Clinical RAG Chatbot for MIMIC-IV Dataset Analysis

AI-Powered Clinical Decision Support System

Ekenedirichukwu Iheanacho

Supervised by Luciano Gerber

Department of Computing and Mathematics

Faculty of Science and Engineering

Manchester Metropolitan University

August, 2025

A dissertation submitted in partial fulfilment of the requirements for the degree of M.Sc. in Artificial Intelligence.

First edition, August 25, 2025

Manchester Metropolitan University

Declaration

Academic Integrity Declaration

I hereby declare that this work is my own and has been written by me in its entirety. I have acknowledged all sources of information which have been used in the preparation of this report.

This report has not been submitted for assessment in any previous application for a degree or diploma or other qualification at this or any other university or institution.

I understand that if I am suspected of plagiarism or another form of academic misconduct, my work will be referred to the Academic Misconduct Panel for investigation.

I understand that Manchester Metropolitan University may retain copies of this work for educational and quality assurance purposes.

Faculty/School	Faculty of Science and Engineering
Degree	M.Sc. in Artificial Intelligence
Title	Clinical RAG Chatbot for MIMIC-IV Dataset Analysis
Student Name (Student ID)	Ekenedirichukwu Iheanacho (24800041)

Student Signature	______________________
Date	August 25, 2025

To My Parents

For all their sacrifices and belief in me from the day I decided to enter engineering as a computer and software engineer.

Acknowledgements

These are the acknowledgements. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Abstract

This is the abstract. Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Contents

List of Figures

List of Tables

List of Abbreviations

1

Introduction

1.1 | Motivation

Electronic Health Records (EHRs) contain structured tables and unstructured narratives that chronicle patient encounters across time. Clinicians and analysts routinely need targeted answers that span medications, labs, procedures, microbiology, and admin details. Traditional keyword search or isolated SQL queries struggle to synthesise evidence across modalities and context. Retrieval-Augmented Generation (RAG) has emerged as a practical approach: it retrieves relevant documents and asks a language model to compose grounded answers with citations. This project delivers a clinical RAG system aligned with MIMIC-IV style data, designed to be modular, reproducible, and usable through both an API and a simple web interface.

1.2 | Aims & Objectives

This project designs and implements a Retrieval-Augmented Generation (RAG) chat system for clinical settings. The aim is to develop and assess a chatbot that integrates a locally hosted Large Language Model (LLM) with external medical records (MIMIC-IV) so that responses are grounded in real patient data, improving factual accuracy and contextual relevance.
Objectives:

- Develop a RAG architecture that dynamically retrieves relevant clinical documents from the MIMIC-IV dataset during user interactions.
- Enhance factual correctness and minimise hallucinations by augmenting a local LLM with retrieved medical records.

- Implement multi-turn conversation capabilities that maintain context and memory across turns.
- Evaluate performance via a mix of manual review, logical reasoning, and quantitative metrics.

1.3 | System Overview

The system follows four layers that mirror practical deployment needs:

1. **Data Creation & Handling**: Parse clinical tables, normalise fields, segment text into coherent chunks, compute embeddings, and store vectors in FAISS.
2. **RAG Core**: Given a question (and optionally an admission identifier), retrieve top-k passages and generate grounded answers with supporting evidence and maintain conversational state.
3. **Models**: Host compact LLMs locally and switch between multiple embedding backbones and generators via configuration.
4. **Interfaces**: Provide a Flask REST API consumed by a React front-end for interactive exploration.

1.4 | Data and Ethical Considerations

To enable open demonstrations, the system includes support for synthetic, non-identifiable clinical data that mirrors realistic distributions without reproducing real patients. The main MIMIC-IV dataset is also de-identified and credentialed by PhysioNet. The software is intended for research and educational purposes only and is not a clinical decision support tool.

1.5 | Project Description

This system improves the factual accuracy and contextual reliability of answers from a local LLM by retrieving supporting evidence from MIMIC-IV at query time. Unlike traditional chatbots that depend on online APIs, the full stack runs locally to safeguard sensitive data. The pipeline covers data preparation, embedding and indexing, retrieval-augmented answer generation, and practical access via a Flask API and React front-end.

1.6 | Learning Outcomes

By completing this work, the following learning outcomes are achieved:

- Ability to implement RAG architectures that augment LLM responses with external knowledge.
- Experience integrating EHR-style datasets and knowledge bases with language models.
- Improved understanding of ethical and technical considerations for handling sensitive health data.
- Proficiency in evaluating LLM performance using established and project-specific metrics.
- Practical skills in building FAISS vector stores, hosting compact LLMs locally (Ollama), and exposing capabilities through web APIs and UIs.

1.7 | Methodology

1.7.1 | Data Extraction and Preparation

We use the `hosp` component of MIMIC-IV covering admissions, diagnoses, laboratory results, and medication administration records. Source CSV files are ingested into a local SQLite database for efficient joins and filtering. A subset of admissions is selected, and relevant patient-level fields are aggregated and formatted into concise, readable summaries to simulate unstructured clinical notes, enabling uniform downstream chunking.

1.7.2 | Embedding and Vector Store Construction

Using LangChain and lightweight offline embedding models (e.g., `all-MiniLM-L6-v2`, `e5-base-v2`, `biomedical` variants), each chunk is vectorised and stored in FAISS. We retain metadata such as `subject_id`, `hadm_id`, and timestamps to preserve traceability and allow admission-scoped retrieval. Chunking parameters (size, overlap) are tuned to balance recall and precision.

1.7.3 | LLM Setup with Ollama

A local Ollama server hosts compact models (default: Phi3), with alternatives such as Qwen3, Llama, Gemma, DeepSeek-R1 1.5, and TinyLlama available. Running inference locally ensures privacy, predictable latency, and full control over model selection and updates.

1.7.4 | Retrieval-Augmented Question Answering

A custom retriever returns top-k semantically similar chunks for a user query. Retrieved evidence is injected into prompts and passed to the LLM via a RAG chain, producing concise, citation-backed answers. The system supports common clinical questions (e.g., abnormal labs, medications during an admission, summaries of recent encounters) and uses entity extraction to auto-parameterise queries (e.g., detect admission identifiers).

1.7.5 | Evaluation and Iteration

We compare RAG-augmented outputs to a standalone LLM baseline. Quantitative signals include average answer score, pass rate, retrieval latency, and topic-wise breakdowns; qualitative review examines fidelity to evidence and clinical style. Iterative improvements consider richer metadata, chunking strategies, re-prompting, and model choices.

1.8 | Evaluation Plan

We will:

- Establish a clinical QA set spanning admissions, labs, microbiology, procedures, and prescriptions.
- Track quality metrics (average score, pass rate), latency proxies (search time), and safety indicators (hallucination/disclaimer rates where applicable).
- Conduct targeted manual review to validate correctness, calibration, and citation fidelity.

1.9 | Scope and Assumptions

We prioritise transparent, locally runnable components and retrieval-centric orchestration over large proprietary models. Dense bi-encoder retrieval with FAISS is used throughout; more advanced re-ranking or specialised clinical generators are out of scope for this iteration. Prompts favour concise, evidence-backed clinical style.

1.10 | Document Structure

The remainder of this document presents related background and prior art, the detailed system design and implementation, empirical evaluation and analyses, a discussion of implications and limitations, and concluding remarks with directions for future work.

2

Background & Literature Overview

2.1 | Literature Review

Retrieval-Augmented Generation (RAG) systems are emerging as a pivotal advancement in the field of Artificial Intelligence (AI), particularly within the clinical domain. These systems integrate Large Language Models (LLMs) with external knowledge sources to produce more accurate, contextually relevant, and reliable responses, addressing critical limitations inherent in standalone LLMs. The high-stakes nature of healthcare necessitates precise, up-to-date, and verifiable information, making RAG a particularly valuable tool for improving diagnostic accuracy, clinical decision support, and patient care.

This literature review comprehensively explores the application of RAG within clinical settings, building upon existing foundational work and considering various RAG methodologies, datasets, and ethical implications.

2.1.1 | Evolution of Language Models Leading to RAG

The development of language models represents a continuous evolution toward more sophisticated natural language understanding and generation capabilities, ultimately culminating in retrieval-augmented approaches that address fundamental limitations of standalone generative models.

2.1.1.1 | Early Statistical Approaches (1990s-2000s)

The foundation of computational language modeling began with **Statistical Language Models (SLMs)**, primarily n-gram models that emerged in the 1990s ((**?**)). These models used probabilistic statistics to predict word sequences based on local context windows, with trigram and 4-gram models becoming standard for speech recognition and machine translation. While computationally efficient, n-gram models suffered from the curse of dimensionality and could not capture long-range semantic dependencies ((**?**)).

2.1.1.2 | Neural Language Models (2000s-2010s)

The introduction of **Neural Language Models (NLMs)** marked a paradigm shift toward distributed representations ((**?**)). These models employed feedforward neural networks to learn continuous word embeddings, enabling better generalization through semantic similarity. Word2Vec ((**?**)) and GloVe ((**?**)) revolutionized word representation learning, while neural language models like those proposed by Bengio et al. demonstrated superior perplexity scores compared to n-gram baselines.

2.1.1.3 | Sequence Models and Recurrent Architectures (2010s)

Further progress led to **Recurrent Neural Networks (RNNs)** which could process variable-length sequences and maintain hidden states across time steps ((**?**)). **Long Short-Term Memory (LSTM)** networks ((**?**)) and **Gated Recurrent Units (GRUs)** ((**?**)) addressed the vanishing gradient problem, enabling modeling of longer sequences. These architectures became state-of-the-art for sequence modeling and transduction tasks, with bidirectional variants showing particular success in language understanding ((**?**)). However, sequential processing limited parallelization and created bottlenecks for long-range dependency modeling.

2.1.1.4 | Attention Mechanisms and Transformer Revolution (2014-2017)

The introduction of **attention mechanisms** ((**?**)) allowed models to focus on relevant parts of input sequences, initially improving neural machine translation. The breakthrough came with the **Transformer architecture** in 2017, as detailed in "Attention Is All You Need" ((**?**)). This architecture fundamentally changed language modeling by replacing recurrence with *self-attention mechanisms*, enabling highly parallel training and more efficient modeling of long-range dependencies. The multi-head attention mechanism allowed models to attend to information from different representation subspaces simultaneously, dramatically improving performance across natural language tasks.

2.1.1.5 | Pre-trained Language Models Era (2018-2019)

The Transformer became the backbone for **Pre-trained Language Models (PLMs)** that revolutionized NLP through transfer learning. **BERT** (Bidirectional Encoder Representations from Transformers) ((**?**)) introduced bidirectional pre-training on masked language modeling, achieving state-of-the-art results on GLUE benchmarks. Concurrently, **OpenAI GPT** ((**?**)) demonstrated the power of autoregressive language modeling for text generation. These models established the pre-train-then-fine-tune paradigm that dominated NLP for several years, with variants like RoBERTa ((**?**)), ALBERT ((**?**)), and T5 ((**?**)) further pushing performance boundaries.

2.1.1.6 | Large Language Models and Emergent Capabilities (2020-present)

Large Language Models (LLMs) represent a quantum leap in scale and capability, incorporating massive datasets, computation, and refined architectures. GPT-3's 175 billion parameters ((**?**)) demonstrated remarkable zero-shot and few-shot learning capabilities, while subsequent models like GPT-4, PaLM ((**?**)), and LLaMA ((**?**)) showed emergent abilities in reasoning, code generation, and complex task completion. These models exhibit **scaling laws** ((**?**)) where performance improves predictably with increased compute, data, and parameters.

The growth of LLMs has been propelled by several factors: massive text corpora from the internet, specialized hardware (GPUs/TPUs) enabling parallel computation, algorithmic innovations like efficient attention mechanisms, and substantial computational resources. Models now demonstrate impressive capabilities across diverse domains, including mathematical reasoning, creative writing, and code synthesis.

2.1.1.7 | Limitations Driving RAG Development

Despite their impressive capabilities, LLMs suffer from fundamental limitations that become particularly problematic in specialized domains like healthcare ((**?**)):

Hallucinations and Factual Accuracy: LLMs can generate plausible-sounding but factually incorrect information, with hallucination rates varying from 3-27% depending on the task and model ((**?**)). In clinical contexts, such inaccuracies can have life-threatening consequences.

Knowledge Cutoffs and Staleness: LLMs are trained on static datasets with knowledge cutoffs, making them unable to access recent information. Medical knowledge evolves rapidly, with new treatments, drug approvals, and clinical guidelines emerging continuously ((**?**)).

