"Mixture of Doctors": an agentic-RAG system for self-management of chronic diseases

Luca Samorè
luca.samore@studio.unibo.it

Lucia Castellucci
lucia.castellucci2@studio.unibo.it

Roberto Mitugno
roberto.mitugno@studio.unibo.it

July 2025

Supporting patients in the self-management of chronic diseases requires continuous, personalized guidance, a task for which LLM-powered chatbots show immense promise. However, their deployment is contingent on overcoming a dual challenge: the inherent risk of LLM "hallucinations", which undermines trust, and the architectural complexity of building a system that can manage multiple medical domains securely and reliably. This report introduces "Mixture of Doctors", an agentic-RAG system designed to address these challenges in tandem. The former is mitigated through a Retrieval-Augmented Generation (RAG) flow, where specialized "Doctor" components, each an expert on a specific chronic disease, ground their responses in curated knowledge bases to ensure factual accuracy. The latter is addressed by a robust, decoupled Orchestrator-worker architecture that leverages asynchronous messaging for resilience and open-source technologies for data privacy. The system's core is an agentic Orchestrator that dynamically analyzes user queries, decomposes complex, cross-domain questions, and routes them to the appropriate specialists. The resulting framework demonstrates a viable path toward creating trustworthy and scalable conversational AI for personalized chronic care.

1 Introduction

Chronic diseases are known to have long-lasting ramifications, extending over an individual's entire lifespan, necessitating ongoing management by both patients and healthcare

practitioners. The enduring nature of these conditions significantly impacts individuals' health related quality of life and generates substantial healthcare costs due to disability, recurrent hospitalizations and treatment measures [1]. In this context, patient self-management is demonstrated to play a central role in patient care, as it not only contributes to improve physical and psychological health outcomes but also alleviates the strain on healthcare resources [2]. In recent years, chatbots are emerging as a powerful tool for supporting patients in managing their chronic conditions. The goal is to provide timely, accurate, and empathetic guidance, including personalized motivational messages that [3] (i) sustain patient engagement in self-management behaviors, (ii) enhance awareness of adherence and performance relative to clinical recommendations, and (iii) foster motivation, thereby potentially promoting healthier lifestyle choices.

Given their ability for interpreting and generating human-like text, Large Language Models (LLMs) have shown potential in improving chatbot-based systems in healthcare. However, despite their extensive contextual understanding and robust question answering abilities, these models face significant challenges when applied in medical settings. One of the most pressing concerns is the phenomenon of *hallucinations*, i.e. the generation of plausible-sounding but factually incorrect or misleading information. This limitation becomes particularly critical in domains with precise and difficult real-world questions, especially in fields such as medicine where high and complex reasoning is required. Moreover, in the absence of a healthcare professional mediating the exchange, ensuring the reliability and safety of chatbot-delivered information becomes paramount. Another critical challenge to be addressed concerns the protection of sensitive patient information, as patients interacting with the chatbot are likely to disclose personal medical data. This need arises not only from ethical and practical considerations, but also from the necessity to comply with established legal frameworks, such as the Health Insurance Portability and Accountability Act (HIPAA) [4] and the General Data Protection Regulation (GDPR) [5].

2 Concept

To address the aforementioned challenges, we propose an agentic workflow that leverages Retrieval-Augmented Generation (RAG) to enhance the accuracy and contextual relevance of responses generated by Large Language Models. The system is composed of multiple specialized "doctors", each focused on a specific chronic condition, and is accessible through a command-line interface (CLI). By exploiting the RAG technique, each "doctor" dynamically retrieves relevant information from an external source and incorporates it into the generative process, thereby ensuring outputs that are more reliable and up-to-date. Cross-domain queries are also supported by carefully analyzing user input, identifying the relevant chronic conditions and intelligently routing the query to the appropriate set of specialists. The individual generated responses are then aggregated and synthesized to produce a cohesive final answer. To meet data privacy needs, open-source Large Language Models are adopted to avoid reliance on proprietary systems, while user queries and corresponding responses are securely retained in locally deployed

data storage. Additionally, a single-user model is implemented: all users interact with the system in the same way, with no role-based differentiation.

Generally speaking, such a system could serve multiple purposes. Indeed, the body of research focusing on LLMs applications has grown significantly in recent years, reflecting the increasing interest within the scientific community working on artificial intelligence applied to medicine. Some possible applications include [6]:

- **Patient Care**: LLM-based systems can enhance patient care by assisting clinicians in decision-making, detecting inconsistencies, and summarizing complex medical information. They also have the potential to empower patients through accessible, empathetic dialogue and personalized advice, ultimately supporting more informed and compassionate care.
- **Research**: LLMs can automate tasks such as data extraction, summarization, and literature review, helping researchers navigate and synthesize vast scientific outputs. While they may foster new insights and democratize knowledge, concerns about accuracy and reproducibility require careful oversight.
- **Education**: LLMs can function as adaptable tutoring tools, providing tailored explanations, simulations, and feedback. Their strong exam performance highlights their potential as study aids, but they must be used responsibly to avoid diminishing students' critical thinking and reasoning skills.

In this project, we concentrate our attention primarily on people living with chronic conditions. Among the wide spectrum of possible diseases, we have selected three well-known conditions as case studies: diabetes, multiple sclerosis and hypertension. This choice is deliberate, as these conditions represent a diverse range of self-management challenges, allowing for a comprehensive evaluation of our system's capabilities. Importantly, recent research has already begun to map the potential and pitfalls of LLMs in these specific areas [7, 8, 3], establishing a timely and relevant context for our contribution.

3 Motivation

A primary challenge in deploying Large Language Models (LLMs) for medical applications is mitigating the risk of hallucinations. To ensure our system provides reliable and safe guidance, we considered three primary mitigation strategies.

The first, prompt engineering, involves carefully crafting inputs to guide the model's output. While useful, this approach is fundamentally constrained by the model's static, internal knowledge base, which may lack the most current or specialized medical information and is prone to generating outdated advice [9].

A more direct approach, fine-tuning, adapts a pre-trained model to a specific domain by continuing the training process on a custom dataset. However, this strategy presents significant practical barriers. It is computationally expensive, demanding substantial

hardware resources and time for training. Even with Parameter-Efficient Fine-Tuning (PEFT) techniques like LoRA [10], which reduce the computational load by updating only a subset of model parameters, the process remains resource-intensive. Furthermore, fine-tuning requires the meticulous construction of large, high-quality, domain-specific datasets, a non-trivial undertaking in itself [11]. This approach also lacks scalability: incorporating new medical knowledge or expanding to new diseases necessitates a complete, costly retraining cycle. Finally, the fine-tuning process carries the inherent risk of catastrophic forgetting, where the model's performance on previously learned general tasks degrades as it specializes [12].

To circumvent these limitations, we adopt a Retrieval-Augmented Generation (RAG) architecture. RAG enhances the LLM by enabling it to dynamically retrieve relevant information from a curated, external knowledge base at inference time. This method effectively mitigates hallucinations by grounding the model's responses in verified, up-to-date sources, ensuring accuracy without altering the model's core weights [13, 14]. Unlike fine-tuning, RAG is highly scalable: the knowledge base can be updated or expanded with new information independently and instantly, without the need for model retraining. This makes RAG a more robust, flexible, and efficient solution for our use case.

Alongside accuracy, a foundational requirement for our system is ensuring user data privacy. Proprietary, closed-source models typically process data on third-party servers, posing unacceptable privacy risks for sensitive health information. Therefore, we exclusively utilize open-source (or more accurately, open-weight) LLMs. This approach allows for local deployment, ensuring that all user interactions and personal data remain within a secure, controlled environment, thereby complying with stringent data privacy standards.

4 Requirements

4.1 Functional Requirements

ID	Requirement	Description	Acceptance Criteria
F1	Query Submission	The system shall allow users to submit natural language queries related to chronic diseases.	The user can input a question via CLI and receive a response.
F2	Response Generation	The system shall generate a response, possibly using RAG technique.	The response is contextually relevant and may include information retrieved from the knowledge base.

ID	Requirement	Description	Acceptance Criteria
F3	Disease-Specific Doctors	The system shall route queries to the appropriate doctor based on the disease topic.	Queries about different diseases are handled by different specialized doctors.
F4	Query Logging	The system shall store all user queries and corresponding responses for future analyses.	A log file or database entry is created for each interaction.
F5	CLI Interaction	The system shall provide a command-line interface for user interaction.	The user can interact with the system entirely through the CLI.
F6	Conversation Persistance	The system shall store user conversation to allow retrieval of past interactions in future sessions	The user can access previous questions and responses upon returning to the system.
F7	User Identification	The system is able to distinguish different users.	The user is recognized and all its information is retrieved upon returning to the system.

4.2 Non-Functional Requirements

ID	Requirement	Description	Acceptance Criteria
NF1	Privacy Compliance	The system shall comply with GDPR and other relevant data privacy regulations.	No personal data is stored unless explicitly authorized; data is anonymized.
NF2	Availability	The system shall be available for use at least 99% of the time.	System uptime is monitored and meets the threshold.

