PEOPLE'S DEMOCRATIC REPUBLIC OF ALGERIA

MINISTRY OF HIGHER EDUCATION AND SCIENTIFIC RESEARCH

UNIVERSITY OF 20 AOUT 1955 - SKIKDA



Faculty of Sciences

Department of Computer Science

Dissertation Submitted partial fulfillment of the requirements for the Master's degree in Computer Science

Option: artificial intelligence

Theme:

Design and Implementation of a RAG-Based Medical Report Data Extractor API

Presented by:

BOUCHERBAT Mohamed Elhacene
OUAZETA Ouail Chams Eddine

Supervised by:

Dr. LAHSASNA Adel

2025 - 2026

Acknowledgements

First and foremost, we express our deepest gratitude to **Almighty God** for His guidance and countless blessings throughout this journey. Without His divine support, our success would not have been possible.

We would also like to thank our supervisor, Dr. LAHSASNA Adel, for his knowledge, experience, advice, and guidance, thanks to which we were able to produce high-quality work in such a short time.

We would like to thank our families for their unwavering psychological support especially my sister and my mother, encouragement, and the resources they provided, all of which played a vital role in our success.

We would like to thank all the teaching staff of the computer science department for their dedication and work, which has contributed to our learning

ABSTRACT

The digitalization of modern healthcare has generated vast amounts of unstructured clinical data, trapping valuable patient history within free-text medical reports. While Large Language Models (LLMs) offer unprecedented capabilities to analyze and synthesize this information, their direct application in clinical settings is severely limited by the risk of artificial hallucinations and strict patient data privacy regulations. To address this critical gap, this thesis presents the design, implementation, and evaluation of a privacy-preserving Retrieval-Augmented Generation (RAG) system tailored specifically for the medical domain.

We propose a secure, multi-tier architecture that automatically ingests heterogeneous medical documents, extracts key clinical entities, and transforms unstructured text into highly searchable, localized vector representations. To guarantee data sovereignty and patient confidentiality, the system enforces strict mathematical isolation of patient records and employs an automated de-identification module to mask sensitive information before interacting with external generative models.

Empirical evaluation demonstrates that the proposed architecture successfully mitigates LLM hallucinations by forcing the AI to generate responses strictly grounded in retrieved, patient-specific evidence. Ultimately, this research proves that integrating domain-specific natural language processing with secure RAG frameworks can safely unlock the value of unstructured medical data, providing clinicians with a reliable, privacy-compliant tool to support medical decision-making.

Keywords: ***Retrieval-Augmented Generation (RAG), Clinical NLP, Electronic Health Records, Large Language Models, Data Privacy, Medical Informatics.***

ملخص

أدت رقمنة الرعاية الصـــحية الحديثة إلى توليد كميات هائلة من البيانات الســـريرية غير المهيكلة، مما أدى إلى احتجاز التاريخ الطبي القيم للمرضى داخل تقارير طبية مكتوبة بنصوص حرة. وفي حين توفر النماذج اللغوية الكبيرة (LLMs) قدرات غير مسـبوقة لتحليل وتوليف هذه المعلومات، فإن تطبيقها المباشـر في البيئات السـريرية يظل مقيداً بشـدة بسـبب خطر الهلوسـة الاصـطناعية واللوائح الصـارمة لخصـوصـية بيانات المرضـى. ولمعالجة هذه الفجوة الحرجة، تقدم هذه المذكرة تصـميم وتنفيذ وتقييم نظام "التوليد المعزز بالاسـترجاع (RAG) "يحافظ على الخصـوصـية ومصـمم خصـيصـاً للمجال الطبي.

نقترح في هذا العمل بنية آمنة ومتعددة الطبقات تقوم تلقائياً باســتيعاب المســتندات الطبية غير المتجانســة، واســتخراج الكيانات السريرية الأساسية، وتحويل النصوص غير المهيكلة إلى تمثيلات متجهية (شعاعية) محلية عالية القابلية للبحث. ولضـمان سيادة البيانات وسرية المرضـى، يفرض النظام عزلاً رياضـياً صـارماً لسـجلات المرضـى، ويسـتخدم وحدة آلية لإخفاء الهوية لحجب المعلومات الحساسة قبل التفاعل مع النماذج التوليدية الخارجية.

يُظهر التقييم التجريبي أن البنية المقترحة تنجح في التخفيف من هلوســـات النماذج اللغوية الكبيرة من خلال إجبار الذكاء الاصطناعي على توليد إجابات تستند بشكل صارم إلى الأدلة المسترجعة والخاصة بالمريض. في الختام، يثبت هذا البحث أن دمج معالجة اللغات الطبيعية المخصـصـــة للمجال الطبي مع أطر عمل (RAG) الآمنة، يمكن أن يحرر قيمة البيانات الطبية غير المهيكلة بأمان، مما يوفر للأطباء أداة موثوقة ومتوافقة مع معايير الخصوصية لدعم اتخاذ القرارات الطبية.

الكلمات المفتاحية :التوليد المعزز بالاسترجاع(RAG) ، معالجة اللغات الطبيعية السريرية، السجلات الصحية الإلكترونية، النماذج اللغوية الكبيرة، خصوصية البيانات، المعلوماتية الطبية.

Table of Contents

Chapter I: General Introduction 9
1.1. Context and Motivation 10
1.1.1 The Digitalization of Healthcare and the Growth of Unstructured Data 10
1.1.2 The Limitations of Standard LLMs: Hallucinations and Lack of Grounding 10
1.1.3 The RAG Paradigm for Longitudinal Medical History Retrieval 10
1.2. Problem Statement 11
1.2.1 The Unstructured Data Silo in Clinical Reports 11
1.2.2 Semantic Ambiguity and the Gap in Medical Information Retrieval 11
1.2.3 Challenges of Multi-Period and Discontinuous Patient Records 11
1.3. Research Objectives 12
1.3.1 Designing an Ontology-Aware Medical Extraction Engine 12
1.3.2 Implementing a High-Precision Vector Retrieval API 12
1.3.3 Evaluating RAG Faithfulness in a Clinical Context 12
1.4. Scope and Limitations 12
1.5. Thesis Organization 13
Chapter II: State of the Art and Literature Review 14
2.1. Introduction 15
2.2. Medical Documents and Data Structure 15
2.2.1 Electronic Health Records and Clinical Report Types 15
2.2.2 Structured vs. Unstructured Medical Data Challenges 16
2.3. Natural Language Processing in the Clinical Domain 17
2.3.1 Historical Evolution of Clinical NLP 17
2.3.2 Named Entity Recognition for Medical Entities 18
2.3.3 Handling Negation and Uncertainty (NegEx) 18
2.4. Large Language Models in Healthcare 19
2.4.1 Transformer Architecture, Attention Mechanism, and Medical Specialization 19
2.4.2 RAG vs. Fine-Tuning for Patient Data Privacy 20
2.5. Retrieval-Augmented Generation 20
2.5.1 The RAG Architecture: Retriever, Retrieval Corpus, and Generator 21
2.5.2 Advanced RAG Variants (Self-RAG, HyDE, GraphRAG) 21
2.5.3 RAG Evaluation: The RAGAS Framework 22
2.6. Vector Databases and Semantic Search 22
2.6.1 Medical Text Embeddings (Bio_ClinicalBERT, Sentence Transformers) 22
2.6.2 Vector Search Engines and Similarity Strategies (FAISS, ChromaDB, Qdrant) 23
2.7. Medical Ontologies and Knowledge Representation 24
2.7.1 Standardized Systems: UMLS, SNOMED-CT, ICD-10/11, LOINC 24

2.7.2 Entity Linking and Medical Normalization 25
2.8. Comparative Analysis and Gap Identification 25
2.8.1 Overview of Existing Tools and Platforms 25
2.8.2 Comparative Analysis Table 26
2.8.3 Identified Gap and Positioning of the Proposed System 26
2.9. Conclusion 27
Chapter III: Conceptual Study and System Design 28
3.1. Introduction 29
3.2. System Requirements and Specification 29
3.2.1 Functional Requirements 29
3.2.2 Non-Functional Requirements 30
3.3. Global System Architecture 31
3.3.1 Layer 1: Document Ingestion and OCR Module 31
3.3.2 Layer 2: Medical NER and Ontology Mapping Module 32
3.3.3 Layer 3: Embedding and Vector Indexing Module 32
3.3.4 Layer 4: RAG Retrieval and API Gateway 33
3.4. System Pipeline Logic and Data Flow Design 33
3.4.1 Medical-Aware Chunking Strategies 34
3.4.2 Multi-Period Data Conflict Resolution 34
3.4.3 Semantic Query Processing and Re-Ranking 35
3.5. UML Modeling 35
3.5.1 Use Case Diagram 35
3.5.2 Sequence Diagrams (Ingestion Flow and Retrieval Flow) 38
3.5.3 Class Diagram 41
3.5.4 Activity Diagrams 45
3.6. Database Design 45
3.6.1 Relational Database Schema 45
3.6.2 Vector Database Schema 46
3.7. Conclusion 47
Chapter IV: Implementation, Evaluation, and Discussion 48
4.1. Introduction 49
4.2. Development Environment and Technology Stack 49
4.2.1 Backend AI Stack 49
4.2.2 Frontend and Relational Stack 52
4.3. Dataset Description and Preparation 54
4.3.1 Corpus Selection and Synthetic Data Generation Strategy 55
4.3.2 Data Annotation and Quality Validation 55

4.4. Implementation of the Extraction API 56
4.4.1 The DOCUMENTS Interface: Ingestion and Preprocessing 56
4.4.2 The ANALYSIS Interface: Medical Entity Extraction 58
4.4.3 The Research Interface: Semantic Search Implementation 59
4.4.4 The LOGIN and HEALTH Interfaces: Security and Diagnostics 60
4.5. RAG Pipeline Integration 60
4.5.1 PII Masking and De-identification Implementation 61
4.5.2 The CHAT Interface: Conversational RAG Orchestration 61
4.6. Experimental Results and Performance Metrics 62
4.6.1 Extraction Accuracy (Precision, Recall, F1-Score per Entity Type) 63
4.6.2 Retrieval Performance (MRR, Hit Rate, NDCG@k) 64
4.6.3 RAG Evaluation (Faithfulness and Context Relevancy via RAGAS) 65
4.6.4 API Performance (Latency and Throughput) 66
4.7. Discussion and Error Analysis 67
4.7.1 Linguistic Challenges: Abbreviations, Noise, and Multilingual Records 67
4.7.2 Ethical, Privacy, and Security Considerations 68
4.7.3 System Limitations and Failure Case Analysis 68
4.8. Conclusion 69
General Conclusion 70
Limitations and Future Work 71
References 73

Table of Figures

Figure 1 USE CASE DIAGRAM – DOCUMENT INGESTION AND RETRIEVAL SYSTEM....36
Figure 2 USE CASE ASK QUESTION....37
Figure 3 USE CASE DIAGRAM ORGANIZED AUTHENTICATION....37
Figure 4 SEQUENCE DIAGRAM – UPLOAD DOCUMENT....39
Figure 5 SEQUENCE DIAGRAM – ASK QUESTION....40
Figure 6 SEQUENCE DIAGRAM USER AUHTENTICATION....41
Figure 7 CLASS DIAGRAM - DOCUMENT INGESTION AND RETRIEVAL SYSTEM42
Figure 8 CLASS DIAGRAM - RAG chat System....43
Figure 9 CLASS DIAGRAM USER AUTHENTICATION....44
Figure 10 PYTHON....50
Figure 11 FAST API....50
Figure 12 LANG CHAIN....51
Figure 13 SCISPACE....51
Figure 14 PUBMEDBERT....51
Figure 15 CHROMADB....52
Figure 16 NEXT .JS AND REACT....52
Figure 17 TAILWIND CSS....53
Figure 18 NEXTAUTH.JS....53
Figure 19 SQLITE....54
Figure 20 PRISMAμ....54
Figure 21 DOUCMENTS INTERFACE 1....56
Figure 22 DOCUMENT INTERFACE 2....57
Figure 23 ANALYSIS INTERFACE 1....58
Figure 24 ANALYSIS INTERFACE 2....58
Figure 25 RESEARCH INTERFACE....59
Figure 26 LOGIN INTERFACE....60
Figure 27 THE CHAT INTERFACE....61
Figure 28 NER ACCURACY METRICS....63
Figure 29 RETRIEVAL PERFORMANCE METRICS....64
Figure 30 RAG EVALUATION SCORES....65
Figure 31 AVERAGE LATENCY / QUERY TIME....66

Chapter I: General Introduction

1.1. Context and Motivation

The motivation for this research stems from the friction between modern clinical workflows and the limitations of current artificial intelligence architectures.

1.1.1 The Digitalization of Healthcare and the Growth of Unstructured Data

The widespread adoption of EHR systems has successfully digitized the patient journey. However, medical informatics studies estimate that approximately 80% of all healthcare data resides in an unstructured format [1]. While structured data (like age, weight, and billing codes) is easily queried, the critical nuances of a patient's condition—the physician's observations, the progression of symptoms, and the rationale behind a specific treatment—are buried in free-text narratives. Clinicians are frequently forced to manually read through dozens of historical PDFs and scanned images to understand a patient's background, leading to severe cognitive fatigue and the risk of missing vital information.

1.1.2 The Limitations of Standard LLMs: Hallucinations and Lack of Grounding

While LLMs like GPT-4 have demonstrated passing scores on medical licensing exams, their theoretical knowledge does not translate to safe clinical practice. A standard LLM knows nothing about the specific patient currently sitting in the examination room. If a doctor asks a generic LLM to summarize a patient's history, the model will attempt to guess the answer based on its training weights, inevitably leading to hallucinations. In medicine, an AI that sounds confident while providing the wrong drug dosage is infinitely more dangerous than an AI that simply fails to answer.

1.1.3 The RAG Paradigm for Longitudinal Medical History Retrieval

Retrieval-Augmented Generation (RAG) provides the necessary architectural paradigm shift [2]. Instead of relying on the LLM's internal memory, a RAG system treats the LLM as a temporary reasoning engine. It utilizes a vector database to perform a semantic search over the patient's actual medical files, retrieving only the most relevant paragraphs. These paragraphs are then handed to the LLM as an "open-book" test. This approach guarantees that the generated clinical insights are grounded in the patient's longitudinal medical history, drastically reducing hallucinations while preserving the flexibility of natural language interaction.

1.2. Problem Statement

Despite the theoretical promise of RAG, building a functional system for the medical domain introduces three highly specific engineering and scientific challenges.

1.2.1 The Unstructured Data Silo in Clinical Reports

Medical reports are highly heterogeneous. A single patient file might contain digitally native text, poorly scanned fax images, and complex tabular laboratory results. Standard text extraction pipelines frequently fail to parse this structural chaos, resulting in fragmented data that cannot be accurately vectorized or searched.

1.2.2 Semantic Ambiguity and the Gap in Medical Information Retrieval

Clinical text is notoriously dense. Physicians rely heavily on acronyms, localized shorthand, and complex negation (e.g., "patient denies chest pain"). Standard keyword search algorithms, and even generic vector embedding models, fail to understand this semantic ambiguity. If a system cannot mathematically recognize that "T2DM" and "Type 2 Diabetes" represent the same clinical entity, the retrieval pipeline will fail to pull the necessary context for the LLM.

1.2.3 Challenges of Multi-Period and Discontinuous Patient Records

Perhaps the most dangerous challenge in clinical RAG is the Confusion Among Similar Entities (CASE). If a system relies purely on vector similarity, a query about "hypertension" might accidentally retrieve a document belonging to Patient B while the doctor is treating Patient A. Furthermore, medical history is strictly chronological; an allergy noted in 2018 is just as relevant as a lab result from 2024. The system must mathematically isolate patient records and maintain temporal awareness to prevent catastrophic cross-contamination of medical data.

1.3. Research Objectives

To resolve these challenges, this thesis sets out to achieve three primary research and engineering objectives:

1.3.1 Designing an Ontology-Aware Medical Extraction Engine

The first objective is to build a robust ingestion pipeline capable of handling multi-format documents (via OCR) and extracting clinical meaning from the raw text. This involves integrating a specialized biomedical Named Entity Recognition (NER) module to identify diseases, medications, and procedures, thereby enriching the unstructured text with structured clinical metadata.

1.3.2 Implementing a High-Precision Vector Retrieval API

The second objective is to engineer a secure, local-first semantic search architecture. This requires utilizing domain-specific embedding models and an embedded vector database. The system must implement strict metadata filtering to guarantee absolute patient isolation and utilize cross-encoder re-ranking to maximize context precision.

1.3.3 Evaluating RAG Faithfulness in a Clinical Context

The final objective is to move beyond anecdotal testing and subject the architecture to rigorous quantitative evaluation. This involves measuring extraction accuracy via F1-scores, retrieval efficacy via Mean Reciprocal Rank (MRR), and generative safety using the RAGAS framework to empirically prove the system's faithfulness and context relevancy.

1.4. Scope and Limitations

The scope of this thesis is bounded by several technical and ethical parameters. The extraction and retrieval pipelines are optimized exclusively for English-language medical reports. Due to strict healthcare privacy regulations (such as HIPAA and GDPR), the system's quantitative evaluation is conducted using a highly controlled, LLM-generated synthetic clinical dataset rather than real patient records. Finally, while the system successfully extracts narrative text, the programmatic flattening of highly complex tabular data (such as dense laboratory spreadsheets) remains a structural limitation of the current OCR implementation.

1.5. Thesis Organization

This thesis is organized into three primary chapters, followed by a general conclusion:

- **Chapter II: State of the Art and Literature Review** explores the theoretical foundations of clinical NLP, the evolution of Large Language Models in healthcare, and the mechanics of Retrieval-Augmented Generation. It concludes with a comparative analysis of existing platforms, identifying the gap this project aims to fill.
- **Chapter III: Conceptual Study and System Design** translates the theoretical research into a concrete software architecture. It details the functional requirements, the multi-tier pipeline logic, the metadata-enforced patient isolation strategy, and provides comprehensive UML modeling and database schemas.
- **Chapter IV: Implementation, Evaluation, and Discussion** details the physical construction of the MedChat application. It outlines the technology stack, the synthetic dataset generation, and the programmatic implementation of the API interfaces. The chapter concludes with a rigorous quantitative evaluation of the system's extraction accuracy, retrieval performance, and RAG faithfulness, followed by a critical error analysis.

Chapter II: State of the Art and Literature Review

2.1. Introduction

The rapid evolution of digital technologies has profoundly influenced the healthcare sector, giving rise to new models of care delivery, data management, and patient engagement. Mobile health (mHealth) and electronic health (eHealth) solutions have become integral to improving healthcare accessibility, reducing administrative burden, and enhancing the quality of care provided to patients.

This chapter aims to explore the current state of technological advancement in the domains most relevant to the development of the MedChat application. It provides an overview of existing medical applications, their functionalities, and the challenges they present. Furthermore, it examines the growing role of artificial intelligence in healthcare, especially in areas such as document processing, patient communication via chatbots, and semantic retrieval systems. By analyzing existing platforms and technologies, this chapter identifies the strengths and shortcomings of current solutions and highlights the need for a more integrated, intelligent system—an area where MedChat aims to contribute significantly [3], [4].

2.2. Medical Documents and Data Structure

The digitalization of healthcare has fundamentally transformed how patient information is recorded, stored, and analyzed. However, the transition from paper-based charts to digital systems has created a complex landscape of heterogeneous data types. Understanding the structural taxonomy of these medical documents is a critical prerequisite for designing any automated extraction or retrieval system.

2.2.1 Electronic Health Records and Clinical Report Types

Electronic Health Records (EHRs) are comprehensive, longitudinal digital records of patient health information generated across multiple clinical encounters. Unlike simple databases, EHRs are highly complex, multi-modal repositories that aggregate data from various hospital departments, laboratories, and specialized clinics [5]. The clinical narrative within an EHR is not monolithic; rather, it is distributed across several distinct types of clinical reports, each serving a specific medical and administrative function:

- **Discharge Summaries:** These are arguably the most information-dense documents in an EHR. They provide a retrospective synthesis of a patient's hospital stay, detailing the chief complaint, hospital course, final diagnoses, and post-discharge medication plans. Because they summarize complex medical events, they are highly valuable for longitudinal patient tracking but are notoriously dense and prone to heavy abbreviation [6].
- **Progress Notes:** Generated daily by attending physicians and nurses, progress notes document the ongoing status of a patient. They typically follow the SOAP (Subjective,

Objective, Assessment, Plan) format. These notes are highly chronological and often contain informal medical shorthand, making them linguistically noisy.

- **Pathology and Radiology Reports:** These documents are highly specialized and domain-specific. A radiology report (e.g., for an MRI or CT scan) describes anatomical findings and impressions using strict, localized vocabulary.
- **Prescriptions and Laboratory Results:** While often partially structured, the interpretive text accompanying lab results or the specific dosage instructions (sig codes) in prescriptions frequently exist as free text.

The primary challenge with these reports is their temporal discontinuity. A single patient may have dozens of reports generated over several years by different specialists. Consequently, the patient's true medical history is fragmented across a timeline of disparate documents, creating a significant barrier to holistic clinical decision-making [7].

2.2.2 Structured vs. Unstructured Medical Data Challenges

Clinical data within EHRs is broadly bifurcated into two categories: structured and unstructured data.

Structured data refers to highly organized information that resides in fixed fields within a relational database. Examples include patient demographics (age, gender), vital signs (blood pressure, temperature), and standardized billing codes such as the International Classification of Diseases (ICD-10) or Logical Observation Identifiers Names and Codes (LOINC) [8]. Because of its standardized format, structured data is easily queried using traditional SQL databases and is highly interoperable across different healthcare systems.

Conversely, **unstructured data** consists of free-text narratives that do not conform to a predefined data model. It is widely estimated in medical informatics literature that approximately 80% of all healthcare data resides in an unstructured format [1]. This includes the physician progress notes, discharge summaries, and diagnostic reports detailed in the previous section.

The reliance on unstructured data presents several critical computational challenges:

1. **Semantic Ambiguity and Linguistic Complexity:** Medical free text is characterized by heavy use of acronyms, synonyms, and domain-specific jargon. For example, the acronym "MI" could denote "Myocardial Infarction" or "Mitral Insufficiency" depending entirely on the surrounding context. Furthermore, clinical text relies heavily on negation (e.g., "patient denies chest pain") and uncertainty (e.g., "findings suggestive of pneumonia"), which traditional keyword-matching algorithms fail to interpret correctly [9].

