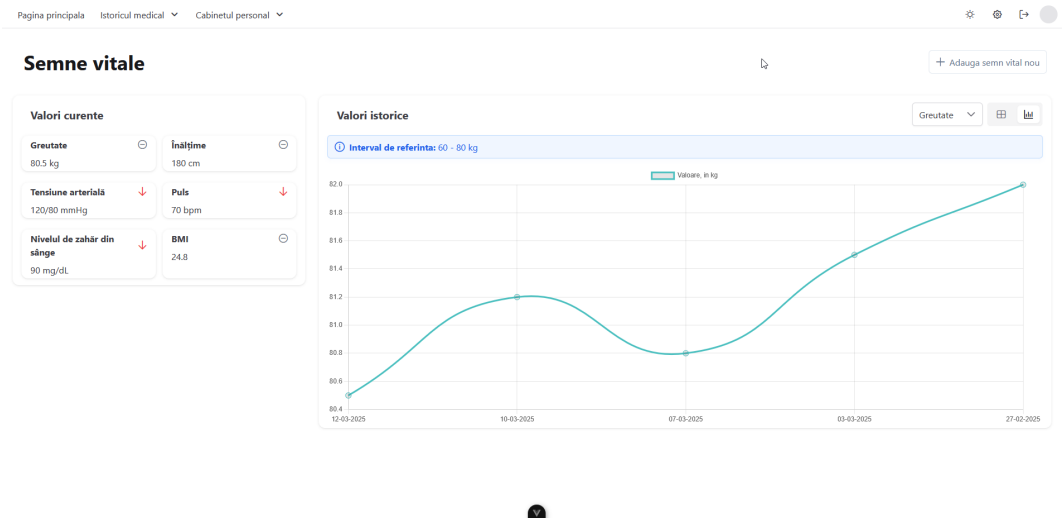


(a) Desktop version



(b) Mobile version

Figure 5.8: Desktop and Mobile version of the Vitals page



(a) Desktop version

(b) Mobile version

Figure 5.9: Desktop and Mobile version of the Vitals page, with Graph

5.3.4 Dashboard

Finally, the other main feature added was the ability for patients to have a dashboard-type view of the most recent records/information added to their account. The decision was made to display only the 5 most recent records for each section, encouraging the user to go to their respective page if they wanted to see more or see the records in more detail. The structure of each record was kept similar as to how they were displayed in their own pages to maintain consistency, so things like tags were still used to ensure a patient would know what they are looking for at the dashboard.

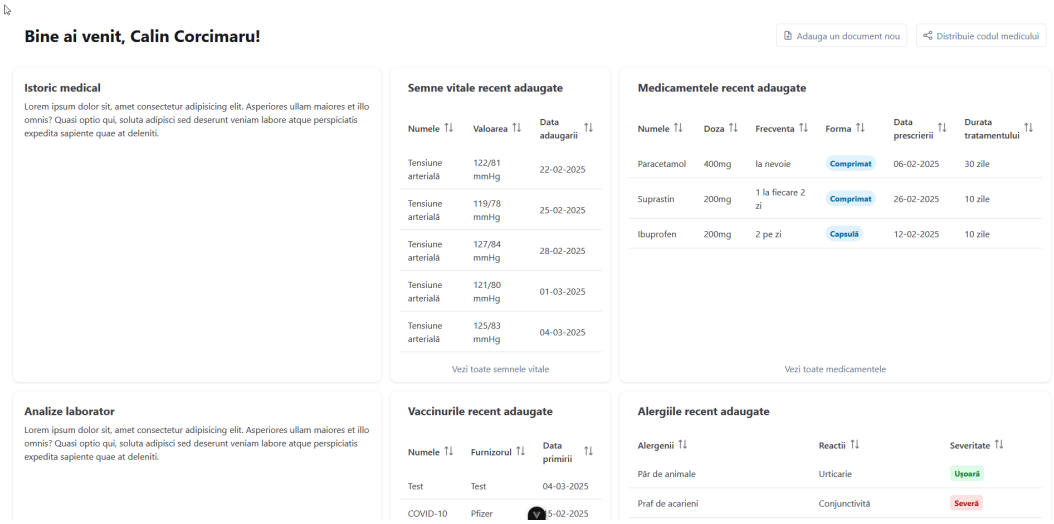
Similar to the vital signs feature, the Data Table view component from PrimeVue was used to create the individual tables for each of the section, making the development of the dashboard feature simple and easy. Each section was an individual Vue component, which was then imported into the dashboard main Vue view. An example of a section code can be seen below:

```
<template>
  <Card :pt="cardStyles" class="h-full">
    <template #title>
      <h2 class="text-xl font-bold p-4">Semne vitale recent adaugate</
    h2>
    </template>
    <template #content>
      <DataTable :value="props.vitals" removableSort>
        <Column field="name" header="Numele" sortable> </Column>
        <Column field="value" header="Valoarea" sortable>
          <template #body="slotProps">
            <span v-if="slotProps.data.value">
              {{ slotProps.data.value }} {{ slotProps.data.unit }}
            </span>
            <span v-else>
              {{ slotProps.data.value_systolic }}/{{ slotProps.data.
    value_diastolic }}
              {{ slotProps.data.unit }}
            </span>
          </template>
        </Column>
        <Column field="date_recorded" header="Data adaugarii" sortable
    >
          <template #body="slotProps">
            <span> {{ slotProps.data.original_date_recorded }} </span>
```

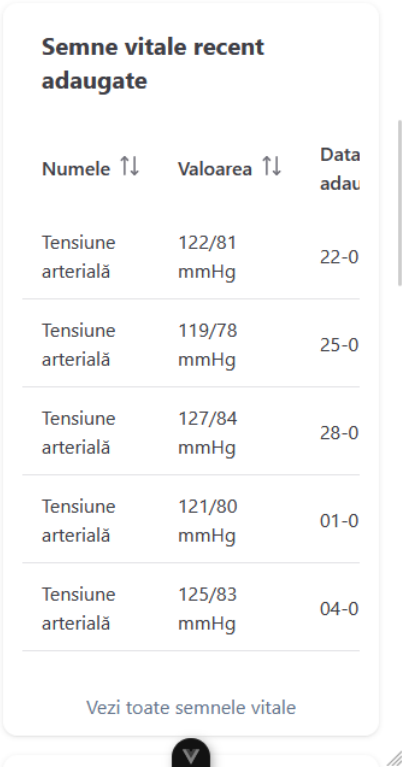
```
          </template>
        </Column>
      </DataTable>
    </template>
    <template #footer>
      <RouterLink to="/vitale" class="p-button p-button-text">Vezi
    toate semnele vitale</RouterLink>
    </template>
  </Card>
</template>
```

Listing 5.15: Vital Signs Dashboard Section Component

Finally, the dashboard view can be seen below in figure 5.10.



(a) Desktop version



(b) Mobile version

Figure 5.10: Desktop and Mobile version of the Dashboard page

5.3.5 Backend changes

While the focus of this sprint was mainly the frontend, there were some parts of the backend that needed to be changed to accommodate some of the implemented changes.

One change made was the addition of a database table for the vital data points, which would contain the type name, such as Weight, the unit of measurement and a normal range, all of which was then used or displayed on the vitals page. Another change to the database was the addition of a new field, called date_added to all of the tables/records in the system. This field was used mainly used to sort and determine the most recently added records to the system, which could be a different value to the other dates already used in the records' tables. An example could be found in vaccines: a vaccine could've been received years ago, but it could be added anytime to the system. As such, the record can have 2 different dates associated with it: the date the vaccine was actually received and the date it was added to the system. As such, the updated database schema can be found below:

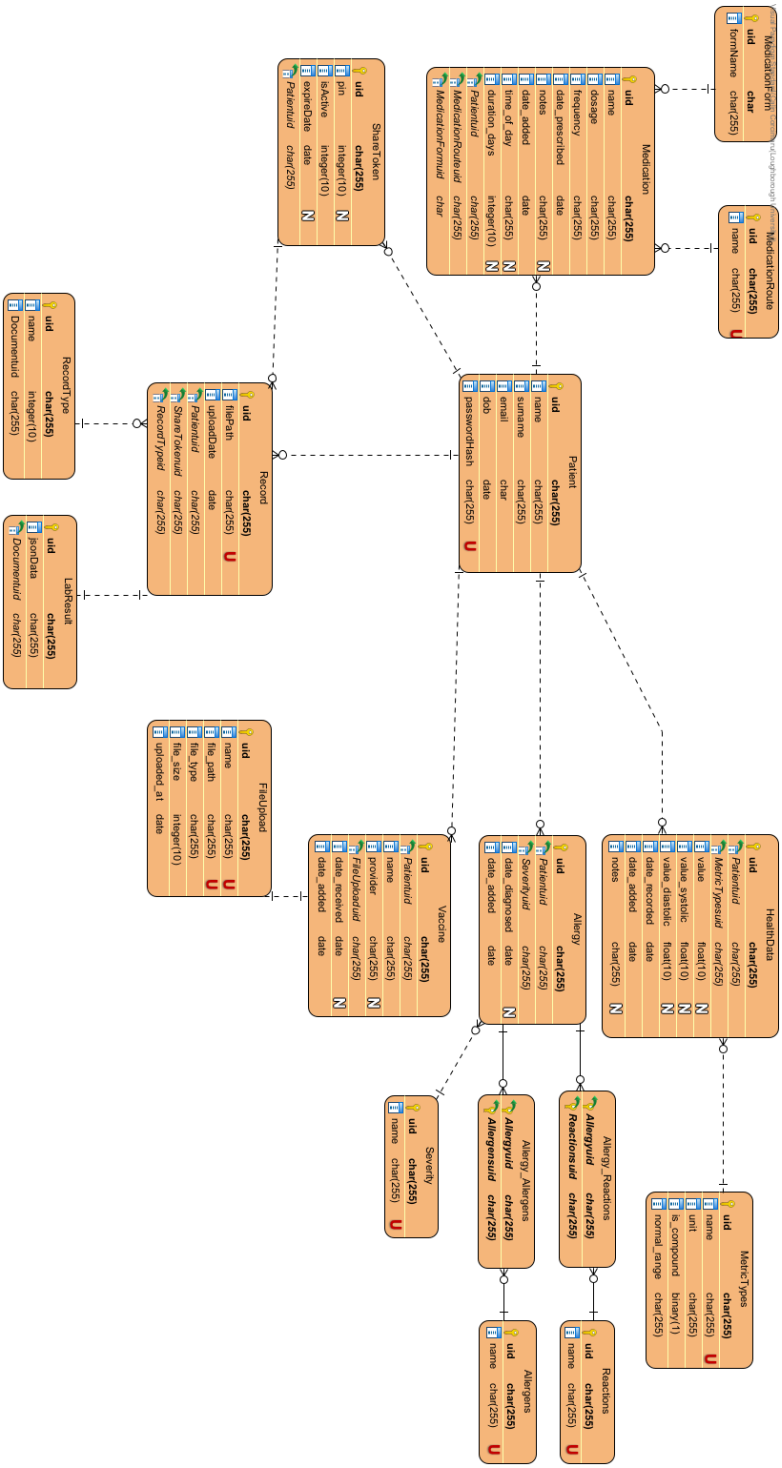


Figure 5.11: Updated Entity Relationship Diagram Sprint #3

Other changes included multiple changes to the API endpoints, mainly relating to the vital signs data, due to the addition of the new table and its contents. Similarly, a new endpoint for the dashboard had to be created, which would use the newly added date_added field to sort through the 5 most recent records in each category and send them back as the response. Below can be found the code that sorts and selects the 5 most recent records for each category:

```
@app.get("/dashboard", response_model=UserDashboard)
async def get_dashboard(user_id: uuid.UUID = Depends(validate_session)
    , session: Session = Depends(get_session)):
    user = session.get(User, user_id)

    if user_id != user.id:
        raise HTTPException(status_code=403, detail="You do not have
    permission to access this endpoint!")

    newest_vaccines = session.exec(select(Vaccine).where(Vaccine.
    user_id == user_id).order_by(col(Vaccine.date_added).desc()).limit
    (5)).all()
    newest_allergies = session.exec(select(Allergy).where(Allergy.
    user_id == user_id).order_by(col(Allergy.date_added).desc()).limit
    (5)).all()
    newest_healthdata = session.exec(select(HealthData).where(
    HealthData.user_id == user_id).order_by(col(HealthData.date_added)
    .desc()).limit(5)).all()
    newest_medications = session.exec(select(Medication).where(
    Medication.user_id == user_id).order_by(col(Medication.date_added)
    .desc()).limit(5)).all()

    # Code that transforms each record into a response object...

    user_dashboard = UserDashboard(
        id = user.id,
        name = user.name,
        vaccines = vaccines_response,
        allergies = allergies_response,
        vitals = healthdata_response,
        medications = medications_response
    )

```

```
24      return user_dashboard
```

Listing 5.16: Dashboard Endpoint

5.3.6 Challenges

Scope Creep

One of the main challenges encountered in this sprint was the scope creep and the change in requirements. While an initial discussion was done with stakeholders and wireframes were created to have a shared understanding regarding how the system should look, further discussions brought up areas of improvement and change opportunities within the system. This led to additional work being done on both the frontend and backend to ensure the system was adhering to stakeholder expectations, and that it provides a pleasant user experience and enough functionality to be a viable solution for the market.

Infrequent communication with stakeholders

One of the main problems encountered in this sprint and the previous ones was the infrequent communication with stakeholders. This was caused by many factors, such as the busy schedules of both the student and the stakeholders, their other commitments such as studies or other work positions, but also the fact that all communication was done virtually, limiting the interaction to meetings or messages via communication apps. This infrequent communication led to situations where the student might have had to make decisions regarding the system design on their own, which could not have been liked by stakeholders, requiring changes or re-doing, adding additional work load on the student.

As such, it was clear that for the upcoming sprints, a more constant communication between the student and stakeholders needed to be established to ensure that what was worked on was in line with the stakeholders' expectations.

Lack of proper testing procedures

Due to the complexity of the system and time pressure, barely any kind of automated testing was implemented. This lead to unexpected issues appearing in the course of development, which created additional work for the student, changing their focus from the sprint items to fixing the bugs, thus delaying the sprint completion. The student also doesn't have much experience creating tests for the frontend, which made the testing process even more difficult, leading to many of the testing to be done manually or skipped entirely. Due to the limited time remaining, it might be necessary to dedicate a part of the upcoming sprints to do some

more thorough testing to ensure that the system as a whole does not have any critical bugs or issues that may affect its functionality.

5.3.7 Requirements completed

- The system must have a dashboard view which displays an overview of the most recent information added to the system (latest lab tests, doctor consultations, vaccinations etc).
- The system should display the health records in both a list or grid view.
- The system should allow the patient to enter vitals information, such as height, weight, blood pressure, etc.
- When multiple vital entries are made, the system could display a historical graph of the patient's vitals.

5.4 Sprint #4

The fourth sprint of the project focused on adding the ability to add health records, such as lab tests, doctor consultations or imaging results. This was one of the main features requested by the stakeholders, as it would allow a patient to have all their medical history in one place, making it easier for them to view and share it with their doctors.

Additionally, at the end of sprint #3, the stakeholders provided feedback regarding the features implemented in the previous sprint. Some of the feedback included:

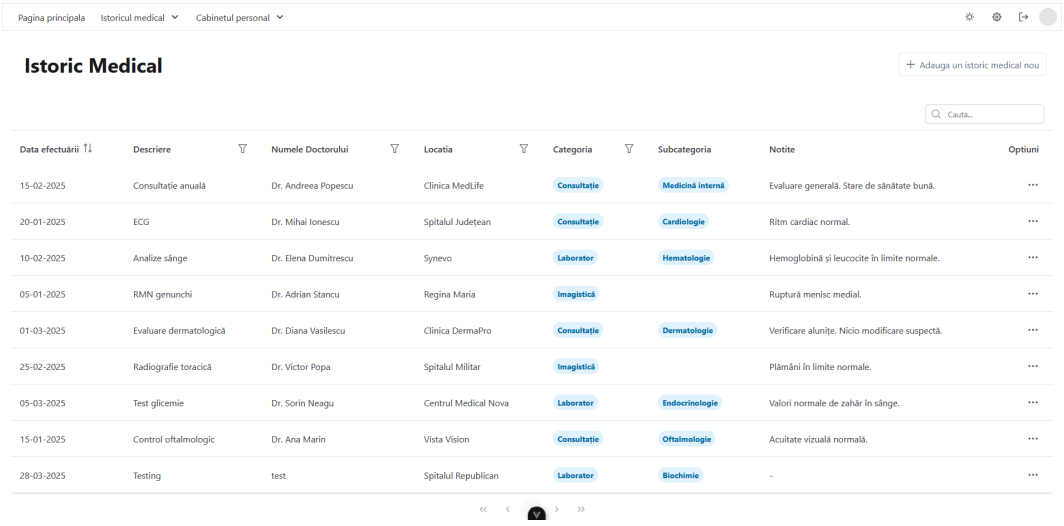
- The graph in the Vitals page should show the normal range for that specific vital type in graph.
- The Vitals signs page should have a date filtering functionality to choose a date range for the signs displayed in the graph or table.
- Weight and height should not have a normal range.
- Previously, the trend most recent vitals signs were compared with the 2nd most recent ones, which was not a good choice for comparison. The most recent vitals signs should be compared with the normal range for that specific vital type.
- On the main dashboard, the most recent vitals section should show the most recent values for one each type of vital signs, instead of the most recent 5 values for each type.

- On the main dashboard, only the allergies with moderate and severe allergies should be displayed.

5.4.1 Health Records

As mentioned in the introduction to this section, the main focus of this sprint was creating the functionality to add health records such as lab tests, doctor consultations or imaging results in the system. This was a crucial feature and a main requirement of the system, as it would allow patients to create their medical history in the system, upload the files associated with these records and then refer to them whenever needed or share them with doctors to allow them to have a more accurate patient history and thus offer better treatment.

The choice was made to display the record history in a tabular format, as that would allow to display the important information in a readable way, but also to sort the information based on different fields such as the date of the consultation or filter by fields such as doctor or institution name for easier search. PrimeVue has a DataTable component, which allowed the inclusion of all these features with little effort that are displayed in a user-friendly way. An example of the health records page can be seen below in figure 5.12.



(a) Desktop version

(b) Mobile version

Figure 5.12: Desktop and Mobile version of the History page

DataTable also comes with an in-built filtering option, which allowed to add a search bar for the whole table, but also search functions for specific fields. Example of the search bar can be seen in the screenshots above, and the filter menu can be seen below in figure 5.13. A code snippet of the filter function in a column can also be seen below.