ID	Requirement	Description	Acceptance Criteria
NF3	Scalability	The system shall be designed to support the addition of new disease-specific doctors.	New doctors can be added without modifying the core architecture.
NF4	Usability	The CLI shall be intuitive and require minimal training.	Users can complete basic tasks without consulting documentation.
NF5	Performance	The system shall return responses within 5 seconds for 95% of queries.	Response times are logged and meet the performance target.

4.3 Implementation Requirements

ID	Requirement	Description	Acceptance Criteria
I1	Use of open-source LLMs	The system shall use open-source LLMs to ensure data privacy and cost control.	All deployed models are open-source and locally hosted.

5 Glossary

Term	Synonymous	Definition
Doctor	RAG Module, Expert, Specialist	A system module that delivers accurate responses to user queries by referencing a knowledge base.
Conversation	Chat	Sequence of question-answer pairs between users and chatbot.

6 Design

6.1 Architecture

The agentic workflow is built upon an Orchestrator-workers architecture, as illustrated in Figure 1. This model comprises a central coordinating component (the Orchestrator) and a set of specialized, independent workers.

The workflow is initiated when the Orchestrator receives a user query. In this phase, it exhibits agentic-like behavior: it reasons about the user's needs, decomposes the high-level request into specific sub-tasks, and delegates them to the appropriate workers. These workers, which we refer to as "Doctors" are specialized modules, each equipped with a dedicated knowledge base for a single chronic condition. Upon receiving a task from the Orchestrator, each Doctor utilizes a Retrieval-Augmented Generation pipeline to generate a response grounded in its specific domain of expertise. For cross-disease queries involving two or more conditions, the Orchestrator routes the request to all relevant Doctors. The resulting set of responses is then passed to a Synthesizer component, which aggregates and fuses them into a single, cohesive answer.

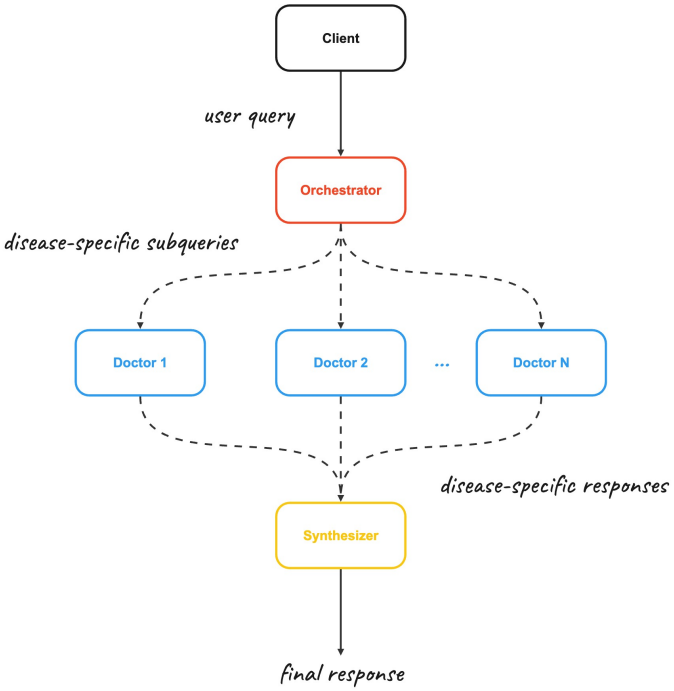

Figure 1: High-level architecture of the agentic workflow.

This orchestrated approach was motivated by several key advantages:

- **Scalability**: support for new medical conditions can be implemented by simply adding new, self-contained Doctor modules without altering the core system logic.

- **Dynamic Task Handling**: this workflow is well-suited for complex tasks where the required sequence of sub-tasks cannot be predetermined. The Orchestrator's ability to dynamically plan and delegate makes the system adaptive to diverse and unpredictable user inquiries.
- **Modularity and Maintainability**: by isolating expertise into distinct modules, we ensure that the system is robust and easy to maintain. Updates to one knowledge domain do not impact the stability of others.

The name "Mixture of Doctors" was chosen as it conceptually mirrors the popular Mixture of Experts (MoE) architecture used in LLMs with hundreds of billions of parameters, where distinct "expert" models are employed to handle different subsets of a problem space [15].

6.2 Infrastructure

The system's infrastructure is implemented as a distributed architecture composed of several interconnected services. User interactions originate from a command-line interface client, with all requests directed to a central API Gateway that serves as the single entry point. The gateway routes traffic based on the request's purpose: new medical queries are forwarded to the Orchestrator's REST API, while requests for past interactions are sent to a dedicated Chat History API. The core response generation workflow, involving the Orchestrator, the specialized Doctor components, and the Synthesizer, leverages a message broker for asynchronous communication. Service discovery within this distributed environment is managed via DNS, enabling components to reliably locate one another through their service names. Each Doctor is coupled with a dedicated vector database, which acts as its knowledge base for the Retrieval-Augmented Generation process. For all natural language processing tasks, the workflow components make calls to a remote LLM cluster. A separate database is utilized to persist user conversation logs, and a caching layer stores recently generated responses to optimize performance. Regarding the deployment topology, all primary services—including the gateway, workflow components, and databases—reside within a common network. The LLM cluster, however, is deployed on an isolated network to separate its computational workload.

6.3 Modelling

6.3.1 Domain Entities

Entity	**Description**
User	An entity representing the patient interacting with the system.
Query	A request for information submitted by a User.

Entity	Description
Response	A system-generated answer corresponding to a User's Query.
Session	An entity that stores the context and history of a User's interaction.

6.3.2 Mapping of Entities to Infrastructural Components

Infrastructural Component	Role
Database	Persists the state of User and Session (chat history) entities.
Message Broker & Cache	Handle the transient flow of Query and Response objects.
Processing Modules	Orchestrator, Doctor and Synthesizer components are responsible for executing the core domain logic related to query processing and response generation.

6.3.3 Message Types and System State

Inter-component communication is achieved through two message patterns:

- **Commands**: imperative messages that instruct a component to perform a task (e.g., a request to the Orchestrator to process a query).
- **Queries**: request-response messages for retrieving data (e.g., a request to the Chat History API for a user's past conversations).

The complete state of the system comprehends all data related to its entities, including persistent user profiles, full chat histories, and logs of all past and current queries and responses.

6.4 Behaviour

6.4.1 Orchestrator

The Orchestrator serves as the central entry point and core component of the entire workflow. It is responsible for ingesting user queries, determining the appropriate execution path, and ultimately formulating the final response. Its primary duties include delegating queries to one or more specialized RAG modules (i.e., Doctors) when necessary or, alternatively, responding directly to the user for queries that are out-of-scope or do not pertain to the supported chronic conditions. In cases involving multiple RAG modules, the Orchestrator is also tasked with decomposing the main query into domain-specific sub-queries.

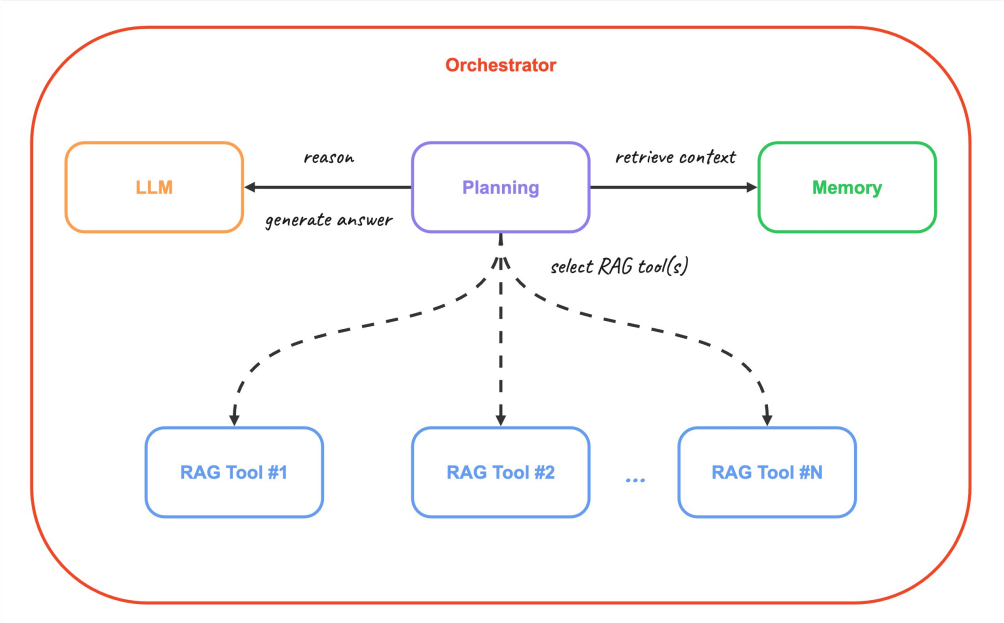

Figure 2: Radiography of the Orchestrator entity.