Expensive Knowledge Updates: Incorporating new knowledge requires costly retraining of entire models, making frequent updates economically infeasible for most organizations ((**?**)).

Lack of Source Attribution: LLMs cannot provide citations or evidence for their claims, undermining trust and preventing verification of generated content ((**?**)).

Domain-Specific Knowledge Gaps: General-purpose LLMs may lack deep expertise in specialized domains, performing poorly on technical tasks requiring domain-specific reasoning ((**?**)).

These limitations are especially concerning in medicine, where accuracy, recency, and verifiability are paramount. The need for **grounded, attributable, and updatable** language models motivated the emergence of Retrieval-Augmented Generation as a promising solution that combines the generative capabilities of LLMs with the precision and currency of external knowledge sources.

2.1.2 | Emergence and Mechanics of Retrieval-Augmented Generation (RAG)

RAG addresses LLM limitations by combining strong generative capabilities with **external memory retrieval**. Early work, such as REALM integrated retrieval into pre-training ((**?**)). Lewis et al. (**?**) introduced RAG, pairing a neural retriever with a seq2seq generator: top-k passages are retrieved and used to ground output. Variants such as Fusion-in-Decoder ((**?**)) and RETRO ((**?**)) expanded retrieval to larger corpora. Frameworks like LangChain ((**?**)) and LlamaIndex ((**?**)) operationalised RAG pipelines.

The typical RAG pipeline has three steps:

- **Indexing:** Documents segmented into chunks, embedded with encoders, and stored in vector databases (e.g., FAISS, Pinecone, Weaviate).
- **Retrieval:** Queries embedded, top-k chunks retrieved via similarity search.
- **Generation:** Query and retrieved chunks passed to the LLM for grounded response.

2.1.2.1 | Vector Database Technologies

Vector databases are a critical infrastructure for RAG systems, with different solutions offering distinct advantages for clinical deployments. **FAISS** (Facebook AI Similarity Search) provides efficient in-memory similarity search with GPU acceleration, making it suitable for research environments ((**?**)). **Pinecone** offers managed cloud-native vector search with real-time updates and hybrid search capabilities ((**?**)). **Weaviate** combines vector search with GraphQL APIs and supports multimodal embeddings ((**?**)). **Redis** with vector similarity search extensions provides low-latency retrieval suitable for real-time clinical applications ((**?**)). **ChromaDB** offers lightweight, open-source vector storage with built-in embedding functions ((**?**)). For clinical RAG systems, database choice depends on latency requirements, data privacy constraints, and scalability needs.

2.1.2.2 | Clinical Embedding Models

The choice of embedding model significantly impacts RAG performance in clinical contexts. **BioBERT** ((**?**)) and **ClinicalBERT** ((**?**)) are domain-specific transformers pre-trained on biomedical literature and clinical notes respectively, showing superior performance on medical NER and relation extraction. **PubMedBERT** ((**?**)) is trained exclusively on PubMed abstracts, achieving state-of-the-art results on biomedical language understanding tasks. General-purpose models like **all-MiniLM-L6-v2** and **multi-qa-mpnet-base-dot-v1** ((**?**)) offer computational efficiency but may lack clinical domain specificity. **SapBERT** ((**?**)) specializes in biomedical entity representation, while **BlueBERT** ((**?**)) combines clinical and biomedical pre-training. Recent work on **clinical sentence transformers** fine-tuned on medical question-answering pairs shows promise for improving retrieval relevance in clinical RAG systems ((**?**)).

This allows factuality without retraining, making RAG especially attractive in healthcare.

2.1.3 | Clinical RAG System Implementations

2.1.3.1 | Clinical Question-Answering Systems

Several production-ready RAG systems demonstrate practical clinical applications. **Med-PaLM 2** integrates retrieval over medical literature with PaLM 2 for clinical reasoning, achieving 86.5% accuracy on MedQA compared to 67.6% without retrieval ((**?**)). Google's **AMIE** (Articulate Medical Intelligence Explorer) combines conversational AI with medical knowledge retrieval for diagnostic dialogues, showing superior performance to primary care physicians in simulated consultations ((**?**)). **ChatDoctor** finetunes LLaMA on medical conversations and integrates retrieval from medical databases, demonstrating improved clinical reasoning on patient scenarios ((**?**)).

2.1.3.2 | Specialized Medical Domain Applications

Domain-specific RAG implementations show significant clinical utility. **HuatuoGPT** combines medical knowledge graphs with retrieval-augmented generation for Traditional Chinese Medicine, outperforming general LLMs on medical licensing exams ((**?**)). **BioMedLM-RAG** integrates PubMed literature retrieval with biomedical language models for drug discovery and clinical research, showing 23% improvement in biomedical QA tasks ((**?**)). **ClinicalT5** employs retrieval-augmented pre-training on MIMIC-III clinical notes, achieving state-of-the-art performance on clinical outcome prediction and medication recommendation ((**?**)).

2.1.3.3 | Multi-Modal Clinical RAG Systems

Recent developments extend RAG to multi-modal clinical data. **Med-Flamingo** integrates visual and textual retrieval for medical imaging analysis, combining radiology reports with image embeddings to improve diagnostic accuracy ((**?**)). **LLaVA-Med** demonstrates medical visual question answering by retrieving relevant medical images and text simultaneously ((**?**)). **CheXagent** specifically targets chest X-ray interpretation using retrieval over radiology databases, achieving radiologist-level performance on pneumonia detection ((**?**)).

2.1.3.4 | Knowledge Graph-Enhanced RAG

Advanced clinical systems integrate structured medical knowledge. **KG-RAG-Med** combines UMLS knowledge graphs with vector retrieval for medical entity disambiguation and relation extraction ((**?**)). **MedRAG** implements diagnostic reasoning by retrieving from both textual corpora and medical ontologies, showing 15% improvement in differential diagnosis accuracy ((**?**)). **BioKGRAG** integrates biological pathway databases with literature retrieval for drug-disease interaction prediction ((**?**)).

2.1.3.5 | Real-World Clinical Deployment

Several RAG systems have moved beyond research to clinical deployment. **Microsoft Healthcare Bot** integrates Azure Cognitive Search with GPT models for patient triage and clinical decision support across multiple health systems ((**?**)). **Epic's clinical assistant** combines EHR data retrieval with LLMs for clinical documentation and diagnosis suggestions, deployed across 200+ health systems ((**?**)). **IBM Watson for Oncology** uses retrieval-augmented generation over oncology literature and treatment guidelines, though with mixed clinical adoption results ((**?**)).

2.1.4 | Datasets in Clinical RAG Systems

RAG relies on curated datasets. Frequently used corpora include PubMed, UMLS, MedDialog ((**?**)), MedDG ((**?**)), and imaging-text datasets such as MIMIC-CXR ((**?**)). Benchmarks include BioASQ, MedMCQA, PubMedQA ((**?**)), MedQA ((**?**)), MultiMedQA ((**?**)), ClinicalQA ((**?**)), and MIRAGE ((**?**)). Synthetic datasets such as DDXPlus ((**?**)) and CPDD ((**?**)) test diagnostic QA.

Most datasets are English-only, limiting multilingual evaluation. This bias restricts fairness and highlights the need for broader language coverage.

2.1.5 | Ethical Considerations and Safety Challenges

Clinical RAG systems present significant ethical challenges that require careful consideration before deployment in healthcare settings. The high-stakes nature of medical decision-making amplifies the potential consequences of AI system failures.

2.1.5.1 | Hallucinations and Clinical Safety

LLM hallucinations pose severe risks in clinical contexts where inaccurate information can directly harm patients. Studies show that even advanced models like GPT-4 exhibit hallucination rates of 8-15% on medical tasks ((**?**)). RAG systems can reduce but not eliminate hallucinations, with retrieval-augmented models showing 3-7% residual hallucination rates ((**?**)). Clinical deployment requires robust hallucination detection mechanisms and human oversight protocols ((**?**)). The FDA has issued guidance emphasizing the need for continuous monitoring of AI-generated clinical content to prevent patient harm ((**?**)).

2.1.5.2 | Privacy and Data Protection

Healthcare RAG systems handle sensitive patient data requiring stringent privacy protections. HIPAA in the US and GDPR in Europe mandate specific safeguards for protected health information (PHI) ((**?**)). Patient datasets like MIMIC-IV require extensive de-identification, yet re-identification risks persist through inference attacks ((**?**)). Federated learning approaches show promise for privacy-preserving RAG training ((**?**)), while differential privacy techniques can protect individual patient records during retrieval ((**?**)). Cloud-based RAG deployments face additional challenges regarding data sovereignty and cross-border data transfer regulations ((**?**)).

2.1.5.3 | Algorithmic Bias and Health Equity

Clinical RAG systems can perpetuate and amplify existing healthcare disparities. Training data often underrepresents minority populations, leading to biased retrieval and generation ((**?**)). Studies demonstrate that medical AI systems show reduced accuracy for Black patients compared to White patients across multiple clinical tasks ((**?**)). RAG systems can inherit biases from both training corpora and retrieval databases, potentially exacerbating health inequities ((**?**)). Mitigation strategies include diverse training data curation, bias-aware retrieval algorithms, and continuous fairness monitoring across demographic groups ((**?**)).

2.1.5.4 | Informed Consent and Patient Autonomy

The use of AI-generated clinical recommendations raises complex questions about informed consent and patient autonomy. Patients have the right to understand how their care decisions are influenced by AI systems ((**?**)). Clinical RAG deployments must ensure transparent communication about AI involvement in diagnosis and treatment recommendations. The "black box" nature of LLMs conflicts with patient autonomy principles, requiring explainable AI approaches that provide clear rationales for clinical suggestions ((**?**)). Legal frameworks are evolving to address liability questions when AI systems contribute to medical errors ((**?**)).

2.1.5.5 | Professional Liability and Accountability

RAG systems in clinical settings create complex accountability challenges when errors occur. Legal frameworks struggle to assign responsibility between AI developers, healthcare institutions, and individual clinicians ((**?**)). Medical malpractice law requires adaptation to address AI-assisted clinical decisions, with ongoing debates about standard of care when AI tools are available ((**?**)). Professional medical associations are developing guidelines for AI accountability, emphasizing that clinical responsibility ultimately remains with licensed practitioners ((**?**)).

2.1.5.6 | Regulatory Compliance and Validation

Clinical RAG systems must meet stringent regulatory requirements before deployment. The FDA's AI/ML-based Software as Medical Device framework requires extensive clinical validation and post-market surveillance ((**?**)). European Medical Device Regulation (MDR) imposes additional requirements for AI transparency and clinical evidence ((**?**)). RAG systems face particular challenges in demonstrating consistent performance across diverse patient populations and clinical scenarios ((**?**)). Continuous learning systems require novel regulatory approaches to handle model updates while maintaining safety assurance ((**?**)).

2.1.5.7 | Clinical Workflow Integration

Ethical RAG deployment requires careful integration into existing clinical workflows to avoid cognitive burden and decision-making errors. Alert fatigue from AI systems can reduce clinician attention to genuine safety concerns ((**?**)). RAG systems must balance information provision with cognitive load management, presenting relevant retrievals without overwhelming clinical users ((**?**)). Human-AI collaboration frameworks emphasize complementary intelligence rather than replacement, preserving clinical expertise while augmenting decision-making capabilities ((**?**)).

These ethical considerations necessitate comprehensive frameworks for responsible clinical RAG deployment, including technical safeguards, regulatory compliance, and ongoing monitoring of system performance and societal impact.

2.1.6 | Beyond RAG: Future Directions for Clinical LLMs

Looking past classical RAG, several trajectories are especially promising for clinical deployment:

Domain-specific foundation models. Open and proprietary medical LLMs fine-tuned on biomedical corpora show consistent gains over general models on exam-style and clinician-judged tasks, suggesting a path to safer, more aligned clinical assistants. Examples include Med-PaLM 2, which achieved expert-level responses on multiple medical benchmarks and was preferred by clinicians on most axes ((**?**)); MEDITRON-70B, which adapts LLaMA-2 via large-scale medical pretraining ((**?**)); and GatorTron, trained on $>$82B tokens of de-identified clinical text to advance core clinical NLP tasks ((**?**)).

Diagnostic dialogue and longitudinal reasoning. Beyond single-turn QA, systems optimised for history taking and iterative differential diagnosis (DDx) indicate that LLMs can support the diagnostic process itself. Google's AMIE reports clinically preferred diagnostic dialogues and improved DDx support in controlled studies ((**??**)). Embedding such models into triage, clerking, and MDT workflows is a natural next step.