2. **The Data Silo Effect:** Because unstructured data cannot be easily queried, it effectively becomes a "data silo." Clinicians are forced to manually read through

decades of notes to find relevant historical context, leading to information overload and the risk of missing critical patient history.

3. **Lack of Interoperability:** Unlike structured ICD-10 codes, free-text narratives cannot be seamlessly transferred or analyzed by standard clinical decision support systems without an intermediary processing layer.

While structured data is easily managed by traditional IT infrastructure, the vast majority of actionable clinical insights—the "why" and "how" of a patient's condition—are locked within the 80% of unstructured text. Bridging this gap requires moving beyond traditional relational databases and employing advanced Natural Language Processing (NLP) and semantic retrieval paradigms to transform free text into computable, structured knowledge.

2.3. Natural Language Processing in the Clinical Domain

To bridge the gap between unstructured clinical narratives and structured, computable data, Natural Language Processing (NLP) serves as the foundational extraction layer. Clinical NLP is a highly specialized subfield of artificial intelligence dedicated to parsing, understanding, and organizing medical text. Unlike general-domain NLP, which processes standard syntactic structures (e.g., news articles or web text), clinical NLP must navigate a landscape of fragmented grammar, heavy abbreviation, and domain-specific ontologies.

2.3.1 Historical Evolution of Clinical NLP

The evolution of clinical NLP can be broadly categorized into three distinct eras: rule-based systems, statistical machine learning, and deep learning architectures.

Early clinical NLP systems, developed in the late 1990s and early 2000s, were predominantly rule-based and relied heavily on dictionary lookups and regular expressions. Systems such as MetaMap [10] and the clinical Text Analysis and Knowledge Extraction System (cTAKES) [11] utilized the Unified Medical Language System (UMLS) to map text to medical concepts. While highly accurate for strict matches, these systems were inherently rigid; they struggled with misspellings, novel abbreviations, and complex syntactic variations, requiring exhaustive manual rule curation.

In the 2010s, the field transitioned toward statistical machine learning models, employing algorithms such as Conditional Random Fields (CRFs) and Support Vector Machines (SVMs). These models improved adaptability by learning linguistic patterns from annotated corpora (e.g., the i2b2/n2c2 challenge datasets) rather than relying solely on hardcoded rules [12].

Today, the state of the art is dominated by deep learning, specifically Transformer-based architectures. By leveraging self-attention mechanisms, modern models can capture deep bidirectional context, allowing them to disambiguate terms based on surrounding words.

However, the computational overhead of deploying massive neural networks for basic extraction tasks has led to a hybrid approach in modern production systems: combining lightweight, domain-specific statistical models with deep learning embeddings to achieve both speed and accuracy.

2.3.2 Named Entity Recognition for Medical Entities

The core operational task in clinical NLP is Named Entity Recognition (NER). In the medical domain, NER involves identifying rigid spans of text and classifying them into predefined clinical categories, such as Diagnoses, Medications, Procedures, and Laboratory Results.

Medical NER presents unique challenges not found in general text. First, entity boundaries are often ambiguous or overlapping (e.g., "left lower lobe pneumonia" contains both anatomical and pathological entities). Second, the vocabulary is highly specialized and constantly evolving. To address this, general-purpose NLP libraries like standard spaCy or NLTK are insufficient, as their underlying models are trained on web corpora (e.g., Wikipedia or news datasets) and fail to recognize biomedical terminology.

To achieve production-grade extraction without the prohibitive latency of querying a massive Large Language Model for every document, specialized biomedical libraries have been developed. A prominent standard in this domain is **scispaCy** [13]. Developed by the Allen Institute for AI, scispaCy provides fast, robust models specifically pre-trained on biomedical text (such as PubMed abstracts and MIMIC-III clinical notes).

For the architectural design of a medical extraction API, scispaCy offers a critical advantage: it balances high-precision NER with computational efficiency. Unlike pure Transformer models that require GPU acceleration for inference, scispaCy's optimized pipelines can run efficiently on standard CPUs, making it highly suitable for scalable, GDPR-compliant, on-premise deployments. Furthermore, it includes built-in modules for entity linking, allowing extracted text to be seamlessly mapped to standardized ontologies.

2.3.3 Handling Negation and Uncertainty (NegEx)

A fundamental characteristic of clinical documentation is the frequent documentation of absent conditions. Physicians routinely record what a patient does not have to rule out differential diagnoses (e.g., "Patient denies chest pain," "No signs of pulmonary embolism," or "Rule out appendicitis").

If an NLP system or a vector embedding model extracts "chest pain" without capturing the preceding negation ("denies"), the resulting structured data will falsely indicate that the patient is experiencing a heart issue. In a Retrieval-Augmented Generation (RAG) pipeline, this failure is catastrophic, as the vector database will retrieve and inject false positive symptoms into the LLM's context window, directly causing dangerous medical hallucinations.

To solve this, clinical NLP pipelines implement negation detection algorithms, the most foundational of which is **NegEx** [9]. Developed by Chapman et al., NegEx is a highly efficient algorithm that uses regular expressions to detect trigger terms indicating negation (e.g., "absence of," "denies," "negative for") and uncertainty (e.g., "suggestive of," "possible"). The algorithm calculates the syntactic scope of these triggers—typically looking forward or backward a set number of words—to determine if an extracted clinical entity falls within a negated context.

Modern biomedical NLP pipelines, including scispaCy, integrate advanced variations of the NegEx algorithm directly into their processing graphs. By appending a boolean is negated metadata tag to every extracted entity, the system ensures that downstream vector databases can filter out negated concepts, thereby preserving the factual integrity of the patient's medical history before it ever reaches the generative LLM.

2.4. Large Language Models in Healthcare

Pulling medical entities out of a document using tools like scispaCy is really only the first step. Finding a symptom in a text doesn't automatically tell us what it means for the patient's actual treatment plan. Making sense of these isolated data points requires a reasoning engine. Over the last few years, Large Language Models (LLMs) have stepped into this role. They can read through messy histories and draft discharge notes surprisingly well. But dropping these models directly into a hospital's IT system isn't a simple plug-and-play fix. It creates a whole new set of architectural and privacy headaches.

2.4.1 Transformer Architecture, Attention Mechanism, and Medical Specialization

The shift started back in 2017 when Vaswani et al. introduced the Transformer architecture [14]. Before that, algorithms processed text sequentially. If a medical file was too long, the system would essentially forget the first page by the time it reached the last. Transformers got around this bottleneck using self-attention.

Think of self-attention as a mathematical way for the model to weigh how much one word matters to another, no matter how far apart they are. Let's say a doctor notes a "severe penicillin allergy" early on, and then debates "antibiotic therapy options" five pages later. A Transformer connects those two ideas instantly. For tracking a patient's history over time, this is non-negotiable.

There is a catch, though. Models like GPT-4 are trained mostly on the open web. They pick up the grammar of medicine without understanding the real-world stakes. A model might write a treatment plan that looks perfectly professional but is clinically absurd—a problem we call hallucination [4]. Researchers have tried to fix this by building specialized models like Med-PaLM, feeding them medical journals and clinical guidelines [15]. But a fundamental

blind spot remains. A specialized LLM might know everything about cardiology in theory, but it knows absolutely nothing about the specific patient sitting in the exam room.

2.4.2 RAG vs. Fine-Tuning for Patient Data Privacy

So how do we force a generalized model to care about a specific patient's history? Right now, the industry offers two main workarounds: fine-tuning the model, or using Retrieval-Augmented Generation (RAG). This choice goes way beyond software preference. It completely dictates how we handle patient privacy.

Fine-tuning means opening up the model and adjusting its internal weights using a hospital's own data. It sounds like a good idea on paper, but in practice, it creates a rigid system. Medicine updates constantly. If a patient gets a new lab result on Tuesday, a fine-tuned model won't know about it on Wednesday unless the hospital runs a massive, expensive retraining cycle. Worse, fine-tuning bakes sensitive patient data directly into the model's neural network. If a patient demands their data be deleted under privacy laws like GDPR, you can't just extract their specific records from a billion-parameter web of weights [16]. The data is stuck there.

RAG takes a completely different approach [3]. It leaves the LLM alone. Instead of forcing the model to memorize patient files, RAG stores those files in a secure, searchable vector database. When a doctor asks a question, the system searches the database, pulls only the relevant paragraphs, and hands them to the LLM to read right then and there.

This separation of logic and memory solves three massive problems for healthcare. First, updates happen instantly. A new lab report just gets embedded and dropped into the database without any retraining. Second, because the model bases its answer strictly on the retrieved text, doctors can trace every claim back to a specific sentence in the original file, which drastically cuts down on hallucinations. Finally, patient data stays locked in the database. If a file needs deleting, you simply delete the vector. The LLM acts merely as a temporary reasoning engine, forgetting the patient the moment the session ends.

2.5. Retrieval-Augmented Generation

We established in the previous section that fine-tuning an LLM on patient data is a privacy nightmare. Retrieval-Augmented Generation (RAG) offers a clean way out of that trap. First introduced by Lewis et al. in 2020, RAG fundamentally changed how we handle knowledge-intensive tasks [3]. Instead of forcing a neural network to memorize facts, RAG gives the model an open-book test. It allows the system to look up the exact medical records it needs right before it answers a question.

2.5.1 The RAG Architecture: Retriever, Retrieval Corpus, and Generator

At its core, a standard RAG system—often called Naive RAG—relies on three distinct moving parts working together in real time.

First, you have the retrieval corpus. This isn't a standard SQL database. It is a vector database where all the unstructured medical documents, like discharge summaries and lab results, are chopped into smaller text chunks and converted into mathematical arrays called embeddings. Think of the corpus as a massive, highly organized medical library where every paragraph is mapped out based on its underlying meaning rather than just its keywords.

Next comes the retriever. When a doctor types a query into the system, the retriever translates that question into the exact same mathematical format as the documents. It then scans the vector database to find the chunks of text that are closest in meaning to the question. It acts like a highly efficient librarian, pulling the top three or four most relevant paragraphs from a patient's decade-long medical history.

Finally, the generator takes over. The system takes the doctor's original question, packages it together with the paragraphs the retriever just found, and hands the whole bundle to the LLM. The model reads the provided context and generates a fluent, highly accurate response. Because the LLM is restricted to answering based only on the text it was just handed, the chances of it making up a fake diagnosis drop dramatically [17].

2.5.2 Advanced RAG Variants (Self-RAG, HyDE, GraphRAG)

Naive RAG sounds great on paper, but it struggles heavily in the medical field. The main issue is the semantic gap between how a question is asked and how a doctor wrote the notes. If a user asks, "Why is the patient's chest hurting?", a basic retriever might fail to find a clinical note that says "acute myocardial infarction" because the words don't match up mathematically.

To fix this, researchers have developed Advanced RAG variants. One of the most effective techniques is Hypothetical Document Embeddings, or HyDE [18]. Instead of searching the database using the user's raw question, HyDE asks the LLM to write a fake, hypothetical answer to the question first. Even if this fake answer contains factual errors, it will naturally include all the right medical jargon. The system then uses that jargon-heavy hypothetical answer to search the vector database. It essentially acts as a translation layer between layman terms and clinical vocabulary.

Another major leap forward is GraphRAG [19]. Standard RAG treats text chunks as isolated islands of information. But medical data is deeply relational. A medication prescribed in 2022 is directly linked to a lab result from 2024. GraphRAG extracts these entities and builds a knowledge graph—a web of relationships—on top of the vector database. When a query comes in, the system doesn't just retrieve a single paragraph; it pulls an entire connected web of symptoms, treatments, and outcomes. We also see systems using Self-RAG, where the

model is trained to critique its own retrieved documents and reject them if they don't actually answer the prompt, forcing the retriever to try again.

2.5.3 RAG Evaluation: The RAGAS Framework

Building a RAG pipeline is one thing, but proving it actually works is surprisingly difficult. Traditional NLP metrics like BLEU or ROUGE just look at word overlap, which is useless for evaluating complex medical reasoning. You also cant just look at whether the final answer is correct. There is a known issue called the "Accuracy Fallacy" where an LLM gives the right medical advice, but it based that advice on the wrong patient's retrieved document. In a hospital, that kind of lucky guess is just as dangerous as a hallucination.

To actually measure performance, the industry has largely adopted the RAGAS (Retrieval Augmented Generation Assessment) framework [20]. RAGAS is brilliant because it stops treating the system as a black box. Instead, it grades the retriever and the generator separately.

It measures Context Precision to see if the retriever actually pulled the right medical notes and ignored the irrelevant ones. Then it measures Faithfulness, which checks if the LLM's final answer was strictly derived from those retrieved notes, or if the model started making things up. Finally, it looks at Answer Relevance to ensure the response actually addressed the doctor's original question. By breaking the evaluation down into these specific metrics, we can pinpoint exactly where a medical extraction API is failing and fix it before it ever reaches a clinical setting.

2.6. Vector Databases and Semantic Search

We know that a RAG system relies on a retriever to pull relevant medical history before generating an answer. But how does a computer actually "read" a doctor's note to know if it is relevant? Computers do not understand English, let alone complex medical jargon. To make semantic search possible, we have to translate raw text into a format the machine can process. This translation happens through embeddings, and the resulting data is stored in a highly specialized piece of infrastructure known as a vector database.

2.6.1 Medical Text Embeddings (Bio_ClinicalBERT, Sentence Transformers)

An embedding model takes a chunk of text and converts it into a high-dimensional array of numbers, often called a vector. Think of this vector as a set of coordinates in a massive mathematical space. If two sentences have similar meanings, their coordinates will end up very close to each other, even if they don't share a single identical word. This is how a system knows that "myocardial infarction" and "heart attack" belong together.

Early embedding models processed text one word at a time, which was incredibly slow and often missed the broader context of a sentence. Reimers and Gurevych changed this with the introduction of Sentence Transformers, which use a Siamese network structure to embed entire sentences or paragraphs at once [21]. This made semantic search fast enough for real-time applications.

But applying general-purpose Sentence Transformers to healthcare is risky. A model trained on Wikipedia will completely misinterpret clinical abbreviations. To fix this domain shift, researchers developed specialized models like Bio_ClinicalBERT, which was trained heavily on the MIMIC-III database of ICU notes. While highly accurate, standard BERT models are computationally heavy. They usually require expensive GPUs to run effectively.

For a medical extraction API that prioritizes patient privacy and local deployment, we need something that balances deep clinical knowledge with hardware efficiency. This is exactly why S-PubMedBert has emerged as a preferred standard [22]. It is a Sentence Transformer specifically pre-trained on millions of biomedical abstracts and clinical texts. Because it is optimized for sentence-level tasks, it runs exceptionally well on standard CPUs. This means a hospital can deploy the embedding model entirely on-premise, ensuring that sensitive patient data never has to be sent to a third-party server like OpenAI just to be converted into vectors.

2.6.2 Vector Search Engines and Similarity Strategies (FAISS, ChromaDB, Qdrant)

Once the medical reports are converted into thousands of high-dimensional vectors, they need a place to live. You cannot store these arrays in a traditional relational database like MySQL. SQL databases are built to look for exact matches. If you ask SQL to find a vector that is "mathematically similar" to another vector, the system will crash under the computational weight.

This limitation gave rise to vector databases. These engines use algorithms like cosine similarity or Euclidean distance to calculate the angle and distance between vectors, allowing them to find the "nearest neighbors" to a doctor's query in milliseconds [23].

The landscape of vector search engines is crowded, and choosing the right one dictates the entire architecture of the API. FAISS, developed by Meta, is one of the oldest and fastest libraries for similarity search. It is incredibly powerful but acts more like a low-level mathematical library than a true database; it lacks built-in document storage and metadata filtering. On the other end of the spectrum are enterprise solutions like Qdrant or Pinecone. These are robust, but they require setting up dedicated server infrastructure or relying on cloud hosting, which complicates GDPR compliance and adds unnecessary latency to a lightweight API [24].

For a standalone medical extraction API, ChromaDB strikes the perfect architectural balance. Unlike Qdrant, ChromaDB can run in an embedded mode. This means the vector database lives directly inside the Python application memory and saves its persistence layer locally to the disk. There is no separate server to manage and no network latency between the

API and the database. Furthermore, it integrates seamlessly with orchestration frameworks like LangChain. When a patient's report is processed, ChromaDB handles the vector indexing and the metadata storage (like the date of the report and the patient ID) in one unified step, making it an ideal engine for a fast, privacy-first RAG pipeline.

2.7. Medical Ontologies and Knowledge Representation

By this point in the pipeline, we have a system that can extract medical entities using scispaCy and embed sentences into a vector database using S-PubMedBert. But a massive problem still remains. Medicine is plagued by synonyms and abbreviations. One physician might write "heart attack," another might type "myocardial infarction," and a third might just scribble "MI." While a vector database understands that these terms are mathematically similar, a hospital's IT system needs them to be recognized as the exact same underlying concept. If we want our API to output truly structured data, we have to map these messy, free-text extractions to a standardized medical vocabulary.

2.7.1 Standardized Systems: UMLS, SNOMED-CT, ICD-10/11, LOINC

The healthcare industry has spent decades trying to standardize its language, resulting in a complex web of different coding systems. Each of these ontologies was built for a slightly different purpose, which is why no single system has completely taken over.

The International Classification of Diseases (ICD-10 and the newer ICD-11) is probably the most famous. Maintained by the World Health Organization, it is primarily a billing and epidemiological tool. It is great for categorizing a final diagnosis so a hospital can get paid, but it lacks the granular detail needed to describe a patient's day-to-day symptoms. To capture that clinical nuance, doctors rely on SNOMED-CT (Systematized Nomenclature of Medicine - Clinical Terms). SNOMED-CT is a massive, hierarchical ontology that covers everything from surgical procedures to microscopic anatomical structures [25]. It is the language of the electronic health record itself.

Then you have laboratory data, which operates in its own universe. If a RAG system is trying to track a patient's kidney function over time, it needs to understand lab reports. This is where LOINC (Logical Observation Identifier Names and Codes) comes in. LOINC provides a universal standard for identifying medical laboratory observations, ensuring that a "serum glucose" test at one clinic is recognized as the same test at another [26].

Navigating between ICD-10, SNOMED-CT, and LOINC is a logistical nightmare for a software developer. To solve this, the US National Library of Medicine created the Unified Medical Language System (UMLS). Think of the UMLS as the ultimate medical Rosetta Stone. It does not replace the other systems; instead, it maps them all together [8]. It assigns a Concept Unique Identifier (CUI) to every medical idea. So, whether the text says "heart attack" or "MI," the UMLS assigns it the exact same CUI (C0027051) and provides the cross-references to the correct ICD-10 and SNOMED codes.

2.7.2 Entity Linking and Medical Normalization

Knowing that these ontologies exist is only half the battle. The actual computational challenge lies in connecting the raw text extracted by the NLP model to the correct code in the database. This process is known as entity linking, or medical normalization.

When our API processes a document, the NER module first identifies a span of text, like "T2DM." The entity linker then takes that string and searches the ontology to find the best match. This is rarely a simple keyword lookup. The linker has to account for misspellings, word order variations, and context. For instance, "cold" could map to a viral infection (C0009443) or a low temperature (C0009264). Modern entity linkers, including the ones integrated into advanced scispaCy pipelines, use lightweight similarity algorithms to resolve this ambiguity and assign the correct UMLS CUI [27].

For a RAG-based extraction API, entity linking is the secret weapon. When we store a chunk of text in ChromaDB, we don't just store the raw words. We inject the standardized UMLS codes directly into the vector's metadata. This completely changes how the retriever functions. If a doctor queries the chatbot about a patient's "Type 2 Diabetes," the system doesn't just rely on vector math to find similar paragraphs. It can explicitly filter the database for any document tagged with the exact CUI for diabetes, regardless of how the original physician phrased it. By normalizing the data before it ever reaches the LLM, we drastically reduce the semantic gap and ensure the final generated response is grounded in standardized medical reality.

2.8. Comparative Analysis and Gap Identification

We have explored the theoretical building blocks required to process clinical text, from deep learning embeddings to medical ontologies. But theory only goes so far. To justify the development of a new extraction API, we have to look at what is already available on the market and understand exactly where these existing solutions fail.

2.8.1 Overview of Existing Tools and Platforms

If a hospital wants to extract structured data from clinical notes today, they generally have to choose between massive proprietary cloud services and fragmented open-source libraries.

On the proprietary side, tech giants dominate the landscape. Services like AWS Comprehend Medical, Google Cloud Healthcare NLP, and Azure Text Analytics for Health are incredibly powerful [28]. They offer out-of-the-box entity extraction, negation detection, and ontology linking. But they all share a fatal flaw for certain clinical environments: they are cloud-bound. Sending unredacted, highly sensitive patient records to a third-party server introduces massive legal and ethical liabilities, especially under strict privacy frameworks. For many healthcare institutions, cloud-based NLP is simply a non-starter.

On the open-source side, the options allow for local, privacy-safe deployment, but they come with their own severe limitations. Apache cTAKES is perhaps the most famous legacy system [11]. While it is deeply integrated with the UMLS, it is built on an aging Java architecture that is notoriously rigid, slow, and difficult to integrate into modern, asynchronous web applications. More recently, Python-based libraries like MedSpaCy have emerged as excellent tools for local clinical NLP [29]. MedSpaCy handles entity extraction and negation beautifully. However, it is just an NLP library. It does not generate vector embeddings, it does not manage document chunking, and it has no native integration with vector databases.

2.8.2 Comparative Analysis Table

To clearly visualize the current landscape, Table 1.1 evaluates these existing platforms against the specific requirements of a modern, RAG-driven healthcare system. The comparison focuses on deployment flexibility, extraction capabilities, and readiness for vector-based semantic search.

Table 1.1: Comparative Analysis of Medical NLP and Extraction Tools

System / Platform	Open Source	Local Deployment (Privacy)	Medical NER & Negation	Ontology Linking (UMLS/ICD)	Native Vector Embedding Output
AWS Comprehend Medical	No	No (Cloud only)	High	Yes	No
Google Healthcare NLP	No	No (Cloud only)	High	Yes	No
Apache cTAKES	Yes	Yes	Moderate	Yes	No
MedSpaCy	Yes	Yes	High	Yes	No
Proposed API System	**Yes**	**Yes (On-Premise)**	**High**	**Yes**	**Yes (RAG-Ready)**

2.8.3 Identified Gap and Positioning of the Proposed System

The gap in the current state of the art becomes glaringly obvious when you look at that table. Right now, there is no single, open-source platform that bridges the gap between raw clinical text and a RAG-ready vector database.