As shown in Figure 2, the Orchestrator is designed as an augmented Large Language Model and comprises four key elements:

- A **planning module**, which contains the core logic that dictates the Orchestrator's behavior.
- A **core LLM** used for reasoning and response generation.
- A set of available **tools**, which in this architecture are the specialized RAG modules. Each tool is identified by the name of the disease.
- A long-term **memory**, implemented as a database that stores the complete user chat history (queries and responses).

When the Orchestrator receives a query, its planning module executes a two-phase decision-making process, inspired by the Adaptive-RAG framework [16]:

1. **Phase 1: Query Classification**. The initial phase analyzes the user's query to determine its complexity, classifying it as `easy`, `medium`, or `hard`. Easy queries are typically simple conversational turns or out-of-domain questions. Medium queries relate to a single supported chronic condition. Hard queries are cross-domain, touching upon multiple conditions.
2. **Phase 2: Orchestration**. The second phase uses the complexity classification to select an execution strategy:

- `easy`: the Orchestrator bypasses the RAG tools and generates a direct response.
- `medium`: the query is routed to the single, relevant RAG module.
- `hard`: the query is decomposed into k sub-queries, each dispatched to its corresponding RAG module.

Rather than implementing this classification and routing logic with hard-coded rules, we leverage an LLM to make these decisions, giving the Orchestrator its agentic capabilities. This is achieved through prompt engineering: the LLM is instructed to analyze the query and return its decision in a predefined JSON schema. This structured output is then programmatically parsed by the system to either invoke the appropriate tool(s) or formulate a direct answer. This approach effectively implements a form of tool-calling, where the LLM decides which function to execute. To maintain conversational context, the Orchestrator is endowed with long-term memory, implemented as a database that persists the entire chat history for each user. Prior to invoking the LLM for any answer generation task, the full conversation history is retrieved and appended to the prompt. This ensures that all interactions are contextually aware, enabling the system to handle follow-up questions and maintain a coherent dialogue.

6.4.2 Doctor

Each Doctor component is conceptualized as a retrieval-augmented Large Language Model, specialized in a single chronic disease. It operates as a specialist worker, receiving tasks from the Orchestrator and generating evidence-based responses through a Retrieval-Augmented Generation (RAG) pipeline. The destination of its generated output is contingent on the query's complexity: for a medium-complexity query, the response is finalized for the user, whereas for a hard (sub-)query, the response is channeled to the Synthesizer for further processing.

The internal architecture of a Doctor, represented in Figure 3, implements a standard RAG pipeline. This architecture is composed of two primary elements: a disease-specific **vector store**, which serves as the knowledge base, and a **generator** module. The generator itself consists of a **retriever** responsible for fetching relevant information and a core LLM for generating the final text.

The operational flow begins with a preliminary **ingestion phase**, where the vector store is populated with embedded text chunks from a curated document corpus; for this case study, documents are sourced from PubMed. At runtime, when a query is received, the retriever performs a similarity search against the vector store to identify the top-k most relevant document chunks. These chunks are then combined with the original query to create an augmented prompt, which is passed to the LLM to produce a grounded, context-aware response. It is important to note that this retrieval process is unimodal, operating exclusively on textual embeddings and not supporting multi-modal information from sources such as images or audio files.

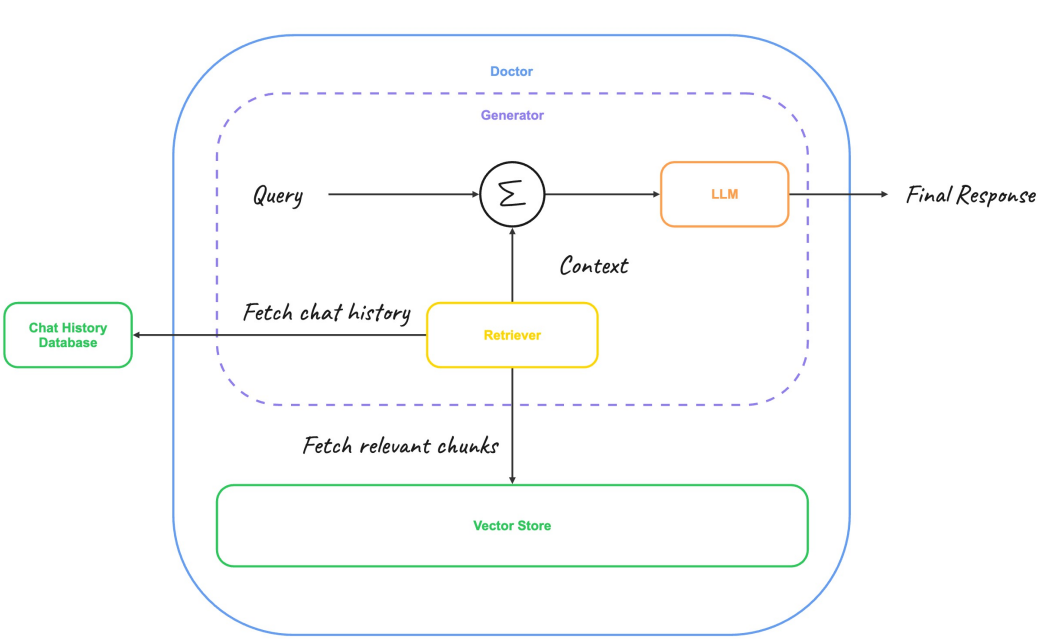

Figure 3: Radiography of the Doctor entity.

6.4.3 Synthesizer

The Synthesizer (Figure 4) is a specialized Large Language Model tasked with aggregating multiple responses from the RAG modules into a single, comprehensive answer. Its purpose is to perform a lossless synthesis of information, integrating the distinct outputs from each specialist Doctor into a unified response for the user.

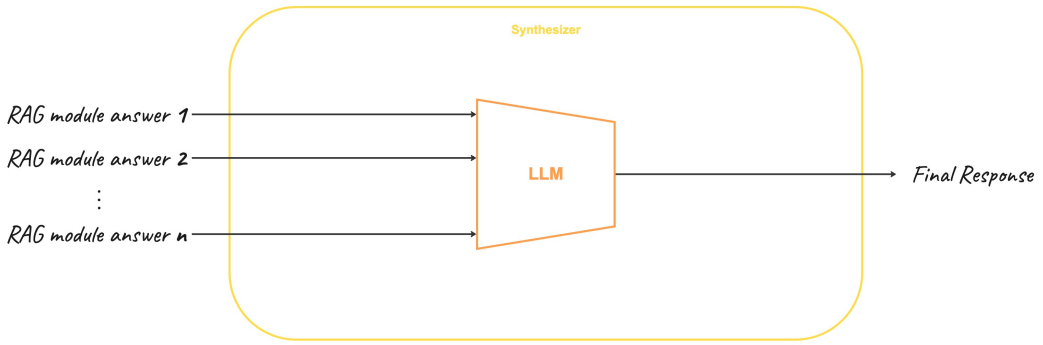

Figure 4: Radiography of the Synthesizer entity.

Two primary architectural options were evaluated for its implementation. The first was to embed the Synthesizer directly within the Orchestrator as an internal set of functions. This would simplify deployment, as both components would share a single unit. However, this approach creates tight coupling, making the Synthesizer's availability entirely dependent on the Orchestrator's operational state. The second option was to implement it as a standalone service, with its own deployment unit and dedicated server. This service-oriented approach decouples the Synthesizer from the Orchestrator, enhancing system resilience and modularity. Despite the increased management complexity associated with an additional service, we selected this latter approach. The benefits of architectural decoupling and improved robustness were deemed to outweigh the simplicity of an embedded design.

6.5 Interaction

The communication between system components is governed by distinct interaction patterns. For easy queries, the system employs a synchronous request-response paradigm. As shown in Figure 5, the user submits a query via the CLI and blocks while awaiting a response. The Orchestrator receives the request, issues a synchronous call to the LLM cluster for reasoning, and queries the Chat History API to retrieve the conversational context. Upon generating the final answer, it is sent back to the client. This synchronous model is well-suited for these simple interactions, as they mirror a direct question-answer flow where the user expects an immediate reply.

In contrast, for medium-complexity queries, the interaction shifts to an asynchronous publish-subscribe mechanism. The Orchestrator publishes the query to a dedicated event queue and can immediately proceed to other tasks. As illustrated in Figure 6, the relevant Doctor component, acting as a subscriber, consumes this message asynchronously and initiates the RAG process. In this flow, the Doctor is responsible for fetching the chat history itself before generating its response. This asynchronous pattern extends to hard queries (Figure 7), where the individual responses from each RAG module are themselves published to another event queue to be consumed asynchronously by the Synthesizer.

The adoption of an asynchronous model for complex queries was a deliberate architectural choice over a purely synchronous alternative. A synchronous implementation would introduce significant points of failure and tight coupling. For instance, if the Orchestrator were to fail while a Doctor is processing a lengthy RAG request, the communication channel for returning the response to the client would be severed. In this case, implementing retry logic would be impractical due to the high latency associated with re-generating RAG responses. A similar issue would arise in the communication between the Orchestrator and the Synthesizer. On the other hand, a publish-subscribe model offers several compelling advantages for this architecture:

- **Durability**: in the event of transient component failure, messages persist within the broker. This ensures no data loss and enables processing to resume once the service is restored, guaranteeing eventual consistency.