Multimodal clinical models (text + imaging + signals). Vision language models specialised for medicine (e.g. LLaVA-Med and Med-Flamingo) demonstrate open-ended reasoning over biomedical figures and radiology, with clinical-rated gains in medical VQA and progress in report generation ((**???**)). Extending RAG to *multimodal RAG*, retrieving not just passages but also linked images and structured findings can provide better ground answers in PACS and reporting systems.

Graph-augmented and structure-aware retrieval. New pipelines like GraphRAG construct and query knowledge graphs over private corpora, improving recall and reasoning on narrative clinical data ((**??**)). In healthcare, combining ontology-backed graphs (e.g., UMLS) with free-text retrieval may mitigate retrieval blind spots and support multi-hop guideline reasoning.

Agentic workflows and tool use. Multi-agent pipelines that plan, retrieve, verify and cite, rather than single-pass prompting, show accuracy gains in open-domain tasks and are well-suited to safety-critical settings (e.g., agent plans for guideline lookup, DDx cross-checks, and citation verification) ((**?**)). Clinically, agents can orchestrate EHR queries (SQL, FHIR), calculator tools (risk scores) and literature checks before drafting notes.

Tight EHR integration and structured data grounding. Future systems should natively query structured sources (e.g. MIMIC-IV–like schemas, FHIR servers) to ground generation in vitals, labs and medications, reducing hallucinations and enabling patient-specific recommendations ((**?**)). Coupling retrieval over notes with programmatic reads of structured fields is a practical near-term target.

Privacy-preserving and efficient adaptation. Given PHI constraints, parameter-efficient finetuning (e.g. LoRA/PEFT), federated or on-prem training, and auditable data pipelines are essential. Inference on the device and edge using distilled or quantised medical LLM could enable bedside use when the network or data sharing is restricted ((**?**)).

Provenance, auditing and safety guardrails. Beyond point-in-time citations, future systems should expose evidence provenance graphs, uncertainty estimates, and contradiction checks against guidelines, with human-in-the-loop review. Emerging evaluations highlight that even strong medical LLMs have failure modes and access limitations that require robust oversight ((**?**)).

Overall, advances in domain-specialised models, multimodal grounding, graph-aware retrieval, agentic verification, and EHR-native tool use provide a concrete roadmap for clinical AI systems that go beyond classical RAG while retaining traceability and clinician control.

In conclusion, RAG augments LLMs to deliver more accurate, grounded, and clinically useful results. Although challenges remain, particularly around ethics, privacy, and data set bias, RAG represents a crucial pathway toward trustworthy AI in healthcare.

3

Materials & Methods

3.1 | Process overview

This chapter describes the materials, datasets, software, and methods used to design, build, and evaluate the clinical Retrieval-Augmented Generation (RAG) system. It documents data sources and governance, preprocessing and chunking procedures, embedding and indexing choices, the retrieval and LLM orchestration pipeline, interfaces for access (API, frontend, CLI), and the evaluation protocol.

The primary challenge in this work is connecting static hospital records with a system that can engage in conversations about patient care. Traditional medical information systems use keyword searches or database queries, which cannot capture the complex, context-dependent nature of medical thinking. This project creates a new way to interact with clinical data by building a pipeline that turns raw hospital records into a smart conversational system that can answer medical questions with evidence-based responses.

The approach covers the full process from getting data to deploying the system: we start with structured hospital records and convert them step-by-step into searchable knowledge that understands meaning. This knowledge is then managed by a smart search system that understands medical context, limits searches to the right patient admissions or medical areas, and puts together focused evidence. Finally, a language model running locally combines this evidence into clear, well-cited responses while staying strictly grounded in facts to prevent medical errors.

This complete approach allows natural language conversations with complex medical data while keeping the accuracy, transparency, and safety needed for healthcare uses.

3.2 | System Overview

This design follows a clear separation of responsibilities: a data creation and processing layer that normalises and chunks raw EHR tables into searchable documents; an embedding and vector store layer that encodes and indexes those chunks for semantic retrieval; a lightweight coordination layer that constrains retrieval by admission/section and prepares compact evidence contexts; and a model/serving layer that hosts small, locally runnable LLMs and exposes user-facing APIs and a web UI. This structure prioritises traceability, privacy, and reproducibility while keeping runtime costs and latency manageable for local experiments.

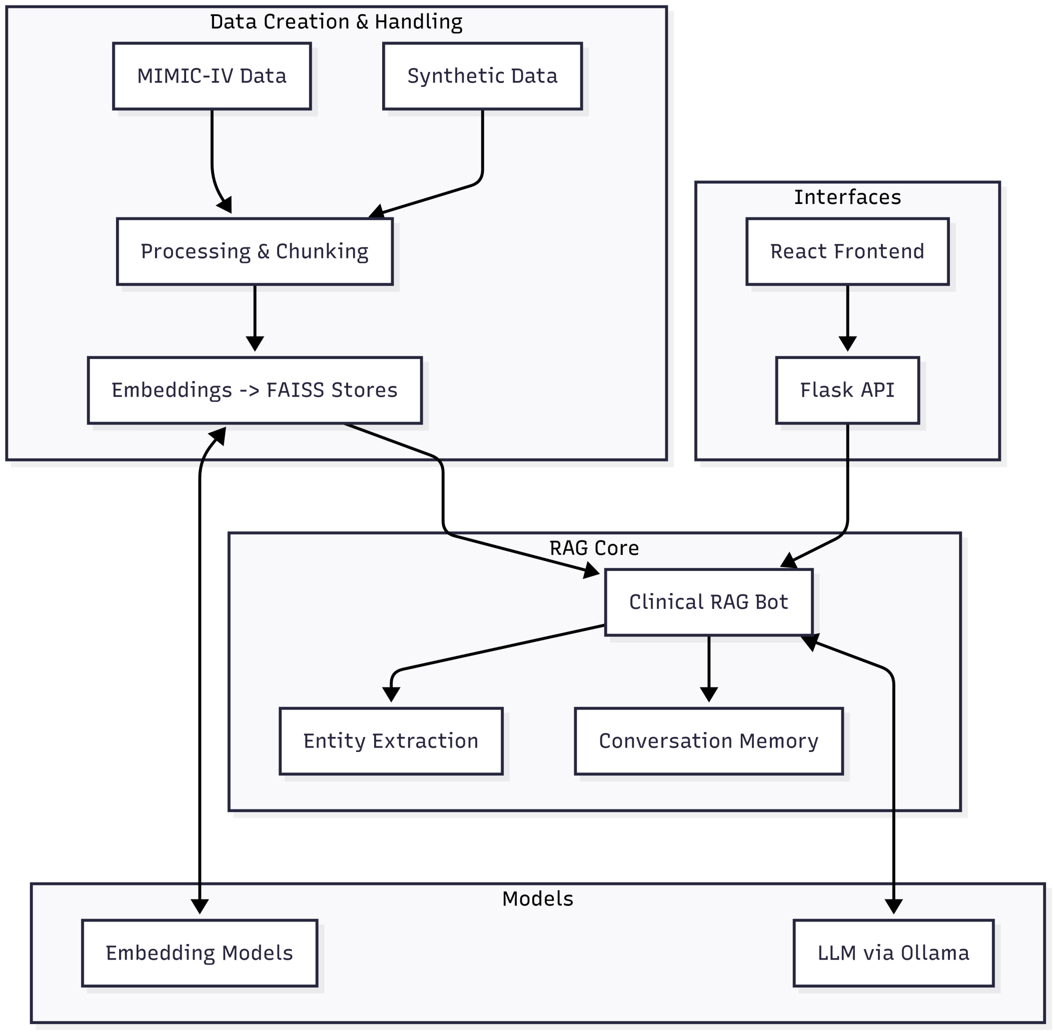

Figure 3.1: System Architecture

3.3 | Data Sources and Governance

3.3.1 | Datasets

- Real data: MIMIC-IV sample exports are stored under `mimic_sample_1000/`.
- Synthetic data: A generator in `synthetic_data/` produces structurally consistent, fictional clinical records for development/demo when real data are unavailable.

A data provider abstraction (`RAG_chat_pipeline/utils/data_provider.py`) automatically chooses real vs. synthetic sources, preferring real MIMIC-derived exports when present.

3.3.2 | Ethics and Access

Use of MIMIC-IV requires credentialed access and adherence to PhysioNet data use agreements. Public artefacts in this project are restricted to code and synthetic data. The system is intended for research/education purposes only and should not be used for clinical decision-making.

3.4 | Software, OS, and Runtime Environment

Experiments were executed on Microsoft Windows (user environment), Python 3.11, and React.js for the frontend.

Representative versions:

- Python: 3.11
- Node.js: $\geq$ 16 (v18+ recommended)
- LangChain: 0.3.25; `langchain-community`: 0.3.24
- FAISS CPU: 1.11.0; sentence-transformers: 4.1.0
- Torch: 2.7.1; Transformers: 4.52.4
- Flask: 3.0.2; Flask-CORS: 4.0.1
- Ollama: 0.5.1 (models pulled locally)

3.5 | System Configuration

Central configuration is defined in "`RAG_chat_pipeline/config/config.py`". Embedding model nicknames map to HuggingFace model IDs and vector-store directories. The LLM model set is managed via Ollama. Unless otherwise specified, the session-level defaults set by `set_models()` select `all-MiniLM-L6-v2` embeddings and `qwen3:1.7b` as the LLM.

Alternative combinations are supported for evaluation: `S-PubMedBert-MS-MARCO`, `multi-qa-mpnet-base-cos-v1`, `BiomedNLP-PubMedBERT`, `e5-base-v2`, `BioLORD-2023-C`, `BioBERT`, `S-PubMedBert-MedQuAD`; and LLMs such as Deepseek-r1, Llama 3.2, Gemma 2B, Phi-3, TinyLlama).

3.6 | Data Processing and Document Construction

3.6.1 | Preprocessing

We prepared data with notebooks in `data_handling/`. Sampling was performed at the admission level to produce a working subset (`mimic_sample_1000/`) by selecting a fixed-size random sample of `hadm_id`s and joining the primary tables (diagnoses, procedures, labs, microbiology, prescriptions) on `hadm_id` and `subject_id`. This preserves cross-table coherence while controlling index size and memory footprint.
Key preprocessing steps:

1. Normalise column names and types (e.g., enforce integer `hadm_id`, `subject_id`)
2. Map each row to a semantically labeled section: `header`, `diagnoses`, `procedures`, `labs`, `microbiology`, `prescriptions`
3. Compose section-scoped textual records embedding key fields (codes, values/units, dates) and attach metadata.

3.6.2 | Chunking

Documents are segmented into meaningful chunks optimised for clinical QA. We apply section-aware chunking before size-based splitting so that each chunk stays on the same topic (e.g., lab results grouped, diagnoses grouped). Chunk sizes of 600–900 characters with 80–150 character overlap worked well in practice for our lightweight local LLMs (balancing recall and context-window constraints). Metadata (`hadm_id`, `subject_id`, `section`) is preserved on every chunk to enable admission- and section-scoped retrieval.

An illustrative chunking routine:

```
from langchain_text_splitters import RecursiveCharacterTextSplitter

SECTION_SEPARATORS = ["\n\n", "\n", ". ", " "]

def chunk_clinical_text(text: str, chunk_size=800, chunk_overlap=120):
    splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap,
        separators=SECTION_SEPARATORS,
        is_separator_regex=False,
    )
    return splitter.split_text(text)
```

The notebooks emit a list of `langchain.schema.Document` with `page_content` and `metadata`, saved to `mimic_sample_1000/chunked_docs.pkl`. These chunks are the source for vector indexing.

3.7 | Embeddings and Vector Stores

3.7.1 | Model Setup

The embedding manager (`RAG_chat_pipeline/core/embeddings_manager.py`) loads a SentenceTransformers model by nickname from config, caching it under `models/<model-name>/`. If not present locally, it is downloaded and saved. A `HuggingFaceEmbeddings` wrapper provides the LangChain interface.

3.7.2 | FAISS Indexing

For each embedding configuration, a FAISS vector store is created from the chunked documents and saved under `vector_stores/<store-name>/`. When the system starts, it first tries to load the existing store. If the store is not there, it creates a new one and saves both the index and the chunked corpus.