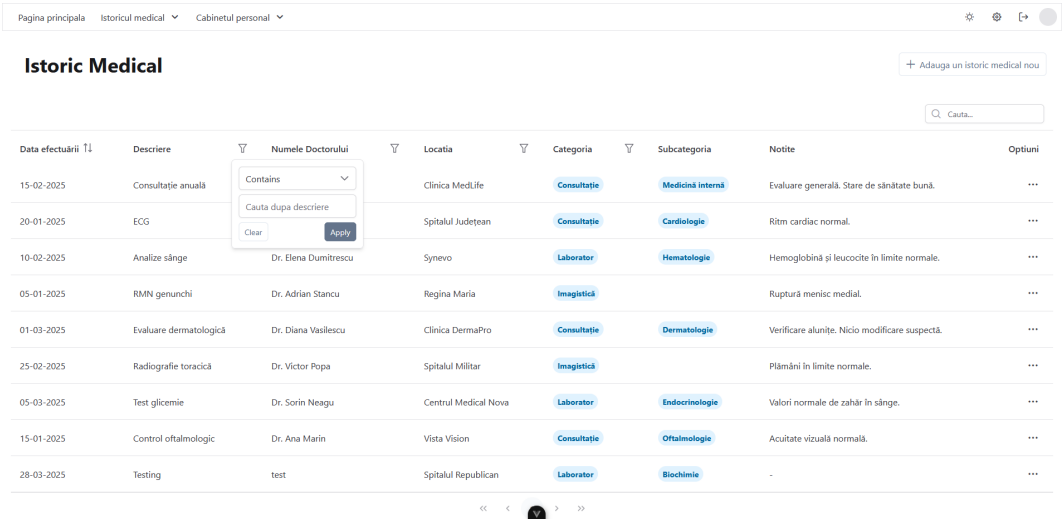


Figure 5.13: Filter Menu for Health Records

```
<template>
<div class="h-full w-full px-4">
  <DataTable
    v-model:filters="filters"
    :value="history"
    :rows="10"
    paginator
    dataKey="id"
    filterDisplay="menu"
    :loading="loading"
    :globalFilterFields="['name', 'doctor_name', 'place', 'category
  ']"
    removableSort
    responsiveLayout="stack"
    breakpoint="960px"
  >
    <template #header>
      <div class="flex justify-end">
        <IconField>
```

```
            <InputIcon>
              <i class="pi pi-search" />
            </InputIcon>
            <InputText size="small" v-model="filters['global'].value"
    placeholder="Cauta..." />
          </IconField>
        </div>
      </template>

      <!-- Previous columns here -->

      <Column field="doctor_name" header="Numele Doctorului" :
    showFilterMenu="true">
      <template #filter="{ filterModel, filterCallback }">
        <InputText v-model="filterModel.value" type="text" class="w-
    full" @input="filterCallback()" placeholder="Cauta dupa doctor" />
      </template>
    </Column>

    <!-- Next columns here -->
    </DataTable>
  </div>
</template>
```

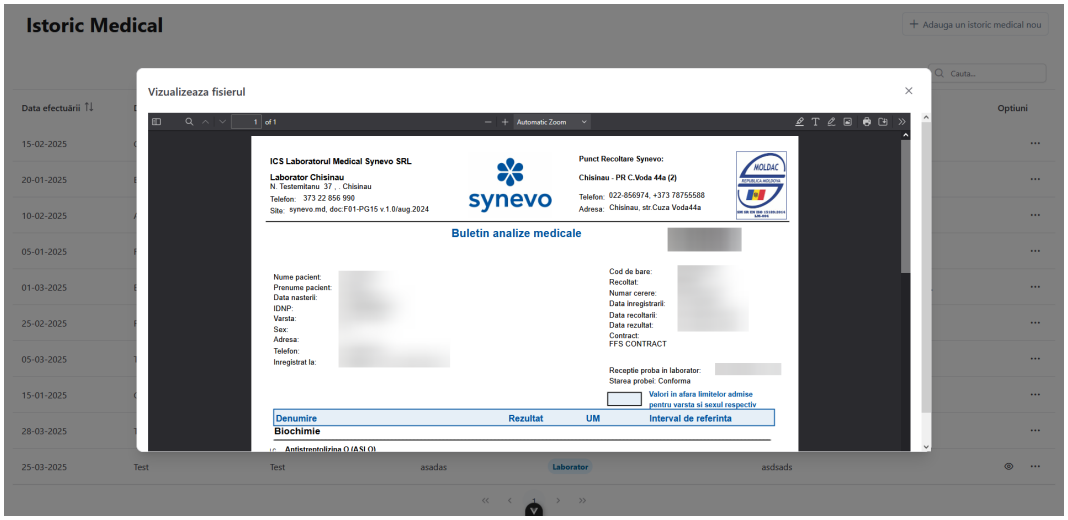
Listing 5.17: Filter Function for Health Records

Finally, with the addition of health records page, the ability to view and upload PDF files was also added to the system. This was an important feature to add, as files related to health records are usually sent in a PDF format. There was a slight challenge as to how to display the PDF files in the frontend, however in the end it was decided to use a simple approach that utilised in-built functionality through the use of object element from HTML. To allow mobile users to view the files, the system instead used a button to allow mobile users to download the file, as mobile browsers do not support the object element. The code for the PDF viewer can be seen below and a screenshot in figure 5.14.

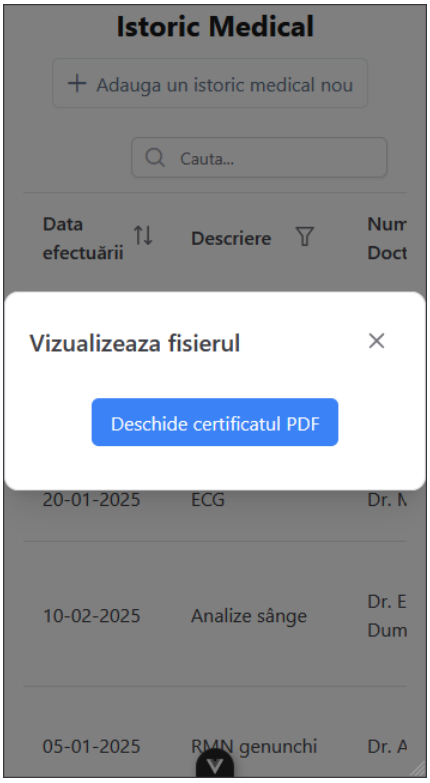
```
<Dialog
  v-model:visible="visible"
  modal
  header="Vizualizeaza fisierul"
  class="w-full md:w-3/4 md:h-3/4"
  @hide="emit('close')"
```

```
  >
    <div v-if="metadata?.file_type == 'application/pdf'">
      <object
        :data="`http://localhost:8000/files/medicalhistory/${props.historyId}`"
        type="application/pdf"
        class="w-full h-[70vh] hidden md:block"
      />
      <div class="block md:hidden text-center mt-2">
        <a
          :href="`http://localhost:8000/files/medicalhistory/${props.historyId}`"
          target="_blank"
          class="bg-blue-500 text-white px-4 py-2 rounded-md inline-block mb-4"
        >
          Deschide certificatul PDF
        </a>
      </div>
    </div>
    <Image
      v-else
      :src="`http://localhost:8000/files/medicalhistory/${props.historyId}`"
      alt="Fisier consultatie"
      class="w-full h-screen"
      preview
    />
  </Dialog>
</template>
```

Listing 5.18: PDF viewer Function for Health



(a) Desktop version



(b) Mobile version

Figure 5.14: Desktop and Mobile version of the PDF viewer

5.4.2 Backend changes

To enable the health record functionality to work properly, new tables had to be created in the database to store the records: MedicalHistory for the main table, MedicalCategory for main categories of the records (lab tests, doctor consultations, etc) and additional tables like MedicalSubCategory and LabSubcategory for the subcategories of the records (blood test, urine test, etc). The new tables can be seen below in figure 5.15.

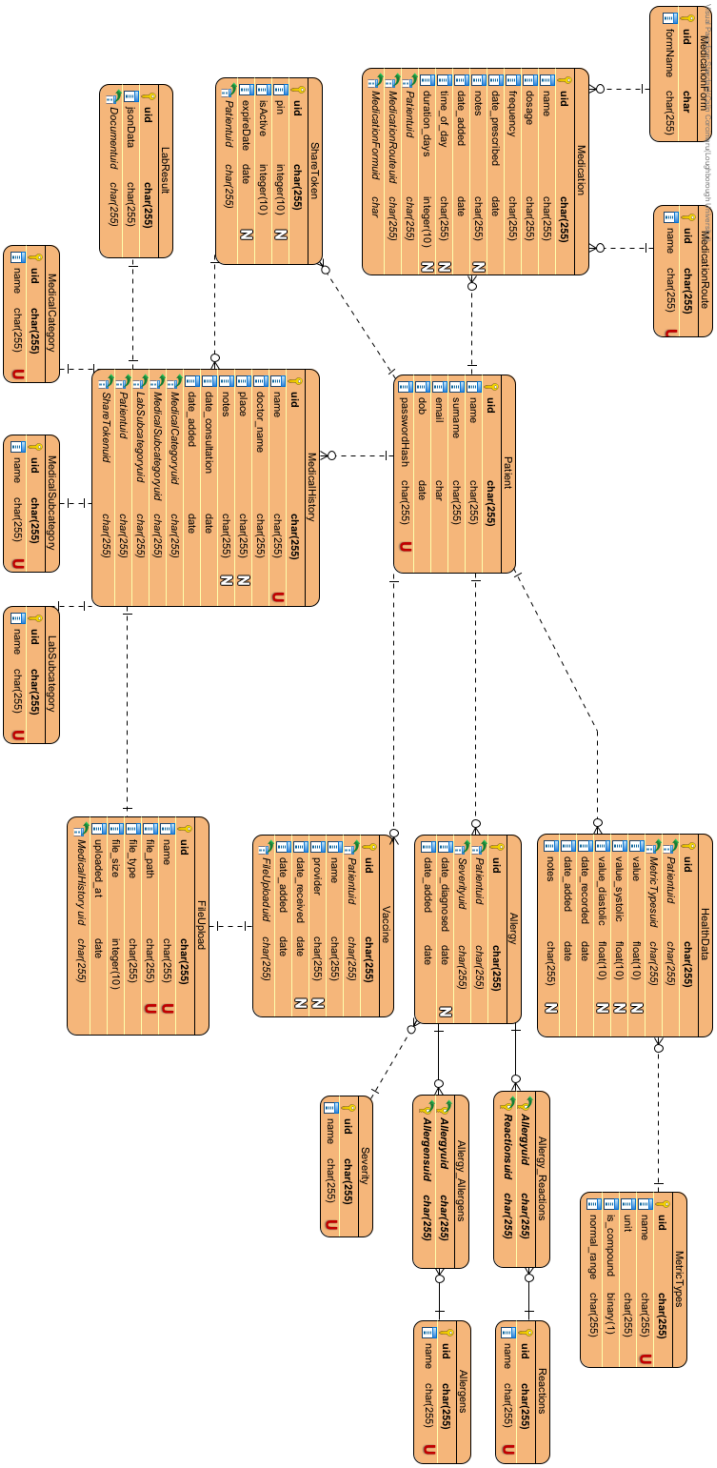


Figure 5.15: Updated Entity Relationship Diagram Sprint #4

Additionally, endpoints were created to allow the user to add, edit and delete health records. The endpoints were created in a similar way to the other endpoints in the system, using FastAPI and SQLModel.

5.4.3 Work on stakeholder feedback

As mentioned in the section introduction, in this sprint work was done to address the feedback received from stakeholder meetings regarding the features implemented in the previous sprint.

The biggest changes were related to how the vital signs are displayed and filtered in the system. The graph now shows lines for the upper and lower ranges for the respective vital signs, and a date range filter was added to the page, allowing the user to select a date range for the data displayed in the graph.

Additionally, the most recent vitals are now compared to the normal range instead of the previous value. This change was implemented to ensure the normal range is used effectively to show whether their vitals signs are within or outside of it to allow for more informed decision making. These changes can be seen below in figure 5.16.

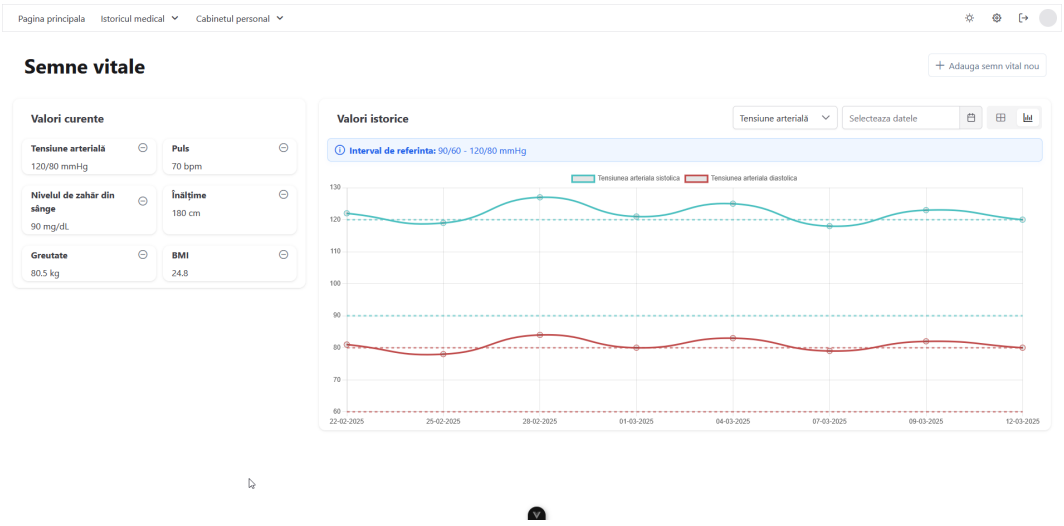


Figure 5.16: Updated Vitals View based on stakeholder feedback

The main dashboard page was also slightly changed to show the most recent vitals signs for each type of vital sign, instead of the most recent 5 values for each type. This was done to ensure that the dashboard page is not cluttered with too much information and that the user can easily see their most recent vitals signs.

Finally, the allergies section on the dashboard page was also changed to only show the moderate and severe allergies, as it was suggested by the stakeholders that this would be more useful for patients to track their most important allergies. The new dashboard can be seen below in figure 5.17.

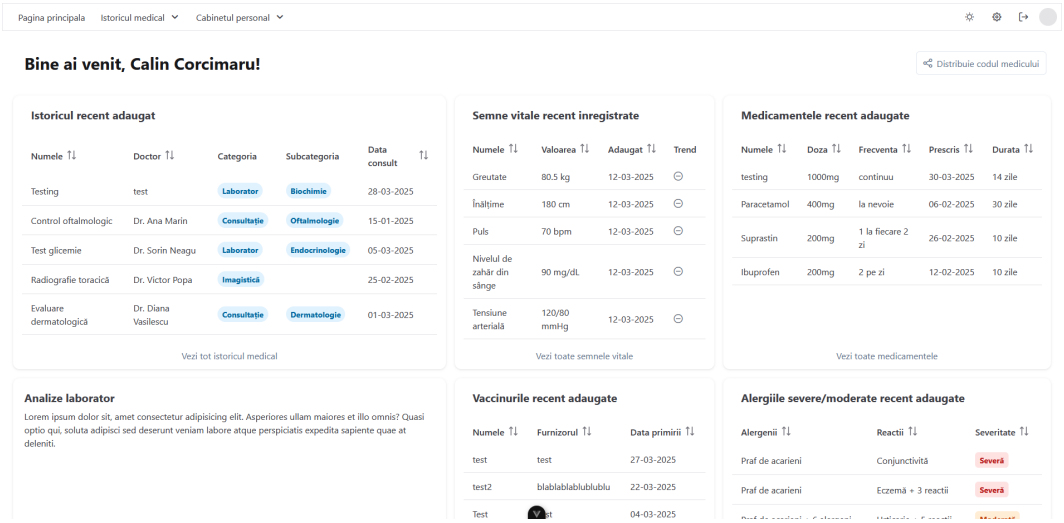


Figure 5.17: Updated Dashboard View based on stakeholder feedback

5.4.4 Challenges

The main challenge encountered in this sprint was actually not related to the project itself. Instead, the biggest challenge that was presented in this sprint was sickness, which led to the student being unable to work on the project for a week. This limited the amount of work that could be done in this sprint, leading to some of the preparatory work that was planned for this sprint to be moved to the next one.

5.4.5 Requirements completed

- The system must allow the patient to specify and categorise the type of document they are uploading (lab test, doctor consultation, etc).
- The system must allow patients to upload their own medical records in a variety of formats (PDF, DOC, etc).
- The system must display the patient's history in a chronological order in the form of a timeline.

- When viewing doctor consultations, the system should divide them into categories based on the domain of the doctor (cardiology, neurology, etc).
- The system must allow the patient to add a description of the document they are uploading.
- The system should allow the patient to add a date for the document they are uploading.
- The system should allow the patient to add a location for the document they are uploading.
- The system should allow the patient to add the doctor name for the document they are uploading.
- The system should allow the patient to sort and filter the documents based on the type of document, date, location, and doctor name.

5.5 Sprint #5

The fifth sprint of the project was focused on adding the ability to add lab tests and extract the lab result values from those documents. The lab results would've been extracted by using a multimodal LLM, which would allow the system to precisely collect the lab results alongside other information such as units, reference ranges, etc without any additional human or system intervention. Using a MLLM would also simplify the extraction as the model would be able to export the results in an easily parseable response, such as JSON, which could be then used to display the information on the frontend or be processed in the backend. This feature was probably one of the most important ones in the whole system, as it represented an innovative way of automating the collection and storage of lab results for patients.

This feature was a continuation of the work done in the previous sprint, as the lab tests were also part of the health records feature. As such, some of the initial work was done on top of existing components, which made development easier and faster. The feature was developed as per the initially agreed requirements and wireframes, with some changes to how the lab results were displayed on the frontend.

The sequence diagram for the lab tests extraction can be seen below in figure 5.18. The diagram shows how the user interacts with the system to upload a lab test, how the system processes the file and sends it to the MLLM API, and how the response is then displayed to the user.