- **Decoupling**: it promotes loose coupling between system components. The Orchestrator (producer) does not need to be aware of the state, location, or even the existence of the specific Doctors (consumers), which enhances modularity.
- **Improved responsiveness**: the Orchestrator's throughput is increased, as it is not blocked while waiting for long-running RAG processes to complete. It can publish a task and immediately become available to handle new incoming requests.

A final, unifying element across all three interaction patterns is the mechanism for delivering the final response. Regardless of query complexity, the answer is streamed to the user as a sequence of tokens. However, this stream is not transmitted directly from the generating component to the client. For medium and hard queries, the asynchronous, decoupled nature of the architecture means that the RAG modules and the Synthesizer have no knowledge of the end-user's identity or session details, making a direct communication channel impossible. To address this challenge, we have introduced an intermediary caching layer engineered to support streaming data. Following query submission, the client initiates a listener on a unique cache key. The component responsible for the final answer—be it the Orchestrator, a Doctor, or the Synthesizer—then publishes the token stream to this designated key, enabling the client to consume the response as it is being generated.

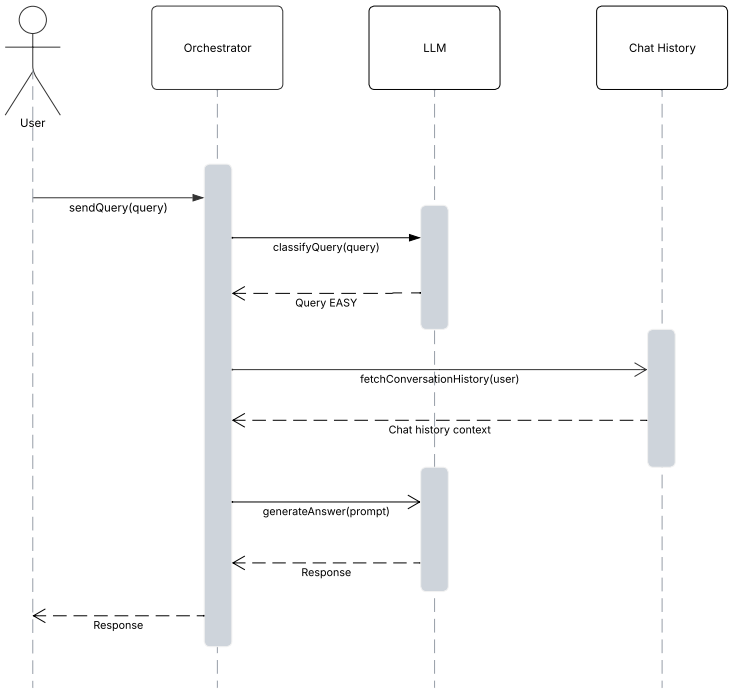

Figure 5: UML sequence diagram of an easy query path.

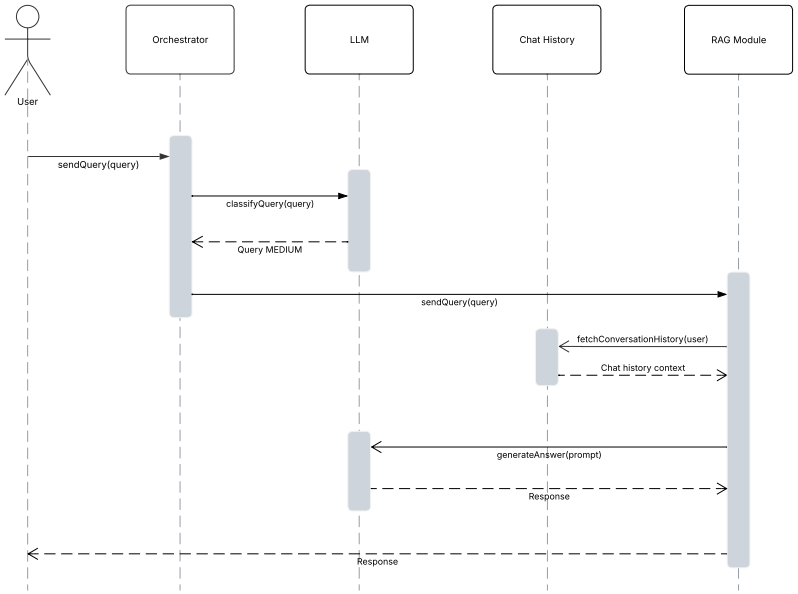

Figure 6: UML sequence diagram of a medium query path.

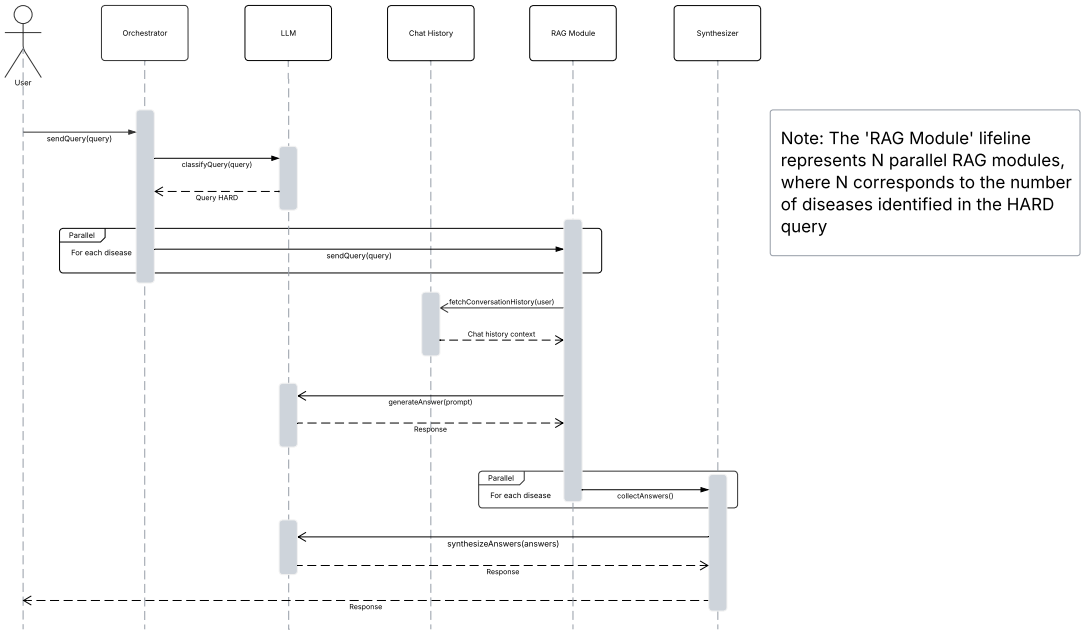

Figure 7: UML sequence diagram of an hard query path.

6.6 Availability

The primary objective of the system's design is to ensure high availability, maximizing the probability that a user receives a timely and relevant response to their query. As a real-time conversational system, perceived uptime is paramount; long delays or outright failures severely degrade the user experience. The strategy to achieve this hinges on eliminating single points of failure within the critical request path. The core components of this path, the API Gateway and the Orchestrator, are therefore designed with resilience in mind. Since both services are designed as stateless entities, they are amenable to horizontal scaling. By deploying multiple, identical replicas of these components behind a load balancing layer, the system can distribute incoming traffic (e.g., following a round robin policy) and maintain operational continuity even if one or more replicas become unavailable. If a replica of either service fails, the load balancer automatically reroutes subsequent requests to a healthy instance, a process that is entirely transparent to the end-user.

6.7 Fault-Tolerance

To achieve the high availability goals, the system incorporates fault-tolerance mechanisms tailored to the nature of its components. A distinction is made between stateless and stateful services. Stateless components, such as the Orchestrator, the Doctor's generator module, and the Synthesizer, are replicated horizontally. The failure of any single replica is non-critical, as its workload can be immediately assumed by another identical instance without loss of system state.

Managing stateful components—namely the document database for chat histories and the vector databases for RAG—presents a more complex challenge. For these services, fault-tolerance is achieved through data replication using a leader-follower model. In this configuration, one replica is designated as the leader, which is responsible for handling all write operations. These writes are then propagated to a set of follower replicas. While read operations can be served by any replica, this model ensures that the system can withstand the failure of a database node. If a follower fails, it is simply replaced. If the leader fails, the system undergoes a failover process where one of the followers is promoted to become the new leader.

A final challenge exists in the user-facing response delivery. The client listens on a cache for a token stream that may never be published due to a producer failure. To address this, the client implements a retry mechanism with a timeout. Upon initiating a listener, a timer starts. If the timer expires before a response is received, the client retries listening a configurable number of times. If all attempts fail, the client concludes that a non-recoverable error has occurred upstream and resubmits the original query to the system, thus ensuring the user's request is not permanently lost and prioritizing the user's interaction over strict system state guarantees.