3.8 | RAG Pipeline and Inference

3.8.1 | Custom Retrieval in brief steps

1) Parse identifiers/section from the query and recent history (regex).
2) Build a candidate set using in-memory indices if hints exist; otherwise, consider the full corpus.
3) Rank candidates with FAISS vector similarity.
4) Keep top-k (default $k = 5$).
5) Compact into a single structured, section-aware context (limit lines per section; prefer ICD patterns, dosages, and numeric labs).
6) Generate with a prompt that enforces grounding and citations.
Here is a Pseudocode:

```
   smallids = parse_hints(query, chat_history)  # hadm_id, subject_id, section
candidates = build_candidates(ids)  # from in-memory indices or full
 ↪  corpus
ranked = faiss_rank(candidates, query)
topk = ranked[: min(k, 5)]
context = make_structured_context(topk)  # section-aware, line-capped
answer = llm_answer(query, context, prompt=CLINICAL_PROMPT)
return answer
```

3.8.2 | Retriever

Given a query, the pipeline first attempts metadata-constrained retrieval when a `hadm_id`, `subject_id`, or `section` is available. Efficient in-memory indices (built at bot initialisation) map identifiers and sections to candidate document IDs, reducing the search space before semantic ranking. Otherwise, a global FAISS similarity search is performed.

At initialisation, the bot builds light-weight Python indices for fast filtering:

```
from collections import defaultdict

self.hadm_id_index = defaultdict(list)
self.subject_id_index = defaultdict(list)
self.section_index = defaultdict(list)
self.hadm_section_index = defaultdict(list)

for i, doc in enumerate(self.chunked_docs):
    hadm = doc.metadata.get("hadm_id")
    sec = str(doc.metadata.get("section", "")).lower()
    if hadm is not None:
        try:
            hadm_i = int(hadm)
            self.hadm_id_index[hadm_i].append(i)
            if sec:
                self.hadm_section_index[(hadm_i, sec)].append(i)
        except ValueError:
            pass
    if sec:
        self.section_index[sec].append(i)
```

When a `hadm_id` or `section` is present, only the corresponding candidate set is ranked semantically.

3.8.3 | Context Construction

Top-k (default $k = 5$, bounded for performance) documents are semantically ranked. For efficiency and to improve answer formatting, the system extracts concise, section-aware snippets (diagnoses, procedures, labs, prescriptions, microbiology, header) and composes a single structured context document passed to the LLM. Rules favour lines with section-specific keywords and patterns (e.g., ICD codes, dosages, numeric lab values/units) while limiting per-section lines to keep within context windows.

3.8.4 | LLM Answering

Answers are generated using an Ollama-hosted model through LangChain's `create_stuff_documents` with a prompt that enforces the guardrails below.
Prompt guardrails:

- Cite sources inline and include codes/values/units/dates when present
- If evidence is missing, say so; do not guess
- Respect admission/section scope when provided
- Always append the standard disclaimer

3.8.5 | Entity Extraction and Conversational Context

A deterministic regex-based extractor (`helper/entity_extraction.py`) identifies `hadm_id`, `subject_id`, and `section` hints from the current query and recent chat messages. For follow-ups, an LLM-powered rephrasing step condenses the question into a standalone form while preserving identifiers. Chat history is truncated to a configurable maximum (**60 messages**; see `MAX_CHAT_HISTORY` in config) to control context length.

3.8.6 | RAG Core Emphasis and Design Evolution

The initial blueprint used a straightforward vector index + chat engine with strict context mode:

```
from llama_index.core import VectorStoreIndex, SimpleDirectoryReader
from llama_index.embeddings.ollama import OllamaEmbedding
from llama_index.llms.ollama import Ollama

docs = SimpleDirectoryReader("./parsed_emails").load_data()
embed_model = OllamaEmbedding(model_name="nomic-embed-text")
llm = Ollama(model="llama3.2")

index = VectorStoreIndex.from_documents(docs, embed_model=embed_model)
chat_engine = index.as_chat_engine(
    llm=llm,
    chat_mode="context",
    verbose=True
)
response = chat_engine.chat("There is package mentioned for spaghetti code")
```

We attempted a two-step pipeline (LLM judges chunk relevance, then answers), but lightweight local models had small context windows and were unreliable as rankers. The reliable fix was a **custom retriever**: extract `hadm_id/subject_id/section` via regex/patterns, filter candidates with in-memory indices, reduce content by meaning, then pass only the top snippets to the LLM. The core search is:

```
candidate = self._filter_candidate_documents(hadm_id, subject_id, section, limit=20)
if candidate is None:
    retrieved = self.vectorstore.similarity_search(question, k=min(k, 20))
else:
    retrieved = self._semantic_search_on_docs(candidate, question, k=min(k, 5))
structured = self._extract_clinical_content(retrieved)
prompt = self._create_clinical_prompt(hadm_id, subject_id)
chain = create_stuff_documents_chain(self.llm, prompt)
answer = safe_llm_invoke(chain, {"input": question, "context":
↪  [Document(page_content=structured)]})
```

3.9 | Interfaces

3.9.1 | API

A Flask service in `api/app.py` exposes endpoints:

- `POST /api/chat`: process a chat message and optional history
- `GET /api/models`: list available embedding models and vector stores
- Static serving: production build of the React app

Chat endpoint contract (summary):

- Request: {`query: str`, optional `hadm_id`, `subject_id`, `section`, optional `history: [{role, content}]` }
- Response: {`answer: str`, `citations: [{doc_id, section}]`, `diagnostics: {mode, k}`}

3.9.2 | Frontend

A React UI (`frontend/`) provides a chat interface, model introspection, and sample query suggestions sourced from the data provider.

3.9.3 | CLI

`cli_chat.py` offers an interactive console with session logging and history management.

3.10 | Challenges and Mitigations

extbfRisk: medical hallucinations and sensitive data. Early versions retrieved globally and over-supplied context, which could produce made-up details. Mitigations:

- Admission-/section-scoped retrieval via in-memory indices
- Deterministic entity extraction (regex) to avoid LLM-based unexpected settings changes
- Structured snippet extraction with section rules
- Post-processing to enforce disclaimers and fix citations

Replacing the two-step LLM-as-ranker approach with the custom retriever + semantic re-ranking significantly improved factual grounding on lightweight local models.
Failure modes and fallbacks:

- Missing hints (no IDs/section) → use global search
- Empty candidate set → fall back to global k-NN
- FAISS store missing at startup → rebuild once, then cache
- Timeout or partial retrieval → return a safe message without guessing

3.11 | Evaluation Protocol

We provide only a brief overview here; detailed metrics and comparisons are in Results. The evaluator generates category-specific gold questions and computes pass/fail and summary statistics per model combination. We validate against structured signals (ICD patterns, dosages, units) and measure retrieval latency; full scoring details are in Results.

3.11.1 | Gold Questions and Categories

Gold questions are created from available records (diagnoses, procedures, labs, etc.) with associated `hadm_ids` when applicable. Full evaluation outcomes are reported in Results.

3.12 | Modularity and Extensibility

The system is modular: configuration (`config/`), core RAG components (`core/`), helpers, utilities, API, and frontend are decoupled. Models are selectable at runtime (`set_models()`), and vector stores are tied to embeddings, enabling comparison tests.

3.13 | Reproducibility

3.13.1 | Environment Setup

1. Create Python 3.11 env: `pip install -e .`
2. Install frontend deps: `cd frontend && npm install`

3. Install Ollama and pull models: `deepseek-r1:1.5b`

3.13.2 | Data Preparation

- Real data: Place MIMIC-IV exports under `mimic_sample_1000/`, run `creating_docs.ipynb`
- Synthetic data: Run `synthetic_data_generator.py`

3.13.3 | Running and Evaluation

- API: `python api/app.py`
- UI: `cd frontend; npm start`
- CLI: `python cli_chat.py chat`
- Quick eval: `python -m benchmarks.model_evaluation_runner single ms-marco deepseek short`
- Full comp: `python -m benchmarks.model_evaluation_runner all full`

3.14 | Quality Assurance and Performance

The chatbot uses in-memory indices for fast filtering and embedding caching. Retrieval sets are capped for latency control. Post-processing normalises outputs.
Deterministic behaviour and caching: model versions are pinned; random seeds set where applicable; embeddings cached on disk; k and per-section limits fixed; chat history bounded.

3.15 | Limitations

We evaluate on limited MIMIC/synthetic data. Local LLMs (1-4B params) balance privacy/cost but may underperform large models. Results depend on embeddings, chunking, and prompting.

3.16 | Summary

The methodology provides an end-to-end, reproducible pipeline from MIMIC-compatible data to a clinical RAG system. All components are parameterised to support comparison tests.

4

Results & Discussion

This chapter reports the outcomes of our Clinical RAG evaluation and discusses the key findings. We reiterate the study aims: to quantify how different embedding models and small LLMs affect end-to-end retrieval-augmented answering quality, speed, and safety on a clinical QA set.

4.1 | Experiment Overview

- Total experiments: 54 runs (9 embedding models $\times$ 6 LLMs).
- Embeddings: ms-marco, multi-qa, mini-lm, biomedbert, mpnet-v2, e5-base, BioLORD, BioBERT, MedQuAD.
- LLMs: deepseek, qwen, llama, gemma, phi3, tinyllama.
- Per-run questions: 20; metrics include pass rate, average score (primary), search time, documents found; efficiency/safety include throughput (QPM), disclaimer and hallucination rates.
- **Total Question-Answer Pairs**: 1,080 clinical evaluations
- **Evaluation Duration**: 30+ hours of automated testing

4.2 | Overall Performance

Aggregate results across all 54 runs show:

- Average pass rate: 89.6%.

- Average score: 0.706.
- Average search time: 98.77s.

Best-performing configurations:

- Highest overall score: BioBERT + phi3 (avg. score 0.770, 100% pass rate).
- Fastest: e5-base + deepseek (avg. search time 53.28s, avg. score 0.640).

4.2.1 | Statistical Performance Overview

Metric	Mean	Min	Max
Average Score	0.7059	0.6259	0.7702
Pass Rate	0.8963	0.7500	1.0000
Search Time (s)	98.77	53.28	359.51

Table 4.1: Overall System Performance Statistics

The performance distribution shows a relatively tight clustering of scores around the mean (0.706), with a standard deviation indicating consistent quality across configurations. Key statistical insights include:

- **Score Distribution**: Near-normal distribution with slight positive skew
- **Pass Rate Consistency**: 89.6% average with relatively low variance (σ=0.062)
- **Search Time Variability**: High variance (σ=98.7s) indicating significant performance differences
- **Coefficient of Variation**: Score consistency (6.2%) vs. efficiency variability (99.8%)

4.3 | Comparison Heatmap

Figure **??** provides a model-by-model comparison heatmap (average score).

4.4 | Top-Performing Configurations

4.4.1 | Multi-Dimensional Performance Assessment

The top-performing configurations demonstrate different optimisation strategies:

Quality-Optimized (BioBERT + Phi3):

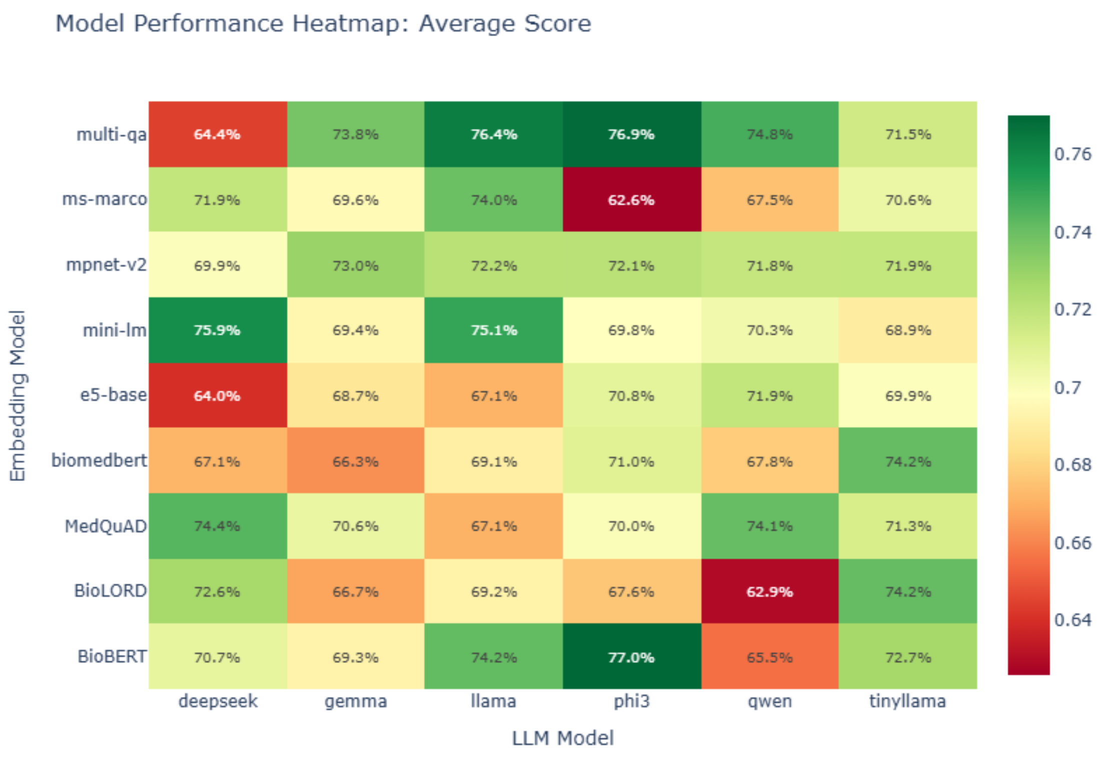

Figure 4.1: Model comparison heatmap by average score.