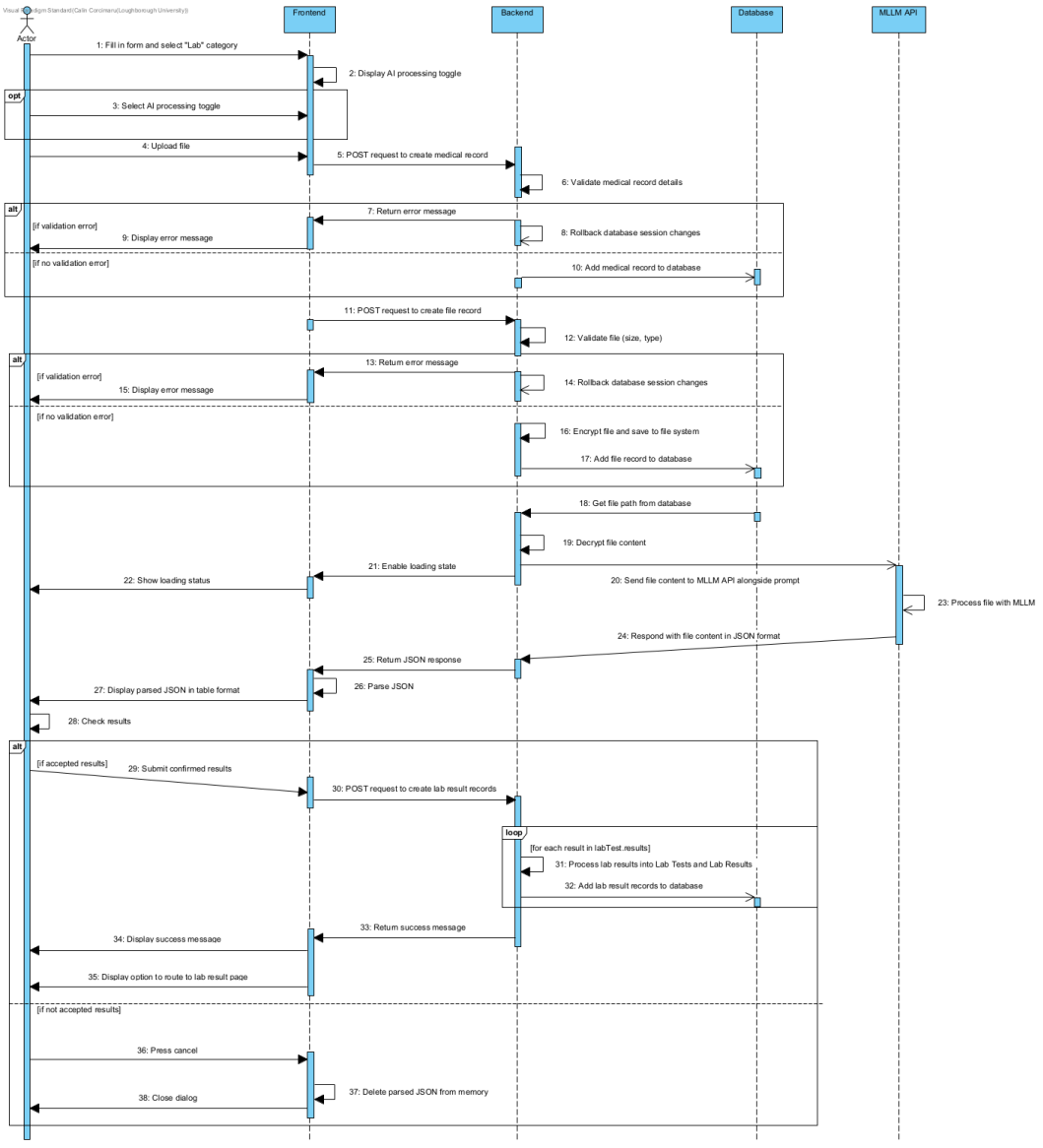
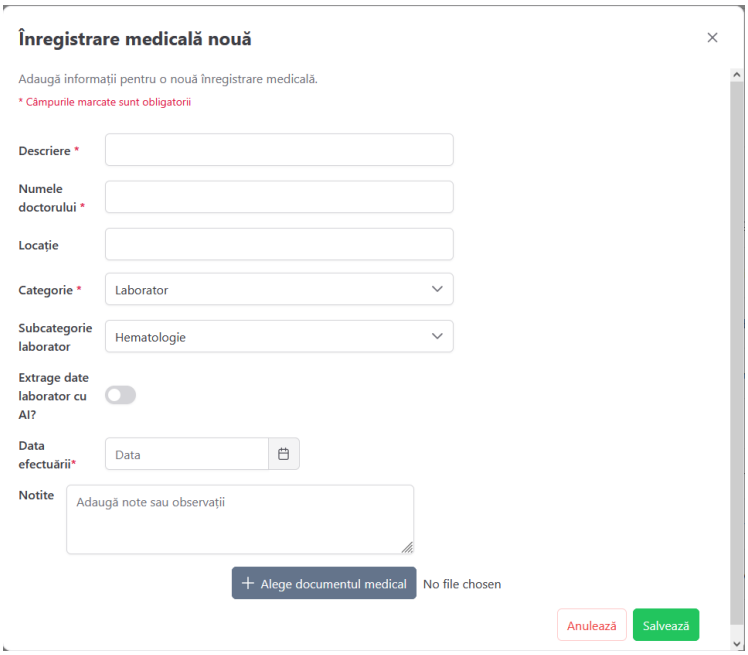


Figure 5.18: UML Sequence Diagram — Extracting Lab Results

5.5.1 Lab Tests Extraction

As lab tests were previously added in sprint #4 as a health record, it was decided to use the existing 'Add Health Record' component to allow the user to choose whether they'd like to extract the lab results besides uploading the file to the system. The decision was made to streamline the user experience of adding new health records, as the user would not have to go through multiple steps to add a lab test, but also because it would speed up development time as an existing component could be reused and made more feature-rich. Finally, this also closely matched the initial wireframe design for the 'New Document' dialog, which can be seen in figure B.5.

The updated dialog can be seen below in figure 5.19. The main change that can be seen is the addition of a toggle switch, which is displayed only when the user selects a lab test as the document type. If used, the toggle would still send the lab file to the backend and save it as a health record, but in addition to this, it would also send it to the MLLM API to be processed.



(a) Desktop version

(b) Mobile version

Figure 5.19: Desktop and Mobile version of the new Add Document dialog

After the record is created and file is uploaded, the backend would find the file and send it the MLLM API. This was done to avoid uploading the file twice and to ensure the file would be successfully uploaded to the system before sending to the API.

The MLLM chosen for this project was Google's Gemini 2.0 Flash, which, as of writing this section, is part of the newly released family of Gemini models (**gemini2**). The choice was made based on the list of available MLLMs that can be accessed via APIs with a free tier from table 3.4. Besides Google's AI Studio, none of the API providers allowed the uploading and processing of files, which made Gemini 2.0 Flash the most suitable choice for the project. It was noted that a local MLLM could've also been used for the project, a point which will be discussed in more detail in section 7.2.

Because an API was used, most of the work with the MLLM involved using Prompt Engineering techniques to ensure the model would process and return the data in the desired format. Prompting techniques from previous section 3.4.3 were used to write the following prompt that was used to extract the lab results from the file:

```
1
2  def extract_with_llm(file_content: bytes, file_type: str):
3      prompt = """
4      You have been given a document that contains lab results that is
         written in the Romanian language. Your job is to extract all lab
         results from this document in JSON format with the following
         fields: test_name, test_code, value, unit, reference_range, method
         . Sometimes the code of the test will be in the name itself, and
         it is your job to determine if the code is there, for example in
         brackets or separated by a comma, and separate the name and the
         code. Sometimes the lab result will not have a method specified,
         and in that case you return an empty string. The document is in
         Romanian, however the JSON keys should be in English.
5
6      EXTREMELY IMPORTANT FORMATTING INSTRUCTIONS:
7      1. Return ONLY the raw JSON array
8      2. DO NOT use code blocks, backticks, or markdown formatting
9      3. DO NOT include ```json or ``` anywhere in your response
10     4. DO NOT include any explanations or text before or after the JSON
11     5. Your response must start with the '[' character and end with the
         ']' character
12     6. The output should be valid JSON that can be parsed directly
13     7. Use period (.) as the decimal separator, not comma (,)
14
```

```
Example of how your output should look, starting from the very first
   character:
[{"test_name":"Hemoglobina","test_code":"HGB","value":"14.3","unit":
  "mg/dL","reference_range":"13.2-17.3", "method":"Chemiluminiscenta
  "}]

if there is no method specified, return an empty string:
[{"test_name":"Hemoglobina","test_code":"HGB","value":"14.3","unit":
  "mg/dL","reference_range":"13.2-17.3", "method":" "}]
"""

response = client.models.generate_content(
    model="gemini-2.0-flash",
    contents=[prompt,
            types.Part.from_bytes(
            data=file_content,
            mime_type=file_type
            )
        ])

return response.text
```

Listing 5.19: Prompt for Lab Results Extraction

The response from the API would be a JSON array containing all the lab results extracted from the document. Before saving, it is important to make sure the results and their values are correct. As such, it was decided to display all the extracted results back to the user in a PrimeVue DataTable component, which allowed for easy filtering of the results, but also for row-based editing. This way, the user could easily edit the values or remove any unwanted results before saving them to the system. An example of how the extracted lab results dialog looks can be seen below in figure 5.20.

A benefit of using PrimeVue's DataTables was not only its row editing feature or its filtering options, but also its ability to dynamically create columns based on the data received. This was perfect for the lab results, as it is not possible to know beforehand how many results will be extracted from a document. A code snippet of the DataTable can be seen below:

```
<DataTable
  :value="extractionResult"
  v-model:editingRows="editingRows"
  class="w-full"
```

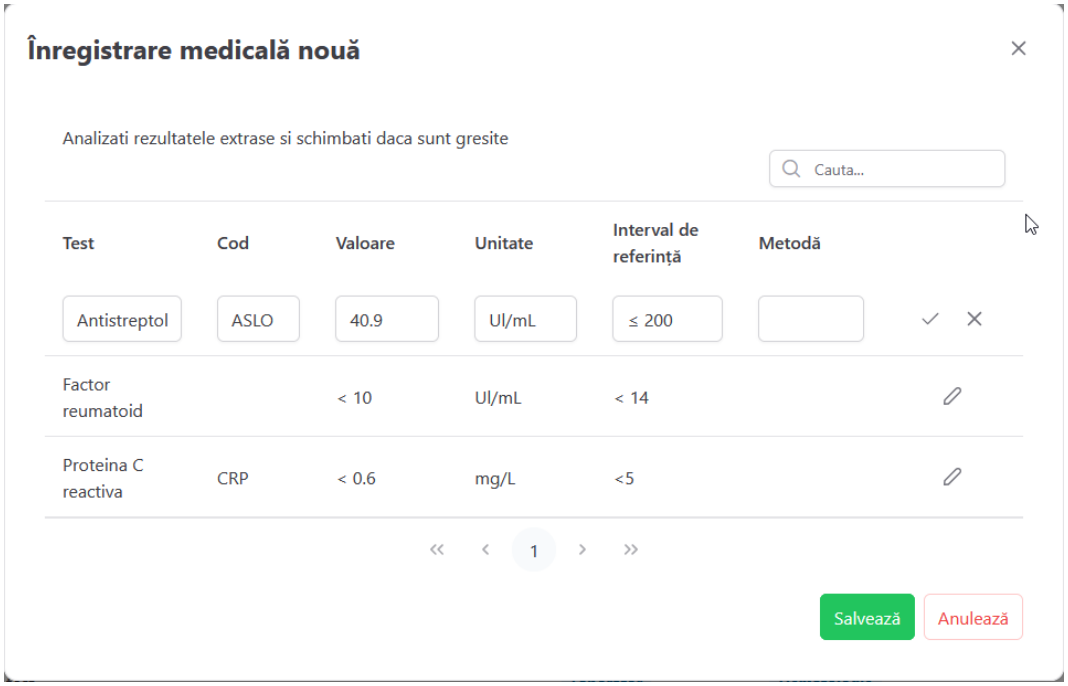


Figure 5.20: Desktop dialog validation of extracted lab results

```
    :rows="10"
    editMode="row"
    paginator
    dataKey="test_name"
    @row-edit-save="onRowEditSave"
    v-model:filters="filters"
    :globalFilterFields="['test_name', 'test_code']"
  >
    <template #header>
      <h2 class="m-0">Analizati rezultatele extrase si schimbati daca sunt gresite</h2>
      <div class="flex justify-end">
        <IconField>
          <InputIcon>
            <i class="pi pi-search" />
          </InputIcon>
          <InputText
            size="small"
            v-model="filters['global'].value"
```

```
            placeholder="Cauta..."
          />
        </IconField>
      </div>
    </template>
    <Column
      v-for="col of columns"
      :key="col.field"
      :field="col.field"
      :header="col.header"
    >
      <template #editor="{ data, field }">
        <InputText v-model="data[field]" class="w-full" fluid />
      </template>
    </Column>
    <Column
      :rowEditor="true"
      style="width: 10%; min-width: 8rem"
      bodyStyle="text-align:center"
    />
  </DataTable>
```

Listing 5.20: Dynamic DataTable for Lab Results

After confirming the extracted test results, they are processed and saved to the backend. The next subsection will discuss in more detail how the backend was changed to accommodate the new lab tests and their results.

Backend changes

To supporrt the processing and addition to database for the newly extracted lab results, two new tables were created in the database: LabTest and LabResult. The decision to use 2 tables was made to avoid duplication of data for fields like name and code of the lab test, as these would be common for all the lab results, regardless of their method or machinery used. This would also prove beneficial as multiple results could be assigned to one test, making the tracking and display of the historical results much easier. Finally, by having 2 separate tables, LabTests could be dynamically populated with new tests straight from uploaded documents, without the need for manual addition or creation of test types. The updated database schema can be seen below in figure 5.21.

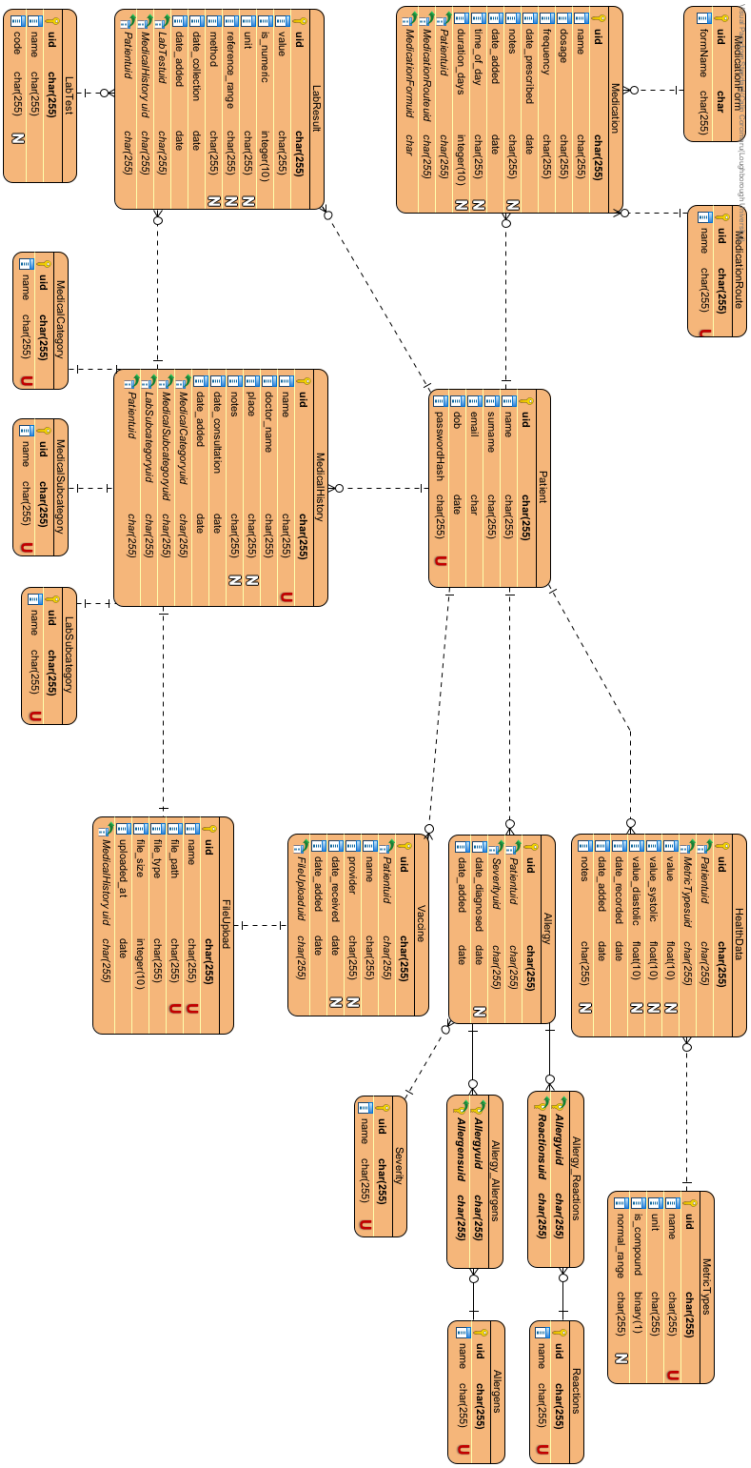


Figure 5.21: Updated Entity Relationship Diagram Sprint #5

As can be seen by the diagram above, there might seem to be a redundant relationship between the Patient and LabResult tables, as LabResult is already related to MedicalHistory, which in turn is related to a Patient. However, the decision was made to introduce the relationship as it would allow for easier querying when getting all the lab results for a patient, in the case of the dashboard view or even in the next sprint when the share functionality is added.

Similarly, a code snippet of the endpoint that handles the creation of lab results can be seen below:

```
  # Create the lab test records in the database
@router.post('/me/labtests/')
async def create_lab_tests(extraction_result: LabsCreate, user_id: uuid.UUID = Depends(validate_session), session: Session = Depends(get_session)):
    medhistory = session.get(MedicalHistory, extraction_result.medicalhistory_id)
    user = session.get(User, user_id)

    if not medhistory:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND, detail="Medical history not found")

    if medhistory.user_id != user_id:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail="Not authorized to access this medical history")

    for lab_item in extraction_result.lab_tests:
        lab_test = session.exec(select(LabTest).where(LabTest.name == lab_item.name)).first()

        if not lab_test:
            lab_test = LabTest(
                name = lab_item.name,
                code = lab_item.code,
            )
            session.add(lab_test)
            session.flush()

        lab_result = LabResult(
            value = lab_item.value,
```

```
            is_numeric = check_is_numeric(lab_item.value),
            unit = lab_item.unit,
            reference_range = lab_item.reference_range,
            date_collection = extraction_result.date_collection,
            test = lab_test,
            medicalhistory = medhistory,
            user = user,
            method = lab_item.method
        )

        session.add(lab_result)
        session.flush()

    session.commit()

    return {
        "status": status.HTTP_201_CREATED,
        "message": "Lab tests created successfully"
        }
```

Listing 5.21: Lab Results Endpoint

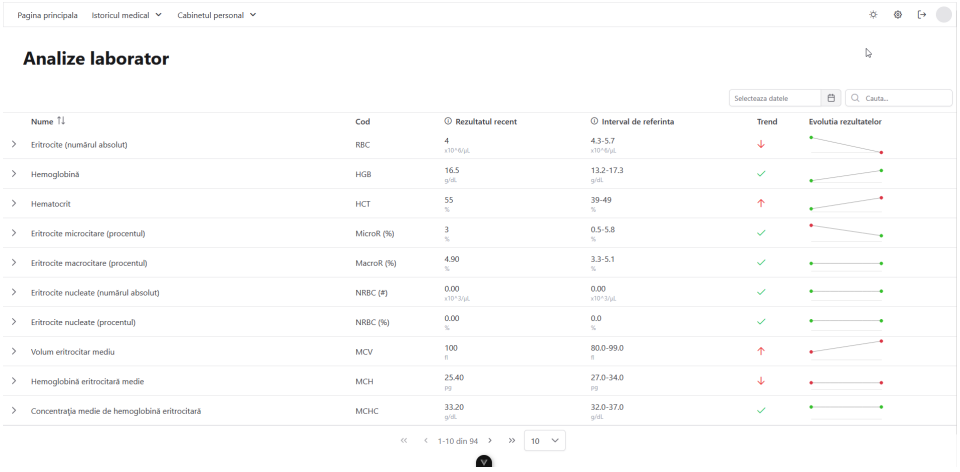
5.5.2 Lab Results Display

The final part of the work on the lab extraction feature was the display of said results within their own page in the system. The initial idea was to reuse the existing design from the vitals page, which can be seen in figures 5.8, 5.9 and 5.16. This would've also followed the initially agreed wireframes and design.

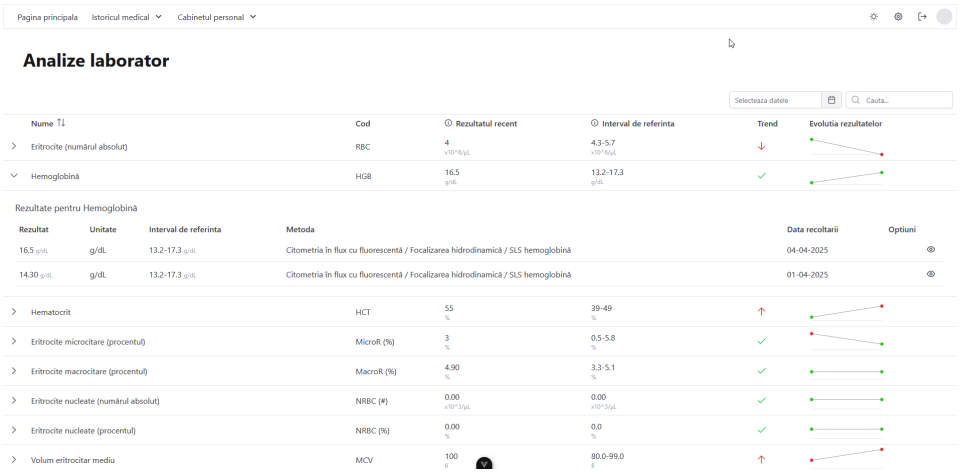
However, upon discussing with stakeholders, an idea and request to change the design was floated in the meeting: instead of using the design from vitals page, the idea was to display everything in a tabular format where rows could be expanded to show the historical resutls from that specific test. This is where the division into 2 tables in the database played an important role: LabTest could be used as the 'parent' element in the table, which could be sorted and filtered, while the LabResult elements would only be shown if the row would be expanded in the table.