6.8 Data and Consistency Issues

The system is architected as an AP system under the CAP theorem, prioritizing availability and partition tolerance over strong consistency. This choice is justified by the application's conversational nature, where receiving a timely response—even if based on momentarily stale data—is preferable to experiencing an error. For instance, if a vector store replica is temporarily unavailable, it is more acceptable for a Doctor to generate a response using only the LLM than to fail the entire query.

Consistency is managed differently across data types. For user-facing chat histories, eventual consistency is the target. A leader-follower replication model may introduce a minor lag, meaning a user's most recent message might not be immediately visible upon a read-from-follower operation, but the data is guaranteed to be persisted on the leader and will propagate. This trade-off is acceptable as it prevents read operations from failing. For the RAG knowledge bases, consistency is primarily a concern during data ingestion, which is an offline or background process; at query time, the system performs read-only operations, making real-time consistency less critical.

7 Implementation

7.1 Infrastructure Components

The system's foundation is built upon a set of scalable infrastructure technologies. These components are not business-logic services themselves but provide the essential groundwork for communication, data storage, and computation.

- **NGINX Reverse Proxy**. NGINX serves as a high-performance reverse proxy and the single gateway for all incoming traffic. It simplifies network topology, enhances security through SSL/TLS termination, and routes requests to the appropriate internal service based on the request path.

- **Redis Streams**. Server-to-client communication is facilitated by Redis Streams. This append-only log data structure within Redis is ideal for publishing real-time response fragments, allowing for token-by-token answer generation in a chatbot application. The choice of Redis was specifically driven by its inherent support for stream data types, which is crucial for guaranteeing real-time writing of continuous response segments. The benefits of an in-memory database, such as extremely low latency for read and write operations, are also leveraged here.

- **Qdrant Vector Database**. The core of the Retrieval-Augmented Generation capability is powered by Qdrant. This specialized vector database stores, indexes, and searches high-dimensional vector embeddings, crucial for finding semantically similar document chunks based on user queries. When used in distributed mode, it employs the Raft consensus algorithm for leader election and ensuring data consistency.

- **MongoDB**. It serves as the persistence layer for storing conversation histories. Its flexible data model, using JSON-like documents, is well suited for varying lengths and structures of conversational data. The choice of a document database over a relational one was driven by the inherent flexibility it offers, avoiding the need for a rigid schema upfront. Document databases excel in handling evolving data structures, which is a significant advantage for storing dynamic conversation histories [17]. MongoDB also utilizes a leader-follower model for replication, enhancing data availability and fault tolerance.

- **Groq API**. Language model inference is provided by Groq, specifically leveraging Llama 3.3 70B model for reasoning and generative tasks, such as query reformulation, answer generation, and answer summarization. This service offers access to high-performance LLMs optimized for low-latency responses.

- **Python**. Python is extensively used throughout the system, leveraging numerous libraries to streamline development. Key frameworks include FastAPI for building APIs, Pydantic for data validation and Typer for creating command-line interfaces.

- **Docker**. Docker is utilized for containerization, ensuring consistent environments across development and deployment. The system employs Docker Swarm for orchestration, managing the deployment and scaling of services, and handling load balancing across service replicas.

- **Apache Kafka**. For asynchronous communication, the system relies on Apache Kafka. This distributed event streaming platform provides high-throughput and fault-tolerant messaging, enabling durable and decoupled communication between services. A detailed explanation of what features of the technology were used follows in the subsection below.

7.1.1 Kafka Integration for Scalable and Resilient Messaging

A fundamental feature utilized in this project is the **consumer group**. Services that consume messages from a specific topic are organized into a single consumer group. This configuration allows for the horizontal scaling of these services. Kafka distributes the topic partitions among all active consumer instances within the same group, ensuring that each message is delivered to only one consumer. This mechanism effectively **balances the processing load** across all available replicas of a service, preventing message duplication and allowing the system to handle a **higher throughput** by simply deploying additional consumer instances.

To guarantee reliable message processing, the system was designed to handle message acknowledgment manually rather than relying on Kafka's default auto-commit behavior: the `enable.auto.commit` **configuration is disabled**, and offsets are committed explicitly only after a message has been successfully processed in its entirety.

This manual offset control is crucial for achieving **fault tolerance**. If a consumer instance fails or encounters an error midway through processing, it will not have committed

the offset for that message. As a result, the message remains in the topic partition and is automatically reassigned by the Kafka broker to another healthy consumer instance within the same group. This ensures "at-least-once" delivery semantics, providing a guarantee that no messages are lost due to application-level failures and maintaining data integrity across the system.

7.2 Components and Interaction Implementation

This section provides a detailed overview of the implementation of the system. It covers the individual components, their underlying technologies, and the communication protocols that enable their interaction within the services architecture.

7.2.1 Frontend Component

The frontend component, implemented as a Python Command-Line Interface (CLI), serves as the primary user interface for the system. Its core responsibility is to capture user input and initiate the agentic workflow. This is achieved by sending the user's query as an HTTP POST request to the NGINX reverse proxy, which routes it to the Orchestrator. While the backend processes the request, the CLI subscribes to a dedicated Redis Stream to asynchronously receive the token-by-token response, leveraging the Redis client library and the RESP/TCP protocol. Additionally, the CLI handles the retrieval of past interactions by making direct HTTP GET requests to the Chat History service's REST API. All data payloads exchanged throughout these interactions are consistently formatted as JSON.

7.2.2 Chat History Component

The Chat History component is a dedicated service responsible for storing and retrieving conversation histories. It serves as a single source of truth for conversational context, queried directly by the frontend and other backend services. The service is implemented using the FastAPI framework. For data persistence, it connects to a MongoDB database. Regarding communication protocols, the Chat History component exposes a REST API, which is accessible directly by other services within the container network. All interactions—whether a POST request to save a message or a GET request to retrieve a conversation—happen through this API. As with the rest of the system, all data payloads exchanged with the Chat History service are formatted as JSON.

7.2.3 Orchestrator Component

The Orchestrator serves as the central coordination hub, initiating the agentic workflow upon receiving a user's query from the frontend via the NGINX proxy. Its first action is to perform a preliminary analysis through an LLM inference call via the Groq API, which classifies the query's complexity and determines the appropriate execution path. The subsequent actions are dictated by this classification.

For easy queries (Figure 8), the Orchestrator then proceeds to retrieve the full conversational context from the Chat History service's REST API. This history is combined with the query to formulate a final prompt, which is used to generate a response that is streamed directly to the frontend via Redis (RESP/TCP). Conversely, for medium and hard queries (Figures 9 and 10), it acts as a Kafka producer, publishing the original query and its metadata to the relevant RAG modules using Kafka's binary protocol. Interoperability across all interactions is ensured by formatting data payloads as JSON.

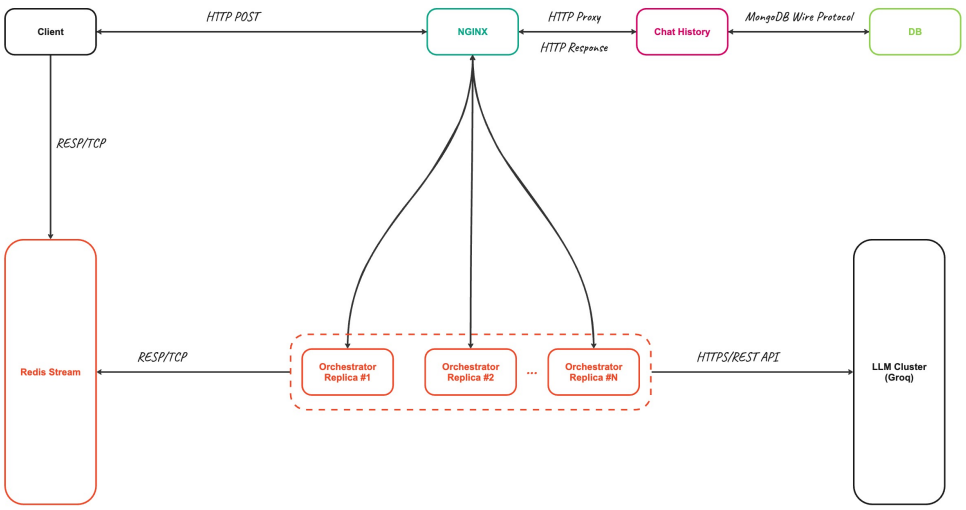

Figure 8: Implementation schema of easy query.

7.2.4 RAG Module Component

A RAG module component includes a Kafka consumer to receive (sub-)queries from the Orchestrator, a Kafka producer for cases requiring summarization, and a Redis publisher for direct responses. It integrates with the Qdrant API for medical knowledge retrieval and the Groq API for LLM inference.

The communication protocols are determined by the workflow. The module receives (sub-)queries from the Orchestrator via a Kafka topic using the TCP binary protocol and retrieves conversational context via a direct request to the Chat History service's REST API. Depending on the scenario, if multiple RAG modules are involved, it sends its generated content to the Synthesizer via a Kafka topic (Figure 10). If the module is solely responsible for the answer, it publishes its complete intermediate response directly to the frontend via Redis Stream using RESP/TCP (Figure 9). For data retrieval and inference, it interacts with the Qdrant and Groq APIs via HTTPS (REST). All data is exchanged in JSON format.