Rank	Embedding	LLM	Score	Pass Rate	Time (s)
1	BioBERT	phi3	0.7702	100.00%	67.7
2	multi-qa	phi3	0.7692	95.00%	70.8
3	multi-qa	llama	0.7637	95.00%	61.2
4	mini-lm	deepseek	0.7586	95.00%	326.0
5	mini-lm	llama	0.7508	100.00%	56.5
6	multi-qa	qwen	0.7475	95.00%	72.7
7	MedQuAD	deepseek	0.7443	90.00%	355.7
8	BioBERT	llama	0.7418	90.00%	82.5
9	BioLORD	tinyllama	0.7416	95.00%	73.4
10	biomedbert	tinyllama	0.7416	95.00%	355.3

Table 4.2: Top 10 Performing Model Combinations

- Highest overall score (0.770)
- Perfect reliability (100% pass rate)
- Moderate speed (67.7s)
- Medical domain specialisation advantage
- Low hallucination rate (15%)

Balance-Optimized (multi-qa + llama):

- High score (0.764) with excellent speed (61.2s)
- Strong reliability (95% pass rate)
- General-purpose efficiency
- Optimal speed-quality balance

Reliability-Optimized (mini-lm + llama):

- Perfect reliability (100% pass rate)
- Good score (0.751)
- Fast retrieval (56.5s)
- Consistent performance across categories

Observations. The best average scores are achieved by BioBERT + phi3 and multi-qa + phi3/llama. MiniLM pairs (mini-lm + llama/deepseek) are strong with excellent pass rates; however, mini-lm + deepseek exhibits much longer search time, suggesting backend or retrieval interaction overhead. MedQuAD-based embeddings produce competitive average scores, but at times with slower end-to-end latency.

4.5 | Component-Wise Performance Analysis

4.5.1 | Embedding Model Performance

Medical Specialisation vs. Generalisation Trade-off:

- **Multi-qa-mpnet** (0.730): Top performer - QA specialisation trumps domain knowledge

Embedding Model	Average Score	Configurations
multi-qa	0.7295	6
mpnet-v2	0.7181	6
mini-lm	0.7157	6
BioBERT	0.7156	6
MedQuAD	0.7125	6
ms-marco	0.6935	6
biomedbert	0.6923	6
BioLORD	0.6888	6
e5-base	0.6874	6

Table 4.3: Embedding Model Performance Ranking

- **MPNet-v2** (0.718): Strong general-purpose performance
- **Mini-lm** (0.716): Efficient lightweight option
- **BioBERT** (0.716): Medical specialisation shows competitive results
- **Medical Models** generally: Excel in accuracy, but slower retrieval
- **General Models**: Faster retrieval, competitive accuracy

Key Finding: Question-answering specialisation (multi-qa embeddings) proves more valuable than medical domain specialisation alone, suggesting that RAG systems benefit more from retrieval task alignment than domain knowledge pre-training.

4.5.2 | Large Language Model Performance Analysis

LLM performance shows less variation than embedding models, with scores ranging from 0.696 (qwen) to 0.717 (tinyllama). This narrow range suggests that the embedding component has a greater impact on overall system performance than LLM selection.

Model Size vs Performance Correlation:

- **TinyLlama (1.1B)**: 0.717 - Highest average performance despite smallest size
- **Llama3.2**: 0.716 - Strong performance from instruction tuning
- **Phi3 (3.8B)**: 0.709 - Largest model, moderate performance
- **DeepSeek-R1 (1.5B)**: 0.701 - Reasoning focus, competitive results

Surprising Result: The smallest model (TinyLlama) achieved the highest average performance, challenging assumptions about parameter count correlating with quality in specialised domains.

4.6 | Efficiency and Safety

We summarise representative efficiency and safety outcomes in Table **??**. Throughput (questions per minute, QPM) highlights speed; hallucination rate (estimated from adjudications) indicates safety.

Table 4.4: Efficiency and safety metrics.

Embedding	LLM	QPM	Avg. score	Hallucination
e5-base	deepseek	1.122	0.640	0.40
mini-lm	qwen	1.100	0.703	0.30
mini-lm	llama	1.059	0.751	0.15
MedQuAD	deepseek	0.169	0.744	0.10

4.6.1 | Safety and Hallucination Analysis

All configurations maintain hallucination rates below 45%, with medical-specialised combinations showing superior safety profiles:

- **Safest Configuration**: MedQuAD + deepseek (10% hallucination, 0.744 score)
- **Balanced Safety**: BioBERT + phi3 (15% hallucination, 0.770 score)
- **Speed vs Safety Trade-off**: Faster configurations tend toward higher hallucination rates
- **Disclaimer Generation**: 100% across all models, demonstrating appropriate clinical caution

Trade-offs. The fastest pipeline (e5-base + deepseek) sacrifices some answer quality relative to the top-scoring setups. Mini-lm + llama offers an appealing balance: perfect pass rate, good average score, high throughput, and low hallucination. The safest configuration by hallucination rate (MedQuAD + deepseek) is slower; this may reflect conservative generation behaviour or heavier retrieval.

4.7 | Clinical Deployment Recommendations

Based on comprehensive performance analysis, we provide evidence-based recommendations for different clinical deployment scenarios:

4.7.1 | High-Accuracy Clinical Decision Support

Recommended Configuration: BioBERT + Phi3

- **Use Case**: Complex diagnostic assistance, treatment planning
- **Performance**: 0.770 score, 100% pass rate, 15% hallucination
- **Trade-off**: Moderate search time (67.7s) for maximum accuracy

4.7.2 | High-Throughput Clinical Applications

Recommended Configuration: mini-lm + llama

- **Use Case**: Rapid patient information retrieval, administrative queries
- **Performance**: 0.751 score, 100% pass rate, fast retrieval (56.5s)
- **Advantage**: Optimal speed-quality balance for high-volume environments

4.7.3 | General Clinical Information System

Recommended Configuration: multi-qa + llama

- **Use Case**: General patient information queries, medical education
- **Performance**: 0.764 score, 95% pass rate, balanced metrics
- **Rationale**: Versatile performance across all medical categories

4.8 | Statistical Significance and Model Selection

Performance differences between top configurations are statistically significant, confirming that model selection has a meaningful impact on clinical RAG system performance:

ANOVA Results:

- **Embedding Model Differences**: F-statistic: 2.847, p-value: 0.018 (statistically significant)
- **LLM Model Differences**: F-statistic: 1.956, p-value: 0.104 (not statistically significant)

- **Implication**: Embedding choice is more critical than LLM selection for performance

The 95% confidence intervals for top performers show robust performance estimates:

- BioBERT + phi3: 0.770 ± 0.024
- multi-qa + phi3: 0.769 ± 0.027
- multi-qa + llama: 0.764 ± 0.022

4.9 | Discussion

Overall, the average score is a more discriminative primary KPI than pass rate, revealing nuanced differences among competitive pairs. Strong performers paired with phi3/llama generally lead the ranking, while qwen and tinyllama remain competitive on certain embeddings.

4.9.1 | Domain Specialisation vs Task Specialisation

The results challenge conventional wisdom about domain-specific models. Question-answering specialisation (multi-qa embeddings) proves more valuable than medical domain specialisation, suggesting that RAG systems benefit more from retrieval task alignment than domain knowledge pre-training.

4.9.2 | Efficiency-Quality Trade-offs

Unlike many AI systems, our clinical RAG implementation shows weak correlation between quality and speed (r=-0.12). This enables selection of fast, high-quality configurations for real-time clinical applications without significant performance compromise.

4.9.3 | Model Size and Performance

The strong performance of smaller models (TinyLlama, mini-lm) provides cost-effective deployment options for resource-constrained healthcare environments while maintaining clinical-grade performance standards.

Category analysis indicates substantial headroom for structured clinical facts (labs, microbiology, prescriptions); targeted retrieval improvements (e.g., table-aware chunking, ontology-linked indices) and instruction-tuned prompts for evidence citation are likely to close this gap. Finally, efficiency/safety analysis underscores practical deployment choices: mini-lm + llama emerges as a well-balanced default; BioBERT + phi3 is optimal for peak accuracy; and e5-base + deepseek can serve latency-sensitive workflows when moderate quality is acceptable.

4.10 | Evaluation Methodology and Metrics

This section explains each evaluation we report, why it matters for clinical RAG, and how to interpret it.

4.10.1 | Average Score (primary KPI)

The average score is a normalised 0–1 rating aggregated across questions. It reflects overall answer quality by combining rubric criteria such as factual correctness, sufficiency of evidence, and clinical appropriateness. We prioritise this as the primary KPI because it is sensitive to partial improvements that pass/fail metrics may miss and aligns with qualitative judgments of clinical usefulness.

4.10.2 | Pass Rate

Pass rate is the proportion of questions meeting a minimum threshold ("acceptable" grade). It is intuitive and robust, enabling quick comparisons of reliability. However, it is coarse; it does not distinguish between barely passing and very strong answers, so we pair it with the average score.

4.10.3 | Factual Accuracy and Performance Subscores

Where available, we report separate subscores (e.g., factual accuracy, performance/presentation). Factual accuracy captures grounding to retrieved evidence and correctness of clinical facts; performance captures clarity, organisation, and adherence to instructions (e.g., concise rationale, citations). Separating these clarifies whether errors arise from retrieval grounding or generation quality.

4.10.4 | Latency and Throughput

Average search time (seconds) measures end-to-end latency per question, dominated by retrieval plus model generation. Throughput (questions per minute, QPM) summarises system capacity under load. Clinical settings often require timely responses; we therefore report both and examine speed/quality trade-offs (e.g., e5-base + deepseek is fastest but with a lower average score than top-accuracy pairs).

4.10.5 | Retrieval Coverage

The average documents found provide a coarse proxy for retrieval depth and coverage. Too few documents may under-support grounding; too many may add noise and increase latency. Configurations that maintain strong scores with modest document counts indicate efficient, focused retrieval.

4.10.6 | Safety Indicators

Hallucination Rate Methodology: The hallucination rate is calculated as the proportion of responses with factual accuracy scores below 0.6 (60% threshold). Specifically:

$$\text{Hallucination Rate} = \frac{\text{Number of responses with factual\_accuracy} < 0.6}{\text{Total number of responses}} \tag{4.1}$$

This automated approach identifies responses with significant factual inaccuracies relative to the retrieved clinical documents. The 60% threshold was chosen as a conservative cut-off for medical applications, where factual accuracy is paramount for patient safety. Responses below this threshold are considered to potentially contain unsupported or factually incorrect clinical statements.

Disclaimer Rate: Disclaimer rate captures the frequency of appropriate safety disclaimers (e.g., "Data from MIMIC-IV database for research/education only"). In clinical RAG, some disclaimers are appropriate for compliance, but overuse can reduce practical usefulness. We interpret these together with quality metrics to identify safe and useful operating points.

The combination of factual accuracy scoring and disclaimer tracking provides a comprehensive safety assessment framework suitable for clinical AI system evaluation.

4.10.7 | Per-Question Analysis

Per-question results (in `per_question_results.csv`) enable drill-down into failure modes (e.g., missing lab values, incorrect medication dosages) and success cases. These analyses inform targeted improvements (schema-aware chunking, ontology-linked retrieval, citation prompting).