This idea was eventually accepted, as it would allow for a more compact view of the lab results, easier filtering/sorting and it enabled for a way to display the most recent results for each test as a separate column without being intrusive. The results were sorted by date in the backend already, which allowed for the most recent results to be easily selected and

displayed. Finally, instead of using a big graph that might take up a lot of space on the page, it was decided to use a small graph, also known as a sparkline, which would only show the trend of the results over time, based on their reference range and collection date. Colors such as red and green were used to indicate whether the results were within the normal range or outside of it. An example of how the new lab results page looks can be seen below in figure 5.22 and 5.23.

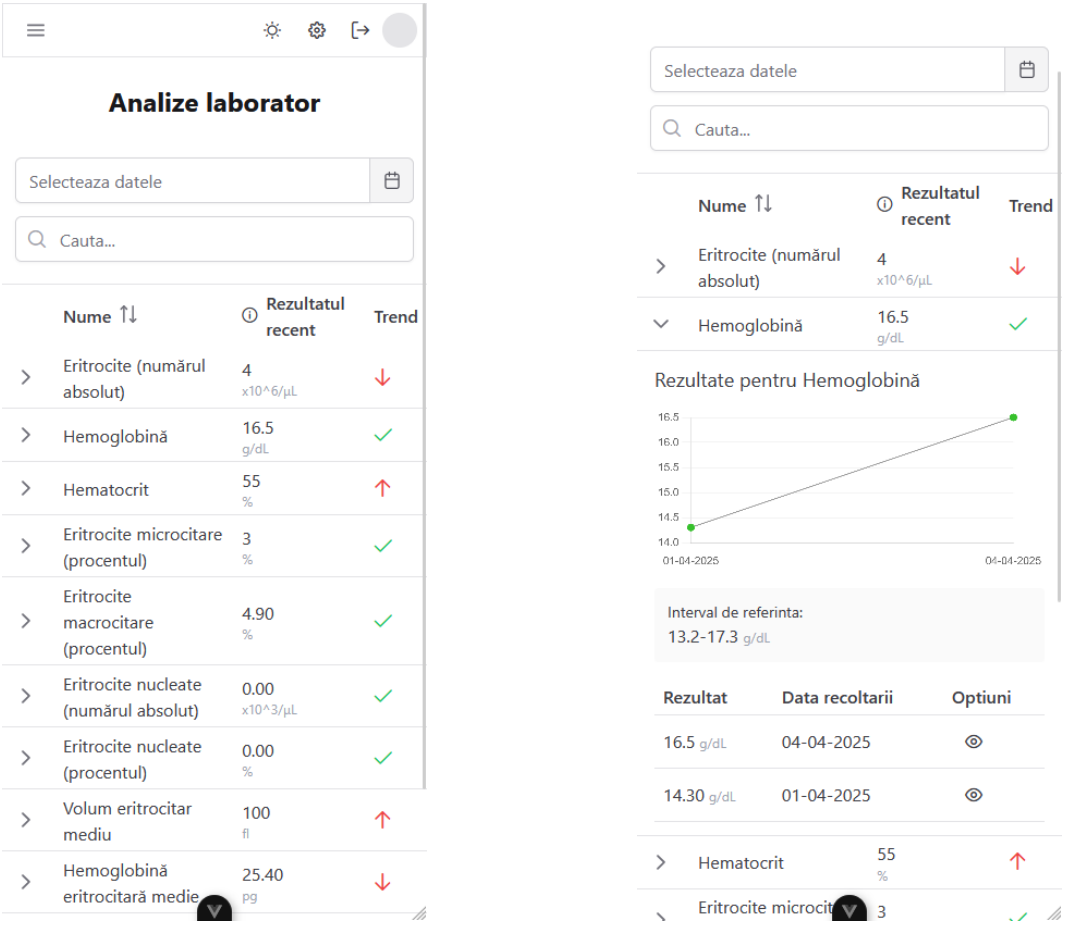


(a) Desktop version



(b) Desktop version #2

Figure 5.22: Desktop version of the new Lab View page



(a) Mobile version

(b) Mobile version #2

Figure 5.23: Mobile version of the new Lab View page

5.5.3 Challenges encountered

This sprint encountered less challenges than the previous ones, meaning that the work was able to be done faster than it was initially planned. However, there were still some challenges that were encountered during the development of the lab tests extraction feature.

One of the biggest challenges was the change in design for the lab results page. Even though the initial design was agreed upon before the development begun, the stakeholders insisted on the change in design, which added a slight overhead in the form of additional design and research on how to implement the new idea. However, it must be noted that the new design also allowed for an easier development experience as a lot of the work was done by

PrimeVue's DataTable component, which ultimately sped up the completion of the feature. Ultimately, the agility of the development process allowed for the change to be made without any major issues, an idea which will be further discussed in section 7.2.

The other challenge was the time pressure, as the student was nearing the deadline of the project. This meant that the focus was on implementing only the most necessary and basic features, without any space for creative or innovative ideas.

5.5.4 Requirements completed

- The system must provide an overview of the patient history through 3 main sections: personal information, lab tests, and doctor consultations.
- The system must allow the doctor to view blood tests in a graphical format.
- The system must allow the doctor to view blood tests in a numerical, tabular format.
- The system must allow the doctor to view the patient's history in a chronological order.
- For blood test results, the system should display the normal range values for each test.
- The system could allow the patient to switch between viewing the lab tests in the document format or in a tabular, numerical format.

5.6 Sprint #6

The last sprint of the project was focused on adding the ability to share the health records with doctors. This was one of the most important features of the system, as it allowed patients to share their medical history with doctors, which was one of the system's main goals of solving the problem of having so many different institution and systems that silo the patient information, without a way to share it.

The purpose of the share link feature was to allow doctors to access the patient's accrued medical history during consultations. It is not aimed to replace their institutions' systems, but rather as a complimentary service that would enable them to view information that might not be available to them, thus creating a more complete picture of the patient's medical history.

This was the hardest feature to implement, as it would combine all the work done in the previous sprints and required designing a way to create these shareable links that would be both secure and easy to use.

5.6.1 Share links

In the initial discussions with stakeholders, it was quickly established that the easiest way to share medical information with healthcare practitioners would be through the use of a shareable link. Since most doctors have access to internet and a computer at their desks, it would represent the fastest and easiet way to access this system. Share links are also very versatile, as they could be sent via email, written on a paper, dictated or even generated as a QR code that can be scanned. As such, one of the first challenges was to design these shareable links so that they would be easy to transmit, but also secure and not easily guessable.

Initially, the idea was to use UUID as the share code, as their length (usually around 36 characters) and randomness would make them very hard to guess. However, this would create a problem when sharing the link, as it would be extremely hard to share unless it is directly copied and pasted. As such, the decision was made to use a string of 8 randomly generated characters as the share code, which would be part of the URL. An example of a share link would be 'https://example.com/share/7zSaQbBd'. The code for the share link generation and share token model can be seen below:

```
  # Helper function to generate random strings for share codes
def generate_random_string(length: int) -> str:
    characters = string.ascii_letters + string.digits
    return ''.join(secrets.choice(characters) for _ in range(length))

# Share Token model for database, used to store information about
    generated share tokens. Share token is the main object that is
    used to share health data with other doctors.
class ShareToken(SQLModel, table=True):
    id: uuid.UUID = Field(default_factory=uuid.uuid4, primary_key=True
    )
    share_code: str = Field(index=True, unique=True, default_factory=
    lambda: generate_random_string(8))
    expiration_time: datetime
    created_at: datetime = Field(default_factory=datetime.now)
    hashed_pin: str
    shared_items: dict = Field(default_factory=dict, sa_column=Column(
    JSON))

    user_id: uuid.UUID = Field(foreign_key="user.id")
    user: "User" = Relationship(back_populates="share_tokens")
```

Listing 5.22: Share Link Generation

To ensure the share links are secure, additional measures were also implemented. One of them was an expiration time, which by default would be set to one hour, with a maximum of 24 hours. As previously mentioned, these links' primary purpose was to complement a doctor's available information during a consultation, s3ince every institution has its own system for storing health information. As such, it was reasonable to assume that they would only be used once, during that visit. To this end, the expiration time was limited to a maximum of 24 hours to avoid share links being leaked or used by unauthorized people.

The other measure was the use a user-set 6 character-long PIN code that would be separate for each share link. The patient would set this PIN when creating the share link, and the doctors would be prompted to enter whenever they wanted to access the shared information. This was added to ensure that, in the event of a share link being leaked, the information would at least be protected by the PIN code. The PIN would be hashed before stored in the database with the same algorithm used for user passwords, to ensure PINs would not be easily accessible in the database.

When combined, these 3 features, randomly generated share code, expiration time and PIN would ensure a secure way to access the information: when accessed, each link's expiration time would be checked first, to ensure its validity. If the link were valid, the user would be prompted to enter the PIN code, which would be compared to the hash stored in the database. Only if the PIN is correct, the user would get access the information. An example code snippet of the share link endpoint can be seen below:

```
@router.post("/share/{share_code}/verify", status_code=status.
    HTTP_200_OK, response_model=ShareItemsResponse)
@limiter.limit("5/minute")
async def verify_share_token(
    request: Request,
    share_code: str,
    pin: Annotated[str, Body(embed=True)],
    session: Session = Depends(get_session)
):
    share_token = session.exec(select(ShareToken).where(ShareToken.
    share_code == share_code)).first()

    if not share_token:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
    detail="Share token not found")

    # Check the PIN against the hashed PIN stored in the database
```

```
15      if not verify_hash(pin, share_token.hashed_pin):
16          raise HTTPException(status_code=status.HTTP_403_FORBIDDEN,
    detail="Invalid PIN")
17
18      user = session.exec(select(User).where(User.id == share_token.
    user_id)).first()
19
20      user_data = {
21          "name": user.name,
22          "dob": user.dob.strftime("%d-%m-%Y") if user.dob else None,
23      }
24
25      # Process the shared items, which are stored as a JSON string in
    the database
26      items_data = get_item_data(share_token.shared_items, session)
27
28      items_response = ShareItemsResponse(
29          expiration_time=share_token.expiration_time,
30          patient=user_data,
31          items=items_data
32      )
33
34      return items_response
```

Listing 5.23: Share Link Endpoint

A sequence diagram of how the share link works, from creation to access can be seen below in figures 5.24 and 5.25.

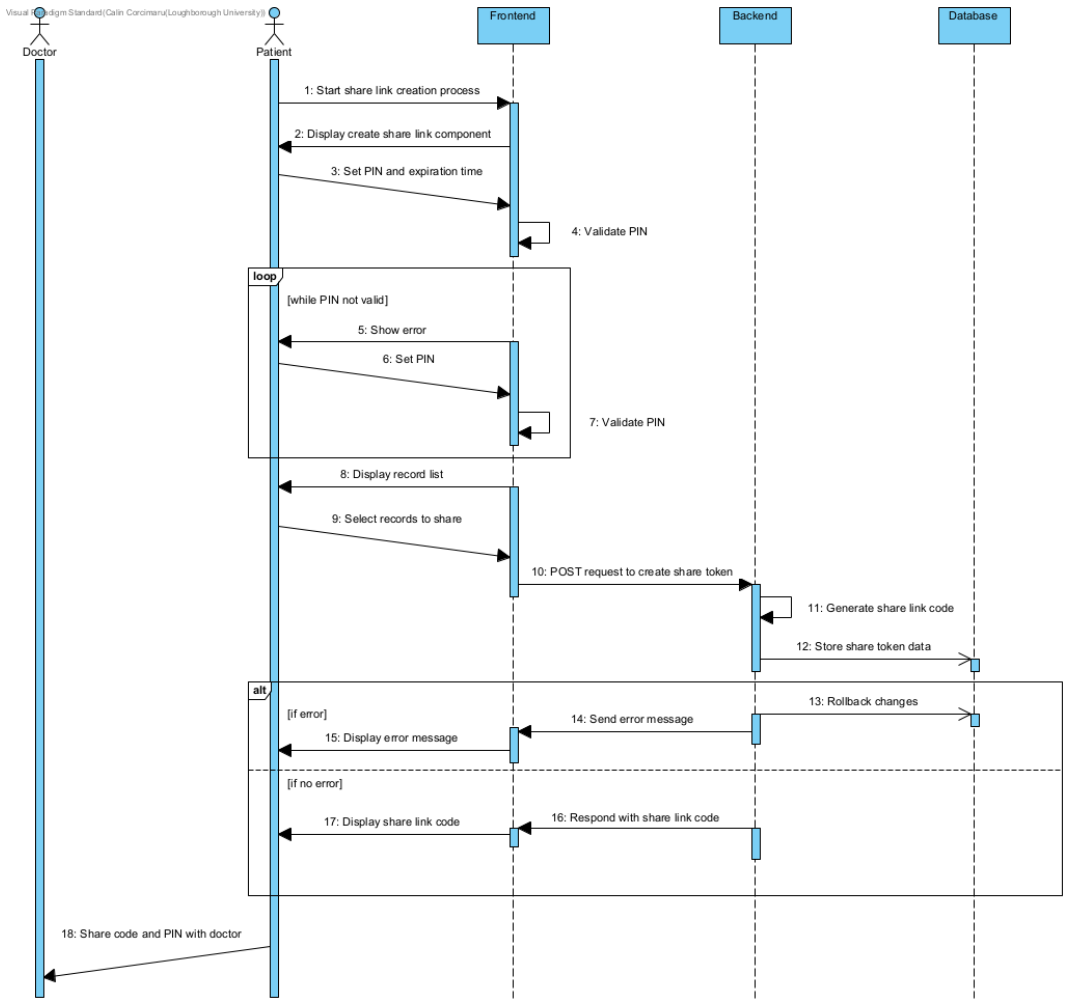


Figure 5.24: UML Sequence Diagram — Creating share link

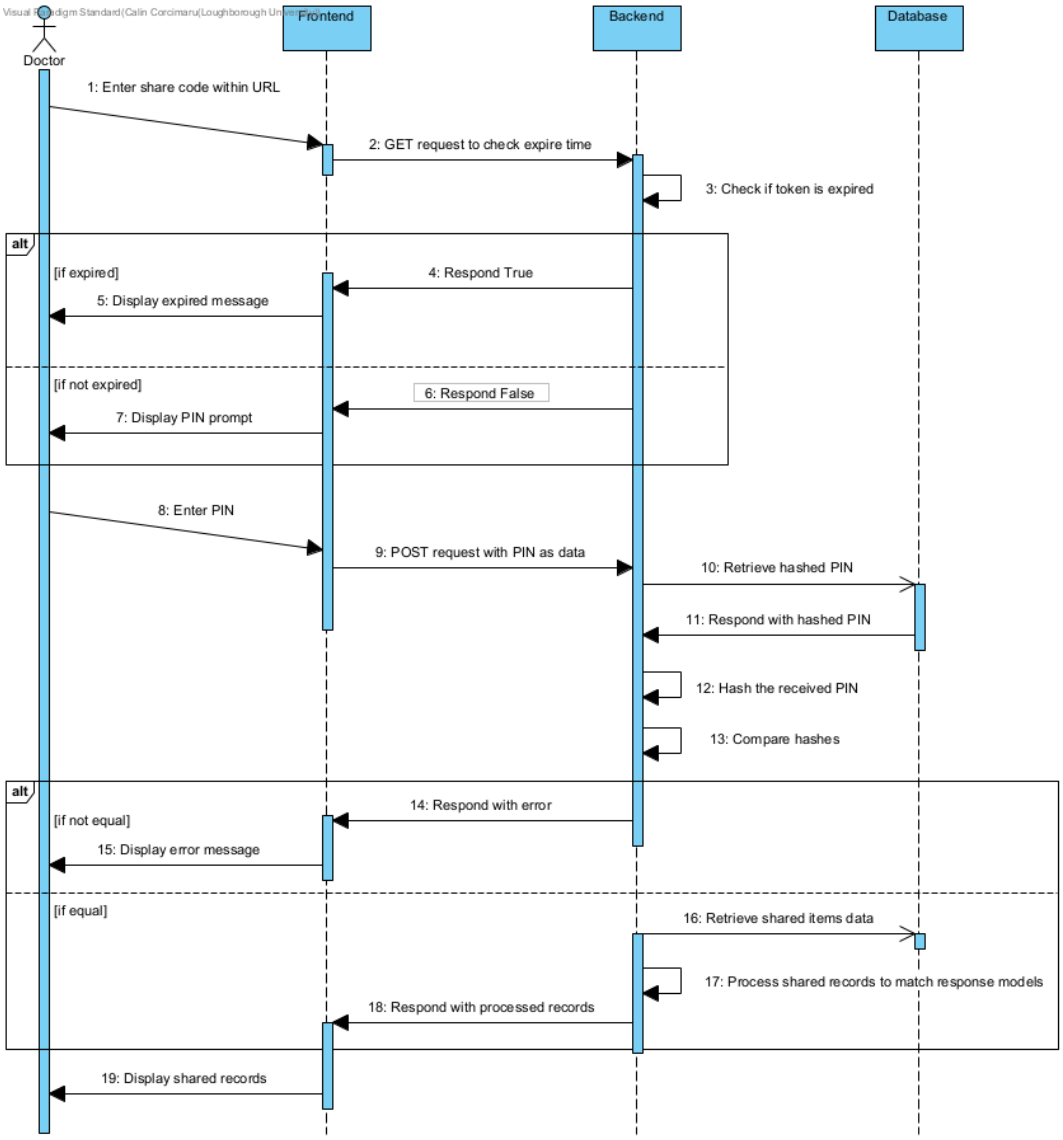


Figure 5.25: UML Sequence Diagram — Accessing shared link

5.6.2 Frontend changes

To allow the user to create share links, a new component was created in the frontend. The component was located in the dashboard page, as this is intended to be the main and first page the user sees after logging in. The component was designed to guide the user through the main 3 stages of share link creation: first is the setting of the expiration time and PIN, next is the selection of the records to be shared and finally the confirmation and display of the share link code. The component was designed to be as simple and easy to use as possible, with clear instructions and a step-by-step approach. The component and the different stages can be seen below in figures 5.26, 5.27, and 5.28.