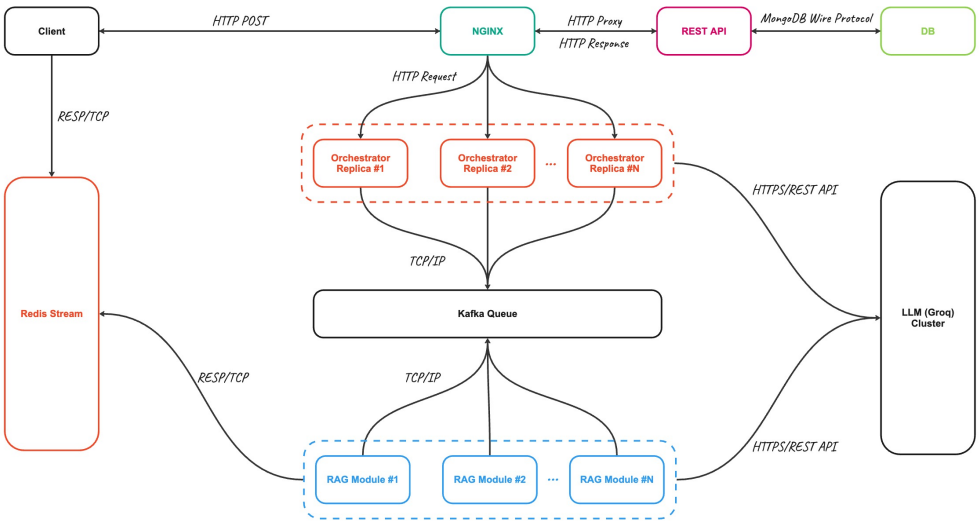

Figure 9: Implementation schema of medium query.

7.2.5 Synthesizer Component

The Synthesizer component aggregates responses from the RAG modules to formulate a comprehensive final answer 10. The component is built as a Python-based service. It features a Kafka consumer and an integration with the Groq API. A Redis publisher streams the final response to the frontend. Its communication patterns are streamlined. It subscribes to messages from the RAG modules via Apache Kafka's binary TCP protocol. It then makes HTTP calls to Groq cloud services for the final inference step. The final answer is published to the frontend via Redis Streams over RESP/TCP. The data format used is JSON.

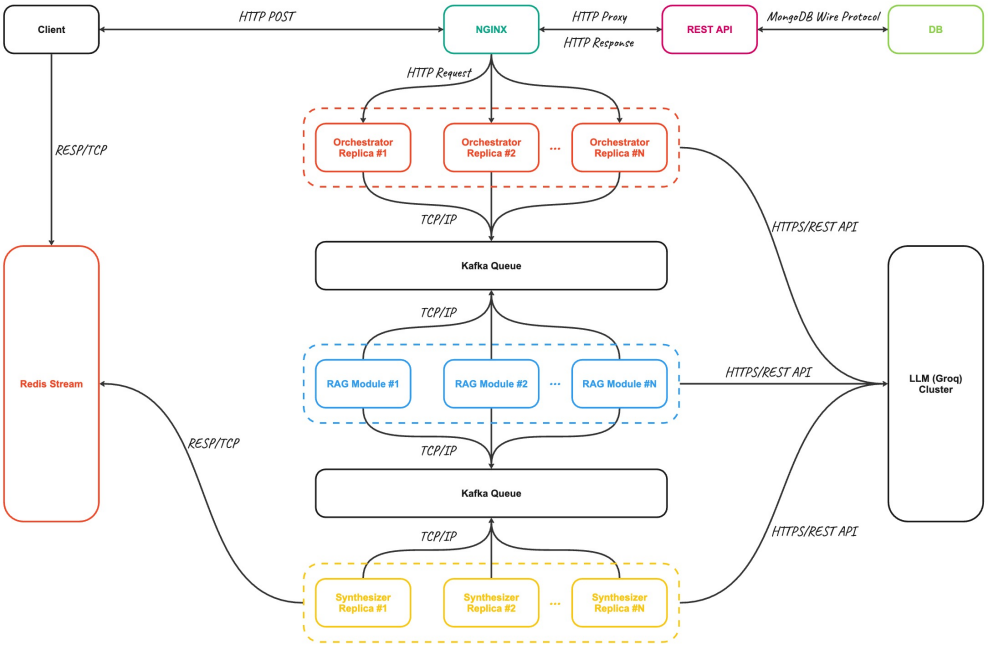

Figure 10: Implementation schema of hard query.

8 Validation

8.1 Automatic Testing

The system's automated validation is centered on a Continuous Integration (CI) pipeline implemented with GitHub Actions. This pipeline automates static code analysis (`Ruff`), type checking (`Pyright`), and the execution of two distinct levels of testing for each module

- **Unit Tests**. These verify individual components in complete isolation. All external dependencies, such as databases and external APIs, are mocked to ensure deterministic and repeatable outcomes.
- **Integration Tests**. These validate inter-service communication using a hybrid approach. Core services are instantiated as real instances, while their collaborators are replaced with mocks at defined boundaries. This strategy allows for the precise validation of communication protocols and data formats without the overhead of a full-system deployment.

End-to-end testing was deliberately omitted from the current strategy to avoid the

complexity and non-determinism of a full production-like environment, prioritizing maintainable and precise tests instead.

8.2 Acceptance Testing

Acceptance testing was performed manually to evaluate the system's emergent properties under realistic clinical query conditions, as these behaviors cannot be reliably validated through automated means. The evaluation focused on the end-to-end user experience, from query submission to the final streamed response. Key qualitative aspects were assessed, including:

- The coherence of responses generated by the language models.
- The relevance of information retrieved by the RAG modules.
- The quality of the final synthesized answer in cross-domain queries.

This manual approach was necessary to assess the nuanced, non-deterministic outputs of the generative models and to provide the domain-specific expertise required for validating medical accuracy.

9 Deployment

The deployment procedure leverages Docker Swarm to orchestrate the containerized services described in the release strategy. Docker Swarm manages the entire lifecycle of the application stack, including service discovery, scaling, and networking across the nodes.

9.1 Deployment

The deployment process requires a host machine with Docker correctly installed and configured. The procedure is initiated as follows:

1. The process begins by cloning the project's Git repository:

   ```
   git clone https://github.com/LucaSamore/Mixture-of-Doctors
   ```

2. Next, the environment is configured by creating a `setup_envs.sh` file from the provided `setup_envs.example.sh` template, populating it with the necessary environment variables.

3. The environment variables are sourced by executing the script:

   ```
   ./scripts/setup_envs.sh
   ```

4. Once the environment is configured, the entire system is deployed by executing a single automated script from the project's root directory:

   ```
   ./scripts/deploy.sh
   ```

Upon execution, this script automatically builds the Docker images for all services (Orchestrator, RAG modules, Chat History, Synthesizer, etc.) and launches them as a coordinated stack of containers. The Command-Line Interface (CLI) then becomes active in the terminal, allowing for direct interaction with the system. Optionally, to enable immediate inference capability upon deployment, the `--ingest` flag can be appended to populate the vector stores of the RAG modules with the domain-specific corpora:

```
./scripts/deploy.sh --ingest
```

9.2 Redeployment

Individual services can be selectively redeployed without impacting the rest of the system. This is achieved through the following script:

```
./scripts/redeploy.sh --<service-name>
```

Available service names include:

- `--orchestrator`
- `--synthesizer`
- `--chat-history`
- `--rag-module`
- `--nginx`
- `--cli`
- `--all` (redeploys all application services without affecting databases and message brokers)

9.3 Undeployment

A controlled shutdown and complete environment cleanup are facilitated via the undeployment script:

```
./scripts/undeploy.sh
```

This script supports two flags to customize the cleanup process:

- `--volumes`: removes all persistent data volumes, including databases and message queues.
- `--points`: clears the vector embeddings used by the RAG modules, resetting the knowledge base without altering configuration files.

10 User Guide

Once the deployment process is complete successfully, the user is presented with an interactive command-line interface, as shown in Figure 11. The CLI allows users to interact with the system through a dedicated Read-Eval-Print-Loop (REPL) interface. Users can issue commands using the `mod` prefix, or type `help` for a list of available options (Figure 12).

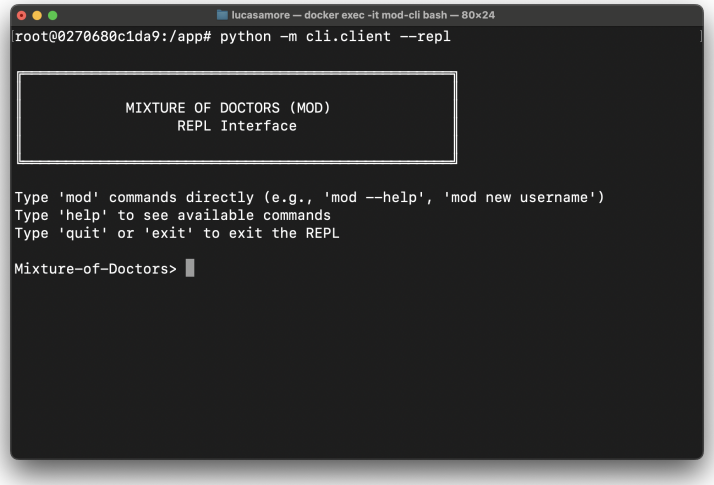

Figure 11: Mixture of Doctors command-line interface.

