Medical Report Diagnosis Application : Documentation :

This documentation provides a comprehensive overview of the medical diagnosis application, covering its technical stack, core features, and architectural design.

1. Introduction

The Medical Diagnosis Application is a full-stack, AI-powered system designed to assist both patients and doctors. It allows patients to securely upload their medical reports and receive a preliminary diagnosis, while providing doctors with a dashboard to review patient records. The application leverages a Retrieval-Augmented Generation (RAG) architecture to provide contextual and accurate information.

2. Technology Stack

The application is built using a modern, scalable technology stack with a clear separation of concerns between the frontend, backend, and data storage layers.

- **Frontend**:
    - **Streamlit**: A Python framework used to create the interactive web user interface. It provides a simple way to build and share data apps.
- **Backend**:
    - **FastAPI**: A high-performance Python web framework for building the RESTful API. Its asynchronous nature and automatic data validation make it highly efficient.
- **Databases**:
    - **MongoDB**: A NoSQL database used for storing structured data such as user credentials and diagnosis history.
    - **Pinecone**: A specialized **vector database** used for efficient storage and retrieval of vector embeddings from medical reports.
- **AI/LLM**:

    - **Google Embedding Model**: Used to convert text data (medical reports, user questions) into numerical vectors.
    - **LangChain**: A framework that orchestrates the workflow between the LLM and the vector database, enabling the RAG pattern.
    - **Groq**: An AI inference engine that provides an extremely fast and efficient LLM for generating real-time diagnoses.

3. Core Features

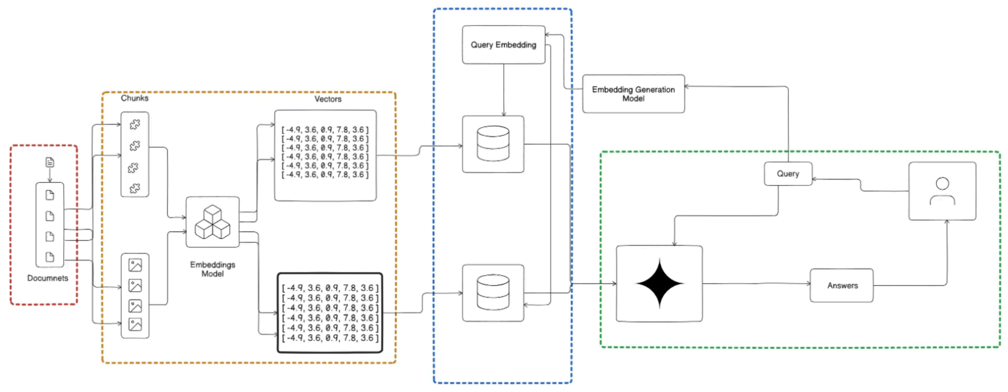
The application provides distinct features tailored to different user roles, enforced by **Role-Based Access Control (RBAC)**.

- **User Management**:
    - **Secure Signup & Login**: Users can create an account and log in with a username, password, and a defined role (`patient` or `doctor`).
    - **Authorization**: API endpoints are protected, ensuring that only users with the correct role can access specific functionalities (e.g., only doctors can view other patients' records).
- **Patient Dashboard**:
    - **Report Upload**: Patients can upload medical reports in PDF or TXT format.
    - **AI Diagnosis**: Patients can ask natural language questions and receive a diagnosis based on the contents of their uploaded reports.
- **Doctor Dashboard**:
    - **Patient Record Search**: Doctors can search for a patient by their username to retrieve a comprehensive history of their diagnoses and related queries.
    - **Formatted History View**: Diagnosis records are displayed in a clean, human-readable format, making it easy to review a patient's history.

4. Core Concepts Used

The application's functionality is built on a few key concepts from modern software engineering and AI.