Figure 5.26: 1st step of share link creation

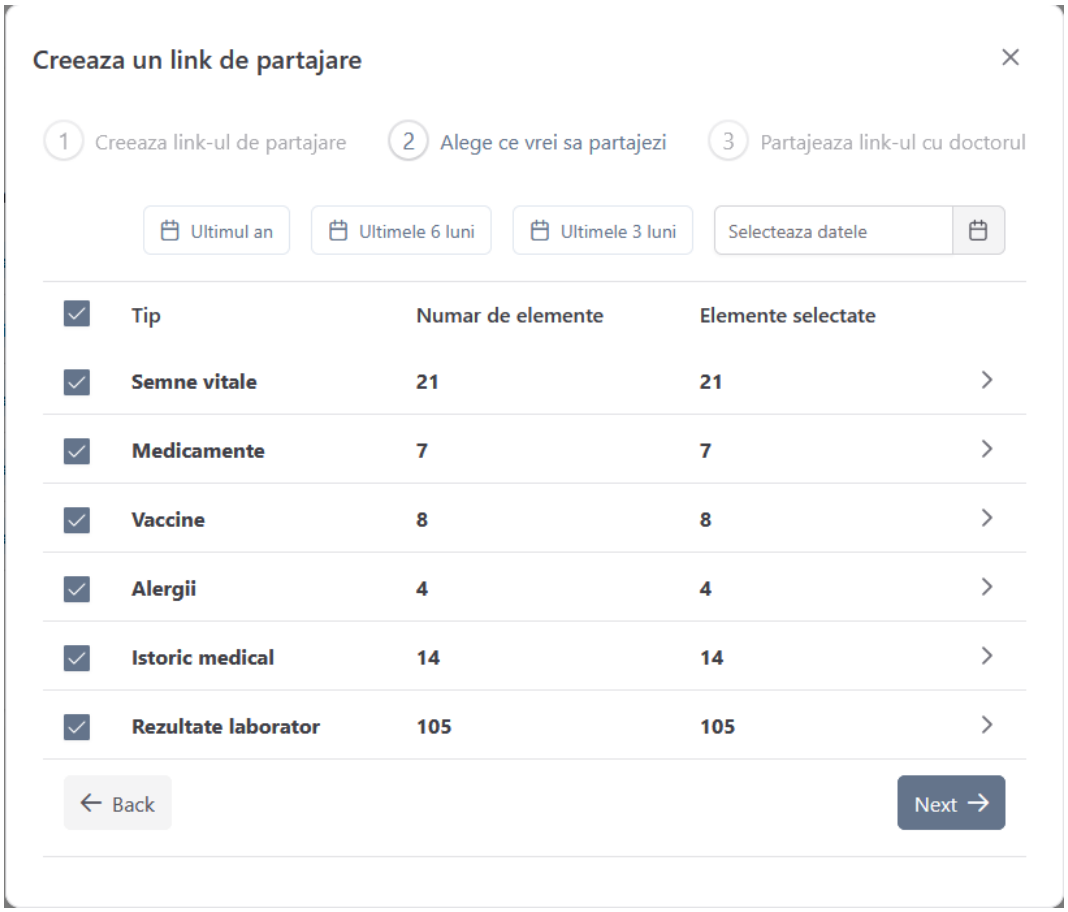


Figure 5.27: 2nd step of share link creation

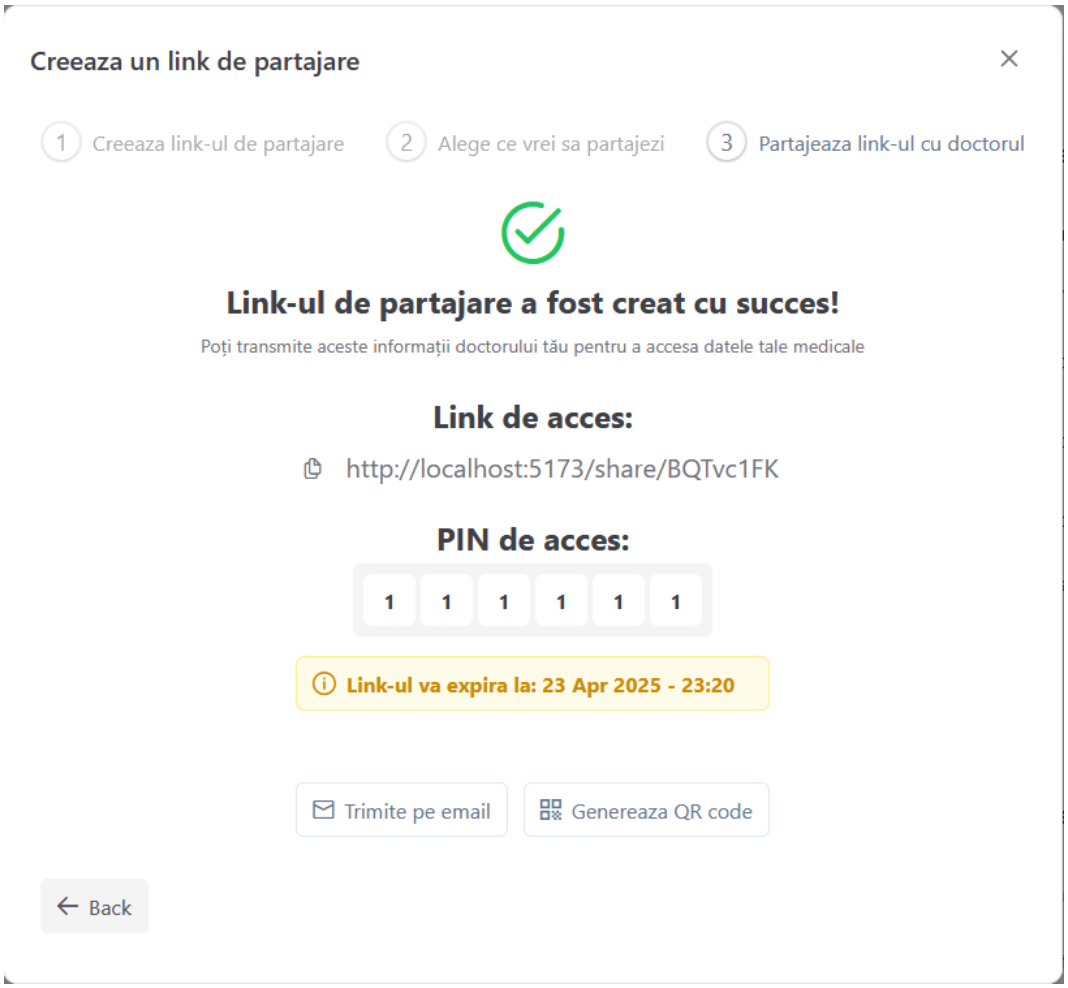


Figure 5.28: 3rd step of share link creation

Next, the actual page that would display the shared information was created. As described in the previous section, the system would first check the expiration time of the share link, and if valid, would prompt the user to enter the PIN code. If the link would not be valid, the user would be shown an error message and offered the option to be redirected to the landing page. Both of these outcomes can be seen in figures 5.30 and 5.29.

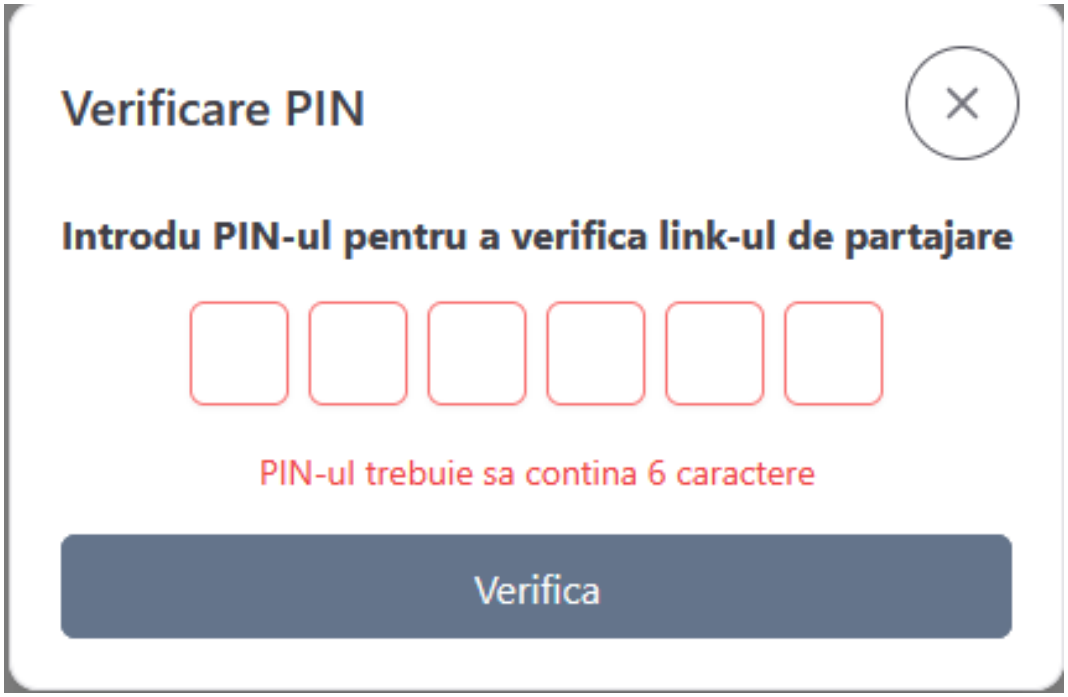


Figure 5.29: PIN prompt dialog

Figure 5.30: Expired link dialog

Finally, to wrap this feature up, the page that would display the shared information was created. Due to the amount of information that could be displayed, it was clear that the previously used tables would not suffice for this task. Since most of the information was displayed in the main system on their own, separate pages, it was important to somehow emulate the same experience without re-creating the pages again.

This is where PrimeVue's TabView component came in handy, as it added in tabs that allowed the user to switch between the different sections (or record categories), while remaining on the same page. This way, each tab could still contain the table with the records, while having the whole page to display them. The tabs and table columns were dynamically created based on the data received from the backend, in case some of the records were omitted by the user.

Additionally, the page would also contain some basic user information such as the name and date of birth, with potential to add more if the user model would be expanded to contain more information. Finally, a live timer was added to the top of the page to show the time remaining until the link would expire, allowing users to understand how much time they have left to access the information. An example of how the page looks can be seen below in figure 5.31.

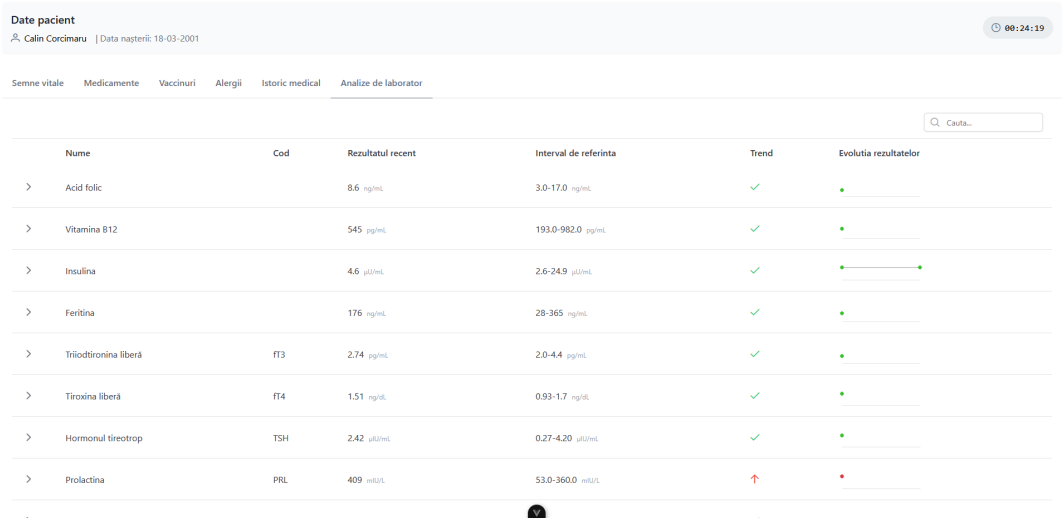


Figure 5.31: Share page — Desktop version

5.6.3 Backend

All the backend changes in this sprint were related to the share link feature. A new model was created for the share tokens, which contains the hashed PIN, share code, expiration time and the actual shared items.

One of the challenges encountered when creating the share token model was the way to store the shared items. The initial idea was to create a new table, which would contain the share item type and their UUID.However, this would've added a lot of computation that needed to be done in the backend when creating the share link, as the database would need need to be queried for all items and then they would've have to be processed into their respective response models for the API.This would've added unncessary complexity to the system, making it less efficient and harder to maintain if new types of records would be added in the future.

A solution to this problem was to use JSON fields in the database to store the shared items. One of the main reasons for using this method is the origin of the records when choosing them to be shared. Since the share creation link dialog was located in the dashboard, it used the same API response from the dashboard view, which meant that the records were already properly formatted in their respective response models. By using these pre-processed records, a whole step of processing could be avoided: records would be pulled from the dashboard API response, chosen in the share link creation dialog and then stored in the database as a big JSON object as-is. Thus, when accessing the share link, the records would

be pulled from the database as a whole JSON object, some light processing done and then directly displayed in the frontend. Additionally, this would reduce the number of queries done to the database, as the items were stored in the same table as the other share token information, such as PIN, share code, etc.

The updated ERD can be seen in figure 5.32, making it the last update to the database schema for this project.

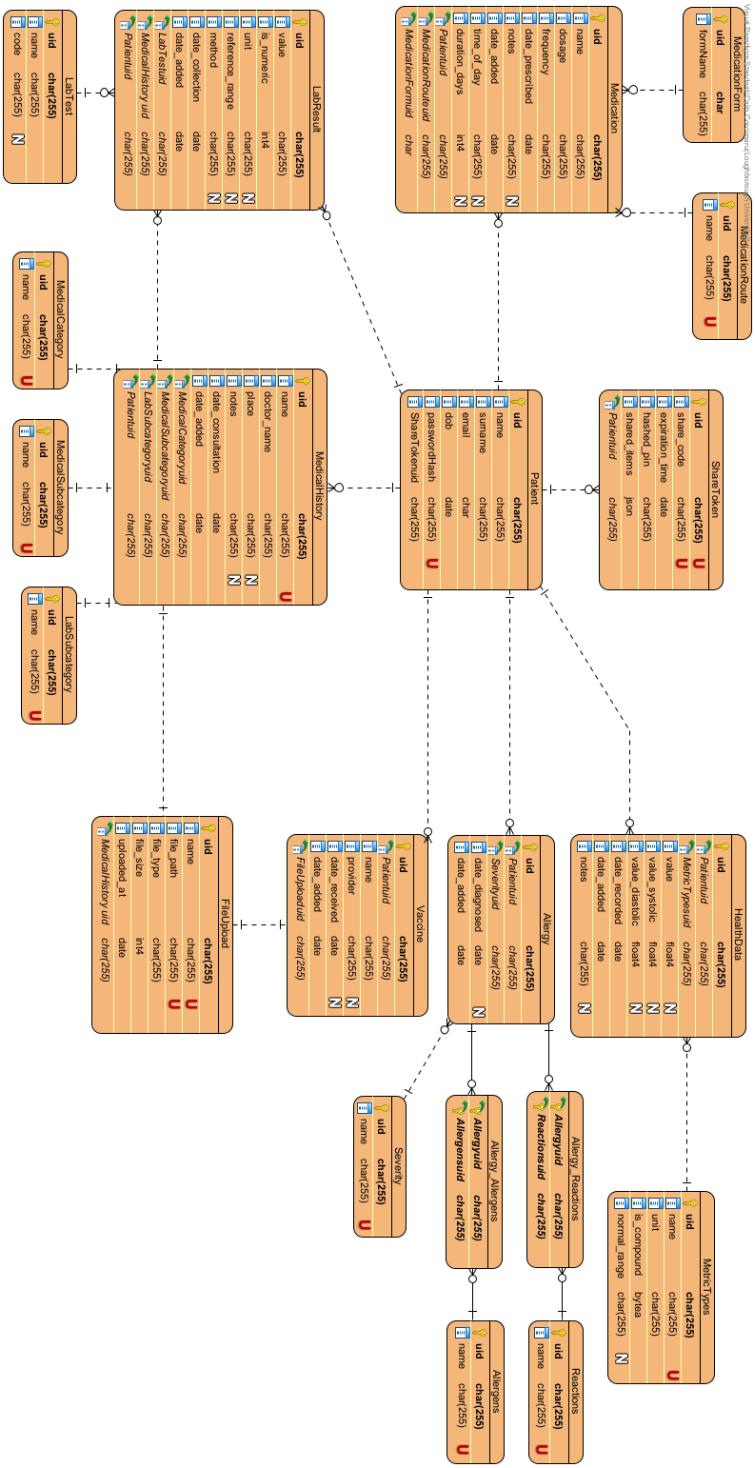


Figure 5.32: Updated Entity Relationship Diagram Sprint #6

5.6.4 Challenges encountered

The main challenge encountered in this sprint was the time pressure, as the project was nearing its deadline. This resulted in some of the features being simplified to their main functionality, without any room for much improvement or additional changes. This was especially true for the frontend part of displaying the shared information, as the amount of data and the combination of the previously added features made it hard to display everything in a single page.

Another interesting challenged encountered was the differences caused by the languages used in the system. As this was a project for patients in Moldova, the frontend was developed in Romanian to accommodate the national language. However most of the code, including the database fields and other models, were written in English as the project was done as part of a degree in a UK-based university. This resulted in some interesting outcomes when data was received from the backend, with many headers being in English, while the rest of the data was in Romanian. In the previously developed pages, it was possible to manually rename the headers to Romanian, but in the case of the share link items it was very difficult as many of the headers were dynamically created based on the data received. Considering this system was just a prototype, it was deemed acceptable, however for a production system, this issue would've had to be addressed.

5.6.5 Requirements completed

- The system must allow the patient to generate a shareable link to provide access to their medical records.
- When creating the shareable link, the system must allow the patient to set an expiration date for the link.
- When creating the shareable link, the system should allow the patient to set an access password for the link.
- When creating the shareable link, the system should allow the patient to select which records to share with the doctor.

Chapter 6

Evaluation

This chapter will discuss the evalution and testing of the system and its components. The next sections will cover the evaluation of the frontend, backend and the LLM used. This chapter will then conclude with a discussion on the client feedback received at the end of the project.

6.1 Frontend Testing

Frontend testing was mainly conducted through manual testing, done at the same time as the development of said components. While initially the plan was to use automated testing, the time constraints, complexity of components and lack of experience with any testing frameworks led to the decision to use manual testing instead.

To ensure some level of robustness in the testing procedure, a checklist or a list of test cases was created for every created component. After finishing development, this list would then be used to ensure the component wasa working as intended. To have a more structured approach, the test cases were usually written using the Gherkin syntax, which follows the Given-When-Then format, allowing for better definition of the test cases (**gherkin**).

An example of some test cases for the selection of records to be shared in a share link is shown below:

```
1    Feature: Record selection for share link
2
3    Scenario: User selects all record types to be shared
4        Given the user is on the share link page
```

```
5         When the user selects the top checkbox to select all record
  types
6         Then all record types should be selected for sharing
7         And the all children checkboxes should be selected as well
8
9     Scenario: User selects specific record types to be shared
10        Given the user is on the share link page
11        When the user selects a specific record type to be shared
12        Then only the selected record types should be selected for
  sharing
13        And the respective children checkboxes should all be selected
  as well
14
15    Scenario: User selects specific children records to be shared
16        Given the user is on the share link page
17        When the user selects specific children records to be shared
18        Then only the selected children records should be selected for
   sharing
19        And the respective parent checkbox should not be selected
20        And the top child checkbox should not be selected
```

Listing 6.1: Test cases for selecting records to be shared in a share link

And the respective frontend result of these test cases can be seen in figures 6.1, 6.2 and 6.3. The first figure shows the selection of all record types, the second figure shows the selection of specific record types and the third figure shows the selection of specific children records.