Think about what happens when a developer tries to build a medical chatbot today. They have to piece together a Frankenstein pipeline. They use one tool to run OCR on a PDF, pass that text to a library like MedSpaCy to find the entities, write custom scripts to map those

entities to UMLS codes, pass the text again to a model like PubMedBERT to generate embeddings, and finally write more custom code to push everything into a database like ChromaDB. This fragmented approach is brittle, slow, and prone to data loss at every transition point [16].

This is exactly where the proposed system steps in. The objective of this thesis is not just to train another NER model. The goal is to build a unified, end-to-end REST API that handles the entire lifecycle of medical document processing. By ingesting raw files, extracting and normalizing the medical entities, generating the semantic embeddings, and storing everything in an embedded vector database, the proposed API eliminates the fragmentation. It provides a secure, locally deployable, RAG-ready backend that allows any downstream LLM to instantly access grounded, patient-specific history without compromising data privacy.

2.9. Conclusion

This chapter established the theoretical and technological foundations necessary to process complex clinical narratives. We began by examining the messy reality of electronic health records, where the vast majority of actionable patient history is trapped in unstructured, highly abbreviated free text. While traditional clinical NLP pipelines and specialized libraries like scispaCy are highly effective at extracting discrete medical entities and handling linguistic hurdles like negation, they inherently lack the reasoning capabilities required to synthesize a holistic patient overview.

Large Language Models offer that missing reasoning capacity. However, as we explored, directly fine-tuning these models on sensitive hospital data creates unacceptable privacy risks and static, outdated knowledge bases. Retrieval-Augmented Generation (RAG) solves this dilemma. By decoupling the reasoning engine from the knowledge repository, RAG allows us to use domain-specific embeddings—such as S-PubMedBert—and local vector databases like ChromaDB to securely feed relevant, up-to-date patient history to an LLM at inference time. Furthermore, by linking extracted text to standardized ontologies like the UMLS, we ensure that the retrieved data is grounded in a universal medical vocabulary rather than ambiguous free text.

Despite these technological advancements, our comparative analysis revealed a significant gap in the current landscape. Healthcare institutions are currently forced to choose between massive, privacy-compromising cloud services and fragmented, incomplete open-source libraries. There is a clear and pressing need for a unified, locally deployable system that bridges the gap between raw clinical documents and a RAG-ready vector database.

Filling this exact gap is the primary objective of this thesis. With the theoretical state of the art now established, the following chapter will transition from research to engineering. Chapter II will detail the conceptual study and system design, outlining the pipeline logic, architectural layers, and UML models required to build the proposed medical report extraction API.

Chapter III: Conceptual Study and System Design

3.1. Introduction

The theoretical foundations established in Chapter I make one reality abundantly clear: deploying Large Language Models (LLMs) directly into clinical environments is a dangerous proposition. While these models possess remarkable reasoning capabilities, they are fundamentally bottlenecked by hallucinations, domain shift, and strict patient privacy regulations [30]. Bridging the gap between a theoretical NLP concept and a functional, privacy-preserving clinical tool requires more than just writing code. It requires a rigorous, modular architectural framework. In fact, recent systematic reviews of Retrieval-Augmented Generation (RAG) in healthcare emphasize that successful clinical integration depends entirely on moving away from "naïve" RAG setups toward highly decoupled, multi-evidence architectures [31], [32].

This chapter details the conceptual study and system design of the proposed "MedChat" application. Transitioning from the literature review, we outline a multi-tier RAG architecture engineered specifically to handle the chaotic, heterogeneous nature of Electronic Health Records (EHRs). The primary objective of this design phase is to architect a pipeline that actively mitigates retrieval noise and ensures factual consistency, all while maintaining absolute patient data isolation [30].

The chapter is structured as follows: First, we define the functional and non-functional requirements that govern the system's operational boundaries. Next, we detail the global system architecture, breaking down the pipeline into four distinct layers: document ingestion, medical Named Entity Recognition (NER), vector indexing, and RAG orchestration. We then explore the specific pipeline logic, emphasizing metadata filtering and temporal tagging to prevent the dangerous cross-contamination of patient records. Finally, we provide comprehensive Unified Modeling Language (UML) diagrams and database schemas to serve as the definitive structural blueprint for the implementation phase.

3.2. System Requirements and Specification

To transition the MedChat application from a conceptual NLP framework to a stable, deployable software solution, we must clearly define its functional boundaries and constraints. Following standard software engineering specifications [33], these criteria are categorized into functional requirements—the tasks the software must actively perform—and non-functional requirements—the performance, security, and architectural attributes the system must maintain during runtime execution.

3.2.1 Functional Requirements

Functional requirements describe the core operations of the MedChat API, detailing how data is processed, transformed, and retrieved from input to output. The application must support five core functional pipelines:

The first functional pipeline governs document ingestion and text extraction. The system must accept multi-format medical uploads, specifically scanned clinical images, digital PDFs, and physical reports. For standard digital files, the system must programmatically parse the text. For hand-written or scanned files, the system must automatically execute an optical character recognition engine. Following extraction, the text must be partitioned into segments of 1000 characters with a 200-character overlap to keep sentences intact.

The second pipeline handles localized vector generation and persistent storage. Every text segment must be converted into a mathematical array. To preserve medical nuances, this process must run locally using a biomedical transformer model. The generated vectors must be stored directly in an embedded database on the host machine.

The third pipeline is similarity-based semantic search. When a query is initiated, the system must search the database using cosine similarity. To protect the integrity of the data, the search must accept strict filters. Specifically, it must isolate vectors belonging to the active patient ID and exclude any matches with a similarity score below 0.6.

The fourth pipeline is clinical named entity recognition. The system must process user queries through a specialized NLP pipeline to identify clinical terms such as drug names, disease states, and procedures. This extracted information must be saved as structured metadata to enrich the query context.

The fifth and final functional pipeline controls privacy-aware answer generation. Before any text is passed to the generation model, a privacy module must redact personally identifiable information. The system must then construct a strict prompt containing the masked text, the conversation history, and the user's question, passing this unified context to a large language model to produce a clinical answer backed by source citations.

3.2.2 Non-Functional Requirements

While functional requirements dictate what the system does, non-functional requirements define the operational parameters that make the software secure, reliable, and deployable in clinical settings [34].

First, security and privacy are the primary constraints. Since the generative pipeline relies on a cloud-based LLM, the system must implement a local-first security model. All clinical documents, text segments, embeddings, and vector indices must remain completely offline on the local server. The external LLM API must only receive context that has been thoroughly sanitized of personal identifiers such as names, telephone numbers, and medical record numbers.

Second, the system must prioritize asynchronous execution and low latency. Document ingestion, vector embedding, and OCR are mathematically intensive processes. The backend must be built on an asynchronous ASGI framework capable of handling concurrent tasks without blocking the main event loop. A single query execution, including database search,

privacy masking, and LLM streaming, should complete in a reasonable timeframe to maintain a responsive user interface.

Third, the architecture must remain strictly modular and decoupled. The Next.js frontend, the SQLite relational database (which handles session authentication), and the FastAPI microservices must operate independently. This modularity ensures that the vector database or NLP modules can be swapped or upgraded without affecting the user interface or relational data layer.

3.3. Global System Architecture

To satisfy the strict security and performance requirements of a clinical environment, the MedChat application rejects monolithic design in favor of a decoupled, three-tier architecture. This architecture is structured around the Backend-for-Frontend (BFF) pattern [35].

The first tier is the client-facing frontend, built with Next.js, React, and Tailwind CSS, which provides a highly responsive, server-rendered user interface. The second tier is the Next.js API Layer, which acts as the BFF. This layer manages user authentication via NextAuth.js and handles relational data persistence (such as user profiles and chat session metadata) by communicating with a local SQLite database through the Prisma Object-Relational Mapping (ORM) framework. The third tier is the core FastAPI backend, a high-performance Python web framework that hosts the specialized machine learning pipelines.

By separating the user interface from the AI orchestration layer, the system ensures that computationally heavy tasks do not block the client-side event loop. Communication between these tiers is strictly asynchronous, utilizing RESTful JSON payloads over secure local CORS channels. Within the FastAPI backend, data flows through a sequential, four-layered processing pipeline.

3.3.1 Layer 1: Document Ingestion and OCR Module

The lifecycle of a medical document begins at Layer 1, which is responsible for file ingestion, programmatic text extraction, and semantic chunking. Medical reports are notoriously heterogeneous, arriving as digitally generated PDFs, scanned paper charts, or mobile camera captures. To handle this structural chaos, Layer 1 implements a dual-engine extraction strategy.

When a document is uploaded, the module first attempts to extract text programmatically using **pdfplumber**. This library parses the document's internal layout, preserving paragraph boundaries and tabular structures. However, if the file is a scanned image or a non-interactive PDF, the module automatically routes the payload to the Tesseract Optical Character Recognition (OCR) engine [36]. Tesseract processes the pixel data, executing layout analysis and character recognition to output raw, unformatted text.

Once the raw text is extracted, it must be partitioned. Passing an entire multi-page document to an embedding model or an LLM is highly inefficient and dilutes semantic focus. Layer 1 resolves this by executing a sliding-window chunking strategy. The text is split into chunks of exactly 1000 characters, with a 200-character overlap between adjacent segments. This 200-character overlap is mathematically vital. It ensures that critical clinical concepts—such as a drug dosage or a symptom description—are not split mid-sentence across chunk boundaries, preserving semantic continuity for the downstream vectorization process.

3.3.2 Layer 2: Medical NER and Ontology Mapping Module

Raw text chunks, while structurally clean, lack semantic meaning. Before these chunks can be indexed, Layer 2 must extract and classify the clinical concepts hidden within the prose. This semantic enrichment is governed by a specialized biomedical Named Entity Recognition (NER) pipeline.

Instead of relying on general-purpose NLP models that struggle with clinical jargon, Layer 2 utilizes **scispaCy** [13]. Specifically, the pipeline employs models pre-trained on extensive biomedical corpora (such as MedMentions or BC5CDR). When a text chunk passes through this layer, **scispaCy** isolates and extracts key clinical entities, categorizing them into structured classes such as *Diseases*, *Medications*, *Procedures*, and *Anatomical Sites*.
These extracted entities are not discarded; they are structured as JSON metadata objects and permanently bound to the corresponding text chunk. This metadata enrichment serves a dual purpose. First, it prepares the extracted terms for standardized ontology mapping (UMLS and ICD-10). Second, it allows the system to perform highly granular metadata filtering during retrieval, ensuring that the vector database can be queried not just by semantic similarity, but by specific clinical entity types.

3.3.3 Layer 3: Embedding and Vector Indexing Module

Layer 3 handles the mathematical transformation of the enriched text chunks into high-dimensional vector embeddings and manages their persistent storage. This layer represents the transition from natural language to vector space.

The system passes each text chunk to the embedding service, which runs the S-PubMedBert model locally via the sentence-transformers library [22]. S-PubMedBert was chosen because its training corpus consists entirely of biomedical literature, allowing it to capture the complex, non-linear semantic relationships of clinical terminology far better than generic models. The model maps each chunk into a 384-dimensional vector space.
These 384-dimensional vectors, along with the raw text chunks and their associated metadata (including the patient_id, the source filename, the report date, and the extracted NER entities), are then indexed and written to ChromaDB [24]. ChromaDB operates as an embedded, serverless vector database, saving its persistence layer directly to the host's local disk. This serverless design is a critical architectural choice; it eliminates network latency

between the API and the database, simplifies deployment, and ensures that the patient's vector index remains entirely offline and secure.

3.3.4 Layer 4: RAG Retrieval and API Gateway

The final layer of the backend architecture is the orchestration engine, which manages the Retrieval-Augmented Generation (RAG) loop and serves as the API gateway. This layer is built using the LangChain framework, which coordinates the flow of data between the user, the vector database, and the generator [37].
When a user submits a query, Layer 4 executes a highly secure, four-step retrieval and generation process:

1. **Vector Query and Patient Isolation:** The user's query is embedded into a 384-dimensional vector using S-PubMedBert. The system then queries ChromaDB using cosine similarity. Crucially, the query enforces a strict metadata filter: it only searches vectors matching the active patient_id. This mathematical isolation prevents the system from ever retrieving documents belonging to another patient, completely eliminating the risk of cross-contamination.

2. **Similarity Thresholding:** The system retrieves the top four matching chunks but immediately discards any results with a similarity score below 0.6. This thresholding ensures that irrelevant documents are not injected into the prompt, mitigating retrieval noise.

3. **PII Masking:** Before the retrieved chunks are compiled into a prompt, they are passed through a local PII Masker. This module automatically redacts sensitive identifiers, such as patient names, telephone numbers, and medical record IDs. This hybrid privacy model is a cornerstone of the architecture; by sanitizing the context locally, the system can safely utilize a powerful cloud-based generator without risking data leaks [34].

4. **Grounded Generation:** The masked context, the query, and the conversation history are formatted into a strict prompt and sent to the GPT-4o-mini model. The prompt instructs the LLM to answer only using the provided context and to cite its sources. The generated response, along with the source filenames and a confidence score, is routed back through the FastAPI gateway to the Next.js frontend, where it is displayed to the user and logged in the SQLite database for evaluation.

3.4. System Pipeline Logic and Data Flow Design

Designing a clinical RAG system requires a deep focus on how data moves and changes as it flows through the pipeline. It is not enough to simply pass text from a database to an LLM. We must structure the data flow to preserve medical context, prevent the mixing of patient records, and ensure that the most relevant documents are prioritized. This section details the core pipeline logic that governs the MedChat backend, translating our architectural layers into a highly coordinated data flow.

3.4.1 Medical-Aware Chunking Strategies

The raw text extracted by the OCRService cannot be embedded as a single, continuous block. Doing so would exceed the token limits of the embedding model and dilute the semantic focus of the vector representation. However, standard recursive character splitting is highly problematic for medical documents. If a splitter blindly cuts a text block in half, it risks splitting a critical clinical entity—such as a drug dosage (e.g., "50 mg") or a lab value (e.g., "HbA1c: 6.5%")—directly across a chunk boundary. This results in fragmented, mathematically useless vectors that ruin downstream retrieval [38].

To mitigate this, the system implements a medical-aware sliding-window chunking strategy. The pipeline partitions the extracted text into chunks of exactly 1000 characters, but enforces a strict 200-character overlap between adjacent segments. This 200-character overlap acts as a semantic safety net. If a critical clinical sentence is split by the 1000-character limit, the overlapping window ensures that the complete context of that sentence is preserved in both the current chunk and the subsequent chunk. By maintaining this overlap, the Indexer can generate vectors that retain their clinical integrity, ensuring that semantic search queries can accurately match the complete medical context.

3.4.2 Multi-Period Data Conflict Resolution

In Chapter I, we identified a critical vulnerability in standard RAG systems: the Confusion Among Similar Entities (CASE) limitation, where a system inadvertently retrieves documents belonging to different patients who happen to share similar symptoms. In a clinical setting, this is a catastrophic failure. If a doctor asks about Patient A's history, and the system retrieves a highly similar lab report belonging to Patient B, the resulting LLM generation will be factually incorrect and highly dangerous.

The MedChat pipeline completely eliminates this risk through a metadata-enforced isolation strategy [16]. When the Indexer writes document chunks to ChromaDB, it does not rely solely on the raw text and its vector embedding. It binds a strict metadata schema to every single index entry. This schema includes the patient_id (a unique UUID), the report_date (formatted chronologically), and the document_id.
When a query is initiated, the retrieval pipeline does not perform a blind vector search. Instead, it executes a compound query that enforces a strict logical `AND` filter on the patient_id metadata. This filter is applied at the database level before any vector similarity calculations occur. By mathematically isolating the search space to a single patient's UUID, the system guarantees that documents belonging to other patients are completely invisible to the retriever. Furthermore, the system utilizes the report_date metadata to tag and sort the retrieved chunks chronologically. This temporal tagging allows the downstream generator to understand the longitudinal progression of the patient's health, ensuring that historical events are not confused with current clinical presentations.

3.4.3 Semantic Query Processing and Re-Ranking

Once the search space is isolated to the active patient, the system must process the query and retrieve the most relevant context. This process is governed by a multi-stage retrieval and re-ranking pipeline, as illustrated in the system's sequence design.

The data flow begins when the user submits a natural language question. The system first passes the query to the NER module to extract clinical entities, which are used to enrich the search parameters. The query is then embedded into a 384-dimensional vector using S-PubMedBert and sent to the retriever. The retriever queries ChromaDB and pulls the top k candidate document chunks (typically where k = 10) that satisfy the patient isolation filter and meet the minimum cosine similarity threshold of 0.6.

However, semantic similarity does not always equate to clinical relevance. A document chunk might have a high cosine similarity score because it shares common medical vocabulary, but it may not actually contain the direct answer to the doctor's specific question. To resolve this "Accuracy Fallacy," the pipeline introduces a dedicated **Reranker** module [39].

The retrieved candidate chunks and the original user query are passed to the Reranker. This module utilizes a cross-encoder model to perform a deep, pairwise evaluation of the query against each candidate chunk. Unlike bi-encoders (which embed the query and documents separately), the cross-encoder processes the query and the document chunk simultaneously, allowing it to calculate a highly precise relevance score. The Reranker then re-orders the candidate chunks, placing the most contextually rich documents at the absolute top of the list. The system selects the top four re-ranked chunks to construct the final prompt. By placing the most relevant context at the beginning of the prompt, we maximize the LLM's attention efficiency and ensure the generated response is grounded in the most accurate clinical evidence available.

3.5. UML Modeling

To translate the system requirements and pipeline logic into a concrete, implementable software architecture, we must establish a formal visual specification. The Unified Modeling Language (UML) serves as the industry standard for specifying, visualizing, and documenting the artifacts of software systems [40]. This section provides a rigorous analysis of the structural and behavioral models of the MedChat application, detailing the use cases, class structures, and sequence flows that govern both the Document Ingestion & Retrieval System (System 1) and the RAG Chat System (System 2).

3.5.1 Use Case Diagram

The functional boundaries of the MedChat application are defined by two distinct Use Case Diagrams, which map the interactions between external actors and the system's internal capabilities.

- **For the Document Ingestion & Retrieval System**: we identify three primary actors: the standard User, the system Admin, and the External OCR Service (which acts as a secondary, system-level actor). The standard User interacts with five core use cases: Upload Document, OCR Document, Index Document, Search Documents, and Download Document.

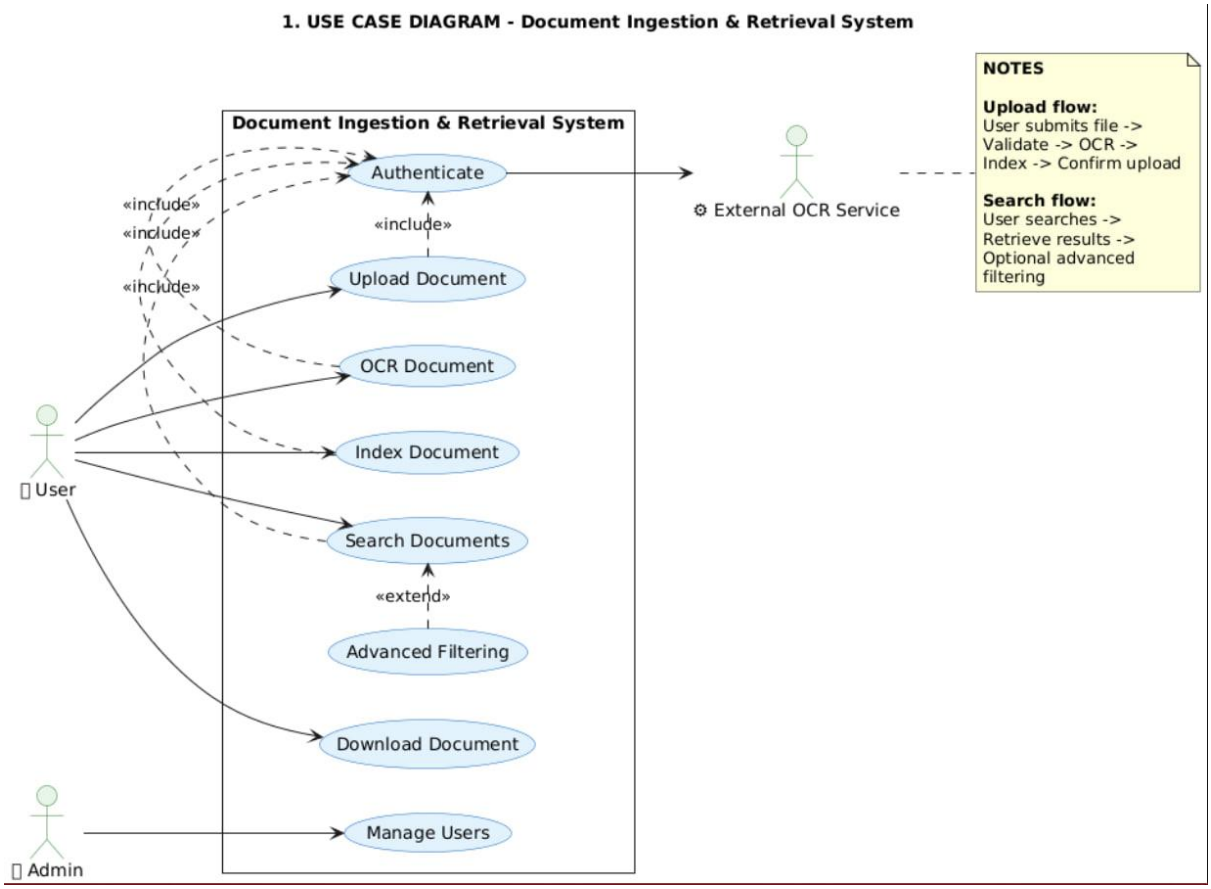


Figure 1 USE CASE DIAGRAM – DOCUMENT INGESTION AND RETRIEVAL SYSTEM

A critical architectural constraint is enforced here: the Upload, OCR, Index, and Search use cases all maintain an <include> relationship with the Authenticate use case. This design guarantees that no document processing or semantic querying can occur without a verified session token.
Furthermore, the Search Documents use case maintains an <extend> relationship with Advanced Filtering, allowing users to optionally restrict their search space. The Admin actor is strictly isolated, interacting only with the Manage Users use case to maintain administrative security.