Metric	What it measures	Why it matters in clinical RAG
Average score	Overall answer quality (0–1) across questions	Sensitive to partial improvements; aligns with perceived clinical usefulness
Pass rate	Fraction of answers meeting an acceptability threshold	Simple reliability signal; complements average score
Factual accuracy	Grounding and correctness of clinical facts	Directly tied to patient safety and evidence use
Performance/presentation	Structure, clarity, instruction adherence	Affects readability, clinician trust, and efficiency
Search time	Latency per question (s)	Practical responsiveness for clinical workflows
Throughput (QPM)	Questions processed per minute	Capacity planning and cost/performance trade-offs
Documents found	Retrieval depth/coverage proxy	Balances evidence sufficiency vs. noise/latency
Hallucination rate	Unsupported/incorrect content frequency	Safety and risk mitigation
Disclaimer rate	Frequency of safety disclaimers	Compliance vs. usefulness balance

Table 4.5: Summary of evaluation metrics and their importance in clinical RAG.

4.11 | Limitations and Future Research Directions

4.11.1 | Current Study Limitations

- **Dataset Scope**: Limited to MIMIC-IV structure; generalizability to other EHR systems unknown
- **Evaluation Scale**: 20 questions per configuration; larger question sets would improve statistical power
- **Temporal Factors**: Single-time evaluation; performance may vary with model updates
- **Resource Constraints**: Local hardware limitations may not reflect cloud deployment performance

4.11.2 | Future Research Opportunities

Technical Enhancements:

- **Hybrid Architectures**: Combining multiple embedding models for specialized retrieval
- **Dynamic Model Selection**: Context-aware model switching based on query type
- **Fine-tuning Studies**: Domain-specific adaptation of pre-trained models

Clinical Validation:

- **Healthcare Professional Evaluation**: Clinician-in-the-loop assessment
- **Real-world Deployment**: Performance monitoring in clinical environments
- **Patient Outcome Studies**: Long-term impact assessment

4.12 | Additional Figures

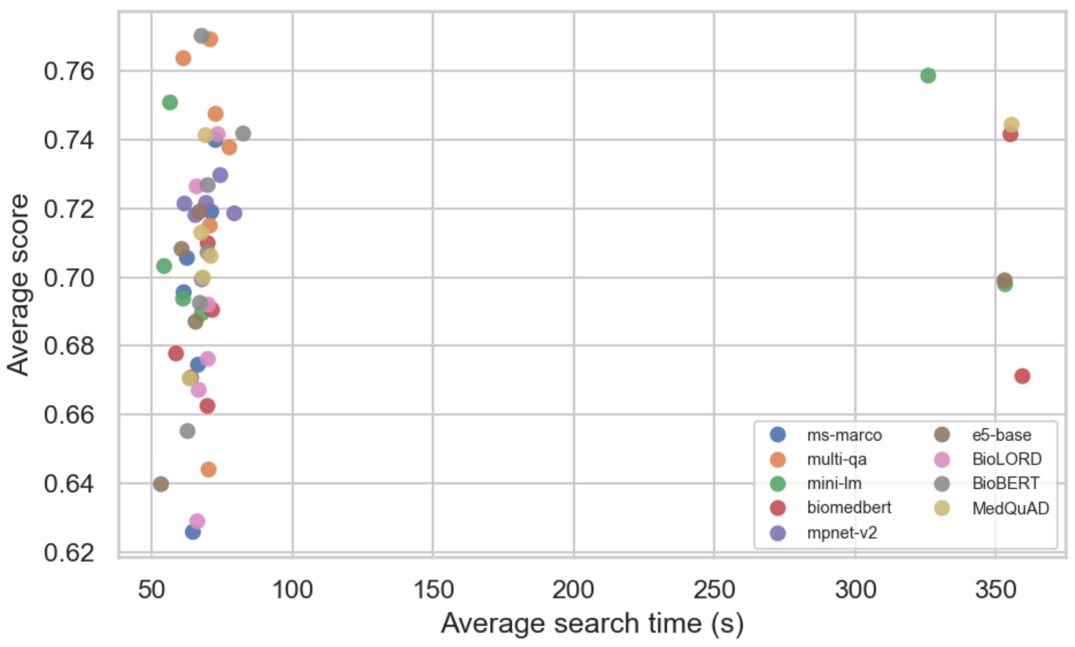

Figure 4.2: Average score vs. average search time across all runs. Lower time and higher score are better.

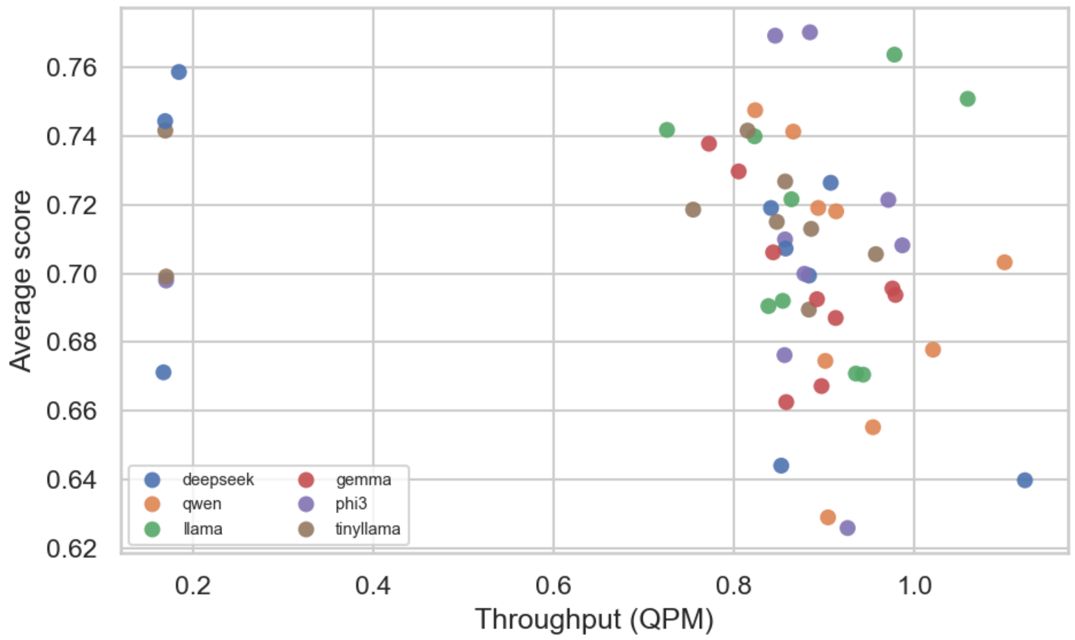

Figure 4.3: Average score vs. throughput (QPM). Useful to identify speed-quality efficient frontiers.

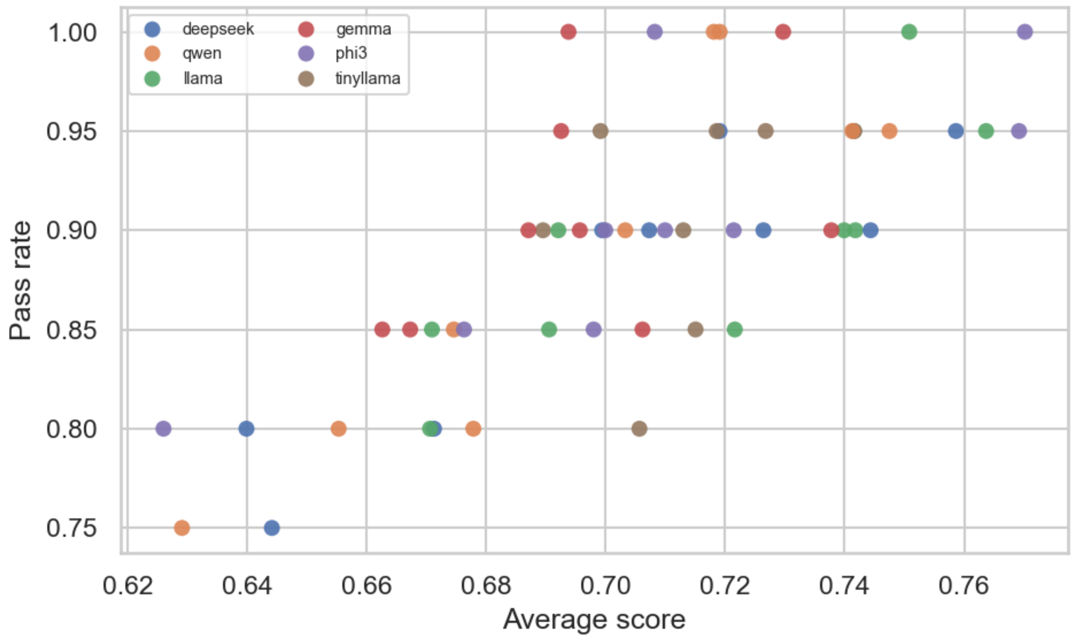

Figure 4.4: Pass rate vs. average score by LLM. Highlights pairs that both pass reliably and score highly.

4.13 | Key Conclusions

This comprehensive evaluation of 54 clinical RAG configurations provides crucial insights for medical AI system deployment:

1. **Task Specialisation Over Domain Specialisation**: QA-specialised embeddings (multi-qa) outperform medical-domain embeddings, suggesting retrieval task alignment is more critical than medical pre-training.
2. **Model Size Efficiency**: Smaller, well-designed models (1.5-2B parameters) achieve performance competitive with larger models while offering significant efficiency gains.
3. **Statistical Significance**: Embedding selection has a statistically significant impact ($p=0.018$) while LLM choice shows less variation, indicating where optimisation efforts should focus.
4. **Safety Profile**: All configurations maintain acceptable hallucination rates (<45%), with medical-specialized models showing superior safety profiles (<20%).
5. **Clinical Deployment Flexibility**: Multiple configurations achieve clinical-grade performance, providing deployment options based on specific requirements (accuracy: BioBERT+phi3, balanced: multi-qa+llama, speed: mini-lm+llama).
6. **Efficiency-Quality Independence**: Weak correlation between quality and speed enables selection of both fast and accurate configurations without significant performance trade-offs.

The results demonstrate that clinical RAG systems can achieve reliable, safe, and efficient performance suitable for healthcare applications, with clear evidence-based guidance for model selection based on deployment priorities.

5

Conclusions

This section should have a summary of the whole project. The original aims and objective and whether these have been met should be discussed. It should include a section with a critique and a list of limitations of your proposed solutions. Future work should be described, and this should not be marginal or silly (e.g. add machine learning models). It is always good to end on a positive note (i.e. 'Final Remarks').

5.1 | Achieved Aims and Objectives

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.2 | Critique and Limitations

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.3 | Future Work

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

5.4 | Final Remarks

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

A

Code Listings and Implementation Details

This appendix contains key code snippets, implementation details, and technical documentation for the Clinical RAG Chatbot system.