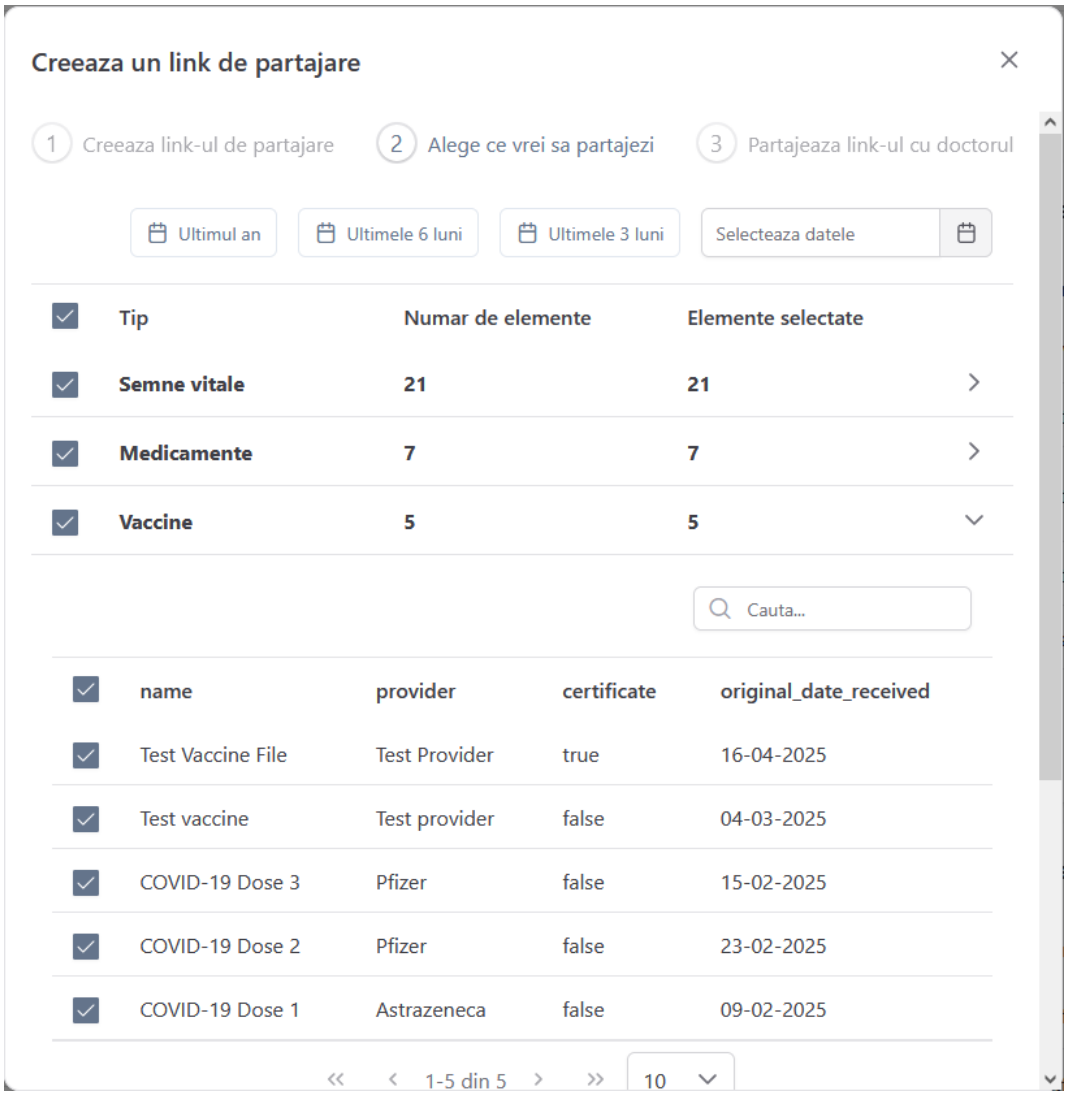


Figure 6.1: Test Scenario #1

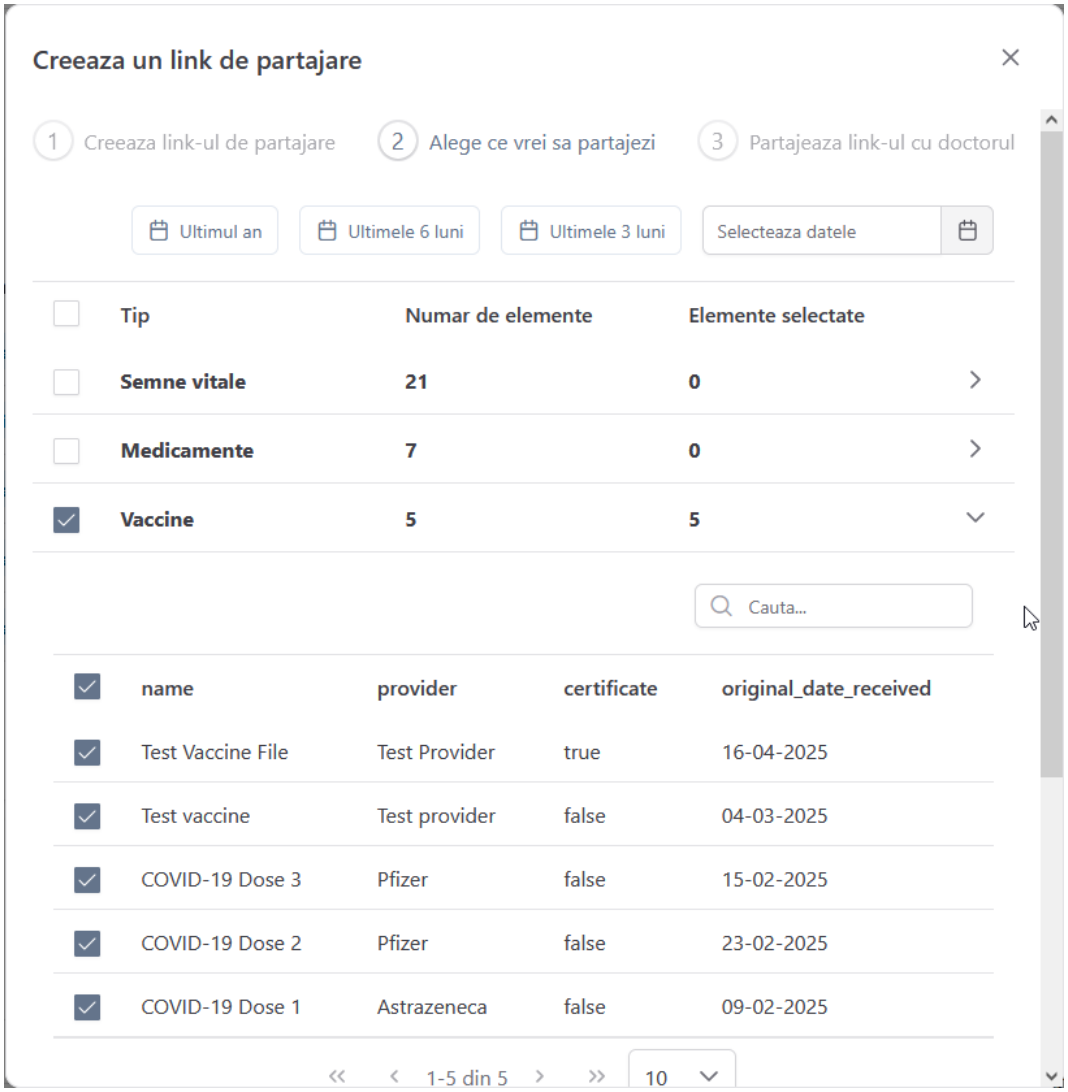


Figure 6.2: Test Scenario #2

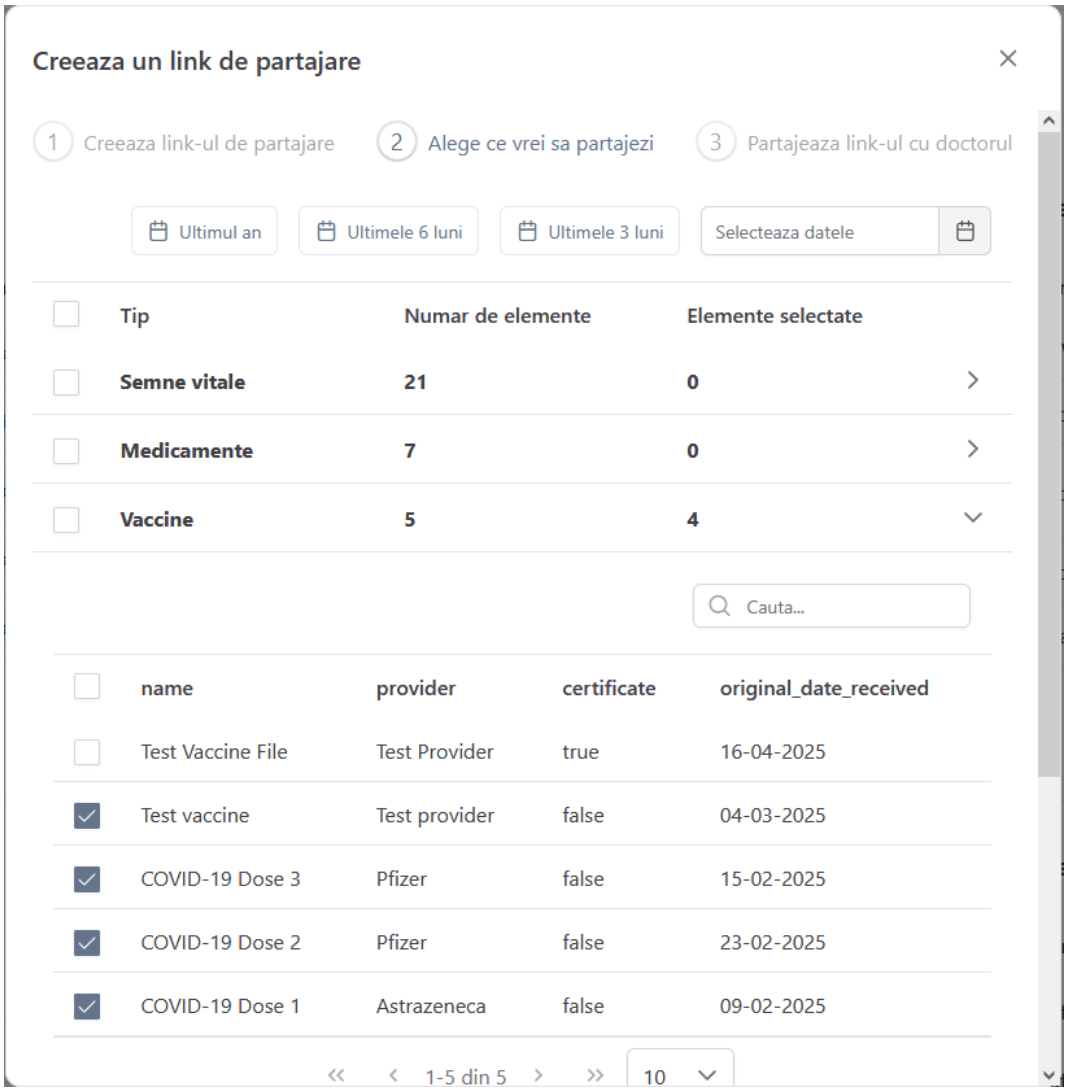


Figure 6.3: Test Scenario #3

In addition to the test cases, a site map was created to track all the pages and their respective components, with relationships showing how each page and component is related to each other. This was done to ensure that all components were created and tested properly. In the end, it allowed for a better, high-level overview of the system and its parts, but also for tracking of the overall progress of the project. The site map can be seen in figure 6.4.

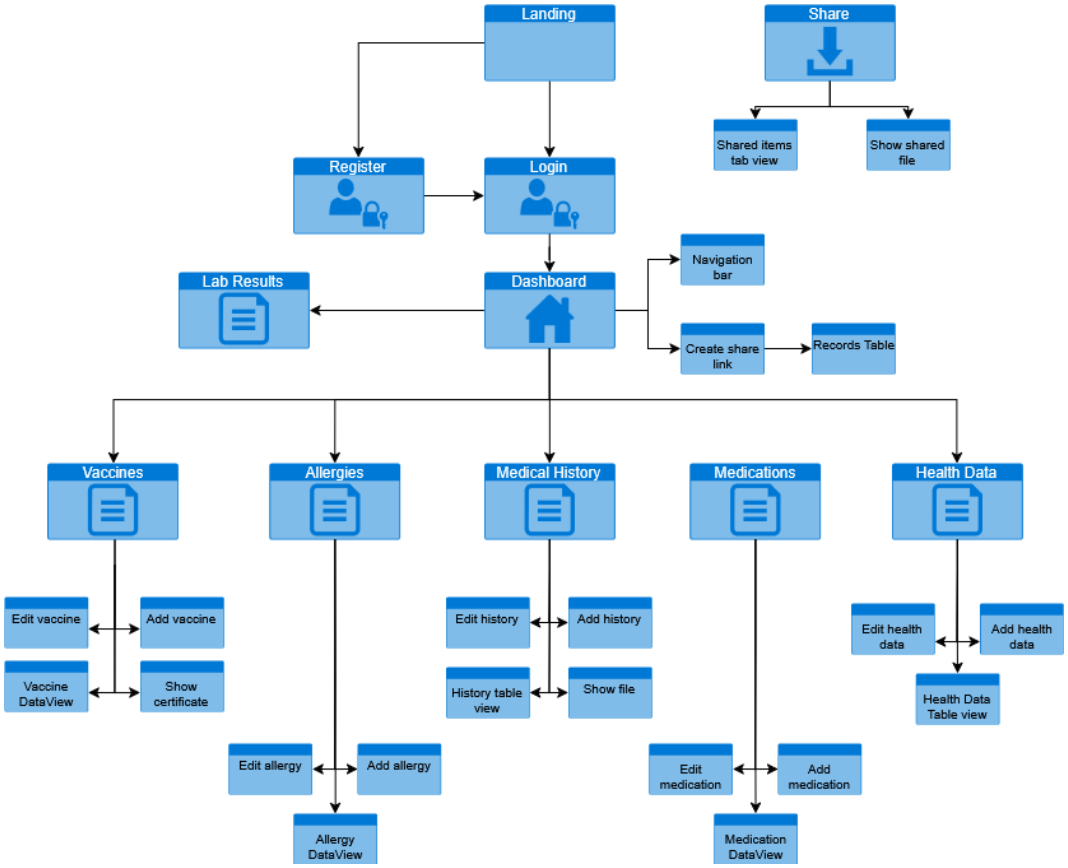


Figure 6.4: PHR System Site Map

6.2 Backend Testing

The backend testing part was also done manually. Thankfully, the backend was less complex than the frontend, so the testing was easier and faster to do. Additionally, much of backend's functionality was already tested simultaneously with the frontend testing, as it was dependent on the backend to work properly. As such, many of the test cases were already covered in the frontend testing.

However, to ensure that the backend was working as intended, additional tests were done using Postman while the API was being developed. This allowed for testing of the endpoints without relying on the frontend components being created. The tests were done by sending requests to the API and checking the responses, and comparing them with the expected results.

An example of how an endpoint was tested will be shown now. An example endpoint can be seen below:

```
    @router.get("/share/{share_code}", status_code=status.HTTP_200_OK)
@limiter.limit("5/minute")
async def check_share_token(
    request: Request,
    share_code: str,
    session: Session = Depends(get_session)
):
    share_token = session.exec(select(ShareToken).where(ShareToken.
    share_code == share_code)).first()

    if not share_token:
        raise HTTPException(status_code=status.HTTP_404_NOT_FOUND,
    detail="Share token not found")

    if share_token.expiration_time < datetime.now():
        raise HTTPException(status_code=status.HTTP_410_GONE, detail="
    Share token has expired")

    return {
        "valid": True
    }
```

Listing 6.2: Example endpoint for checking a share link for expiration time

This endpoint checks if a share link is valid and if it has expired. The endpoint takes in a share code and checks if it exists in the database. If it does, it checks if the expiration time is still valid. If both checks pass, it returns a valid response. If either check fails, it raises an HTTP exception with the appropriate status code and message.

As such, for this endpoint, 3 test cases can be created: one where the share code is not valid, another where the code is valid but the expiration time is not valid and a third one where everything is valid. The test cases in Postman can be seen in figures 6.5, 6.6 and 6.7.