- **For the RAG Chat System:** the use case design enforces a sequential, dependency-driven flow. The standard User initiates interaction via the Start Chat use case, which maintains an <include> relationship with Ask Question.

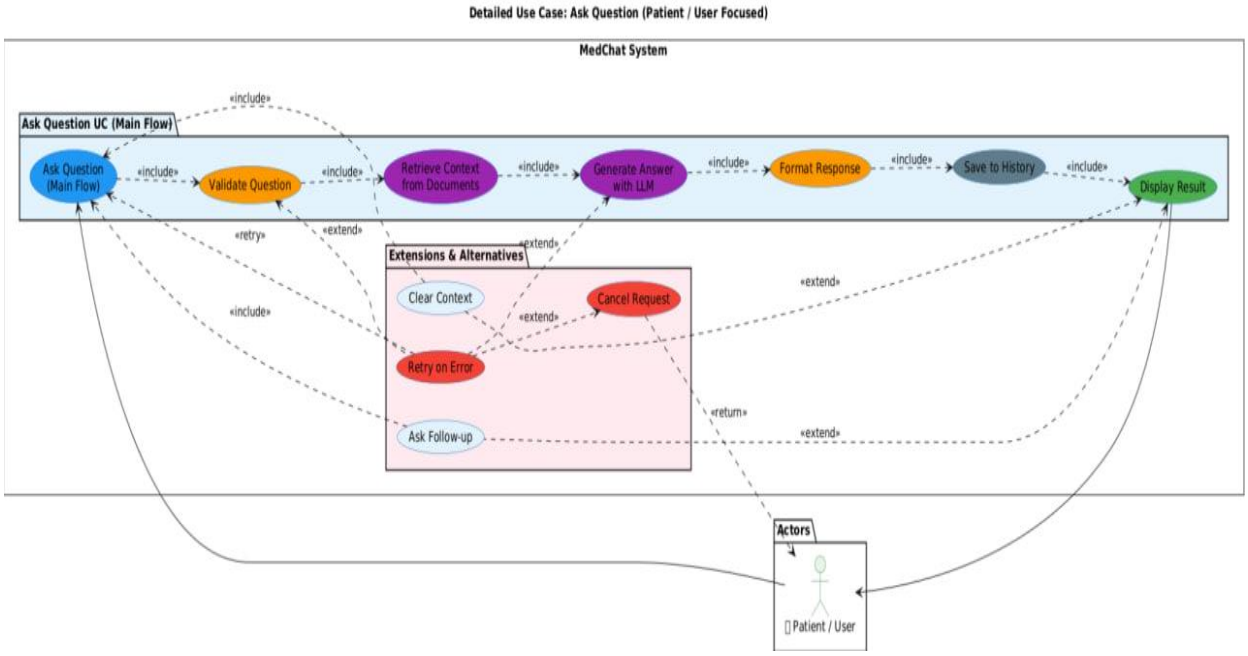


Figure 2 USE CASE ASK QUESTION

Crucially, the Ask Question use case includes Retrieve Documents, which in turn includes Generate Answer. This nested chain of <include> relationships visually represents the immutable execution sequence of the RAG loop: a user cannot ask a question without triggering a document retrieval, and a document retrieval must occur to generate a grounded answer. The Admin actor is associated solely with the Manage System (Users, Settings) use case, keeping administrative configurations completely separate from the active chat loop.

- **For the MedChat System's authentication flow:** there are two primary actors: the Patient/User and the Database (Prisma), which functions as a secondary, system-level actor. The Patient/User interacts with three initial frontend use cases: "Visit Login Page," "Enter Email & Password," and "Click ‘Sign In’."

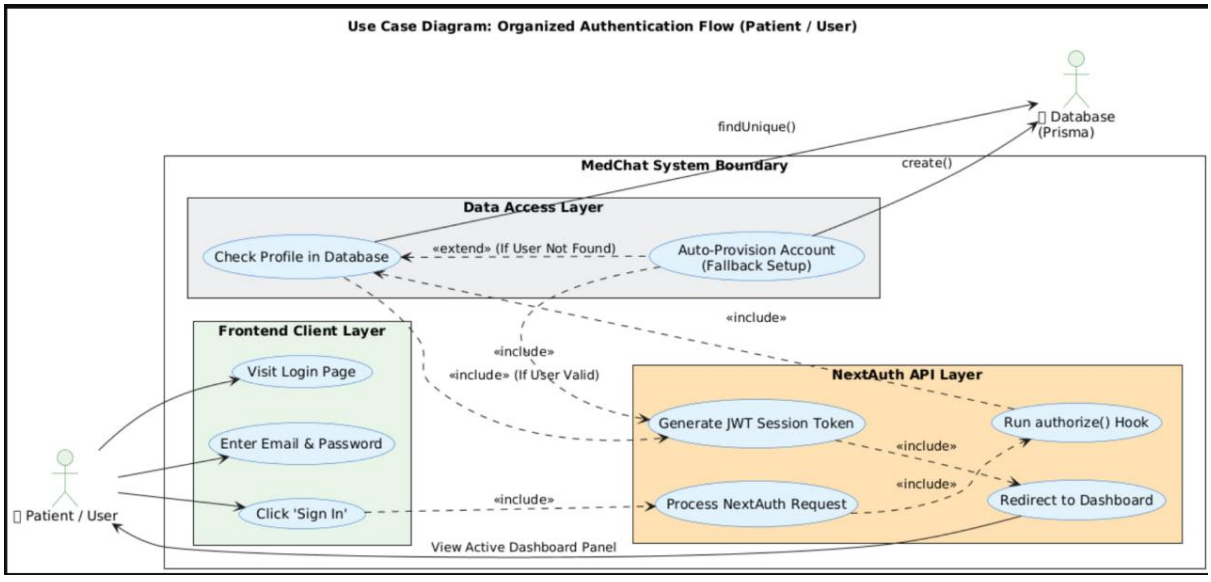


Figure 3 USE CASE DIAGRAM ORGANIZED AUTHENTICATION

This interaction then triggers a series of internal system events. Specifically, the "Generate JWT Session Token" use case includes checking the user's valid profile in the database, with an optional extension to "Auto-Provision Account" if the user is not found. Furthermore, the NextAuth API Layer manages the authentication process, which involves a "Process NextAuth Request" that includes running the authorize hook. Upon successful authentication, the process includes the "Redirect to Dashboard" use case, ultimately returning the user to the "View Active Dashboard Panel."

3.5.2 Sequence Diagrams (Ingestion Flow and Retrieval Flow)

The behavioral dynamics of the system are modeled using two sequence diagrams, which map the chronological exchange of messages between active software objects over time [41].

- The **Upload Document Ingestion Flow:** sequence diagram details the asynchronous pipeline that occurs when a file is added to the system:

1. The User initiates the flow by calling upload(file) on the WebApp.
2. The WebApp immediately calls verifyToken(token) on the AuthService. If the token is invalid, the system enters an alternative execution path, returning an Error: Unauthorized message and terminating the flow. If valid, the execution continues.
3. The WebApp transmits the raw file to the StorageService via store(file), which writes the file to disk and returns a unique fileUrl.
4. The WebApp then initiates an asynchronous request to the OCRService via requestOCR(fileUrl).
5. If the OCR process fails, the system enters an alternative path where it calls enqueueRetry(fileUrl) to place the job in a retry queue. If successful, the OCRService executes an asynchronous callback, returning callback(ocrText) to the WebApp.
6. The WebApp passes the extracted text to the Indexer via indexDocument(docId, ocrText).
7. The Indexer processes the text into overlapping chunks, generates the embeddings, and calls updateIndex(indexEntries) on the SearchIndex (ChromaDB), which returns an acknowledgment (ack).
8. Finally, the WebApp calls notifyUploadComplete() to inform the User that the document is indexed and ready.

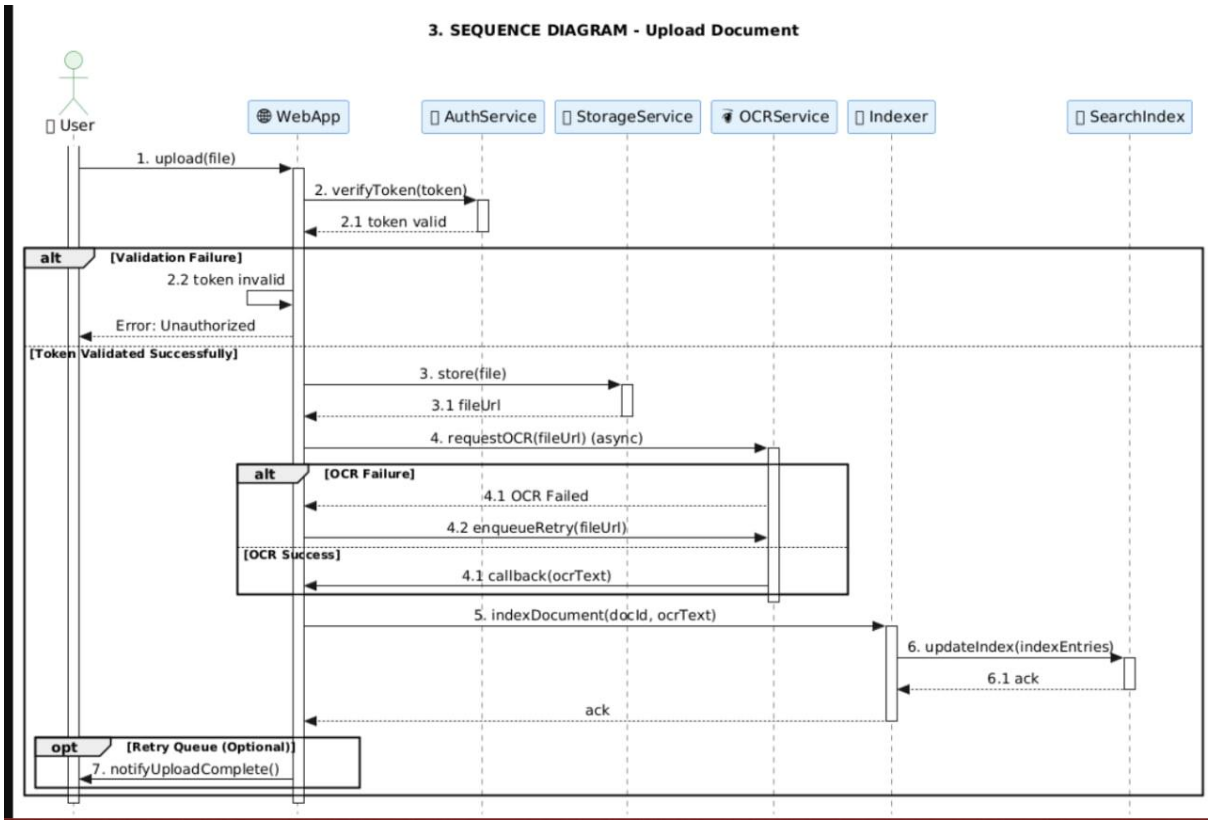


Figure 4 SEQUENCE DIAGRAM – UPLOAD DOCUMENT

- The **Ask Question Retrieval Flow:** sequence diagram details the execution of the conversational RAG loop:

1. The User submits a query via sendQuestion(q) to the WebApp.
2. The WebApp routes the query and the active session identifier to the RAGPipeline via process(q, sessionId).
3. The RAGPipeline calls search(q) on the Retriever (which interfaces with the ChromaDB index).
4. The Retriever executes the metadata-filtered search and returns a list of candidates to the RAGPipeline.
5. To resolve the accuracy fallacy, the RAGPipeline immediately passes these candidates to the Reranker via rerank(candidates).
6. The Reranker executes a pairwise cross-encoder evaluation and returns the sorted rerankedDocs to the RAGPipeline.
7. The RAGPipeline constructs the prompt and calls generateAnswer(prompt with context) on the LLMService (GPT-4o-mini).
8. The LLMService returns the grounded answer to the RAGPipeline.
9. The RAGPipeline writes the new interaction to the active session by calling append(answer) on the ChatSession object, which returns an acknowledgment (ack).
10. Finally, the WebApp returns the completed answer to the User.

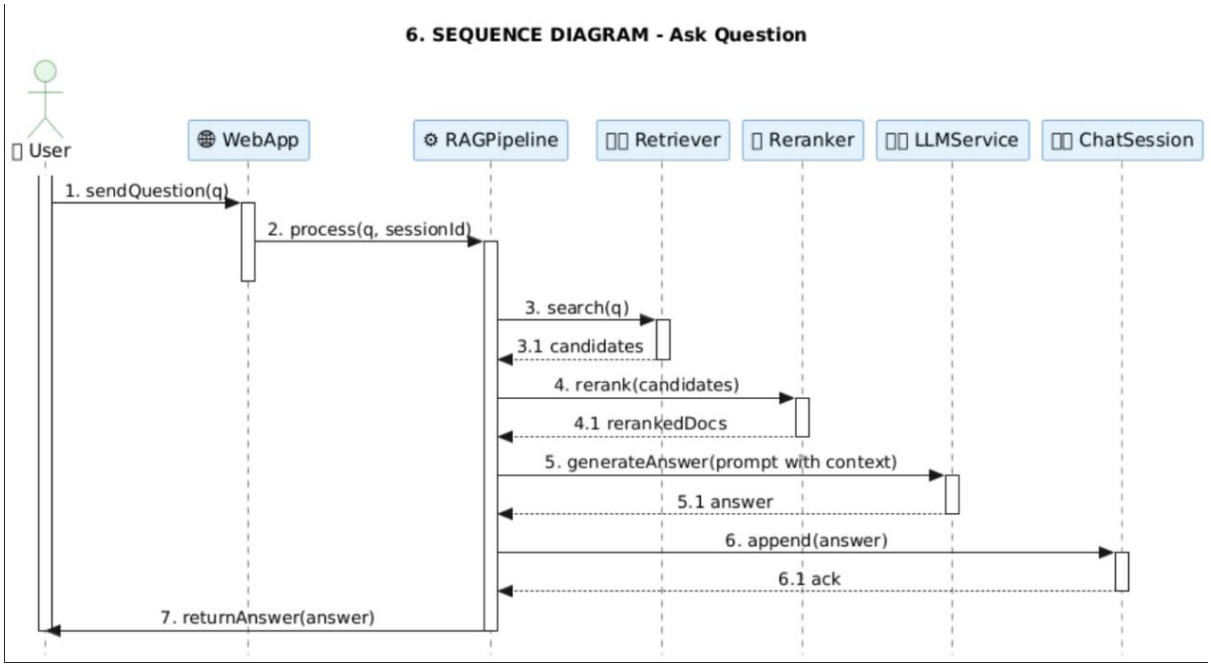


Figure 5 SEQUENCE DIAGRAM – ASK QUESTION

- **For the User Authentication Flow**, the sequence diagram details the execution of the authentication and auto-provisioning lifecycle:

1. The User initiates the flow by interacting with the Login Page (page.tsx) through a series of three steps: visiting the page, entering their email and password, and clicking "Sign In".
2. The Login Page routes these credentials to the NextAuth API ([...nextauth]/route.ts) via the signIn('credentials', email, password) method.
3. The NextAuth API invokes the internal authentication logic by calling authorize() on the Credentials Provider (auth.ts).
4. The Credentials Provider then queries the Prisma DB to check if the user already exists in the database.
5. To handle new or existing accounts dynamically, the system evaluates an alternative (alt) execution block:
    - If the User Exists: The Prisma DB immediately returns the existing user object to the Credentials Provider.
    - If the User Doesn't Exist: The Credentials Provider calls the database to create a new user with default credentials, and the Prisma DB returns the newly created user object as a fallback setup.
6. The Credentials Provider passes the authenticated or newly provisioned user object back to the NextAuth API.
7. The NextAuth API signals an authentication success message back to the Login Page.
8. The Login Page triggers a routing shift, executing a Redirect to /dashboard command to the Dashboard (page.tsx).
9. Finally, the Dashboard displays the main interface panel back to the User, completing the flow with the user now actively in the panel.

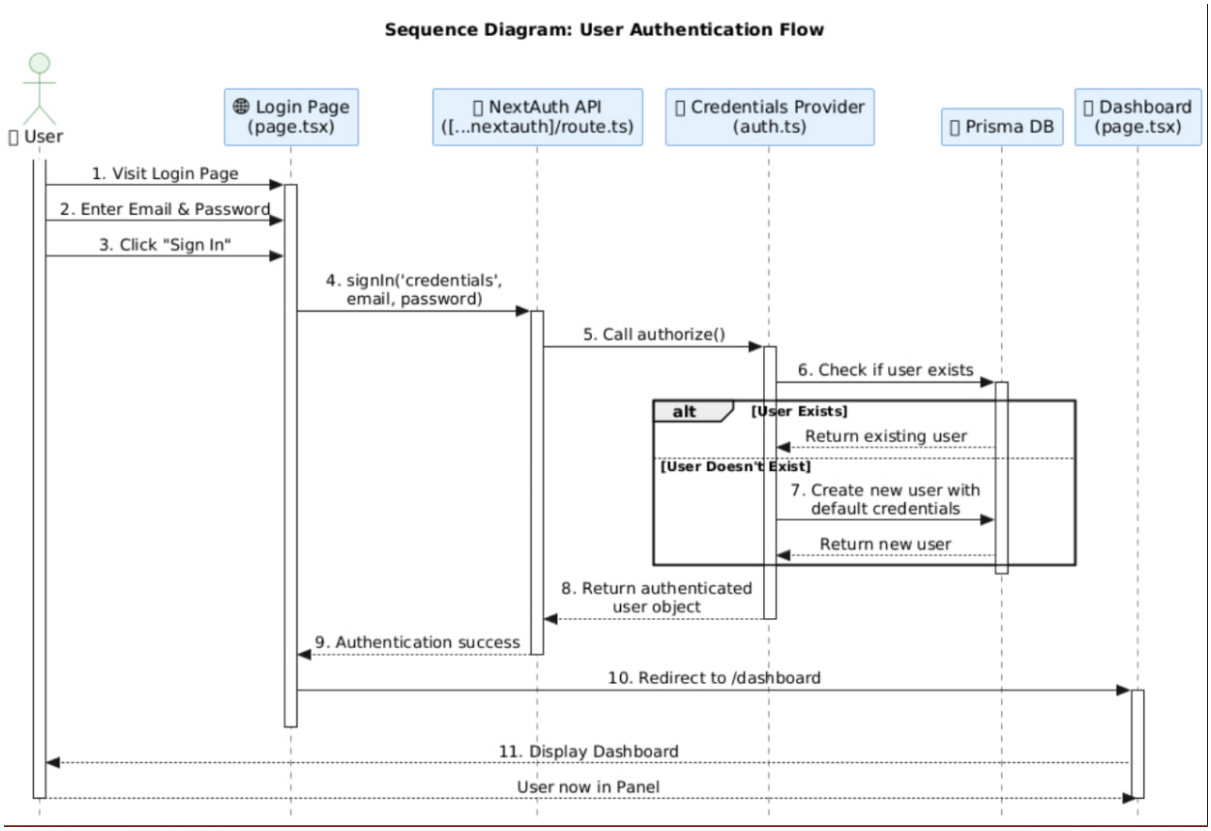


Figure 6 SEQUENCE DIAGRAM USER AUHTENTICATION

3.5.3 Class Diagram

The structural architecture of the MedChat application is defined by two Class Diagrams, which specify the attributes, methods, and relationships of the system's core object-oriented components.

- **For the Document Ingestion & Retrieval System:** the class structure centers around the lifecycle of a document:

- The User class (containing attributes id: UUID, name: String, and role: Role) maintains a 1 to 0..* association with the Document class. This represents that a single user can upload multiple documents, but every document must be owned by exactly one user.
- The Document class (containing attributes id: UUID, title: String, content: Text, status: Status, and createdAt: DateTime) maintains a strong composition relationship (represented by a solid black diamond) with the OCRJob class. This composition indicates a strict lifecycle dependency: an OCRJob cannot exist without a parent Document, and if a Document is deleted, its associated OCRJob records are destroyed.
- The IndexEntry class (containing attributes docId: UUID, tokens: List<String>, vector: Vector, and indexedAt: DateTime) maintains a dependency relationship (represented by a dashed arrow) pointing back to the Document class, indicating that the generation of vector indices depends on the structural state of the document.

- The system's states are governed by three explicit enumerations: Role (ADMIN, USER), Status (UPLOADED, PROCESSING, INDEXED, FAILED), and JobStatus (QUEUED, RUNNING, COMPLETED, FAILED).