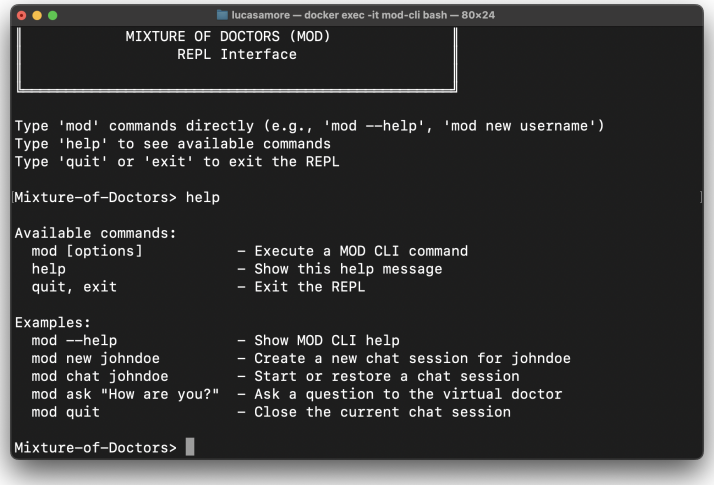

Figure 12: Output of the `help` command.

To begin using the system, users must first establish a session. The `mod new` command creates a chat session for a specified username, initializing the authentication context and displaying a welcome message along with the available commands (Figure 13). This process establishes the foundation for all subsequent interactions with the virtual doctor system.

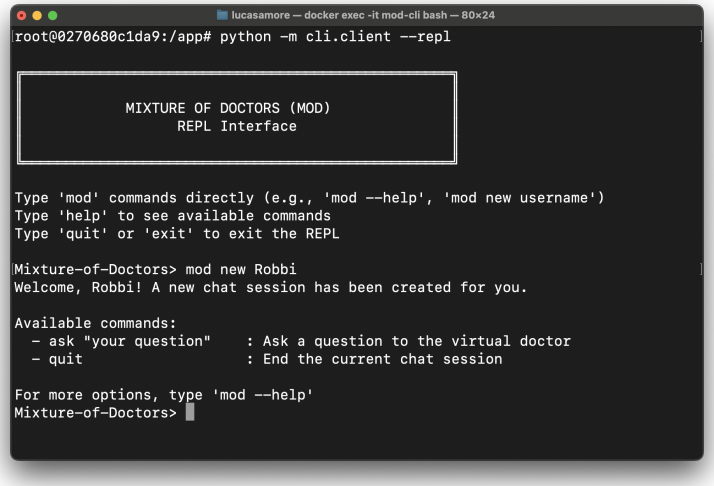

Figure 13: Create a new chat session with `mod new <username>` command.

For users returning to continue previous consultations, the `mod restore` command loads existing chat history and resumes the session context. This functionality displays the complete conversation history in chronological order, allowing users to review their previous interactions before proceeding with new questions (Figure 14).

The `mod chat` command provides automated session management by determining whether to create a new session or restore an existing one based on the current system state. This command streamlines the user experience by eliminating the need to manually select between new and restore operations (Figure 15). Once a session is established, users can ask questions to the chatbot using the `mod ask` command (Figure 16). By default, all interactions are automatically saved to the chat history to maintain consultation continuity. For temporary consultations that should not be stored in the persistent chat history, the `mod ask` command supports the `--oneshot` flag. This mode processes the query and provides a response without updating the stored conversation history, useful for preliminary inquiries or privacy-sensitive questions.

The system provides two distinct methods for termination: exiting the interactive REPL and ending a user session. To simply exit the REPL interface, users can execute either the `quit` or `exit` command, both of which gracefully close the interactive shell. Proper session termination, which includes the cleanup of temporary authentication files, is facilitated by the `mod quit` command. Upon successful execution, the system concludes the session and provides a confirmation message.

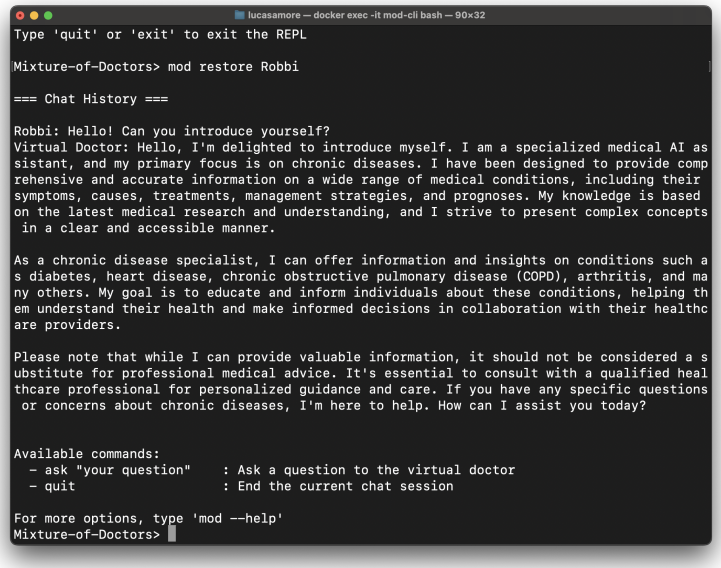

Figure 14: Restore an existing session with `mod restore <username>` command.

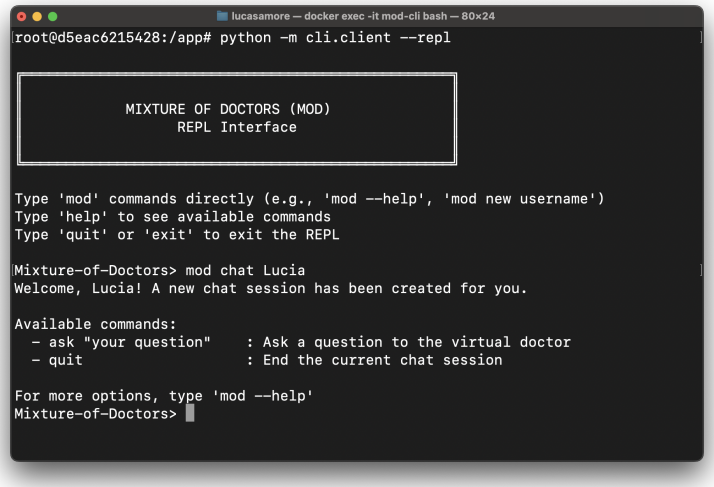

Figure 15: Use `mod chat <username>` command for automatic session management.

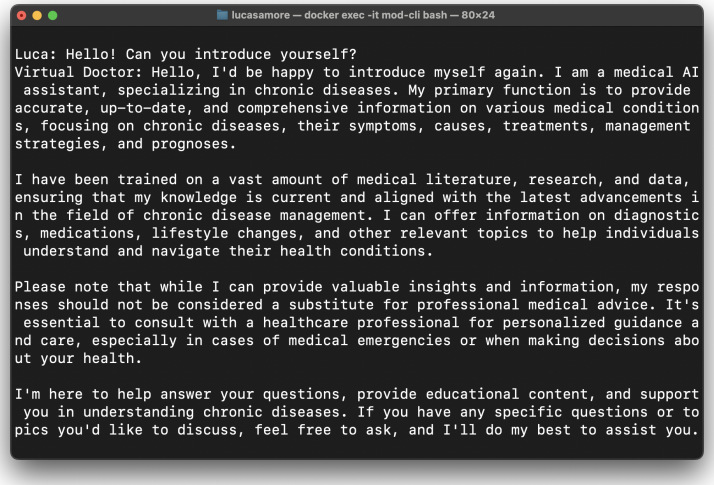

Figure 16: Example of consultation with `mod ask "<question>"` command.

11 Conclusions

This report presented "Mixture of Doctors", an agentic-RAG system designed to deliver reliable support for chronic disease self-management. By combining a distributed architecture of specialized, retrieval-augmented "Doctors" with an agentic Orchestrator, our work addresses the critical challenges of LLM hallucinations and data privacy. The resulting proof-of-concept demonstrates a scalable and resilient framework for building trustworthy, multi-domain conversational AI in sensitive fields like healthcare.

11.1 Limitations and Future Challenges

While the current implementation provides a strong foundational framework, we have identified several limitations and avenues for future work that are critical for enhancing its capabilities and overcoming its current constraints.

11.1.1 Current Limitations

- **Finite Conversational Memory**. The system's long-term memory is constrained by the context window of the underlying LLM. For extended conversations, the current approach of prepending the entire chat history is not scalable. As a temporary measure, the system would need to initiate a new session once the limit is reached, but more sophisticated context summarization or memory management techniques are required for a seamless user experience.
- **Standard RAG Implementation**. The current architecture employs a traditional, single-pass RAG pipeline. This approach, while effective, may be insuffi-

cient for highly complex queries that require multi-step reasoning or the synthesis of information from disparate document chunks.