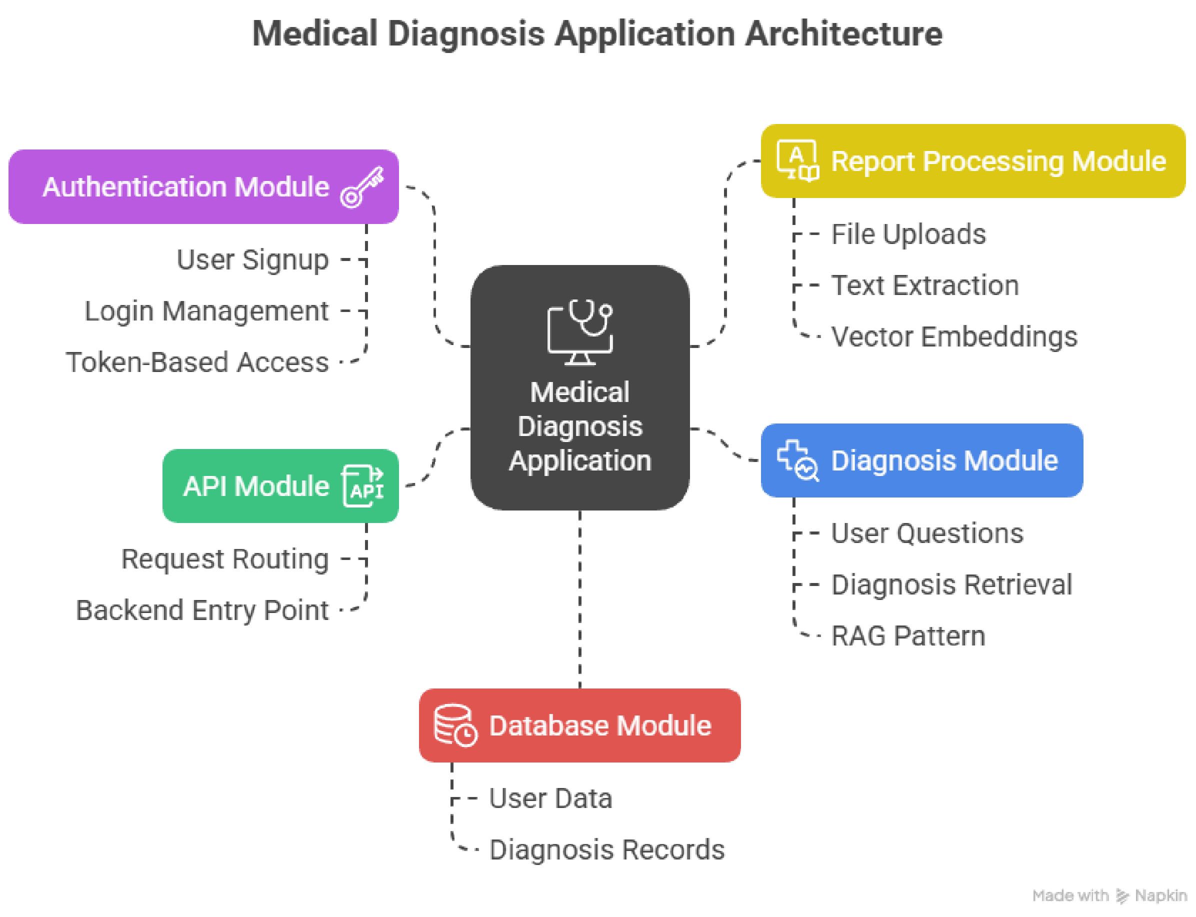
- **Microservices Architecture**: The application is broken down into small, independent services (e.g., Frontend, Backend, Database) that communicate with each other. This makes the system more maintainable, scalable, and resilient.
- **Role-Based Access Control (RBAC)**: A security model that restricts system access based on a user's role. This ensures that users can only perform actions that are appropriate for their function (e.g., a patient cannot search for another patient's records).
- **Retrieval-Augmented Generation (RAG)**: A GenAI technique that combines the power of an LLM with information retrieval. Instead of relying solely on the LLM's pre-trained knowledge, the system retrieves relevant information from a private knowledge base (the patient's reports) and uses it as a source for generating the final answer. This enhances accuracy and reduces hallucinations.



- **Vector Embeddings**: Text data (reports, questions) is converted into numerical representations called vectors. The similarity between these vectors is used to find semantically relevant information, which is a core part of the RAG process.

5. Core Modules

The following diagram illustrates the core components.



6. Application Workflow: Step-by-Step

This application works by orchestrating a series of steps that depend on the user's role. Here is a breakdown of the process from start to finish.

---

1. User Authentication

A user first interacts with the Streamlit frontend. They must either **sign up** with a role (`patient` or `doctor`) or **log in** with existing credentials. The frontend sends this information to the FastAPI backend, which authenticates the user and provides an authorization token. This token is used to protect all subsequent API calls.

2. Patient Workflow (Report Upload & Diagnosis)

Step 2.1: Report Upload

A patient uploads a medical report (e.g., a PDF or TXT file). The Streamlit frontend sends the file to the FastAPI backend's `reports/upload` endpoint.
The backend then performs a critical step: it reads the document's content, and using the **Google Embedding Model**, it converts the text into **vector embeddings**. These vectors are then stored in the **Pinecone vector database** for fast, semantic search. The backend returns a unique **Document ID** to the patient.