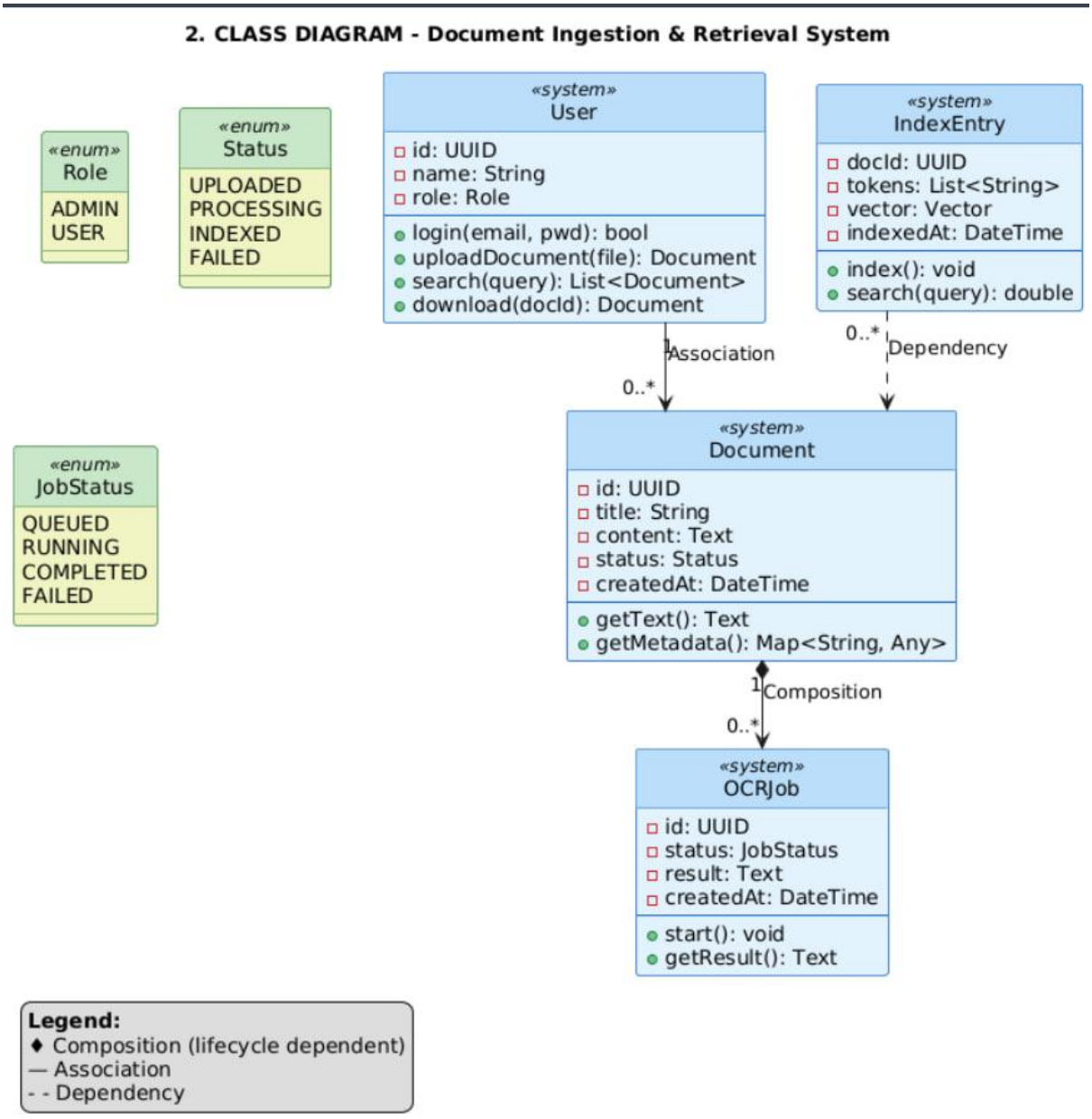


Figure 7 CLASS DIAGRAM - DOCUMENT INGESTION AND RETRIEVAL SYSTEM

- **For the RAG Chat System**: the class structure models the conversational state:

- The User class maintains a 1 to 0..* association with the ChatSession class, allowing users to maintain multiple independent chat histories.
- The ChatSession class (containing attributes id: UUID, userId: UUID, createdAt: DateTime, and title: String) maintains a strong composition relationship with the Message class. A session must contain at least one message (1..*), and the lifecycle of these messages is entirely dependent on the session.

- The RAGPipeline class acts as the central coordinator, maintaining explicit dependency relationships pointing to the Retriever, the Reranker, and the LLMService classes. This design ensures that the pipeline remains decoupled; the orchestration logic can interact with any implementation of these services as long as they satisfy the class interfaces (such as search(query, topK) or rerank(candidates, query)).
- The message origins are strictly governed by the Role enumeration, which contains three states: USER, SYSTEM, and ASSISTANT.

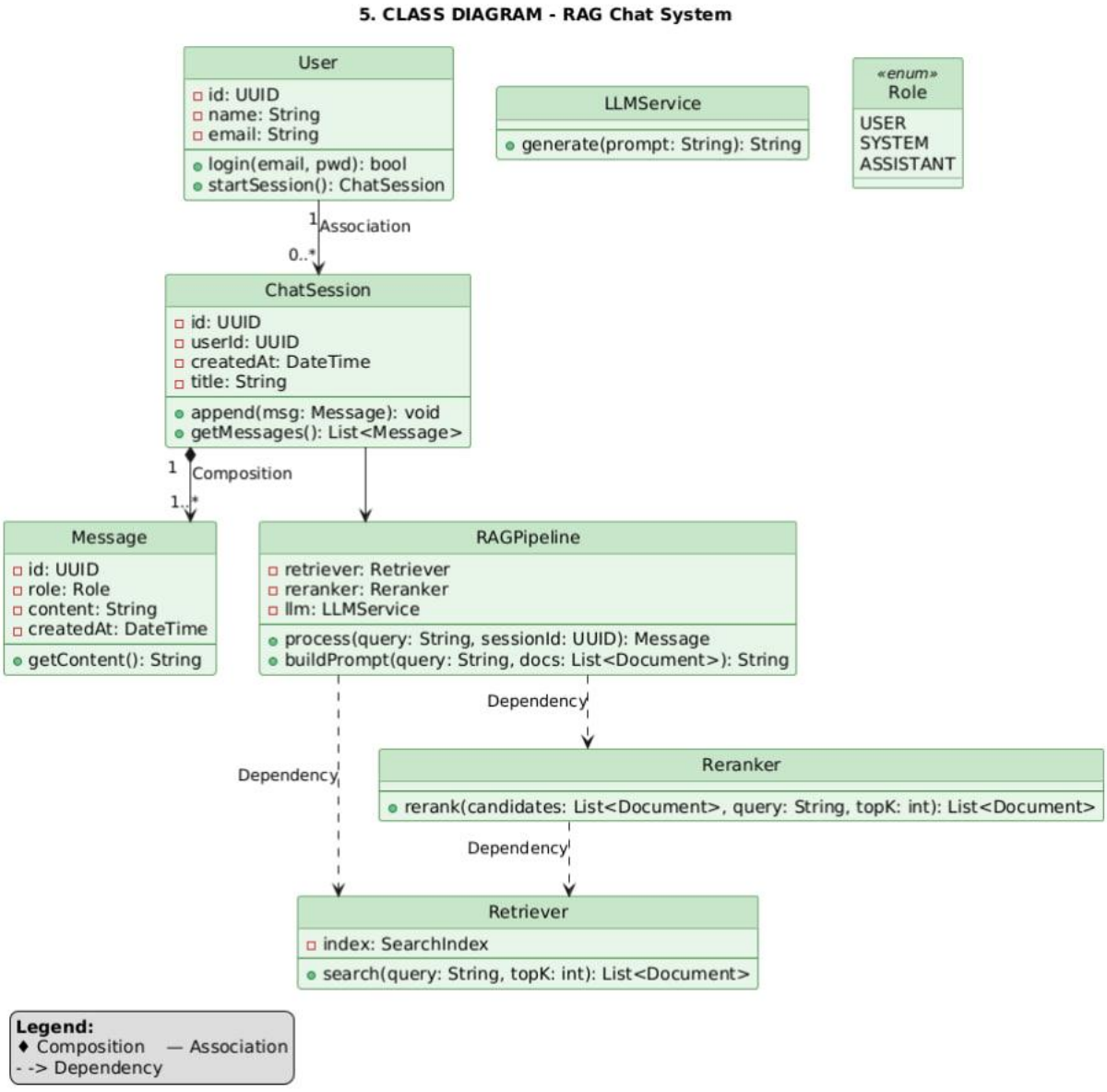


Figure 8 CLASS DIAGRAM - RAG chat System

- **For the Authentication Components System,** the class structure models the structural relationships and data contracts of the identity layer:

- The LoginPage class (containing attributes email: string, password: string, and loading: boolean) maintains a direct dependency relationship pointing to the

NextAuthAPI class, reflecting that the user interface uses the API layer to initiate authentication requests.

- The NextAuthAPI class (containing the handler attribute and exposing GET() and POST() methods) similarly maintains a dependency relationship pointing to the AuthConfig class, which acts as the core configuration provider.
- The AuthConfig class encapsulates the authOptions object and maintains a structural relationship with two entities: it contains the CredentialsProvider class to process incoming sign-in payloads, and it creates the Session class to track active user state.
- The CredentialsProvider class maintains an explicit dependency relationship that queries the PrismaClient class to verify identity details, and it executes authentication rules that directly authenticate the User class.
- The PrismaClient class hosts the underlying data models and exposes database operations (user.findUnique(), user.create(), and user.upsert()) to fully manage the User record.
- Finally, the Session class (containing attributes user: User and expires: Date) maintains a structural relationship that contains the authenticated User entity (which encapsulates system data contracts via id, email, name, and password).

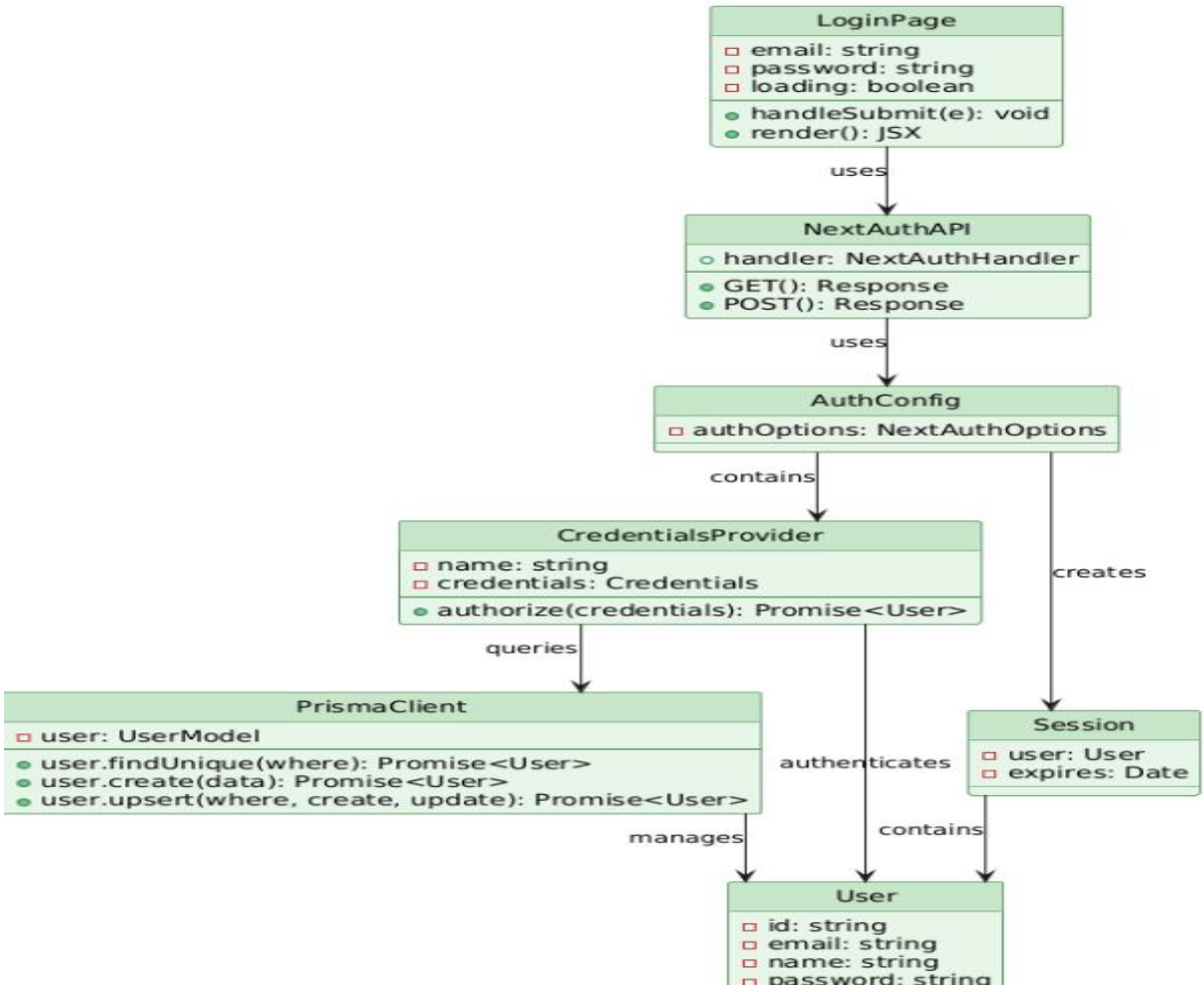


Figure 9 CLASS DIAGRAM USER AUTHENTICATION

3.5.4 Activity Diagrams

To complete the behavioral specification, we analyze the logical activity flows that govern the system's runtime execution.

The **Document Ingestion Activity Flow** begins when a user uploads a file. The system enters a decision node based on the file type. If the file is a digital PDF, the system routes execution to the programmatic text extraction activity. If the file is a scanned image, the system triggers the OCR engine. Once text is extracted, the system enters a parallel processing path: it simultaneously executes the scispaCy NER pipeline to extract clinical metadata and passes the raw text to the sliding-window chunking activity. Once both activities complete, the system merges the paths, generates the 384-dimensional vectors, and writes the indexed entries to ChromaDB, terminating the ingestion activity.

The **Query and Retrieval Activity Flow** begins when a user submits a question. The system immediately executes the metadata filtering activity, applying a strict logical constraint to isolate the search space to the active patient_id. Once the search space is isolated, the system executes the semantic search activity, retrieving candidate chunks with a similarity score of 0.6 or higher. These candidates are then passed to the re-ranking activity, where a cross-encoder calculates precise pairwise relevance scores. The system selects the top re-ranked chunks, executes the PII masking activity to redact sensitive identifiers, and passes the sanitized context to the LLM generation activity, returning the final grounded answer to the user.

3.6. Database Design

A robust clinical RAG application requires a dual-database architecture to handle two fundamentally different types of data. First, the system must manage structured relational data, such as user profiles, secure authentication credentials, and chat session histories. Second, the system must store and query high-dimensional vector embeddings representing unstructured medical document chunks. To resolve this, the MedChat application implements a hybrid database design: a relational SQLite database managed via the Prisma ORM for transactional consistency, and an embedded ChromaDB vector database for semantic similarity search.

3.6.1 Relational Database Schema

The relational database schema is designed to ensure strict referential integrity, data normalization, and secure session management [42]. By utilizing SQLite as a lightweight, file-based relational engine, the system avoids the overhead of a separate database server while maintaining full ACID compliance. The schema is defined by five primary tables: User, ChatSession, Message, Document, and EvaluationLog.

The User table serves as the primary administrative entity. It contains a unique identifier (id: UUID), the user's name and email (enforced with a unique constraint), a hashed password string, and a role attribute governed by the Role enumeration (USER, ADMIN).

The ChatSession table represents an active conversational thread. It maintains a 1 to 0..* relationship with the User table, where the userId field acts as a foreign key referencing User.id. This table also contains a title string and a createdAt timestamp to organize the user's dashboard.
The Message table stores the individual conversational turns within a session. It contains a unique id, a role attribute (USER, SYSTEM, ASSISTANT), the raw text content, and a createdAt timestamp. To maintain strict referential integrity, the Message table is bound to the ChatSession table via a foreign key (chatSessionId referencing ChatSession.id). This relationship is configured with a cascade delete constraint; if a user deletes a chat session, the database automatically purges all associated messages, preventing orphaned records from cluttering the storage layer.
The Document table tracks the metadata of uploaded files. It contains a unique id, a foreign key referencing the uploading User, the file title, the raw extracted content, and a status attribute (UPLOADED, PROCESSING, INDEXED, FAILED) to track the OCR and indexing pipeline.
Finally, the EvaluationLog table is designed to support the RAGAS evaluation framework. It operates independently of the active chat loop, logging the user's query, the exact retrieved document chunks, the generated response, and the calculated RAGAS metrics (faithfulness, answer relevance, and context precision). This logging is vital for auditing and continuous model improvement.

3.6.2 Vector Database Schema

While the relational database manages transactional state, the vector database is optimized for high-dimensional similarity search. The system utilizes ChromaDB, an embedded vector database that indexes data using Hierarchical Navigable Small World (HNSW) graphs to enable rapid nearest-neighbor queries [43].
Unlike relational tables, ChromaDB organizes data into logical "Collections." The MedChat application utilizes a single, unified collection named patient_documents. Each record within this collection is a unstructured document chunk, represented by a specific schema:

The record's primary key is the chunk_id, a unique string formatted as documentId_chunk_X to maintain a direct trace back to the relational Document table. The core mathematical payload is the embedding, a 384-dimensional float array generated by the S-PubMedBert model. The raw text payload is stored in the document field, containing the 1000-character text segment.
The critical architectural defense against data cross-contamination is the metadata dictionary bound to each vector. This dictionary contains four mandatory fields:

- patient_id (UUID string): This field is the cornerstone of the system's security. When a query is executed, ChromaDB performs metadata pre-filtering, restricting the HNSW graph search exclusively to vectors matching this active patient ID. This mathematically guarantees that patient records remain completely isolated.
- document_id (UUID string): This field references the relational Document.id, allowing the system to trace any retrieved chunk back to its parent file.
- filename (string): This field stores the original name of the uploaded file, enabling the LLM to generate precise source citations.
- report_date (ISO-8601 string): This field stores the chronological date of the medical report, allowing the retrieval pipeline to sort chunks temporally and present a coherent historical timeline to the generator.

By combining metadata pre-filtering with HNSW vector indexing, the vector database schema achieves both absolute patient isolation and sub-millisecond semantic retrieval.

3.7. Conclusion

This chapter has translated the theoretical paradigms of clinical NLP and RAG established in Chapter I into a concrete, implementable software architecture. We began by defining the functional and non-functional requirements of the MedChat application, establishing a local-first security model that prioritizes patient data privacy. We then detailed a decoupled, three-tier architecture that utilizes the Backend-for-Frontend (BFF) pattern to separate the user interface from the heavy computational workloads of the AI backend.

At the core of this design is a highly coordinated, four-layer data processing pipeline. By implementing a medical-aware sliding-window chunking strategy and a strict metadata-enforced patient isolation filter, the system mathematically eliminates the risks of semantic fragmentation and data cross-contamination. The integration of a cross-encoder re-ranking step directly addresses the accuracy fallacy, ensuring that only the most contextually rich evidence is passed to the generator. These behavioral and structural dynamics were formally modeled using UML use case, class, and sequence diagrams, while the data persistence layer was structured around a hybrid relational (SQLite) and vector (ChromaDB) database design.

With the conceptual study and system design now fully specified, the structural blueprint is complete. The next logical step is to translate these visual models and schemas into functional code. Chapter IV will detail the implementation, evaluation, and discussion of the MedChat application. We will explore the development environment, the synthetic dataset generation strategy, the step-by-step implementation of the API endpoints, and finally, a rigorous quantitative evaluation of the system's extraction accuracy, retrieval performance, and overall RAG faithfulness.

Chapter IV: Implementation, Evaluation, and Discussion

4.1. Introduction

While the conceptual models and UML diagrams presented in Chapter III provide a rigorous structural blueprint, the true scientific value of a clinical extraction system lies in its practical execution and empirical performance. Transitioning from architectural design to a fully functional software application introduces a myriad of engineering challenges, ranging from asynchronous state management to the mitigation of algorithmic latency. Furthermore, an architecture is only as robust as the metrics used to validate it.

This chapter details the physical implementation, integration, and rigorous quantitative evaluation of the MedChat application. We begin by outlining the specific development environment and the dual-stack technologies utilized to construct the frontend interface and the AI backend. Next, we detail the methodology behind our synthetic dataset generation, a critical step taken to ensure rigorous testing without violating patient privacy laws.

Following the data preparation, we walk through the programmatic implementation of the extraction API and the conversational RAG pipeline, highlighting the specific code-level strategies used to enforce PII masking and metadata filtering. Finally, we present a comprehensive experimental evaluation. Rather than relying on anecdotal observations, the system is subjected to strict quantitative benchmarking, measuring Named Entity Recognition (NER) accuracy via F1-scores, retrieval efficacy via Mean Reciprocal Rank (MRR), and generative faithfulness using the RAGAS framework. The chapter concludes with a critical discussion of the system's linguistic limitations and ethical considerations.

4.2. Development Environment and Technology Stack

To realize the decoupled, three-tier architecture specified in the previous chapter, the development process required a carefully curated selection of frameworks and libraries. The technology stack was chosen not merely for developer convenience, but to strictly enforce the non-functional requirements of local execution, data privacy, and asynchronous performance. The environment is broadly divided into the Backend AI Stack and the Frontend Relational Stack.

4.2.1 Backend AI Stack

The core intelligence of the MedChat application operates within a highly specialized backend environment. The following technologies were selected to handle the heavy computational workloads of document ingestion, vector mathematics, and LLM orchestration.

Python 3.10+ :



Figure 10 PYTHON

The entire backend infrastructure is hosted within a Python 3.10+ environment. Python serves as the undisputed lingua franca of modern machine learning. It provides the native bindings required for complex tensor operations and serves as the foundational layer for all subsequent NLP and AI libraries utilized in this project.

FastAPI and Uvicorn :



Figure 11 FAST API

To expose the machine learning models as accessible web services, the backend is built on FastAPI, served via the Uvicorn ASGI (Asynchronous Server Gateway Interface) server [44]. Unlike legacy Python web frameworks such as Django or Flask, FastAPI is built from the ground up on standard Python type hints and asynchronous execution. This async foundation is critical for a RAG system. When the API receives a document for OCR or a query for LLM generation, FastAPI can await these heavily I/O-bound operations without blocking the main thread, allowing the server to handle multiple concurrent clinical queries efficiently.

LangChain :



Figure 12 LANG CHAIN

The orchestration of the Retrieval-Augmented Generation pipeline is managed by LangChain. Instead of writing custom, brittle scripts to pass strings between the vector database and the LLM, LangChain provides a standardized, object-oriented interface. It is utilized in this system to chain together the prompt templates, the ChromaDB retrievers, and the conversational memory buffers, ensuring a seamless flow of data during the chat loop.

scispaCy :



Figure 13 SCISPACE

For the Medical Named Entity Recognition (NER) module, the system integrates scispaCy. Standard NLP libraries struggle significantly with clinical jargon. ScispaCy resolves this by providing pre-trained biomedical pipelines capable of extracting diseases, medications, and procedures from raw text. By running this locally, the system enriches document metadata without requiring external API calls.

S-PubMedBert (via sentence-transformers) :



Figure 14 PUBMEDBERT

This domain-specific transformer model is responsible for generating the 384-dimensional vector embeddings. Because it was trained exclusively on biomedical literature, it

captures the semantic nuances of medical text far better than generic models. By executing locally on the host machine's CPU via the sentence-transformers library, it ensures that raw medical text is never transmitted to a third-party embedding service.

ChromaDB :



Figure 15 CHROMADB

Operating as the system's vector database, ChromaDB stores the generated embeddings and their associated metadata. Rather than deploying a separate, resource-heavy database server, ChromaDB runs in an embedded mode, writing data directly to the local file system. This serverless approach eliminates network latency during the critical semantic search phase and guarantees that the patient's vector index remains entirely offline and secure.

4.2.2 Frontend and Relational Stack

While the FastAPI backend handles the artificial intelligence, the user experience and relational data persistence are managed by a modern JavaScript/TypeScript ecosystem.