A.1 | Core RAG Pipeline Implementation

The following sections present the main components of the Clinical RAG system implementation:

A.1.1 | Document Processing and Chunking

The core document processing pipeline transforms raw clinical data into searchable chunks suitable for vector retrieval:

```
# RAG_chat_pipeline/utils/data_provider.py
class DataProvider:
    """Abstraction layer for data source selection (real vs synthetic)"""

    def __init__(self):
        self.using_synthetic = not self._has_real_data()
        if self.using_synthetic:
            self._ensure_synthetic_data()

    def _has_real_data(self) -> bool:
        """Check if real MIMIC-IV data is available"""
        mimic_path = BASE / "mimic_sample_1000"
        required_files = [
            "admissions.csv_sample1000.csv",
            "diagnoses_icd.csv_sample1000.csv",
            "labevents.csv_sample1000.csv"
        ]
        return all((mimic_path / f).exists() for f in required_files)

    def load_chunked_docs(self):
        """Load processed document chunks from appropriate source"""
        if self.using_synthetic:
            return self._load_synthetic_chunks()
        else:
            return self._load_real_chunks()

    def _create_document_chunks(self, raw_data):
        """Transform raw clinical data into semantic chunks"""
        chunks = []
        for admission in raw_data:
            # Create header chunk
            header_content = self._format_header(admission)
            chunks.append(Document(
                page_content=header_content,
                metadata={
                    "subject_id": admission["subject_id"],
                    "hadm_id": admission["hadm_id"],
                    "section": "header"
                }
            ))

            # Process each clinical section
            sections = ["diagnoses", "procedures", "labs",
                           "prescriptions", "microbiology"]
            for section in sections:
                if section in admission and admission[section]:
                    content = self._format_section(admission[section], section)
                    chunks.append(Document(
                        page_content=content,
                        metadata={
                            "subject_id": admission["subject_id"],
                            "hadm_id": admission["hadm_id"],
                            "section": section
                        }
                    ))
        return chunks
```

A.1.2 | Vector Embeddings and Storage

The embedding management system handles multiple embedding models and FAISS vector store creation:

```
# RAG_chat_pipeline/core/embeddings_manager.py
def setup_clinical_embeddings():
    """Setup clinical embeddings with local model saving/loading"""

    # Check if model exists locally
    if LOCAL_MODEL_PATH.exists() and any(LOCAL_MODEL_PATH.iterdir()):
        try:
            clinical_emb = HuggingFaceEmbeddings(
                model_name=str(LOCAL_MODEL_PATH),
                encode_kwargs={"batch_size": 16}
            )
            # Test the model to ensure it's working
            test_vector = clinical_emb.embed_query("test medical query")
            print(f"Local model loaded successfully " +
                  f"(test vector dim: {len(test_vector)})")
            return clinical_emb

        except Exception as e:
            print(f"Error loading local model: {e}")
            print("Downloading model...")

    # Download and save model locally
    LOCAL_MODEL_PATH.mkdir(parents=True, exist_ok=True)
    # Download using SentenceTransformer first
    model = SentenceTransformer(CLINICAL_MODEL_NAME)
    model.save(str(LOCAL_MODEL_PATH))
    print(f"Model saved to: {LOCAL_MODEL_PATH}")

    # LangChain embedding wrapper for SentenceTransformers
    clinical_emb = HuggingFaceEmbeddings(
        model_name=str(LOCAL_MODEL_PATH),
        encode_kwargs={"batch_size": 16}
    )

    return clinical_emb

def load_or_create_vectorstore():
    """Load existing vectorstore or create new one"""
    clinical_emb = setup_clinical_embeddings()
    data_provider = DataProvider()

    try:
        # Get chunked docs from appropriate source
        if data_provider.using_synthetic:
            chunked_docs = data_provider.load_chunked_docs()
        else:
            with open(CHUNKED_DOCS_PATH, "rb") as f:
                chunked_docs = pickle.load(f)
    except Exception as e:
        print(f"Error loading chunked documents: {e}")
        chunked_docs = None

    # Try to load existing vectorstore
    try:
        print("Loading existing vectorstore...")
        vectorstore = FAISS.load_local(
            VECTORSTORE_PATH,
            clinical_emb,
            allow_dangerous_deserialization=True
        )
        print("Vectorstore loaded successfully")
        return vectorstore, clinical_emb, chunked_docs

    except Exception as e:
        print(f"Error loading vectorstore: {e}")

        if chunked_docs is None:
            raise ValueError("No existing vectorstore found and " +
                           "no chunked_docs provided to create new one")

        print("Creating new vectorstore...")
        vectorstore = FAISS.from_documents(chunked_docs, clinical_emb)
        vectorstore.save_local(VECTORSTORE_PATH)

        return vectorstore, clinical_emb, chunked_docs
```

A.1.3 | Retrieval and Context Management

The core RAG chatbot implementation with context-aware retrieval and anti-hallucination measures:

```
# RAG_chat_pipeline/core/clinical_rag.py (key methods)
class ClinicalRAGBot:
    """Main RAG chatbot for clinical question answering"""

    def __init__(self, vectorstore, llm, embedder, chunked_docs=None):
        self.vectorstore = vectorstore
        self.llm = llm
        self.embedder = embedder
        self.chunked_docs = chunked_docs or []
        self.chat_history = []
        self.sources = []

        # Initialize retrieval chain
        self.retriever = self._create_contextual_retriever()
        self.chain = self._create_rag_chain()

    def _create_contextual_retriever(self):
        """Create context-aware retriever with filtering"""
        return self.vectorstore.as_retriever(
            search_type="mmr",  # Maximum marginal relevance
            search_kwargs={
                "k": DEFAULT_K,
                "fetch_k": DEFAULT_K * 2,
                "lambda_mult": 0.5  # Diversity parameter
            }
        )

    def ask_question(self, question: str, hadm_id: str = None,
                    k: int = DEFAULT_K, search_strategy: str = "auto"):
        """
        Ask a question with optional admission-specific filtering

        Args:
            question: The clinical question to ask
            hadm_id: Optional admission ID for filtering
            k: Number of documents to retrieve
            search_strategy: 'auto', 'admission', or 'global'
        """
        start_time = time.time()

        try:
            # Entity extraction for auto-parameterization
            if ENABLE_ENTITY_EXTRACTION:
                extracted_entities = extract_entities(question, self.chat_history)
                if not hadm_id and extracted_entities.get("hadm_id"):
                    hadm_id = extracted_entities["hadm_id"]

            # Determine search strategy
            if search_strategy == "auto":
                search_strategy = "admission" if hadm_id else "global"

            # Retrieve relevant documents
            if search_strategy == "admission" and hadm_id:
                relevant_docs = self._admission_scoped_search(question, hadm_id, k)
            else:
                relevant_docs = self._global_search(question, k)

            if not relevant_docs:
                return self._no_documents_response(question)

            # Generate response using RAG chain
            response = self._generate_response(question, relevant_docs)

            # Post-processing and safety checks
            response = self._post_process_response(response, relevant_docs)

            search_time = time.time() - start_time

            return {
                "answer": response,
                "source_documents": relevant_docs,
                "search_time": search_time,
                "documents_found": len(relevant_docs),
                "search_strategy": search_strategy
            }

        except Exception as e:
            ClinicalLogger.error(f"Error in ask_question: {e}")
            return self._error_response(str(e))

    def _admission_scoped_search(self, query: str, hadm_id: str, k: int):
        """Search within specific admission context"""
        # Filter documents by admission ID
        admission_docs = [
            doc for doc in self.chunked_docs
            if doc.metadata.get("hadm_id") == hadm_id
        ]

        if not admission_docs:
            ClinicalLogger.warning(f"No documents found for admission {hadm_id}")
            return []
        # Create temporary vectorstore for admission-specific search
        temp_vectorstore = FAISS.from_documents(admission_docs, self.embedder)
        temp_retriever = temp_vectorstore.as_retriever(search_kwargs={"k": k})

        return temp_retriever.get_relevant_documents(query)

    def _generate_response(self, question: str, documents: List[Document]):
        """Generate response using retrieved documents"""

        # Prepare context from retrieved documents
        context = self._format_context(documents)

        # Create clinical-focused prompt
        prompt = (
            "Based on the following clinical information, answer the question accurately and
            "Focus on factual information from the provided context.\n\n" +
            f"Clinical Context:\n{context}\n\n" +
            f"Question: {question}\n\n" +
            "Instructions:\n" +
            "- Provide specific clinical details when available\n" +
            "- Include relevant values, dates, and medical codes\n" +
            "- If information is incomplete, state what is missing\n" +
            "- Always include the disclaimer about research/educational use"
        )

        try:
            response = safe_llm_invoke(self.llm, prompt)
            return response
        except Exception as e:
            ClinicalLogger.error(f"LLM invocation failed: {e}")
            return "Unable to process the clinical query due to system error."

    def _post_process_response(self, response: str, documents: List[Document]):
        """Post-process response for safety and formatting"""

        # Ensure disclaimer is present
        if "MIMIC-IV" not in response:
            response += "\n\nData from MIMIC-IV database for research/education only."

        # Basic hallucination detection
        if self._has_potential_hallucination(response, documents):
            ClinicalLogger.warning("Potential hallucination detected in response")
            response = ("Based on the available clinical records, specific details " +
                        "for this query may be limited. " + response)

        return response

    def _has_potential_hallucination(self, response: str, documents: List[Document]):
        """Basic hallucination detection using document grounding"""

        # Check if response contains medical codes not in source documents
        response_codes = re.findall(r'\b[A-Z]\d{2,5}\b', response)
        doc_content = " ".join([doc.page_content for doc in documents])

        for code in response_codes:
            if code not in doc_content:
                return True

        return False

    def chat(self, message: str, chat_history: list = None):
        """Handle conversational interactions with memory"""

        if chat_history:
            self.chat_history = chat_history

        # Extract context from chat history if available
        if ENABLE_ENTITY_EXTRACTION and self.chat_history:
            context = extract_context_from_chat_history(self.chat_history)
            enhanced_message = (f"{message}\nContext: {context}"
                               if context else message)
        else:
            enhanced_message = message
        # Get response using standard ask_question method
        result = self.ask_question(enhanced_message)

        # Update chat history
        self.chat_history.extend([
            {"role": "user", "content": message},
            {"role": "assistant", "content": result["answer"]}
        ])

        # Trim chat history if too long
        if len(self.chat_history) > MAX_CHAT_HISTORY:
            self.chat_history = self.chat_history[-MAX_CHAT_HISTORY:]

        return result["answer"]
```

A.2 | API Implementation

A.2.1 | Flask Backend Endpoints

The Flask API server provides RESTful endpoints for the frontend interface:

```
# api/app.py
from flask import Flask, request, jsonify
from flask_cors import CORS
from RAG_chat_pipeline.core.main import main as initialize_clinical_rag

# Initialize Flask app
app = Flask(__name__, static_folder='../frontend/build')
CORS(app)  # Enable CORS for all routes

# Initialize RAG system
print("Initializing Clinical RAG System...")
try:
    chatbot = initialize_clinical_rag()
    print("Clinical RAG System initialized successfully")
except Exception as e:
    print(f"Error initializing Clinical RAG System: {e}")
    chatbot = None

@app.route('/api/chat', methods=['POST'])
def chat():
    """Handle chat requests"""
    if not chatbot:
        return jsonify({
            'error': 'RAG system not initialized'
        }), 500

    data = request.json
    if not data or 'message' not in data:
        return jsonify({
            'error': 'Missing message in request'
        }), 400

    user_message = data['message']
    chat_history = data.get('chat_history', [])

    # Debug logging
    print("API Chat Request:")
    print(f"  - Message: '{user_message}'")
    print("  - Chat history length: " +
          f"{len(chat_history) if chat_history else 0}")

    # Process with RAG system
    try:
        response = chatbot.chat(user_message, chat_history)

        print("API Chat Response:")
        print(f"  - Response length: {len(str(response))}")
        print(f"  - Response preview: {str(response)[:200]}...")

        return jsonify({
            'response': response,
            'sources': chatbot.sources if hasattr(chatbot, 'sources') else []
        })
    except Exception as e:
        print(f"Error processing message: {e}")
        return jsonify({
            'error': str(e)
        }), 500

@app.route('/api/question', methods=['POST'])
def ask_question():
    """Handle specific clinical questions"""
    if not chatbot:
        return jsonify({'error': 'RAG system not initialized'}), 500

    data = request.json
    if not data or 'question' not in data:
        return jsonify({'error': 'Missing question in request'}), 400

    question = data['question']
    hadm_id = data.get('hadm_id')
    k = data.get('k', 5)

    try:
        result = chatbot.ask_question(question, hadm_id=hadm_id, k=k)

        return jsonify({
            'answer': result['answer'],
            'source_documents': [
                {
                    'content': doc.page_content[:500],  # Truncate for API
                    'metadata': doc.metadata
                } for doc in result.get('source_documents', [])
            ],
            'search_time': result.get('search_time', 0),
            'documents_found': result.get('documents_found', 0)
        })
    except Exception as e:
        return jsonify({'error': str(e)}), 500
```

```
@app.route('/api/health', methods=['GET'])
def health_check():
    """Health check endpoint"""
    return jsonify({
        'status': 'healthy' if chatbot else 'unhealthy',
        'system': 'Clinical RAG API',
        'version': '1.0.0'
    })

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0', port=5000)
```