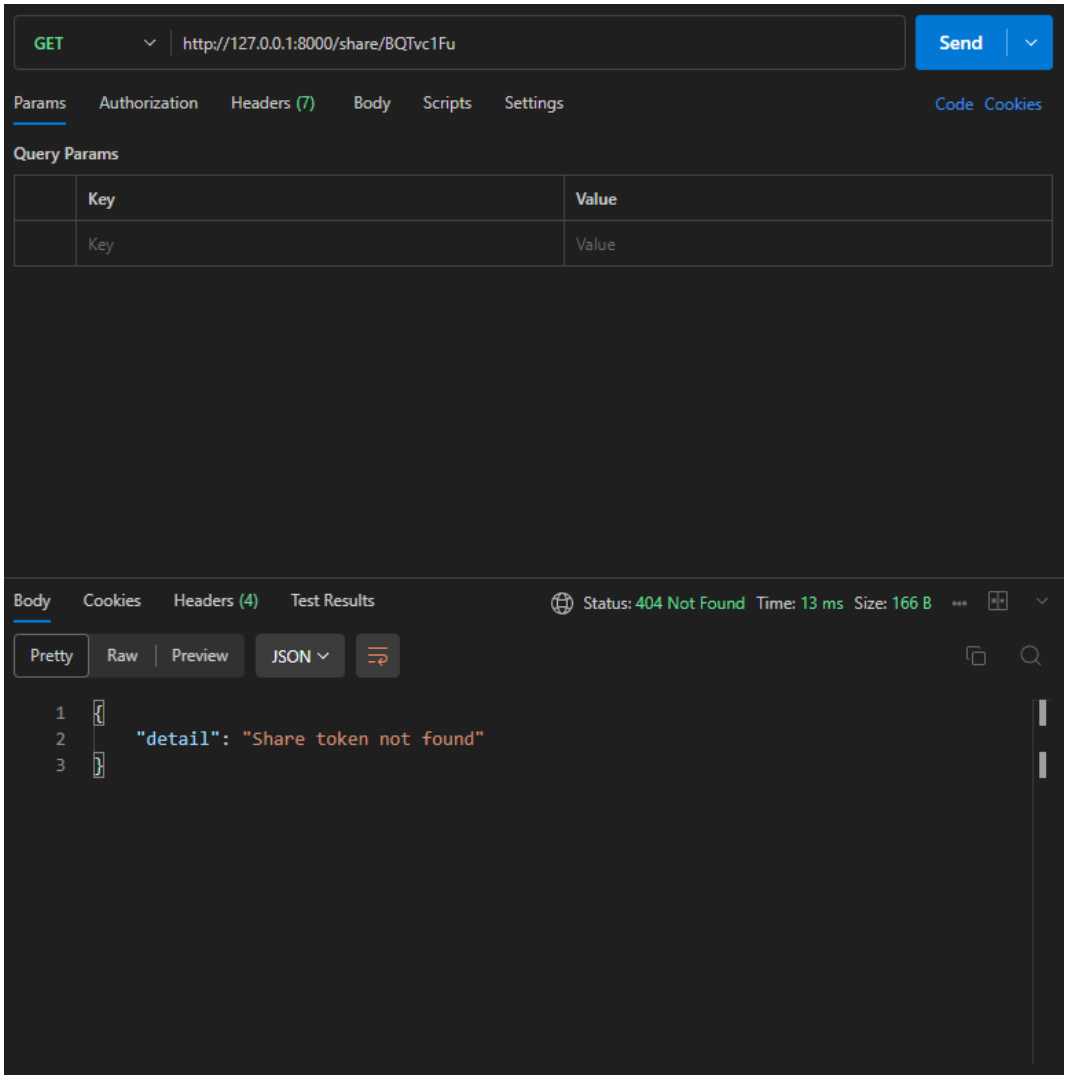


Figure 6.5: Postman test — share token not found

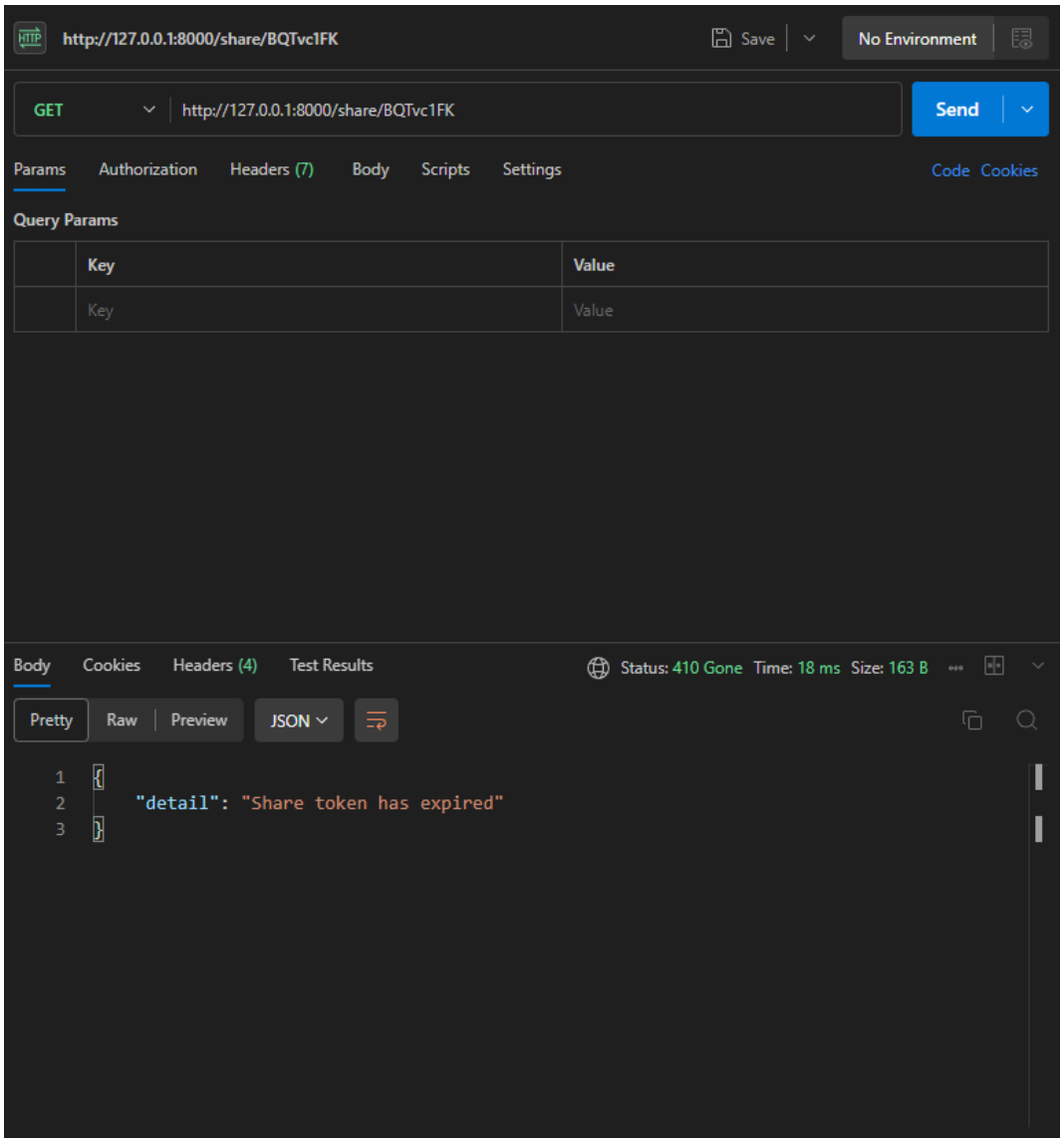


Figure 6.6: Postman test — share token expired

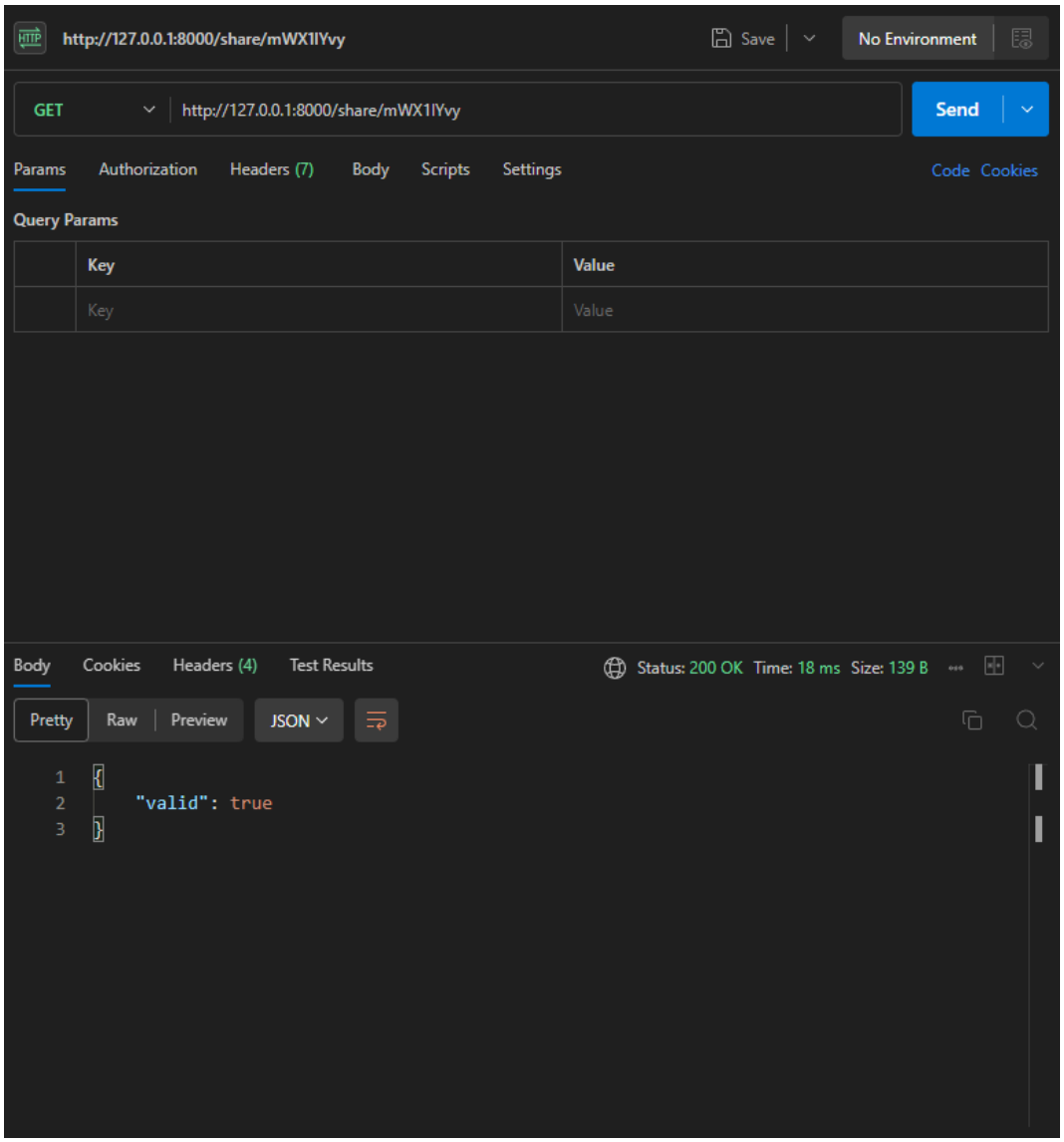


Figure 6.7: Postman test — share token valid

6.3 LLM Evaluation

As previously mentioned in the Development chapter, the MLLM used in this project was Gemini 2.0 Flash. One of the main reasons for using this LLM was its generous free tier, which allowed for up to 15 requests per minute with a maximum of 1,000,000 tokens per minute. Alongside the generous free tier, the model also sported robust multimodal capabilities, which was crucial for processing lab results, which were often in PDF or image format.

However, it is also important to note that while the free tier and its multimodal capabilities were a big factor in choosing this model, it was also chosen for its performance and speed, which rivaled that of other, similarly sized models. As of writing this section, Gemini 2.0 Flash occupies a spot in the top 10 of the Chatbot Arena leaderboard for both language and vision tasks, authored by **chatbotarena**. As described by its authors, the leaderboard is an 'open platform for crowdsourced AI benchmarking', which generates live leaderboards based on user feedback from comparing responses from different models. As such, it is quite likely that this leaderboard is a good indicator of the performance of the models, and good place to compare them.

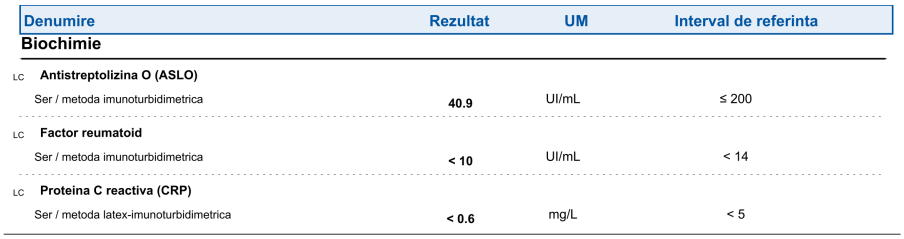
To ensure the MLLM was working as intended and was up to the task, the same prompt used in the actual system was used to test the model across multiple lab result documents from various institutions. Documents from different institutions such as labs and hospitals were used to ensure that the model was able to handle different formats of the document and the language used within it. The prompt can be seen below:

```
"""
You have been given a document that contains lab results that is
written in the Romanian language. Your job is to extract all lab
results from this document in JSON format with the following
fields: test_name, test_code, value, unit, reference_range, method
. Sometimes the code of the test will be in the name itself, and
it is your job to determine if the code is there, for example in
brackets or separated by a comma, and separate the name and the
code. Sometimes the lab result will not have a method specified,
and in that case you return an empty string. The document is in
Romanian, however the JSON keys should be in English.

EXTREMELY IMPORTANT FORMATTING INSTRUCTIONS:
1. Return ONLY the raw JSON array
2. DO NOT use code blocks, backticks, or markdown formatting
3. DO NOT include ```json or ``` anywhere in your response
```

```
    4. DO NOT include any explanations or text before or after the
    JSON
    5. Your response must start with the '[' character and end with
    the ']' character
    6. The output should be valid JSON that can be parsed directly
    7. Use period (.) as the decimal separator, not comma (,)

    Example of how your output should look, starting from the very
    first character:
    [{"test_name":"Hemoglobina","test_code":"HGB","value":"14.3","unit
    ":"mg/dL","reference_range":"13.2-17.3", "method":"
    Chemiluminiscenta"}]

    if there is no method specified, return an empty string:
    [{"test_name":"Hemoglobina","test_code":"HGB","value":"14.3","unit
    ":"mg/dL","reference_range":"13.2-17.3", "method":" "}]
    """
```

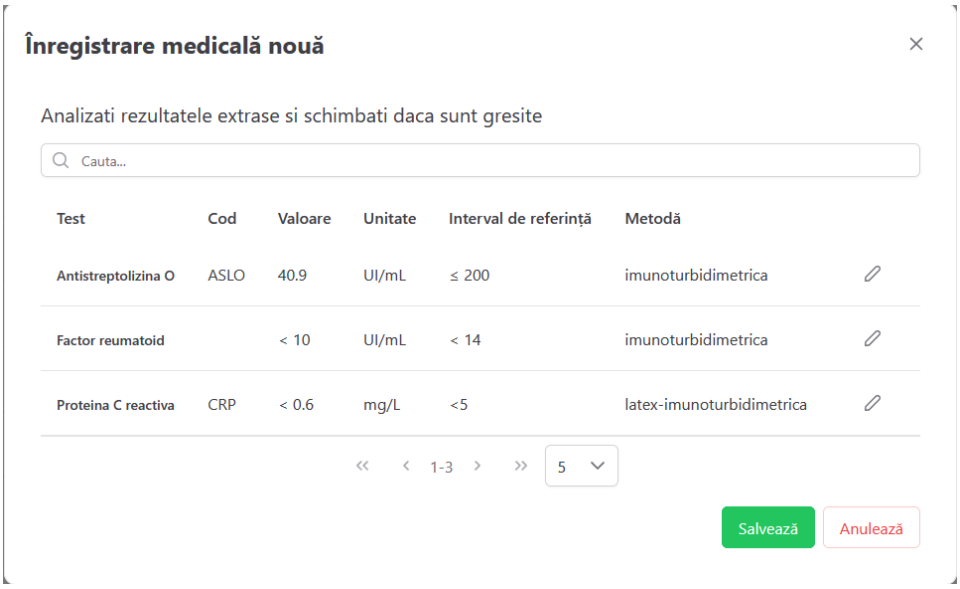
In the end, it was possible to obtain 3 different lab results documents — 2 from different labs and 1 from a private hospital. At the time of writing, the labs that were chosen were Alfalab and Synevo, for being one of the most popular in Moldova and having a wide network of testing sites across the whole country (**alfalab**; **synevo**). The private hospital chosen was Medpark, which is also a popular destionation for many of Moldova's capital residents (**medpark**). The anonymised documents, with the exported results alongside them can be seen in figures.

The documents, along with their extracted results in the system, can be seen in the following figures: 6.8, 6.9 and 6.10.



Denumire	Rezultat	UM	Interval de referinta
Biochimie			
LC **Antistreptolizina O (ASLO)**			
Ser / metoda imunoturbidimetrica	**40.9**	UI/mL	≤ 200
LC **Factor reumatoid**			
Ser / metoda imunoturbidimetrica	**< 10**	UI/mL	< 14
LC **Proteina C reactiva (CRP)**			
Ser / metoda latex-imunoturbidimetrica	**< 0.6**	mg/L	< 5

(a) Lab document



(b) Extraction result

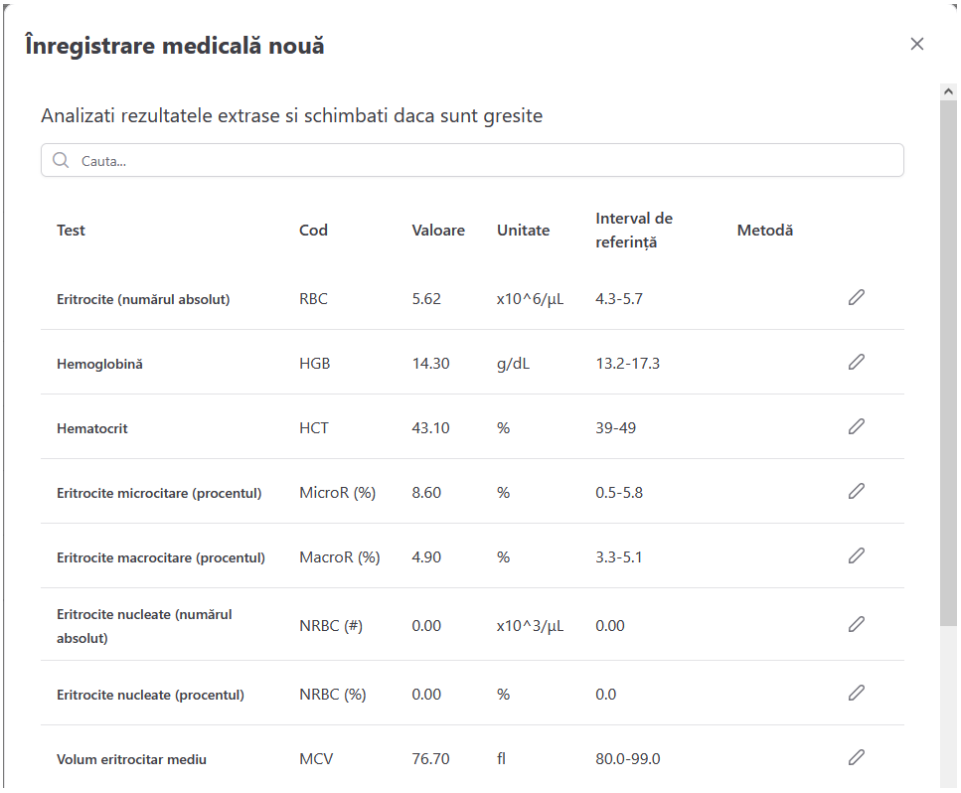
Figure 6.8: Synevo extraction results

Hemograma cu formula leucocitară

Sânge integral. Metoda: Citometria în flux cu fluorescentă / Focalizarea hidrodinamică / SLS hemoglobină

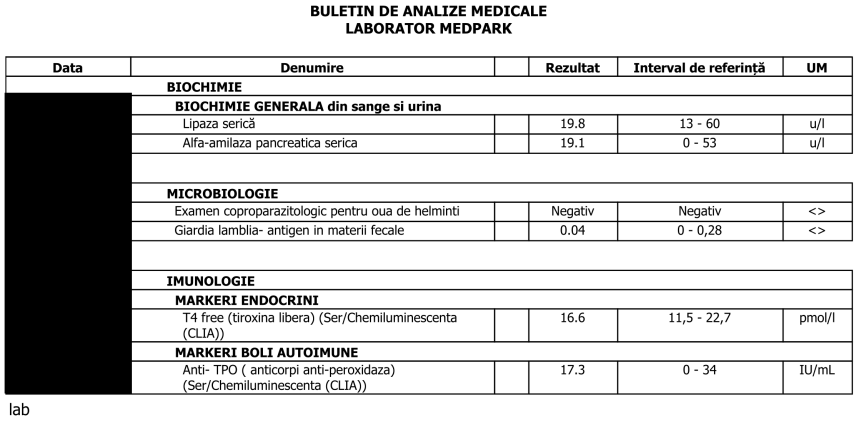
Parametru		Rezultat		Valori de referință	Unități de măsură
Eritrocite (numărul absolut)	RBC	5,62		4,3 — 5,7	x10^6/µL
Hemoglobină	HGB	14,30		13,2 — 17,3	g/dL
Hematocrit	HCT	43,10		39 — 49	%
Eritrocite microcitare (procentul)	MicroR (%)	**8,60**	▲	0,5 — 5,8	%
Eritrocite macrocitare (procentul)	MacroR (%)	4,90		3,3 — 5,1	%
Eritrocite nucleate (numărul absolut)	NRBC (#)	0,00		0,00	x10^3/µL
Eritrocite nucleate (procentul)	NRBC (%)	0,00		0,0	%
Volum eritrocitar mediu	MCV	**76,70**	▼	80,0 — 99,0	fl

(a) Lab document



(b) Extraction result

Figure 6.9: Alfalab extraction results

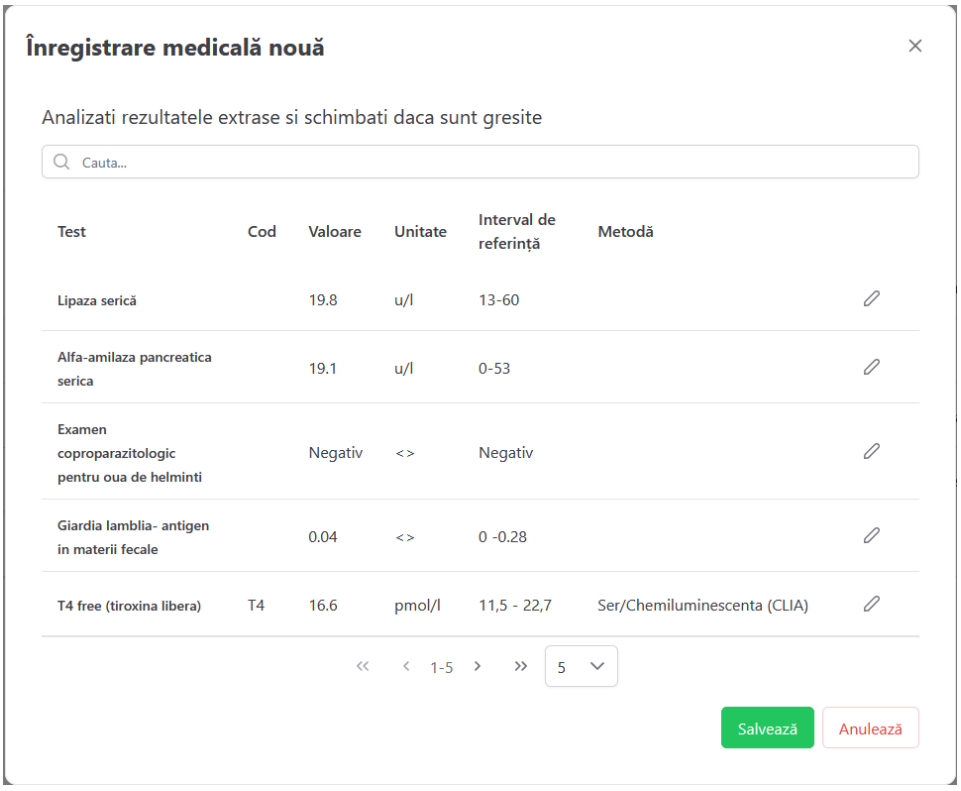


BULETIN DE ANALIZE MEDICALE
LABORATOR MEDPARK

Data	Denumire		Rezultat	Interval de referință	UM
	BIOCHIMIE				
[redacted]	**BIOCHIMIE GENERALA din sange si urina**				
	Lipaza serică		19.8	13 - 60	u/l
	Alfa-amilaza pancreatica serica		19.1	0 - 53	u/l
	MICROBIOLOGIE				
	Examen coproparazitologic pentru oua de helminti		Negativ	Negativ	<>
	Giardia lamblia- antigen in materii fecale		0.04	0 - 0,28	<>
	IMUNOLOGIE				
	MARKERI ENDOCRINI				
	T4 free (tiroxina libera) (Ser/Chemiluminescenta (CLIA))		16.6	11,5 - 22,7	pmol/l
	MARKERI BOLI AUTOIMUNE				
	Anti- TPO (anticorpi anti-peroxidaza) (Ser/Chemiluminescenta (CLIA))		17.3	0 - 34	IU/mL

lab

(a) Lab document



(b) Extraction result

Figure 6.10: Medpark extraction results

As it can be seen by the results, the model was able to extract the lab results and their details such as code, reference range, unit and even method. These excellent results were achieved with a single, common prompt and delivered consistent results across all 3 documents, whether they were edited in any way or not. In conclusion, the model seemed to be an excellent choice for the task at hand, and these results confirmed that Gemini 2.0 Flash was indeed a suitable choice for extracting lab results from documents.