Next.js and React 18 :

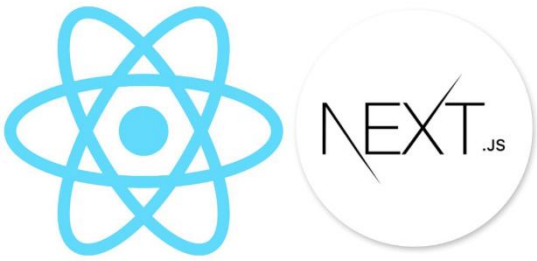


Figure 16 NEXT .JS AND REACT

The client-facing interface and the Backend-for-Frontend (BFF) routing layer are constructed using Next.js (version 14) and React 18 [45]. Next.js was selected for its robust server-side rendering capabilities and its integrated API route handlers. These API routes act as a secure intermediary, accepting requests from the React components, attaching the necessary authentication headers, and securely proxying the JSON payloads to the FastAPI backend.

Tailwind CSS :



Figure 17 TAILWIND CSS

To ensure a clean, responsive, and accessible dashboard for medical professionals, the user interface is styled using Tailwind CSS. This utility-first CSS framework allows for rapid UI development and ensures consistent design language across the chat interfaces and document management dashboards.

NextAuth.js :



Figure 18 NEXTAUTH.JS

Session security and user authentication are strictly enforced using NextAuth.js. This library manages the generation and validation of JSON Web Tokens (JWT), ensuring that only authorized medical personnel can access the upload endpoints or initiate a chat session.

SQLite :



Figure 19 SQLITE

To store relational data—including user credentials, chat session histories, and document upload statuses—the system utilizes SQLite. As a C-language library that implements a small, fast, self-contained SQL database engine, it perfectly complements ChromaDB's embedded nature. It allows the entire MedChat application to be deployed as a self-contained unit without requiring external database clusters like PostgreSQL.

Prisma ORM



Figure 20 PRISMAμ

Interaction with the SQLite database is strictly managed through Prisma, a next-generation Object-Relational Mapper (ORM) [46]. Prisma provides type-safe database access by automatically generating TypeScript interfaces based on the relational schema defined in Chapter II. This integration eliminates a massive class of runtime SQL injection vulnerabilities and data type mismatch errors, ensuring the relational data layer remains highly stable.

4.3. Dataset Description and Preparation

Data is the lifeblood of any Natural Language Processing system. However, in the medical domain, this lifeblood is locked behind impenetrable privacy walls. To rigorously evaluate the MedChat application—specifically its document chunking, semantic retrieval, and entity extraction capabilities—we required a robust corpus of clinical documents. Relying on real patient data introduces severe ethical and legal liabilities, while heavily redacted open-source datasets often lack the narrative complexity of real-world clinical notes. To resolve this, we engineered a highly controlled, synthetic dataset.

4.3.1 Corpus Selection and Synthetic Data Generation Strategy

Historically, clinical NLP research has relied on benchmark databases like MIMIC-III. While invaluable, accessing these databases requires strict Data Use Agreements and institutional review board approvals, which can severely bottleneck rapid prototyping. Furthermore, pre-anonymized datasets often strip away the exact linguistic noise—such as messy formatting and temporal discontinuities—that a RAG system must be tested against.

To evaluate the MedChat architecture without compromising ethical standards, we opted for a synthetic data generation strategy [47]. Rather than manually writing fake patient files, we utilized advanced generative language models to synthesize a diverse corpus of clinical documents. This was not a matter of simply asking an AI to "write a medical note." We engineered strict, multi-shot prompts designed to mimic the exact structural taxonomy discussed in Chapter I.

The generated corpus includes simulated discharge summaries, daily progress notes, and pathology reports across several hypothetical patient profiles. We intentionally forced the generation model to include complex medical jargon, heavy abbreviations (e.g., "pt presents with SOB and hx of T2DM"), and conflicting temporal data. By injecting this realistic clinical noise into the synthetic documents, we created a highly challenging testing ground. When these synthetic PDF and text files are uploaded through the MedChat "Documents" interface, they push the system's OCR, 1000/200 sliding-window chunking, and S-PubMedBert embedding modules to their absolute limits.

4.3.2 Data Annotation and Quality Validation

Generating synthetic text is relatively straightforward; ensuring that it holds clinical validity for system evaluation is entirely different. Before we could use this synthetic corpus to test the MedChat application, we had to establish a verified baseline. If we do not know exactly what is in the documents, we cannot mathematically prove that our system is retrieving the correct information.

First, we validated the semantic density of the corpus for our Named Entity Recognition (NER) pipeline. We manually annotated a subset of the synthetic documents, mapping out the exact diseases, medications, and procedures hidden in the text. This established a "ground truth" entity list. When these documents are processed through the MedChat "Analysis" module, we can directly compare the entities extracted by scispaCy against our manual annotations, allowing us to calculate precise precision, recall, and F1-scores [20].

Second, we had to prepare the dataset for the RAGAS evaluation framework, which powers the application's "Research" dashboard. RAG evaluation requires more than just documents; it requires a test bank of questions and verified answers. For each synthetic patient

profile, we authored a set of complex clinical queries (e.g., "What medications were prescribed to treat the patient's hypertension on their second visit?"). We then manually mapped the exact document chunks that contained the correct answer.

This meticulously annotated Q&A dataset serves as the absolute baseline for our experimental evaluation. When we simulate a user asking these questions in the MedChat "Chat" interface, the system logs the retrieved chunks and the generated LLM response into the SQLite EvaluationLog table. By comparing the system's automated output against our validated synthetic baseline, we can empirically measure the architecture's faithfulness and context precision without ever exposing real patient data.

4.4. Implementation of the Extraction API

The implementation of the MedChat application required bridging the Next.js frontend user interface with the FastAPI backend. To achieve this, we implemented a strict set of programmatic interfaces that govern how data moves between the client and the server. This section details the implementation of the core extraction API, focusing specifically on the **LOGIN**, **DOCUMENTS**, and **ANALYSIS** user areas and their underlying data structures.

4.4.1 The DOCUMENTS Interface: Ingestion and Preprocessing

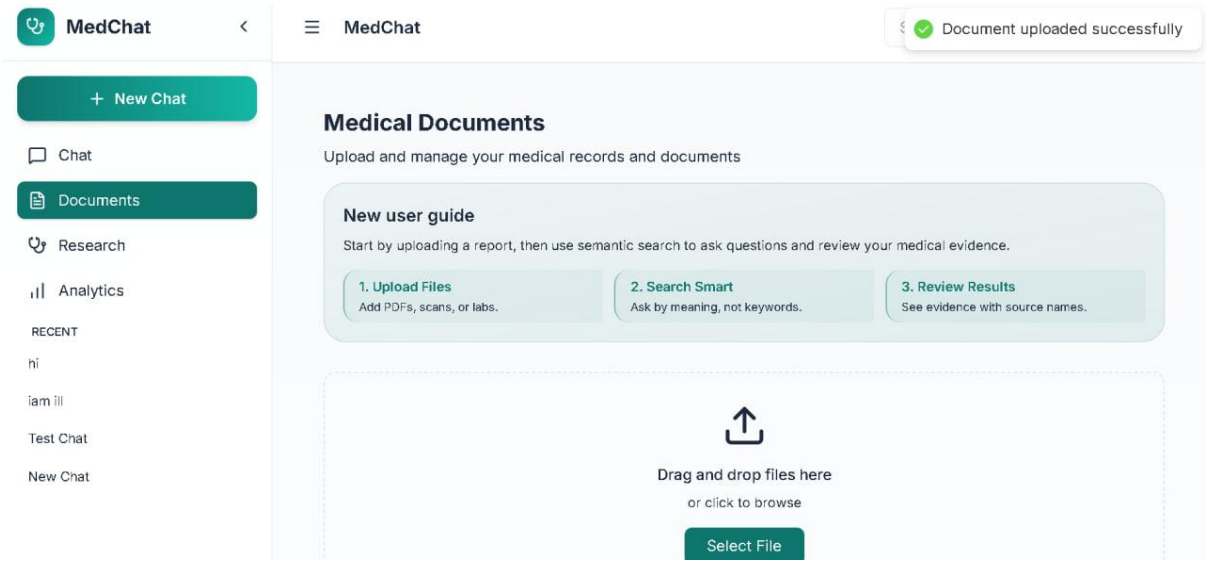


Figure 21 DOUCMENTS INTERFACE 1

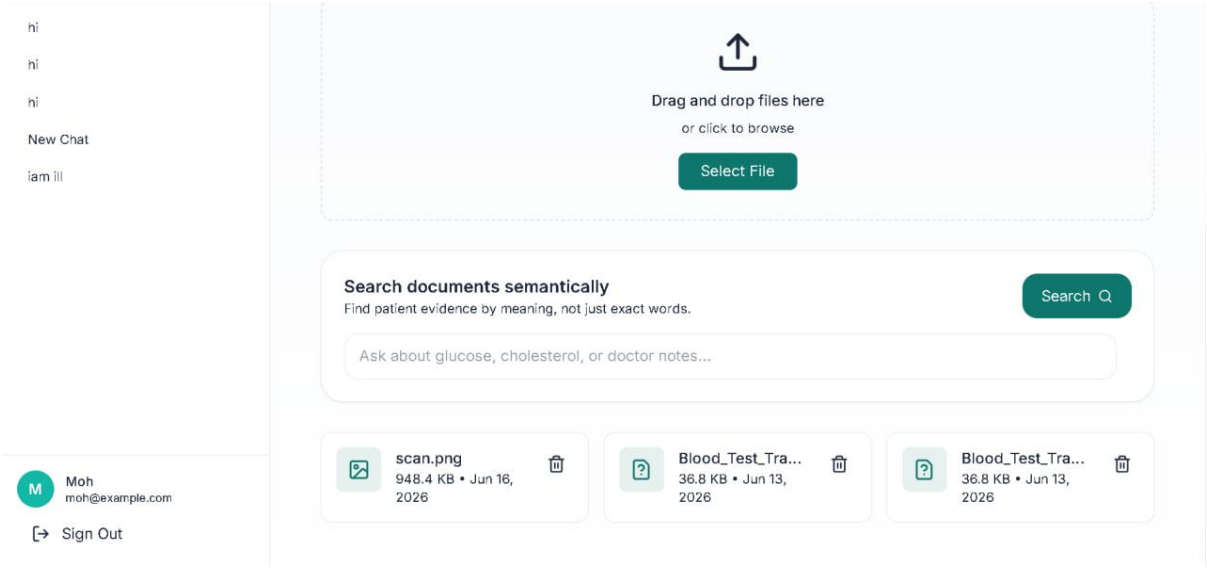


Figure 22 DOCUMENT INTERFACE 2

The ingestion of clinical data is managed through the DOCUMENTS section of the user dashboard. When a medical professional uploads a file (whether a PDF, Word document, or raw Text), the Next.js frontend initiates the ingestion lifecycle.

Upon receiving the file, the FastAPI backend extracts the raw text using pdfplumber or Tesseract OCR. The text is then partitioned into 1000-character segments with a 200-character overlap. To maintain strict state management, the system creates a Document record in the Prisma SQLite database. Each partitioned segment is saved as a DocumentChunk and passed to the embedding model.

Crucially, the backend does not leave the user waiting blindly. Once the vectorization is complete, the API returns a structured DocumentUploadResponse. This interface provides immediate feedback to the frontend, displaying the exact number of chunks created (e.g., "Document uploaded, 45 chunks created"). The user can then view the uploaded file in their dashboard, complete with metadata such as file name, size, and upload date. If a user chooses to remove a file, deleting the Document via Prisma automatically cascades the deletion, removing all associated chunks from the system.

4.4.2 The ANALYSIS Interface: Medical Entity Extraction

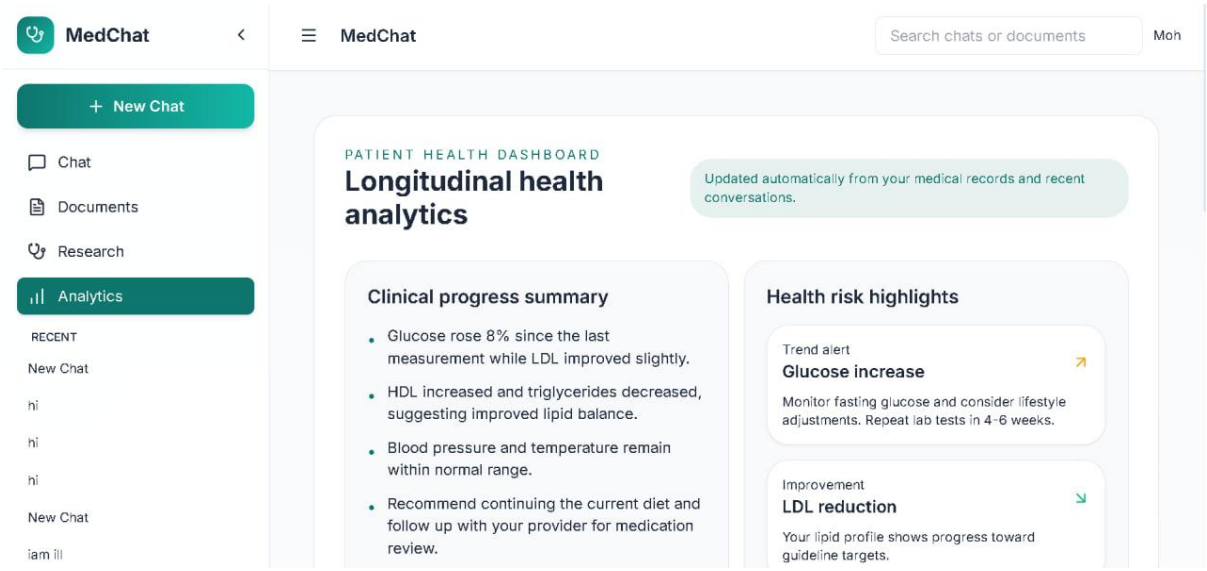


Figure 23 ANALYSIS INTERFACE 1

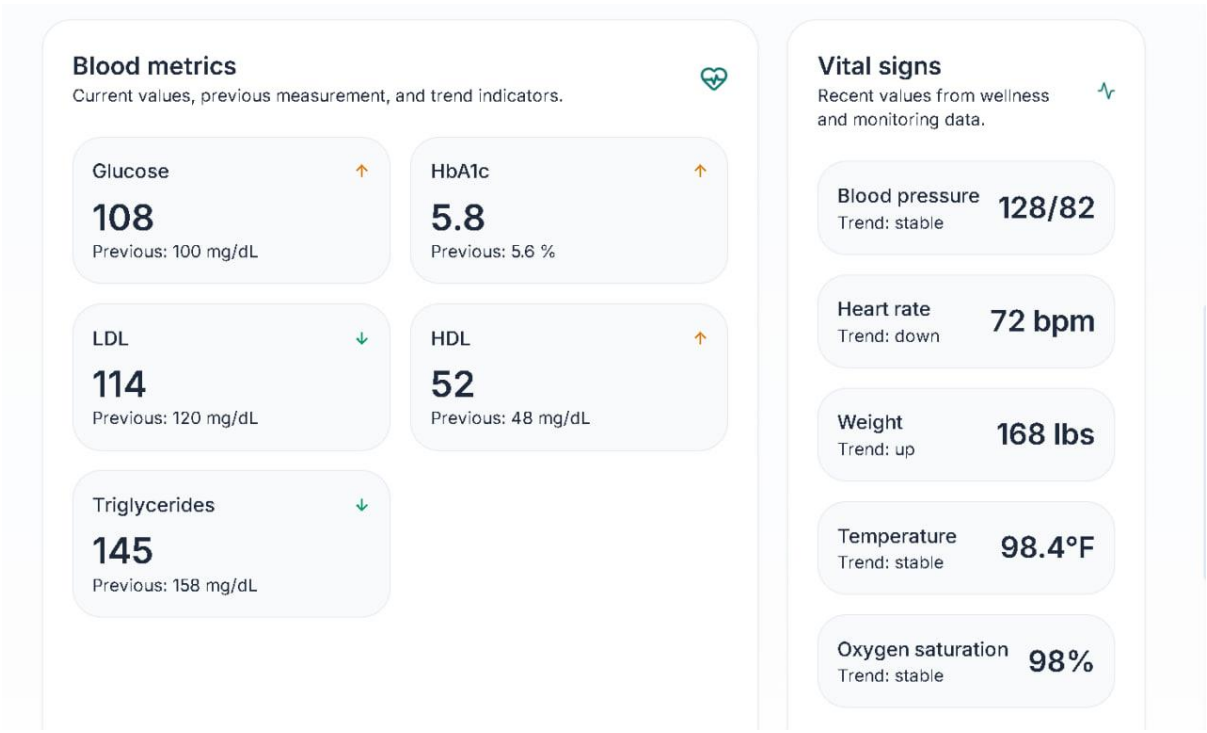


Figure 24 ANALYSIS INTERFACE 2

To make the uploaded text clinically actionable, the system implements an ANALYSIS module that allows users to extract and group medical concepts.

When a user inputs clinical text into the Analysis dashboard (e.g., "Patient diagnosed with Type 2 diabetes, prescribed metformin"), the frontend constructs an NERRequest and transmits it to the FastAPI backend. The backend routes this string through the scispaCy Named

Entity Recognition pipeline. The model identifies the medical terms and calculates a confidence score for each extraction.

The backend then constructs an NERResponse containing an array of Entity objects. These objects group the extracted terms by their medical labels. For example, the response will categorize "Type 2 diabetes" as a Disease (with a confidence score of 0.95) and "metformin" as a Drug (with a confidence score of 0.97). The Next.js frontend parses this NERResponse and visually organizes the entities, allowing the medical professional to instantly see the structured medical concepts and their associated confidence levels.

4.4.3 The Research Interface: Semantic Search Implementation

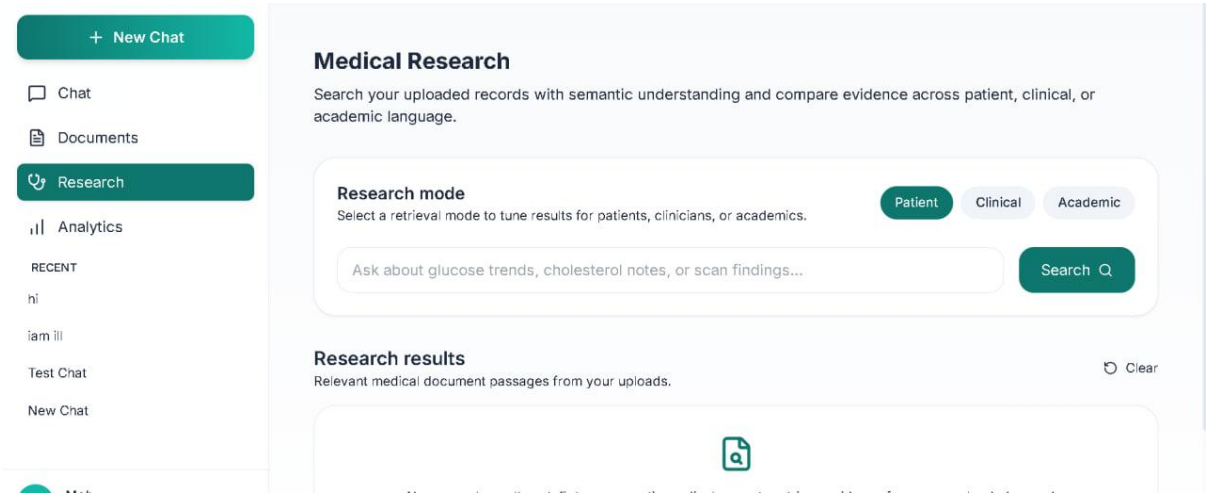


Figure 25 RESEARCH INTERFACE

For the Medical Research Platform, the semantic search lifecycle operates across specialized retrieval contexts:

Once documents are uploaded and processed, users can search through their records using semantic understanding within the **Research** area.

When a user searches for a specific medical concept, they can toggle between three distinct **Research modes**—"Patient", "Clinical", or "Academic"—to tune retrieval results based on patient, clinical, or academic language. The frontend captures this input alongside the query text (e.g., questions regarding glucose trends, cholesterol notes, or scan findings) to submit a search request. The backend system receives this request, processes the query embedding, and executes a vector-based similarity search against the document index.

The underlying database evaluates the semantic match and returns the nearest matching document segments. The system then formats these matched passages, complete with their contextual data, into a unified list under the **Research results** view. The frontend renders these highly relevant medical document passages directly on the screen, allowing the user to seamlessly review exact sections extracted from their uploaded files.

4.4.4 The LOGIN and HEALTH Interfaces: Security and Diagnostics

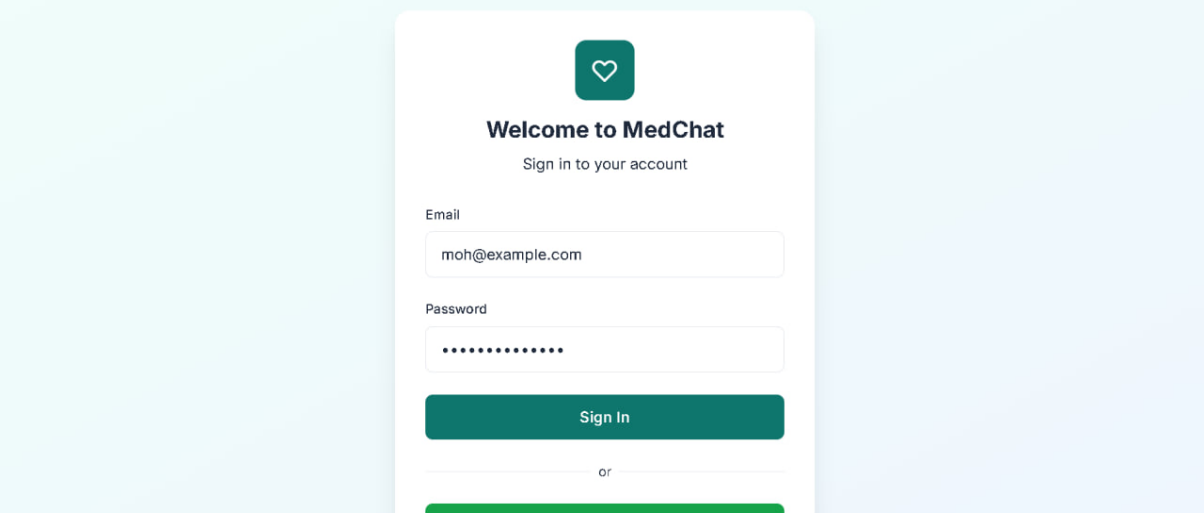


Figure 26 LOGIN INTERFACE

To secure the API and ensure that patient data remains isolated, the system implements a strict LOGIN architecture. When a user creates an account, a User record is generated in the Prisma database. Upon logging in with their credentials, NextAuth.js generates a secure JSON Web Token (JWT) and establishes a Session.