Step 2.2: Diagnosis Generation

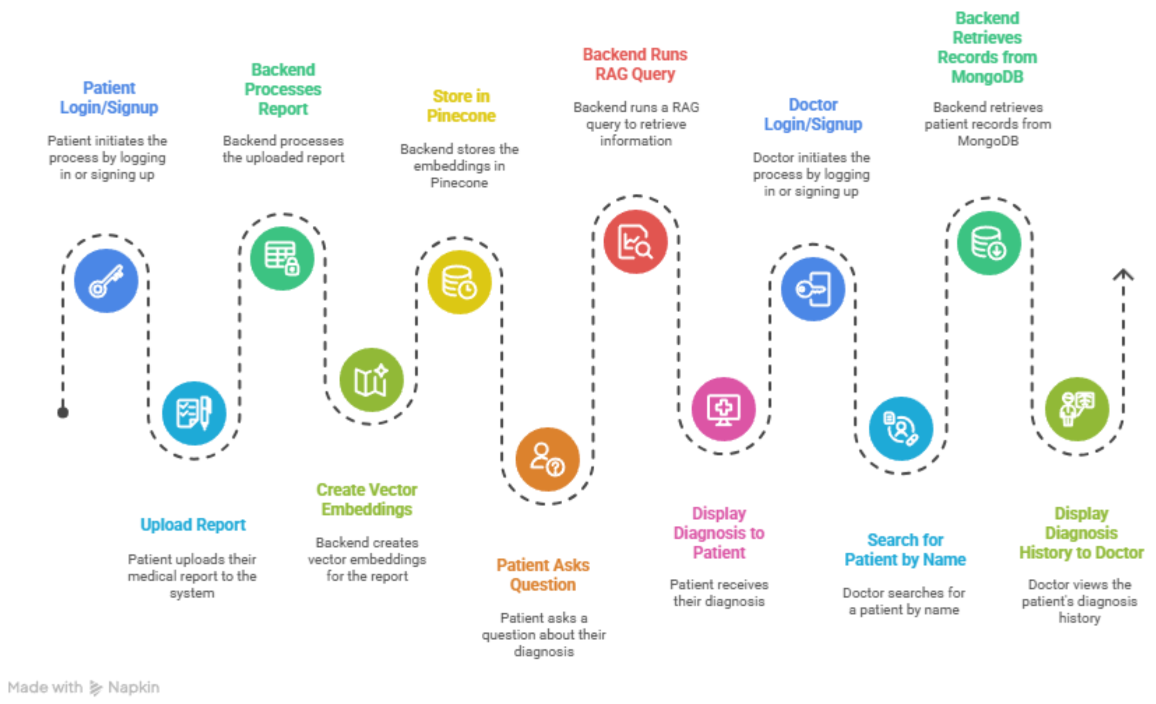
The patient enters a question and the Document ID they received. The frontend sends this to the `diagnosis/from_report` endpoint. The backend uses **LangChain** to coordinate the following:

1. It takes the patient's question and converts it into a vector embedding.
2. It queries the **Pinecone database** to find the most relevant sections of the patient's report by comparing the question's vector with the stored document vectors. This is the **retrieval** part of the RAG pattern.
3. LangChain then constructs a prompt that includes both the patient's question and the retrieved, relevant context from their report.
4. This comprehensive prompt is sent to the **Groq LLM**, which generates a final, contextual diagnosis.
   The backend saves this diagnosis record (including the question, answer, and source document) in **MongoDB** before sending it back to the frontend.

3. Doctor Workflow (Viewing Patient Records)

A doctor logs in and navigates to their dashboard. They can enter a patient's username into the search bar. The Streamlit frontend sends a request to the `diagnosis/by_patient_name` endpoint. The FastAPI backend queries the **MongoDB database** for all diagnosis records associated with that patient's username. The retrieved records are sent back to the frontend, which displays them in a clean, organized format for easy review.

Medical Diagnosis Application Flowchart


Patient Login/Signup
Patient initiates the process by logging in or signing up
Upload Report
Patient uploads their medical report to the system
Backend Processes Report
Backend processes the uploaded report
Create Vector Embeddings
Backend creates vector embeddings for the report
Store in Pinecone
Backend stores the embeddings in Pinecone
Patient Asks Question
Patient asks a question about their diagnosis
Backend Runs RAG Query
Backend runs a RAG query to retrieve information
Display Diagnosis to Patient
Patient receives their diagnosis
Doctor Login/Signup
Doctor initiates the process by logging in or signing up
Search for Patient by Name
Doctor searches for a patient by name
Backend Retrieves Records from MongoDB
Backend retrieves patient records from MongoDB
Display Diagnosis History to Doctor
Doctor views the patient's diagnosis history
Made with Napkin