- **Minimal Security**. The current prototype operates without a formal authentication or authorization layer, making it suitable only for single-user, trusted environments. It also lacks robust safeguards against prompt injection attacks, where a user could craft inputs to bypass the Orchestrator's intended logic.

11.1.2 Future Work and Enhancements

- **Advanced User Interface**. A primary next step is to develop a full-fledged graphical user interface (e.g., a web or mobile application) to replace the current CLI, significantly enhancing system accessibility and user experience.

- **Comprehensive Security Framework**. The transition to a multi-user interface necessitates a robust security framework. This would involve introducing a dedicated authentication service responsible for managing user credentials and issuing access tokens (e.g., JSON Web Tokens). The existing API Gateway (NGINX) would then be configured to act as an enforcement point, routing login requests to this new service and validating tokens on all subsequent requests to the Orchestrator. Implementing this layer of authentication and authorization is essential to ensure that only authenticated users can access the system's core functionalities.

- **Rigorous RAG Evaluation**. A quantitative evaluation of the RAG pipeline's output is necessary. This would involve using established metrics like faithfulness, answer relevance, and context recall to objectively measure the quality and accuracy of the generated responses.

- **GraphRAG Migration**. Transitioning from a standard vector-based retrieval to a GraphRAG [18] approach could unlock more sophisticated reasoning capabilities. By representing knowledge as a graph, the system could leverage explicit relationships between medical concepts to answer more nuanced questions.

- **Multi-Agent System**. The natural evolution of this architecture is to transform the Doctor modules into specialized agents, each equipped with its own internal state and reasoning loop (e.g., a ReAct cycle [19]). Instead of returning a simple text response, a Doctor-agent would communicate its findings to the Orchestrator via a structured object. This object could contain not only the generated answer but also metadata such as confidence scores or identified knowledge gaps. By doing so, the Orchestrator would receive richer, structured feedback from its workers, enabling it to generate more nuanced and reliable final answers.

- **Automated Ingestion Pipeline**. Establishing an automated data ingestion pipeline is crucial for keeping the Doctors' knowledge bases current with the latest medical research, ensuring the system's long-term reliability and relevance.

12 Self-evaluation

12.1 Luca Samorè

My primary responsibility within Mixture of Doctors was to define and oversee its technical and architectural vision, and I am immensely proud of the final system we have produced. This role allowed me to drive the key design decisions, particularly in shaping the system's strategies for availability, consistency, and fault-tolerance, which I believe are fundamental to its success. On the implementation side, my efforts were directed towards developing the core Orchestrator component and configuring the essential infrastructure, including the containerized deployment managed with Docker and Docker Swarm. This also involved leading the selection of the underlying technologies used throughout the project. A potential weakness is that, in focusing on architectural robustness, some finer-grained implementation details could be further optimized. Finally, I would like to extend my sincere gratitude to my teammates, Lucia and Roberto, for their unwavering commitment and patience throughout this journey.

12.2 Lucia Castellucci

My contribution to Mixture of Doctors focused on developing the foundational infrastructure and the Retrieval-Augmented Generation (RAG) module, which represents the system's functional core. I designed and implemented the entire RAG pipeline, managing the medical data lifecycle from ingestion and pre-processing to indexing in a vector database. This included developing components for semantic text chunking and embedding generation, establishing a modular architecture. Concurrently, I established the project's technical foundations by configuring the development environment and its dependencies. I also developed the command-line interface (CLI), a tool that proved essential for testing and debugging the end-to-end flow in the initial stages. The RAG pipeline and the infrastructural foundations I helped create represent a solid and functional starting point. However, I recognize that there is room for improvement. Looking ahead, it will be important to continue iterating on the system's components, particularly the RAG module, to progressively refine its relevance and accuracy, thereby increasing the overall effectiveness of the solution.

12.3 Roberto Mitugno

In the Mixture of Doctors project, I managed the implementation of the conversational memory and synthesis components: the Chat History service and the Synthesizer. For the first, I developed an asynchronous API with FastAPI, combined with a MongoDB cluster in set replication to ensure high availability and fault tolerance. I introduced atomic transactions for each write operation, preventing inconsistent states during failures. The Synthesizer, on the other hand, was designed as a Kafka consumer with manual offset management, ensuring no messages are lost before processing is complete. The components I developed constitute a solid foundation for the system's functionalities.

However, I am aware of current limitations, which represent important areas for improvement. The Chat History mechanism, while robust, is not scalable in the long term due to the context window limitations of the LLM. Similarly, the Synthesizer could be enhanced to more effectively manage potential contradictions in aggregated responses, a necessary step to further increase the quality and reliability of the system.

References

[1] Moh Heri Kurniawan, Hanny Handiyani, Tuti Nuraini, Rr Tutik Sri Hariyati, and Sutrisno Sutrisno. A systematic review of artificial intelligence-powered (AI-powered) chatbot intervention for managing chronic illness. *Annals of Medicine*, 56(1), March 2024.

[2] Sarah Dineen-Griffin, Victoria Garcia-Cardenas, Kylie Williams, and Shalom I. Benrimoj. Helping patients help themselves: A systematic review of self-management support strategies in primary health care practice. *PLOS ONE*, 14(8):e0220116, August 2019.

[3] Sara Montagna, Stefano Ferretti, Lorenz Cuno Klopfenstein, Antonio Florio, and Martino Francesco Pengo. Data Decentralisation of LLM-Based Chatbot Systems in Chronic Disease Self-Management. In *Proceedings of the 2023 ACM Conference on Information Technology for Social Good*, GoodIT '23, pages 205–212. ACM, September 2023.

[4] U.S. Department of Health & Human Services. Summary of the HIPAA Privacy Rule, 2023. Available at: `https://www.hhs.gov/hipaa/for-professionals/privacy/laws-regulations/index.html`.

[5] European Parliament and Council of the European Union. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation), 2016.

[6] Jan Clusmann, Fiona R. Kolbinger, Hannah Sophie Muti, Zunamys I. Carrero, Jan-Niklas Eckardt, Narmin Ghaffari Laleh, Chiara Maria Lavinia Löffler, Sophie-Caroline Schwarzkopf, Michaela Unger, Gregory P. Veldhuizen, Sophia J. Wagner, and Jakob Nikolas Kather. The future landscape of large language models in medicine. *Communications Medicine*, 3(1), October 2023.

[7] Waqar Hussain and John Grundy. Advice for Diabetes Self-Management by ChatGPT Models: Challenges and Recommendations. *arXiv preprint arXiv:2501.07931*, 2025.

[8] Hernan Inojosa, Isabel Voigt, Judith Wenk, Dyke Ferber, Isabella Wiest, Dario Antweiler, Eva Weicken, Stephen Gilbert, Jakob Nikolas Kather, Katja Akgün, and

Tjalf Ziemssen. Integrating large language models in care, research, and education in multiple sclerosis management. *Multiple Sclerosis Journal*, 30(11–12):1392–1401, September 2024.

[9] Tianyu Gao, Yiming Chen, and Bing Li. A Survey on Mitigating Hallucination in Natural Language Generation. *arXiv preprint arXiv:2302.04896*, 2023.

[10] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-Rank Adaptation of Large Language Models. *arXiv preprint arXiv:2106.09685*, 2021.

[11] Fan Xia, Yichong Zhang, and Furu Wei. Fine-tuning Large Language Models: Survey and Challenges. *arXiv preprint arXiv:2308.12393*, 2023.

[12] Fuzhen Zhuang, Zheng Qi, Ke Duan, Dongbo Xi, Yanchi Zhu, He Zhu, Hui Xiong, and Qing He. A Comprehensive Survey on Transfer Learning. *Proceedings of the IEEE*, 109(1):43–76, 2021.

[13] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474, 2020.

[14] Gautier Izacard and Edouard Grave. Leveraging Passage Retrieval with Generative Models for Open Domain Question Answering. *arXiv preprint arXiv:2007.01282*, 2021.

[15] Weilin Cai, Juyong Jiang, Fan Wang, Jing Tang, Sunghun Kim, and Jiayi Huang. A Survey on Mixture of Experts in Large Language Models. *arXiv preprint arXiv:2407.06204*, 2024.

[16] Soyeong Jeong, Jinheon Baek, Sukmin Cho, Sung Ju Hwang, and Jong C. Park. Adaptive-RAG: Learning to Adapt Retrieval-Augmented Large Language Models through Question Complexity. *arXiv preprint arXiv:2403.14403*, 2024.

[17] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[18] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A. Rossi, Subhabrata Mukherjee, Xianfeng Tang, Qi He, Zhigang Hua, Bo Long, Tong Zhao, Neil Shah, Amin Javari, Yinglong Xia, and Jiliang Tang. Retrieval-Augmented Generation with Graphs (GraphRAG). *arXiv preprint arXiv:2501.00309*, 2025.

[19] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv preprint arXiv:2210.03629*, 2022.