6.4 Client feedback

Chapter 7

Conclusion and Future Work

7.1 Areas of Improvement

7.2 Future Work

7.3 Lessons Learned and Reflections

7.4 Final Thoughts

Appendix A

Requirements

ID	Requirement	Priority
1.1	The system must be accessible on all modern desktop and mobile-based browsers.	Must Have
1.2	The system must be accessible from any location by using an internet connection.	Must Have
1.3	The system must store the data in a secure manner, ensuring that only the patient and the doctor can access the data.	Must Have
1.4	When shared with the doctor via a link, the system must load within 3 to 5 seconds when accessed via a desktop browser.	Should Have
1.5	When shared with the doctor via a link, the system should secure the data with a unique token or PIN that expires after the specified time frame.	Could Have
1.6	The system could be accessible on all modern mobile devices via a mobile application.	Could Have

Table A.1: Non-functional Requirements

ID	Requirement	Priority
2.1	The system must provide a secure login mechanism for patients by using a combination of login and password.	Must Have
2.2	The database must store the user credentials in a secure manner.	Must Have
2.3	The system could provide an option for multi factor authentication.	Could Have
2.4	The system could provide an option for password recovery.	Could Have
2.5	If used on mobile, the system could allow the patient to use biometric authentication for logging in.	Should Have

Table A.2: Login Requirements

ID	Requirement	Priority
3.1	The system must allow patients to upload their own medical records in a variety of formats (PDF, DOC, etc).	Must Have
3.2	The system should allow to upload files/documents for other types of records (vaccine certificates, etc).	Should Have
3.2	The system must allow the patient to specify and categorise the type of document they are uploading (lab test, doctor consultation, etc).	Must Have
3.3	The system must allow the patient to add a description of the document they are uploading.	Must Have
3.4	The system should allow the patient to add a date for the document they are uploading.	Should Have
3.5	The system should allow the patient to add a location for the document they are uploading.	Should Have
3.6	The system should allow the patient to add the doctor name for the document they are uploading.	Should Have
3.7	The system should allow the patient to sort and filter the documents based on the type of document, date, location, and doctor name.	Should Have
3.8	If used on mobile, the system should allow the patient to take a picture of the document and upload it.	Could Have

Table A.3: Document Upload Requirements

ID	Requirement	Priority
4.1	The system must allow the patient to generate a shareable link to provide access to their medical records.	Must Have
4.2	When creating the shareable link, the system must allow the patient to set an expiration date for the link.	Must Have
4.3	When creating the shareable link, the system should allow the patient to set an access password for the link.	Should Have
4.4	When creating the shareable link, the system should allow the patient to select which records to share with the doctor.	Should Have

Table A.4: Patient Shareable Link Requirements

ID	Requirement	Priority
5.1	The patient personal cabinet must provide an overview of the patient's history through 3 main sections: personal information, lab tests, and doctor consultations.	Must Have
5.2	The system must display the patient's history in a chronological order in the form of a timeline.	Must Have
5.3	The system must have a dashboard view which displays an overview of the most recent information added to the system (latest lab tests, doctor consultations, vaccinations etc).	Must Have
5.4	The system must allow patients to add their own personal information, such as name or date of birth.	Must Have
5.5	The system must allow the patient to add their own allergies.	Must Have
5.6	The system must allow the patient to add their own vaccinations.	Must Have
5.7	When viewing doctor consultations, the system should divide them into categories based on the domain of the doctor (cardiology, neurology, etc).	Should Have
5.8	The system should allow the patient to enter vitals information, such as height, weight, blood pressure, etc.	Should Have
5.9	The system should display the information in both a list or grid view.	Should Have
5.10	When multiple vital entries are made, the system could display a historical graph of the patient's vitals.	Could Have
5.11	The system could allow the patient to switch between viewing the lab tests in the document format or in a tabular, numerical format.	Could Have

Table A.5: Patient Personal Cabinet Requirements

ID	Requirement	Priority
6.1	The system must provide an overview of the patient history through 3 main sections: personal information, lab tests, and doctor consultations.	Must Have
6.2	When shared with the doctor, the system must allow the doctor to only view the patient's history, not edit it.	Must Have
6.3	The system must allow the doctor to view blood tests in a graphical format.	Must Have
6.4	The system must allow the doctor to view blood tests in a numerical, tabular format.	Must Have
6.5	The system must allow the doctor to view the patient's history in a chronological order.	Must Have
6.6	The system must display the doctor consultation and every lab test, except for blood tests, in a free text or document format.	Must Have
6.7	When viewing blood test results, the system should show the source document of the blood test value.	Should Have
6.8	For blood test results, the system should display the normal range values for each test.	Should Have

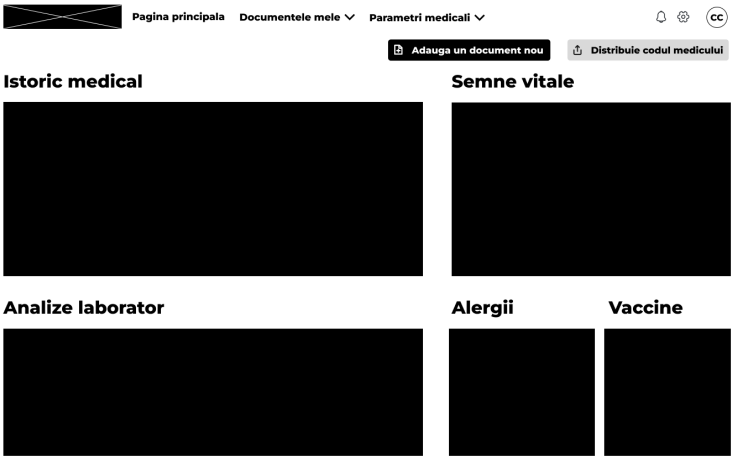
Table A.6: Shared Patient Information Requirements (Doctor View)

ID	Requirement	Priority
7.1	The system must allow patients to enter their current medication including details such as the name of the drug, dosage, frequency and start/end date.	Must Have
7.2	The system must allow patients to add new medication to their list.	Must Have
7.3	When adding medication, the system should have 2 options: add a simplified version of the medication or add a detailed version of the medication.	Should Have
7.4	When choosing the simplified version, the system should allow the patient to just add the name, dosage and duration of the medication.	Should Have
7.5	When choosing the detailed version, the system should allow the patient to add the name, dosage, frequency, start/end date, and the reason for taking the medication.	Should Have
7.6	The system should allow patients to add their past medication	Should Have
7.7	The system should allow patients to set medication reminders.	Should Have
7.8	After entering the medication, the system could allow the patient to track the medication intake.	Could Have

Table A.7: Patient Medication Requirements

Appendix B

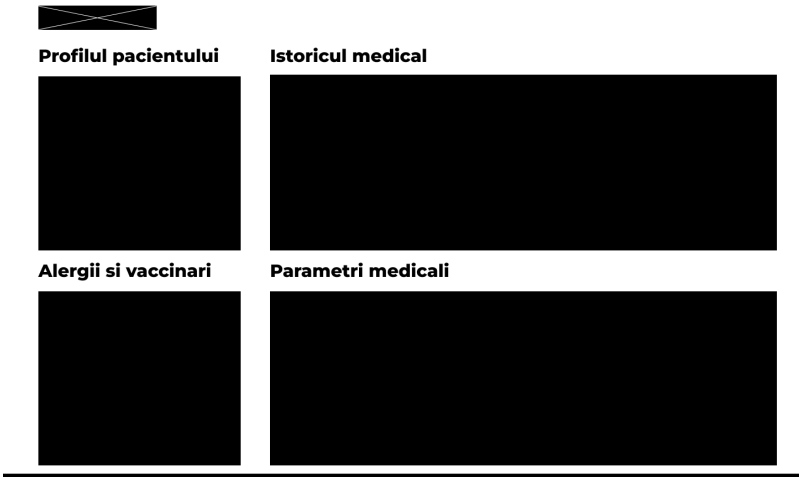
Wireframes



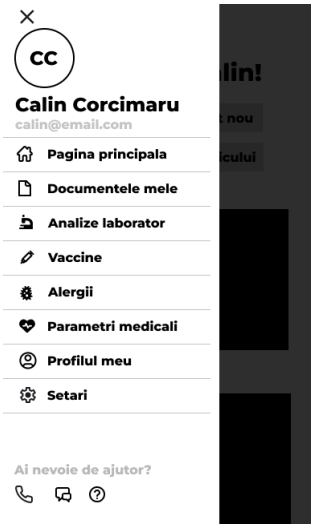
(a) Desktop version

(b) Mobile version

Figure B.1: Desktop and Mobile version of the Dashboard screen

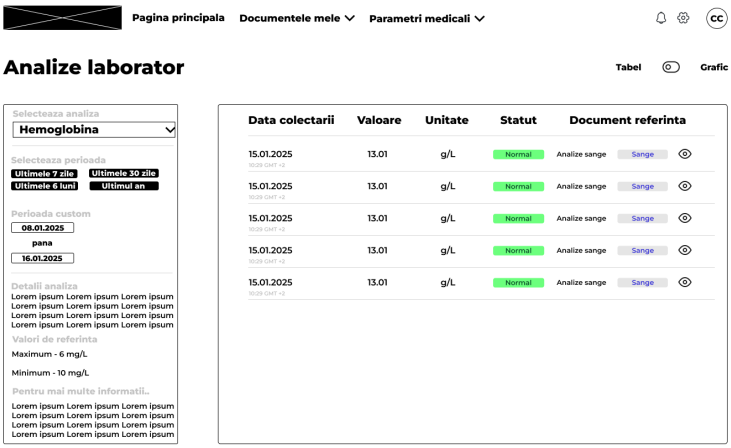


(a) Desktop version



(b) Mobile version

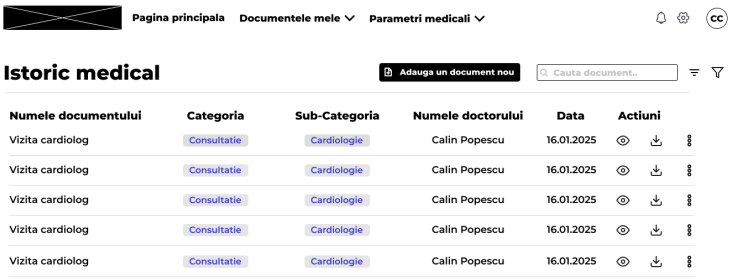
Figure B.2: Desktop and Mobile version of the Doctor View screen



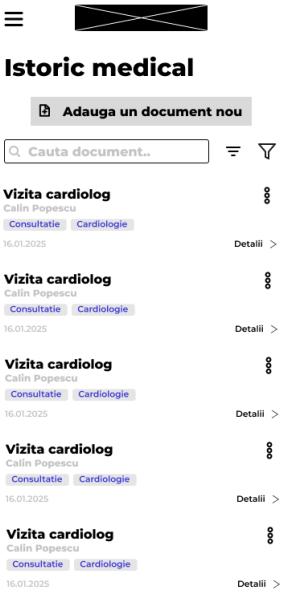
(a) Desktop version

(b) Mobile version

Figure B.3: Desktop and Mobile version of the Lab Test screen

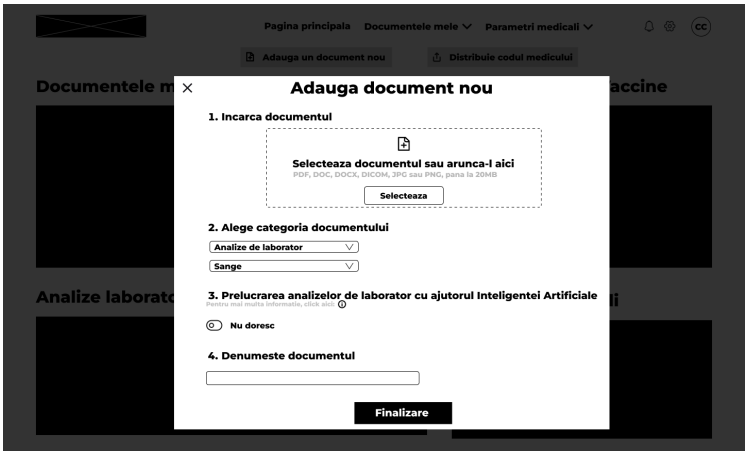


(a) Desktop version

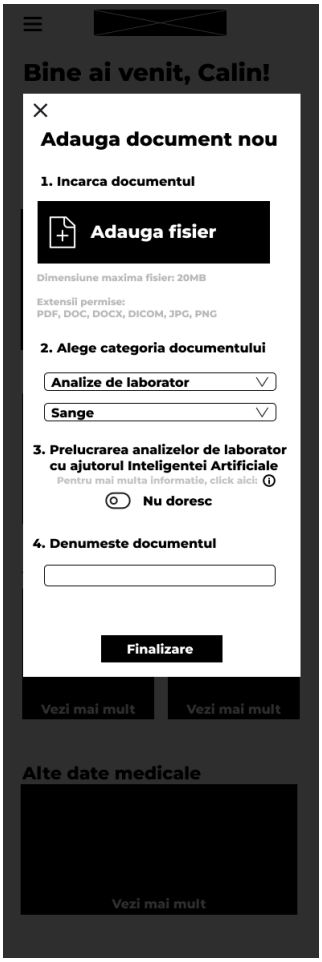


(b) Mobile version

Figure B.4: Desktop and Mobile version of the Medical History screen

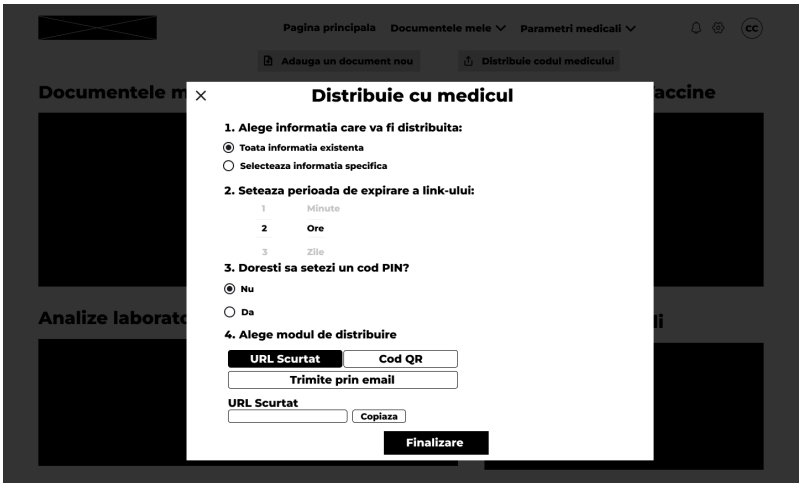


(a) Desktop version

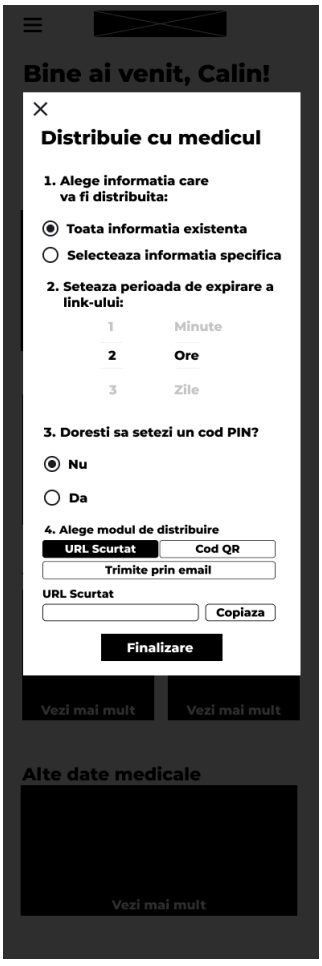


(b) Mobile version

Figure B.5: Desktop and Mobile version of the New Document screen

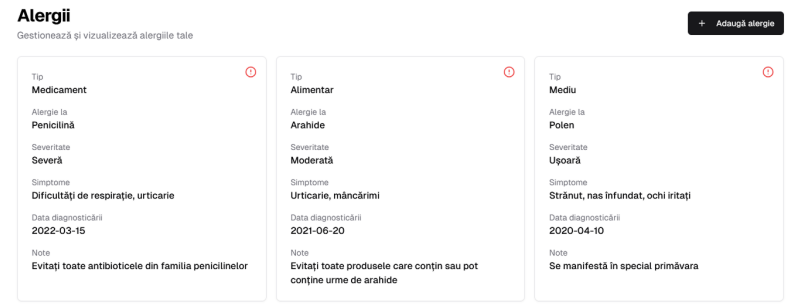


(a) Desktop version



(b) Mobile version

Figure B.6: Desktop and Mobile version of the Share Doctor screen



(a) Desktop version

(b) Mobile version

Figure B.7: Desktop and Mobile version of the Allergies screen

Figure B.8: Mobile version of the Vaccines screen