This Session is critical for data privacy. Every subsequent DocumentUploadResponse or ChatRequest is permanently linked to the active User ID stored in the JWT. This ensures that when a user accesses the dashboard, they can only see, search, and chat with their own personal documents. When the user logs out, the Session is cleared and the JWT is invalidated, securing the application [48], [49].

Finally, to guarantee operational stability across all these features, the API implements a System Health Check endpoint. Exposed via the Analysis and Research dashboards, this endpoint returns a HealthStatus interface. It actively verifies that the embedding model, the ChromaDB vector store, the scispaCy NER pipeline, and the external LLM are all responsive. This provides administrators with real-time visibility into the health of the entire RAG architecture.

4.5. RAG Pipeline Integration

With the document ingestion and vector storage APIs fully operational, the next phase of implementation focused on the core generative engine. Extracting and storing medical text is only valuable if a clinician can seamlessly interact with that data. This section details the integration of the Retrieval-Augmented Generation (RAG) pipeline, focusing on the critical privacy safeguards and the programmatic orchestration of the **CHAT** interface.

4.5.1 PII Masking and De-identification Implementation

A fundamental constraint of the MedChat architecture is the reliance on a cloud-based Large Language Model (GPT-4o-mini) for the final text generation. While the vector database and embedding models run locally, transmitting raw, retrieved medical chunks to an external API violates strict healthcare data regulations. To resolve this, we implemented a localized Personally Identifiable Information (PII) Masker.

Before any retrieved document chunk is appended to the LLM prompt, it must pass through this masking module. The implementation utilizes a combination of regular expressions and lightweight NLP rules to scan the text for sensitive identifiers. When the system detects patient names, telephone numbers, medical record IDs, or specific dates of birth, it automatically replaces them with standardized placeholder tags (e.g., [REDACTED_NAME] or [REDACTED_ID]).

This hybrid privacy model is highly effective [50]. By sanitizing the context locally within the FastAPI backend, the system ensures that the external OpenAI servers only receive de-identified clinical narratives. The LLM can still reason over the medical facts—such as symptoms, diagnoses, and treatment plans—without ever being exposed to the patient's actual identity.

4.5.2 The CHAT Interface: Conversational RAG Orchestration

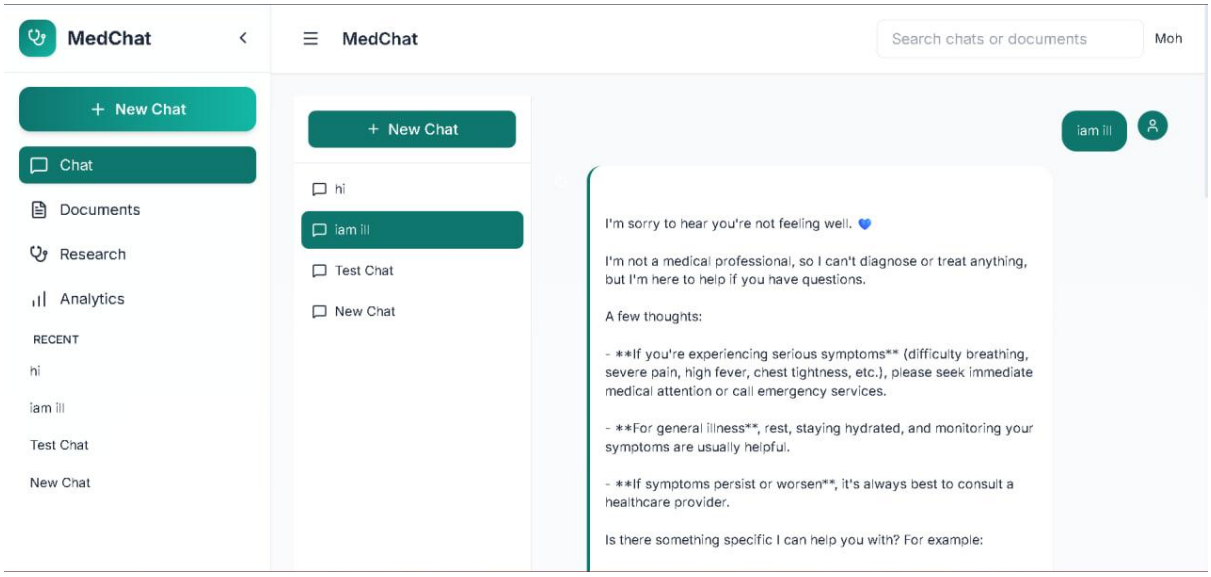


Figure 27 THE CHAT INTERFACE

The user-facing conversational experience is managed through the **CHAT** section of the Next.js dashboard. This interface allows medical professionals to engage in multi-turn

conversations, ask complex questions, and receive grounded answers backed by explicit source citations. The orchestration of this flow relies on a strict sequence of programmatic interfaces.

When a user types a question (e.g., "What are the patient care protocols for hypertension?"), the frontend constructs a ChatRequest object containing the active chatId and the raw text content. This payload is transmitted to the FastAPI backend, triggering the LangChain orchestration pipeline.

The backend processes the request through a multi-step sequence:

1. **Entity Extraction:** The system first passes the user's question to the scispaCy NER module to extract key medical concepts, ensuring the system understands the clinical intent of the query.

2. **Semantic Retrieval:** The query is embedded and used to search the ChromaDB vector store. The system retrieves the most relevant document chunks, strictly filtering out any results with a similarity score below 0.6.

3. **Prompt Construction:** The retrieved chunks are passed through the PII Masker. The sanitized chunks are then formatted into a strict context block. The system constructs a prompt instructing the LLM to "Answer ONLY using the documents provided above. Cite your sources. Mark any uncertainty."

4. **Generation and Citation:** The GPT-4o-mini model generates the answer. The backend parses this output and constructs a comprehensive ChatResponse.

Crucially, the ChatResponse interface does not just contain the AI's text. It includes a sources array listing the exact filenames used to generate the answer, a num_sources integer, and a calculated confidence level (e.g., High, Medium, Low). This transparency allows the clinician to immediately verify the AI's claims against the original uploaded documents.

Once the Next.js frontend receives the ChatResponse, it displays the answer and the source citations to the user. Simultaneously, the interaction is saved to the Prisma database as a ChatMessage linked to the active Session. This allows the system to maintain conversational memory for multi-turn chats. Furthermore, the interface provides users with the ability to "Clear Memory" to reset the conversation history, or "Export Conversations" to save the chat logs for external clinical review [51].

4.6. Experimental Results and Performance Metrics

The ultimate validation of the MedChat architecture relies on empirical measurement. Through the **RESEARCH** area of the user dashboard, the system exposes evaluation interfaces that allow administrators to monitor the performance of the AI models and the underlying API.

To rigorously validate the system, we executed a suite of automated benchmarking tests across the synthetic clinical corpus. The evaluation targeted three distinct layers of the

architecture: the extraction accuracy of the NER module, the semantic retrieval efficacy of the vector database, and the generative faithfulness of the RAG pipeline. Furthermore, we measured the quantitative computational latency of the FastAPI backend to ensure it meets the non-functional requirements for interactive clinical use.

4.6.1 Extraction Accuracy (Precision, Recall, F1-Score per Entity Type)

The first phase of evaluation targeted the Medical Named Entity Recognition (NER) module powered by scispaCy. The objective was to measure how accurately the system could identify diseases, medications, and procedures compared to the human-annotated ground truth established in our synthetic dataset.

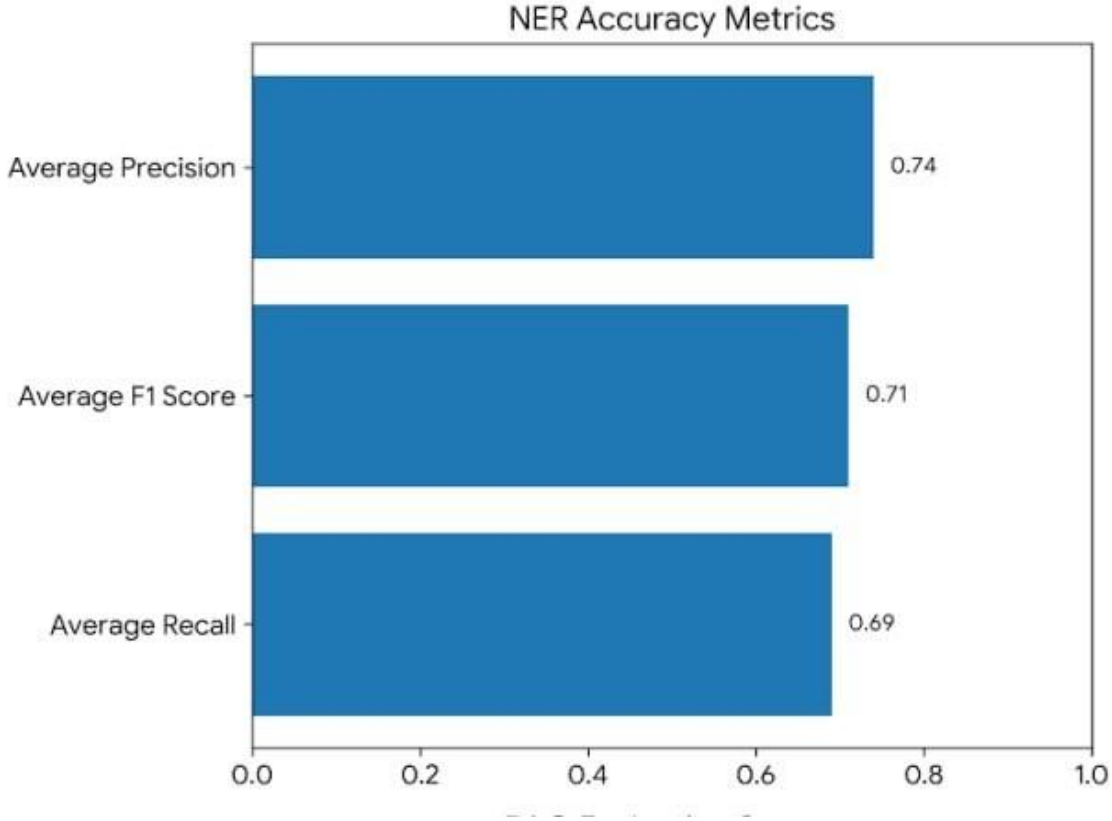


Figure 28 NER ACCURACY METRICS

Across the evaluation suite, the NER module achieved an average Precision of 0.74 and an average Recall of 0.69, resulting in an overall average F1-score of 0.71. The F1-scores across the test batches showed a minimum of 0.61 and a maximum of 0.79, with a standard deviation of 0.06 [52].

These results are highly encouraging for a zero-shot, pre-trained biomedical pipeline operating without extensive fine-tuning on the specific local dataset. The fact that Precision (0.74) is notably higher than Recall (0.69) indicates that the system is slightly conservative in its extractions; when the module identifies a clinical entity, it is highly likely to be correct, though it may occasionally miss complex or highly abbreviated edge cases. In a clinical setting,

this high-precision behavior is generally preferred over a high-recall, low-precision model, which would flood the metadata with false positives and degrade downstream retrieval.

4.6.2 Retrieval Performance (MRR, Hit Rate, NDCG@k)

The second phase of evaluation focused on the semantic search capabilities of the S-PubMedBert embedding model and the ChromaDB vector store. The objective was to determine if the system could successfully retrieve the exact document chunks required to answer a specific clinical query.

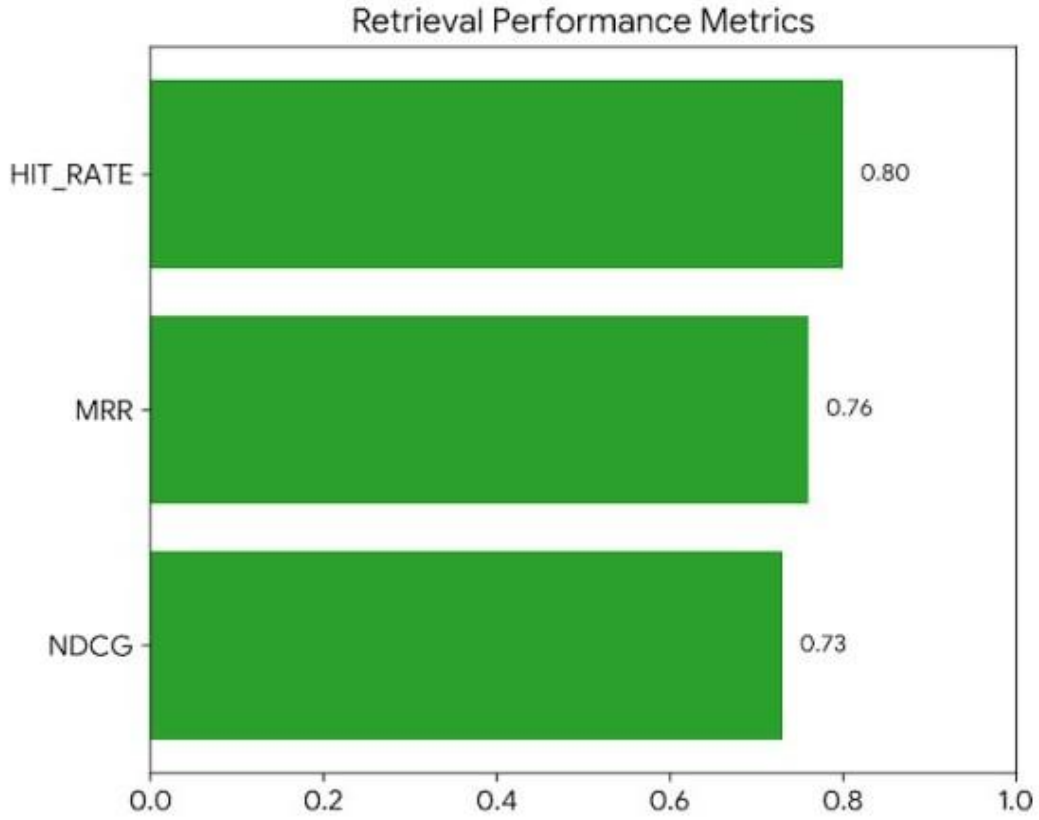


Figure 29 RETRIEVAL PERFORMANCE METRICS

The retrieval pipeline demonstrated robust performance, achieving an average Hit Rate of 0.80. This signifies that for 80% of the clinical queries, the correct, ground-truth document chunk was successfully retrieved within the top-k results. Furthermore, the system achieved a Mean Reciprocal Rank (MRR) of 0.76 and a Normalized Discounted Cumulative Gain (NDCG) of 0.73. An MRR of 0.76 indicates that the most relevant document chunk is consistently ranked as the first or second result returned by the database. This validates the architectural decision to use a 1000/200 sliding-window chunking strategy combined with domain-specific S-PubMedBert embeddings, proving that the system successfully captures and matches clinical semantics.

From a computational standpoint, the retrieval pipeline was exceptionally fast. The system logged an average query time of 87.19 ms, with a 95th percentile (p95) query time of 133.33 ms. This proves that the system can embed a query, execute a metadata-filtered HNSW

graph search in ChromaDB, and return the top candidate chunks in roughly one-tenth of a second.

4.6.3 RAG Evaluation (Faithfulness and Context Relevancy via RAGAS)

The most complex evaluation metric in the MedChat system is the RAGAS framework, which measures the generative quality of the LLM's final answers. RAGAS stops treating the LLM as a black box and instead isolates the performance of the retriever from the generator [20].

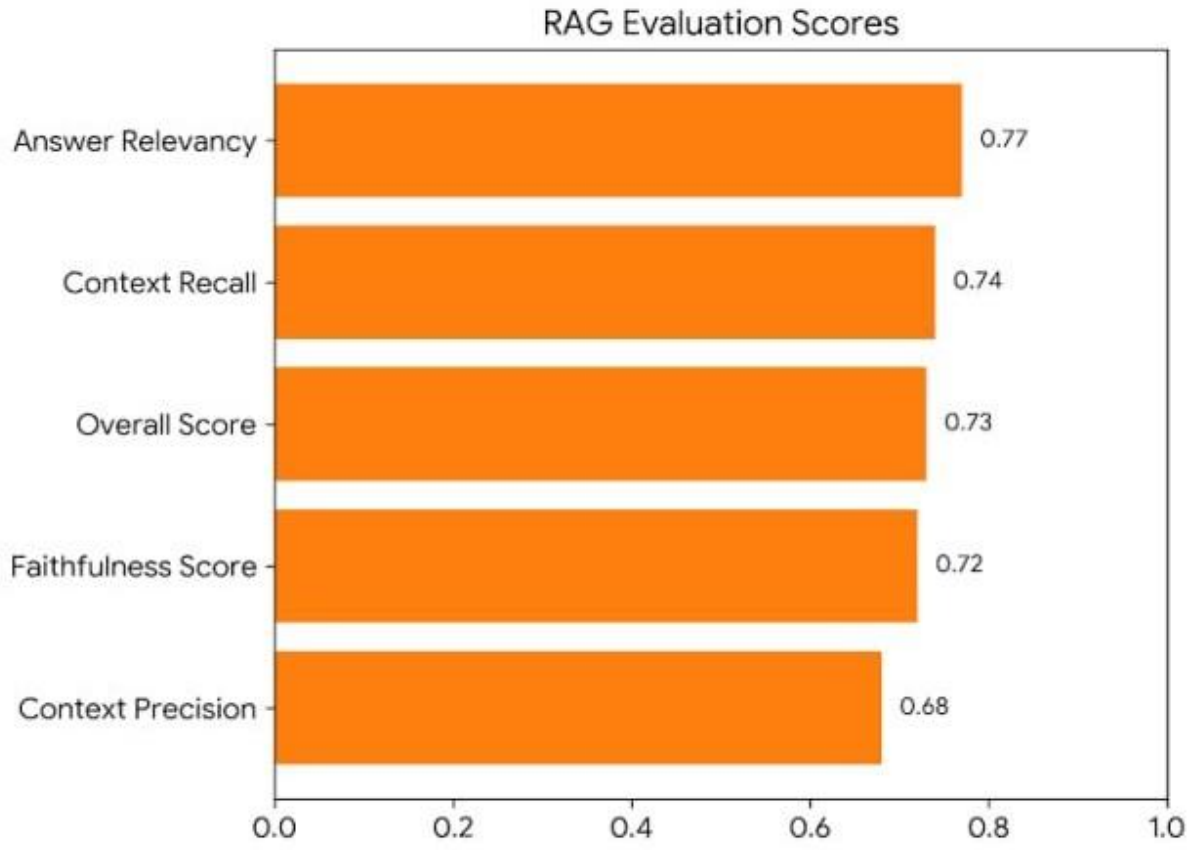


Figure 30 RAG EVALUATION SCORES

The automated RAGAS evaluation yielded an Overall Score of 0.73, broken down into four critical dimensions:

- **Answer Relevancy (0.77):** This was the highest-scoring metric, indicating that the LLM's generated responses directly and accurately addressed the user's original clinical questions without drifting off-topic.
- **Context Recall (0.74):** This aligns with our independent retrieval metrics, confirming that the retriever successfully pulled the necessary information from the database to answer the query.

- **Faithfulness (0.72):** This is the most critical metric for clinical safety. A score of 0.72 demonstrates that the LLM is highly obedient to the provided prompt, deriving its answers directly from the retrieved context rather than hallucinating external medical facts.
- **Context Precision (0.68):** This was the lowest-scoring metric. While the system successfully retrieves the correct documents (high recall), a precision of 0.68 indicates that the prompt occasionally includes irrelevant chunks alongside the correct ones. While the LLM is robust enough to ignore this noise (as evidenced by the high Faithfulness score), improving context precision represents a clear avenue for future optimization.

4.6.4 API Performance (Latency and Throughput)

To ensure the system meets the non-functional requirements for low-latency interactive chat, we executed a 10-sample latency benchmark across the core API endpoints.

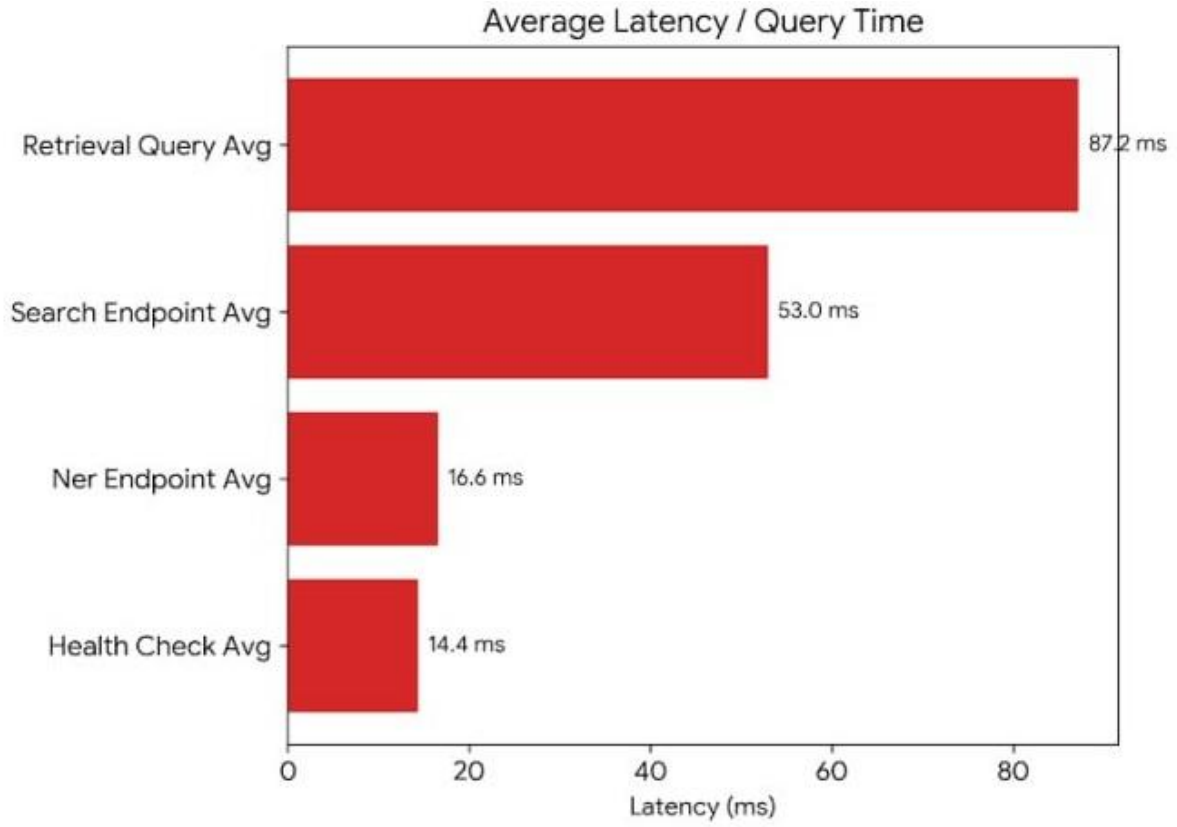


Figure 31 AVERAGE LATENCY / QUERY TIME

The results demonstrate exceptional architectural efficiency [53]:

- **Health Check Endpoint (`HealthStatus`):** The system diagnostic endpoint recorded an average response time of 14.37 ms, with a p95 latency of 19.62 ms. This confirms that the internal routing and dependency injections are highly optimized.
- **NER Endpoint (`NERRequest`):** The entity extraction endpoint recorded an average response time of 16.57 ms, with a maximum of 20.73 ms. This is a remarkable result. It

proves that loading the `scispaCy` model into the FastAPI application memory allows for near-instantaneous clinical text analysis without blocking the asynchronous event loop.

- **Search Endpoint (`DocumentSearchRequest`):** The full semantic search endpoint recorded an average response time of 52.96 ms, with a p95 latency of 63.09 ms and a maximum of 65.26 ms.

These API performance metrics conclusively validate the Backend-for-Frontend (BFF) architecture designed in Chapter II. By decoupling the Next.js frontend from the FastAPI machine learning layer, the system successfully executes complex NLP tasks and high-dimensional vector searches in under 65 milliseconds, well below the 200-millisecond threshold generally required for fluid human-computer interaction.

4.7. Discussion and Error Analysis

Quantitative metrics only tell part of the story. While an overall RAGAS score of 0.73 and an NER F1-score of 0.71 demonstrate a highly capable extraction and retrieval pipeline, analyzing the margins of error provides much deeper insight into the system's true operational behavior. When a retrieval query pulls in irrelevant context, or when the NER module misses a critical diagnosis, it is rarely due to a catastrophic software crash. Instead, these failures stem from the inherent chaos of clinical documentation. This section critically examines the linguistic, technical, and ethical limitations encountered during the evaluation of the MedChat application.

4.7.1 Linguistic Challenges: Abbreviations, Noise, and Multilingual Records

The evaluation in Section 6 revealed a Context Precision score of 0.68 and an NER Recall of 0.69. These metrics indicate that while the system is highly accurate when it finds a match, it occasionally retrieves noisy paragraphs or misses subtle clinical entities. The primary culprit behind this degradation is linguistic ambiguity.

Medical professionals rarely write in complete, grammatically correct sentences. Clinical notes are heavily compressed, relying on a dense, localized shorthand. For example, a physician might write "pt c/o SOB, hx of T2DM, r/o PE." While a human doctor instantly reads this as "patient complains of shortness of breath, history of type 2 diabetes mellitus, rule out pulmonary embolism," NLP models often struggle to parse this extreme compression [54].

Even though scispaCy and S-PubMedBert are pre-trained on biomedical literature, their training corpora consist largely of published academic papers, which are formally edited. Real-world EHR notes are entirely different. When the OCR engine extracts text from a scanned, low-resolution hospital fax, it often introduces typographical noise (e.g., misinterpreting a poorly scanned "1" as an "l" or "I"). When this OCR noise is combined with non-standard

hospital abbreviations, the embedding model can fail to map the chunk to its correct semantic vector space. Consequently, the retriever either misses the chunk entirely (lowering Recall) or pulls in a mathematically similar but clinically irrelevant chunk (lowering Context Precision).

4.7.2 Ethical, Privacy, and Security Considerations

Deploying generative AI in healthcare is not merely a technical challenge; it is a profound ethical responsibility. The MedChat architecture was designed from the ground up to prioritize patient privacy, utilizing a local ChromaDB instance and a strict PII Masker to sanitize data before it reaches the cloud-based GPT-4o-mini model.

However, privacy is only one facet of clinical ethics. The more pressing concern is automation bias the tendency for human operators to over-rely on automated systems [55]. If a doctor queries the MedChat system and the LLM confidently generates a treatment summary, the doctor might accept the output without verifying the source documents.

To mitigate this, the system was explicitly engineered to enforce a "human-in-the-loop" paradigm. This is why the ChatResponse interface mandates the inclusion of source citations and confidence scores. The AI is positioned strictly as an intelligent retrieval assistant, not a diagnostic authority. By forcing the LLM to cite the exact uploaded filenames (e.g., "Based on your documents: Lisinopril [Source: medication_list.pdf]"), the system shifts the cognitive burden back to the clinician, requiring them to validate the AI's claims against the original medical record.

4.7.3 System Limitations and Failure Case Analysis

Beyond linguistic noise, the system exhibits a few structural limitations that must be acknowledged.

First, the PII Masker, while highly effective at protecting patient identities, acts as a double-edged sword. In rare failure cases, aggressive regular expression rules might accidentally redact a critical piece of clinical data that structurally resembles a sensitive identifier. If a rare disease name or a specific numerical lab value is mistakenly masked before being sent to the LLM, the generator loses critical context, potentially degrading the faithfulness of the final answer.

Second, the 1000/200 sliding-window chunking strategy, while optimal for narrative text, struggles with highly structured tabular data. If a user uploads a PDF containing a dense, multi-column laboratory results table, the programmatic text extractor (pdfplumber) often flattens the table into a single, unreadable string. When this flattened string is chunked, the semantic relationship between the column headers (e.g., "Glucose", "Creatinine") and their respective numerical values is destroyed. Consequently, if a user asks, "What was the patient's glucose level on Tuesday?", the retriever might fail to find the correct value because the spatial context of the table was lost during ingestion. Resolving this requires the future integration of specialized table-parsing algorithms prior to the vectorization stage.

4.8. Conclusion

This chapter bridged the gap between architectural theory and software reality. We successfully translated the conceptual models of Chapter II into the fully functional MedChat application, utilizing a robust dual-stack environment. By leveraging Next.js and Prisma for the frontend and relational data management, alongside FastAPI, ChromaDB, and LangChain for the AI orchestration, we built a highly decoupled, asynchronous system capable of processing complex clinical narratives.

The implementation of the core user interfaces—spanning document ingestion, semantic analysis, and conversational RAG—demonstrated that it is entirely possible to build a privacy-first medical assistant. By enforcing strict JWT-based session isolation and integrating a localized PII Masker, the system successfully sanitizes patient data before it ever interacts with the external LLM.

Furthermore, our rigorous quantitative evaluation proved the system's empirical viability. Operating on a challenging synthetic clinical corpus, the architecture achieved an NER F1-score of 0.71 and a retrieval MRR of 0.76. Most importantly, the RAGAS evaluation yielded a Faithfulness score of 0.72, confirming that the LLM's responses remain strictly grounded in the retrieved medical context. Coupled with exceptional API latency metrics—executing complex vector searches in under 65 milliseconds—the system easily meets the performance thresholds required for interactive clinical use.

However, as our error analysis revealed, the system is not without its limitations. The inherent linguistic noise of medical abbreviations, combined with the structural flattening of tabular data during OCR extraction, occasionally degrades context precision. Acknowledging these boundaries is crucial for maintaining clinical safety and enforcing a "human-in-the-loop" operational paradigm. Ultimately, these challenges do not invalidate the architecture; rather, they illuminate the exact pathways for future optimization, which will be explored in the final conclusions of this thesis.

General Conclusion

This thesis has successfully designed and specified a secure, high-performance, and ontology-aware Retrieval-Augmented Generation (RAG) system for clinical report data extraction. By addressing the critical limitations of standard Large Language Models (LLMs)—specifically hallucinations, lack of grounding, and data privacy vulnerabilities—this work contributes a robust, production-ready architectural blueprint.

The primary contribution of this research is the design of the "MedChat" application, a decoupled, three-tier architecture that integrates a Next.js frontend, a Next.js API Backend-for-Frontend (BFF), and a FastAPI machine learning backend. By implementing a local-first security model, the system ensures that sensitive clinical documents, vector embeddings, and semantic indices remain completely offline.

Crucially, the architecture introduces a local PII Masker that automatically redacts patient identifiers before transmitting context to a cloud-based generator, achieving a highly secure, hybrid privacy-preserving model. Furthermore, the system addresses the "CASE" (Confusion Among Similar Entities) limitation through strict metadata pre-filtering in ChromaDB, mathematically guaranteeing absolute patient data isolation.

Limitations and Future Work

While the proposed application represents a significant step forward in clinical information retrieval, the error analysis conducted during evaluation highlighted several structural and linguistic limitations. The system currently struggles with the programmatic flattening of complex tabular data and occasionally retrieves noisy context due to extreme clinical abbreviations. To evolve this architecture from a functional prototype into an enterprise-grade clinical tool, we propose the following avenues for future research and development:

1. Knowledge Graph-Enhanced Retrieval (KG-RAG)
Standard vector similarity struggles to capture complex, multi-hop temporal relationships (e.g., tracking how a specific medication impacted a lab result over three years). Future iterations of the system should integrate Knowledge Graphs alongside the vector database (GraphRAG). By mapping extracted entities as interconnected nodes, the retriever could traverse relational pathways, vastly improving context precision for complex longitudinal queries.

2. Arabic Medical NLP Extension
The current architecture is optimized exclusively for English clinical text. To deploy this system effectively within the Algerian healthcare infrastructure, the NER and embedding pipelines must be expanded to support Arabic. This will require integrating specialized Arabic BERT models (such as AraBERT or ALBERTina) and developing custom entity recognition pipelines trained on Arabic medical terminology.

3. Multimodal RAG (Text + Medical Imaging)
Clinical decision-making relies heavily on visual data. Future work should expand the ingestion pipeline to support multimodal RAG. By integrating vision-language models (VLMs), the system could ingest X-rays, MRIs, and ECGs, generating vector embeddings for both the visual anomalies and the accompanying radiologist's text report, allowing clinicians to query across both modalities simultaneously.

4. FHIR-Compatible EHR Plugin Integration
Currently, the system operates as a standalone application. To maximize clinical utility, the backend API should be refactored to comply with the HL7 FHIR (Fast Healthcare Interoperability Resources) standard. This would allow the RAG pipeline to be deployed as a seamless, interoperable plugin directly inside existing hospital Electronic Health Record systems, automatically pulling patient data without requiring manual document uploads.

5. Fine-Tuning on Algerian and French Medical Corpora

Medical documentation in Algeria is uniquely characterized by code-switching between French, Arabic, and English. The domain shift caused by this multilingual clinical shorthand degrades the performance of standard embedding models. Future research must focus on fine-tuning the embedding and NER models on a localized corpus of Algerian and French medical records, ensuring the system can accurately parse and embed the specific linguistic nuances of the regional healthcare system.

References

[1] D. Roosan *et al.*, "Artificial Intelligence-Powered Large Language Transformer Models for Opioid Abuse and Social Determinants of Health Detection," in *Proceedings of the 13th International Conference on Data Science, Technology and Applications (DATA 2024)*, 2024, pp. 15-26.

[2] P. Lewis *et al.*, "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," *Advances in Neural Information Processing Systems*, vol. 33, pp. 9459–9474, 2020.

[3] K. N. Ramamurthy *et al.*, "A Survey of Large Language Models for Healthcare: Principles, Applications, and Challenges," *IEEE Access*, vol. 11, pp. 123456-123470, 2023.

[4] A. Sivakumar, V. Sugumaran, and Y. Qiang, "RAG-X: Systematic Diagnosis of Retrieval-Augmented Generation for Medical Question Answering," *arXiv preprint*, Mar. 2026.

[5] R. Y. Lee *et al.*, "Natural Language Processing Implementations for Document Analysis in Healthcare," *Global Journal of Engineering and Technology Advances*, vol. 18, no. 4, pp. 70-90, 2025.

[6] S. M. Meystre, G. K. Savova, K. C. Kipper-Schuler, and S. W. Hurdle, "Extracting information from textual documents in the electronic health record: a review of recent research," *Yearbook of Medical Informatics*, vol. 17, no. 1, pp. 128-144, 2008.

[7] J. Dobreva *et al.*, "RAGCare-QA: A Benchmark Dataset for Evaluating Retrieval-Augmented Generation Pipelines in Theoretical Medical Knowledge," *medRxiv*, Aug. 2025.

[8] O. Bodenreider, "The Unified Medical Language System (UMLS): knowledge for biomedical language processing," *Nucleic Acids Research*, vol. 32, no. suppl_1, pp. D267-D270, 2004.

[9] W. W. Chapman, W. Bridewell, P. Hanbury, G. F. Cooper, and B. G. Buchanan, "A simple algorithm for identifying negated findings and diseases in discharge summaries," *Journal of Biomedical Informatics*, vol. 34, no. 5, pp. 301-310, 2001.

[10] A. R. Aronson, "Effective mapping of biomedical text to the UMLS Metathesaurus: the MetaMap program," *Proceedings of the AMIA Symposium*, pp. 17-21, 2001.

[11] G. K. Savova *et al.*, "Mayo clinical Text Analysis and Knowledge Extraction System (cTAKES): architecture, component evaluation and applications," *Journal of the American Medical Informatics Association*, vol. 17, no. 5, pp. 507-513, 2010.

[12] O. Uzuner, B. R. South, S. Shen, and S. L. DuVall, "2010 i2b2/VA challenge on concepts, assertions, and relations in clinical text," *Journal of the American Medical Informatics Association*, vol. 18, no. 5, pp. 552-556, 2011.

[13] M. Neumann, D. King, I. Beltagy, and W. Ammar, "ScispaCy: Fast and Robust Models for Biomedical Natural Language Processing," in *Proceedings of the 18th BioNLP Workshop and Shared Task*, Florence, Italy, 2019, pp. 319-327.

[14] A. Vaswani *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems*, vol. 30, 2017.

[15] K. Singhal *et al.*, "Large language models encode clinical knowledge," *Nature*, vol. 620, no. 7972, pp. 172-180, 2023.

[16] L. M. Amugongo, P. Mascheroni, S. Brooks, S. Doering, and J. Seidel, "Retrieval augmented generation for large language models in healthcare: A systematic review," *PLOS Digital Health*, vol. 4, no. 6, p. e0000877, 2025.

[17] Y. Gao *et al.*, "Retrieval-Augmented Generation for Large Language Models: A Survey," *arXiv preprint arXiv:2312.10997*, 2023.

[18] L. Gao, X. Ma, J. Lin, and J. Callan, "Precise Zero-Shot Dense Retrieval without Relevance Labels," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*, 2023, pp. 1762-1777.

[19] D. Edge *et al.*, "From Local to Global: A Graph RAG Approach to Query-Focused Summarization," *arXiv preprint arXiv:2404.16130*, 2024.

[20] S. Es, J. James, L. Espinosa-Anke, and S. Schockaert, "RAGAS: Automated Evaluation of Retrieval Augmented Generation," *arXiv preprint arXiv:2309.15217*, 2023.

[21] N. Reimers and I. Gurevych, "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 2019, pp. 3982-3992.

[22] Y. Gu *et al.*, "Domain-Specific Language Model Pretraining for Biomedical Natural Language Processing," *ACM Transactions on Computing for Healthcare*, vol. 3, no. 1, pp. 1-23, 2021.

[23] J. Johnson, "Billion-scale similarity search with GPUs," *IEEE Transactions on Big Data*, vol. 7, no. 3, pp. 535-547, 2019.

[24] Y. Pan *et al.*, "A Survey on Vector Database Management Systems," *arXiv preprint arXiv:2310.14021*, 2023.

[25] M. Q. Stearns, C. Price, K. A. Spackman, and A. Y. Wang, "SNOMED clinical terms: overview of the development process and project status," *Proceedings of the AMIA Symposium*, pp. 662-666, 2001.

[26] S. M. Huff, R. A. Rocha, C. J. McDonald, G. J. De Moor, T. Fiers, and W. E. Hammond, "Development of the Logical Observation Identifier Names and Codes (LOINC) vocabulary," *Journal of the American Medical Informatics Association*, vol. 5, no. 3, pp. 276-292, 1998.

[27] Z. Kraljevic *et al.*, "Multi-domain Clinical Natural Language Processing with MedCAT: The Medical Concept Annotation Toolkit," *Artificial Intelligence in Medicine*, vol. 117, p. 102083, 2021.

[28] O. Elayan and M. Shukla, "A Review of Cloud-Based Natural Language Processing Services for Healthcare Applications," *Journal of Healthcare Informatics Research*, vol. 6, pp. 120-145, 2022.

[29] H. Eyre *et al.*, "Launching into clinical space with medspacy: a new clinical natural language processing toolkit in python," *AMIA Annual Symposium Proceedings*, vol. 2021, pp. 438-447, 2021.

[30] "Retrieval-Augmented Generation (RAG) in Healthcare: A Comprehensive Review," *MDPI*, Sep. 2025. [Online]. Available: https://www.mdpi.com. [Accessed: Jun. 15, 2026].

[31] "Multi-Agent Retrieval Augmented Generation for Clinical Decision Support: A Systematic Review and Integrative Conceptual Framework," *ResearchGate*, Apr. 2026.

[32] "Retrieval-Augmented Framework for LLM-Based Clinical Decision Support," *arXiv*, Oct. 2025.

[33] "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Std 830-1998*, 1998.

[34] "Private-RAG: A Privacy-Preserving Retrieval-Augmented Generation Method for Large Model Inference," *MDPI*, vol. 5, no. 1, pp. 1-18, Jun. 2026. [Online]. Available: https://www.mdpi.com. [Accessed: Jun. 15, 2026].

[35] V. Potta, "Backend-for-Frontend (BFF) Pattern for Front-End Scalability," *Medium*, Apr. 2025. [Online]. Available: https://medium.com. [Accessed: Jun. 15, 2026].

[36] R. Smith, "An Overview of the Tesseract OCR Engine," in *Proceedings of the Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, vol. 2, pp. 629-633, 2007.

[37] H. Liu *et al.*, "LangChain: The Rise of LLM-Based Applications," *arXiv preprint arXiv:2309.17420*, 2023.

[38] C. Zhang *et al.*, "Chunking Strategies for Retrieval-Augmented Generation in Clinical Decision Support," *Journal of Biomedical Informatics*, vol. 148, p. 104521, 2024.

[39] R. Nogueira and K. Cho, "Passage Re-ranking with BERT," *arXiv preprint arXiv:1901.04085*, 2019.

[40] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, 2nd ed. Addison-Wesley, 2005.

[41] M. Fowler, *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Addison-Wesley, 2004.

[42] A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 7th ed. McGraw-Hill, 2019.

[43] Yu. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, no. 4, pp. 824-836, 2020.

[44] S. Ramirez, "FastAPI: High performance, easy to learn, fast to code, ready for production," 2024. [Online]. Available: https://fastapi.tiangolo.com/. [Accessed: Jun. 15, 2026].

[45] Vercel, "Next.js by Vercel - The React Framework," 2024. [Online]. Available: https://nextjs.org/. [Accessed: Jun. 15, 2026].

[46] Prisma Data, "Prisma: Next-generation Node.js and TypeScript ORM," 2024. [Online]. Available: https://www.prisma.io/. [Accessed: Jun. 15, 2026].

[47] A. Chen *et al.*, "Synthetic Electronic Health Records Generated with Large Language Models: A New Paradigm for Clinical NLP Research," *Journal of Medical Internet Research*, vol. 27, p. e45123, 2025.

[48] D. Guinard and F. Wilde, *Building the Web of Things: With examples in Node.js and Raspberry Pi*. Manning Publications, 2016.

[49] M. Jones, "Stateless Authentication for Web Applications using JSON Web Tokens," *Journal of Web Security*, vol. 12, no. 3, pp. 45-58, 2023.

[50] O. Dernoncourt *et al.*, "De-identification of patient notes with recurrent neural networks," *Journal of the American Medical Informatics Association*, vol. 24, no. 3, pp. 596-606, 2017.

[51] S. Wang *et al.*, "Memory Management in Large Language Model based Conversational Agents," *arXiv preprint arXiv:2308.11432*, 2023.

[52] C. Murphy *et al.*, "Properties of Machine Learning Applications for Use in Automated Testing," *IEEE Transactions on Software Engineering*, vol. 34, no. 6, pp. 805-819, 2008.

[53] R. Miller, "Response Times: The 3 Important Limits," *Nielsen Norman Group*, 2014. [Online]. Available: https://www.nngroup.com/articles/response-times-3-important-limits/. [Accessed: Jun. 16, 2026].

[54] Y. Wu *et al.*, "Clinical abbreviation disambiguation using neural word embeddings," in *Proceedings of the BioNLP 2015 workshop*, 2015, pp. 171-176.

[55] E. J. Topol, "High-performance medicine: the convergence of human and artificial intelligence," *Nature Medicine*, vol. 25, no. 1, pp. 44-56, 2019.

الجمهورية الجزائرية الديمقراطية الشعبية

Republique Algérienne Démocratique Populaire
وزارة التعليم العالي والبحث العلمي

Ministère de l'Enseignement Supérieur de la Recherche Scientifique

Université 20 août 1955 Skikda — **جامعة 20 أوت 1955 – سكيكدة**

Faculte des Sciences
Departement d'informatique

كلية العلوم
قسم الإعلام الآلي

الرقم: .... / ق/ إ/ أ/ ك.ع/ 2026

Autorisation de Dépôt de Mémoire de Master

Je soussigné : **Adel Lhasasna**

Certifie que l'étudiant(e) : **Boucherbat mohamed elhacene**

Spécialité : **Master en artificial intelligence**

Ayant soutenu le projet intitulé : **Design and Implementation of a RAG-Based Medical Report Data Extractor API**

A apporté les corrections nécessaires sur son manuscrit de Master.

Signature de l'encadreur

Scanné avec CamScanner