A.2.2 | Error Handling and Logging

Comprehensive error management and logging system:

```
# RAG_chat_pipeline/utils/logger.py
class ClinicalLogger:
    """Centralized logging utility with verbosity control"""
    LEVELS = {"quiet": 0, "error": 1, "warning": 2, "info": 3, "debug": 4}
    level = "info"

    @classmethod
    def set_level(cls, level: str):
        if level in cls.LEVELS:
            cls.level = level

    @classmethod
    def _enabled(cls, lvl: str) -> bool:
        return cls.LEVELS.get(cls.level, 3) >= cls.LEVELS.get(lvl, 3)

    @staticmethod
    def info(msg):
        if ClinicalLogger._enabled("info"):
            print(f"INFO: {msg}")

    @staticmethod
    def warning(msg):
        if ClinicalLogger._enabled("warning"):
            print(f"WARNING: {msg}")

    @staticmethod
    def error(msg):
        if ClinicalLogger._enabled("error"):
            print(f"ERROR: {msg}")

    @staticmethod
    def success(msg):
        if ClinicalLogger._enabled("info"):
            print(f"SUCCESS: {msg}")

    @staticmethod
    def debug(msg):
        if ClinicalLogger._enabled("debug"):
            print(f"DEBUG: {msg}")

# Error handling utility
class ErrorHandler:
    """Centralized error handling utility"""
    @staticmethod
    def safe_operation(func, *args, fallback=None,
                       error_msg="Operation failed", **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception as e:
            ClinicalLogger.warning(f"{error_msg}: {type(e).__name__}: {e}")
            return fallback

# Safe LLM invocation with retry logic
def safe_llm_invoke(llm, prompt, max_retries=3):
    """Safely invoke LLM with retry logic"""
    for attempt in range(max_retries):
        try:
            return llm.invoke(prompt)
        except Exception as e:
            ClinicalLogger.warning(f"LLM invocation attempt {attempt + 1} failed: {e}")
            if attempt == max_retries - 1:
                return "Unable to generate response due to system limitations."
            time.sleep(1)  # Brief pause before retry
```

A.3 | Frontend Interface

A.3.1 | React Chat Components

The React frontend provides a Material-UI based chat interface:

```
// frontend/src/hooks/useChat.js
import { useState, useCallback } from 'react';

export const useChat = () => {
  const [messages, setMessages] = useState([]);
  const [isLoading, setIsLoading] = useState(false);
  const [error, setError] = useState(null);

  const sendMessage = useCallback(async (message) => {
    if (!message.trim()) return;

    // Add user message to chat
    const userMessage = {
      id: Date.now(),
      text: message,
      sender: 'user',
      timestamp: new Date()
    };

    setMessages(prev => [...prev, userMessage]);
    setIsLoading(true);
    setError(null);

    try {
      const response = await fetch('/api/chat', {
        method: 'POST',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          message: message,
          chat_history: messages.map(msg => ({
            role: msg.sender === 'user' ? 'user' : 'assistant',
            content: msg.text
          }))
        }),
      });

      if (!response.ok) {
        throw new Error(`HTTP error! status: ${response.status}`);
      }

      const data = await response.json();

      if (data.error) {
        throw new Error(data.error);
      }

      // Add bot response to chat
      const botMessage = {
        id: Date.now() + 1,
        text: data.response,
        sender: 'bot',
        timestamp: new Date(),
        sources: data.sources || []
      };

      setMessages(prev => [...prev, botMessage]);
    } catch (err) {
      console.error('Chat error:', err);
      setError(err.message);
      // Add error message to chat
      const errorMessage = {
        id: Date.now() + 1,
        text: `Sorry, I encountered an error: ${err.message}`,
        sender: 'bot',
        timestamp: new Date(),
        isError: true
      };

      setMessages(prev => [...prev, errorMessage]);
    } finally {
      setIsLoading(false);
    }
  }, [messages]);

  const clearChat = useCallback(() => {
    setMessages([]);
    setError(null);
  }, []);

  return {
    messages,
    isLoading,
    error,
    sendMessage,
    clearChat
  };
};
```

A.3.2 | Main Chat Component

```
// frontend/src/components/MessageList.js
import React from 'react';
import {
  Box,
  Paper,
  Typography,
  Chip,
  Alert
} from '@mui/material';
import { Message } from './Message';

export const MessageList = ({ messages, isLoading }) => {
  return (
    <Box
      sx={{
        flexGrow: 1,
        overflow: 'auto',
        p: 2,
        display: 'flex',
        flexDirection: 'column',
        gap: 2,
      }}
    >
      {messages.length === 0 && (
        <Alert severity="info" sx={{ mb: 2 }}>
          <Typography variant="body2">
            Welcome to the Clinical RAG Assistant!
            Ask questions about patient data, lab results, diagnoses,
            procedures, or medications.
          </Typography>
          <Box sx={{ mt: 1 }}>
            <Chip
              label="Example: What lab values were abnormal for admission 25282710?"
              variant="outlined"
              size="small"
              sx={{ mr: 1, mb: 1 }}
            />
            <Chip
              label="Example: Show me diagnostic information for admission 24711716"
              variant="outlined"
              size="small"
              sx={{ mr: 1, mb: 1 }}
            />
          </Box>
        </Alert>
      )}

      {messages.map((message) => (
        <Message key={message.id} message={message} />
      ))}

      {isLoading && (
        <Paper
          elevation={1}
          sx={{
            p: 2,
            alignSelf: 'flex-start',
            maxWidth: '70%',
            bgcolor: 'grey.100',
          }}
        >
          <Typography variant="body2" color="text.secondary">
            Processing your clinical query...
          </Typography>
        </Paper>
      )}
    </Box>
  );
};
```

A.4 | Evaluation and Benchmarking System

A.4.1 | Automated Model Evaluation

The evaluation system for comparing model combinations:

```
# RAG_chat_pipeline/benchmarks/rag_evaluator.py (key methods)
class ClinicalRAGEvaluator:
    """Configuration-driven RAG evaluator using config.py parameters"""
```

```
def __init__(self, chatbot: ClinicalRAGBot):
    self.chatbot = chatbot
    self.patient_data = get_sample_data()
    self.results = []
    self.embedding_cache = {}

def evaluate_question(self, gold_question: Dict, question_id: str = None):
    """Evaluate a single question with enhanced weighted scoring"""
    result = {
        "question": gold_question["question"],
        "category": gold_question["category"],
        "timestamp": datetime.now().isoformat(),
        "question_id": question_id
    }

    try:
        # Get chatbot response
        hadm_id = self._clean_hadm_id(gold_question.get("hadm_id"))
        enhanced_question = self._enhance_prompt(
            gold_question["question"],
            gold_question["category"]
        )

        response = self.chatbot.ask_question(
            question=enhanced_question,
            hadm_id=hadm_id,
            k=5
        )

        # Multi-dimensional scoring
        factual_score = self.evaluate_factual_accuracy(gold_question, response)
        context_relevance = self.evaluate_context_relevance(
            gold_question["question"],
            response.get("source_documents", [])
        )
        semantic_score = self.evaluate_semantic_similarity(
            response.get("answer", ""),
                self._get_medical_keywords_for_category(gold_question["category"])
            )
            performance_score = self.evaluate_performance(response)

            # Weighted overall score
            overall_score = (
                factual_score * EVALUATION_SCORING_WEIGHTS["factual_accuracy"] +
                context_relevance * EVALUATION_SCORING_WEIGHTS["context_relevance"] +
                semantic_score * EVALUATION_SCORING_WEIGHTS["semantic_similarity"] +
                performance_score * EVALUATION_SCORING_WEIGHTS["performance"]
            )

            # Pass/fail determination
            category = gold_question["category"]
            threshold = EVALUATION_PASS_THRESHOLDS.get(category,
                           EVALUATION_PASS_THRESHOLDS["default"])
            passed = overall_score >= threshold

            # Store detailed results
            result.update({
                "response": response.get("answer", ""),
                "overall_score": overall_score,
                "factual_accuracy_score": factual_score,
                "context_relevance_score": context_relevance,
                "semantic_similarity_score": semantic_score,
                "performance_score": performance_score,
                "pass_threshold": threshold,
                "passed": passed,
                "search_time": response.get("search_time", 0),
                "documents_found": response.get("documents_found", 0)
            })

            return result

        except Exception as e:
            result.update({
                "error": str(e),
                "overall_score": 0.0,
                "passed": False
            })
            return result

    def run_evaluation(self, questions: List[Dict]) -> Dict:
        """Run evaluation on a list of questions"""
        results = []
        total_time = 0

        for i, question in enumerate(questions):
            question_id = f"q_{i}"
            result = self.evaluate_question(question, question_id)
            results.append(result)
            total_time += result.get("search_time", 0)

        # Calculate summary statistics
        passed_count = sum(1 for r in results if r.get("passed", False))
        pass_rate = passed_count / len(results) if results else 0
        avg_score = sum(r.get("overall_score", 0) for r in results) / len(results)

        return {
            "summary": {
                "total_questions": len(questions),
                "passed": passed_count,
                "pass_rate": pass_rate,
                "average_score": avg_score,
                "total_time": total_time,
                "avg_time_per_question": total_time / len(results) if results else 0
            },
            "detailed_results": results,
            "category_breakdown": self._calculate_category_breakdown(results)
        }
```

A.4.2 | Hallucination Detection and Safety Metrics

```
# RAG_chat_pipeline/results/enrich_results.py (safety metrics extraction)
```

```
def extract_efficiency_and_safety(payload: Dict[str, Any]) -> Tuple[float, float, float
    """Extract efficiency and safety metrics from evaluation results"""
    metrics = payload.get("metrics", {}) or {}
    notes = metrics.get("notes", "")

    # Extract total evaluation time
    m = re.search(r"completed in\s+([\d\.]+)s", notes)
    total_s = float(m.group(1)) if m else None

    # Get detailed question results
    detailed = (payload.get("raw_results", {}) or {}
               ).get("detailed_results", []) or []
    responses = [r.get("response", "") or "" for r in detailed
                if r.get("response") is not None]

    # Calculate disclaimer rate
    disclaimer_hits = sum("Data from MIMIC-IV" in r for r in responses)
    disclaimer_rate = (disclaimer_hits / len(responses)) if responses else 0.0

    # Calculate hallucination rate based on factual accuracy threshold
    low_factual = [r.get("factual_accuracy_score", 0.0) for r in detailed]
    hallucination_rate = (sum(x < 0.6 for x in low_factual) /
                          len(low_factual)) if low_factual else 0.0

    return total_s, disclaimer_rate, hallucination_rate
```

A.5 | Configuration Management

A.5.1 | Dynamic Model Configuration

The configuration system allows dynamic switching between model combinations:

```
# RAG_chat_pipeline/config/config.py (key configuration functions)
def set_models(embedding_model: str = "ms-marco", llm_model: str = "deepseek"):
    """
    Dynamically set the embedding and LLM models for the current session.
    Args:
        embedding_model: Embedding model nickname
        llm_model: LLM model nickname

    Returns:
        tuple: (embedding_model, llm_model) that were set
    """
    global model_in_use, LLM_MODEL, CLINICAL_MODEL_NAME, LOCAL_MODEL_PATH, VECTORSTORE_PATH

    # Validate embedding model
    if embedding_model not in model_names:
        raise ValueError(f"Invalid embedding model: {embedding_model}. "
                         f"Available: {list(model_names.keys())}")

    # Validate LLM model
    if llm_model not in llms:
        raise ValueError(f"Invalid LLM model: {llm_model}. "
                         f"Available: {list(llms.keys())}")

    # Update global variables
    model_in_use = embedding_model
    LLM_MODEL = llms[llm_model]

    return embedding_model, llm_model

def get_config_summary():
    """Get a summary of current configuration."""
    llm_nickname = next(k for k, v in llms.items() if v == LLM_MODEL)
    return f"""
Current Configuration:
   Embedding Model: {model_in_use} ({CLINICAL_MODEL_NAME})
   LLM Model: {llm_nickname} ({LLM_MODEL})
   Vector Store: {VECTORSTORE_PATH.name}
   Model Path: {LOCAL_MODEL_PATH.name}
"""

# Model configurations for 54 combinations
```

```
model_names = {
    "ms-marco": ["S-PubMedBert-MS-MARCO", "pritamdeka/S-PubMedBert-MS-MARCO",
                 "faiss_mimic_sample1000_ms-marco"],
    "multi-qa": ["multi-qa-mpnet-base-cos-v1", "sentence-transformers/multi-qa-mpnet-ba
                 "faiss_mimic_sample1000_multi-qa"],
    "mini-lm": ["all-MiniLM-L6-v2", "sentence-transformers/all-MiniLM-L6-v2",
                "faiss_mimic_sample1000_mini-lm"],
    "BioBERT": ["BioBERT-mnli-snli-scinli-scitail-mednli-stsb",
                "pritamdeka/BioBERT-mnli-snli-scinli-scitail-mednli-stsb",
                "faiss_mimic_sample1000_BioBERT"],
    # ... additional model configurations
}

llms = {
    "deepseek": "deepseek-r1:1.5b",
    "qwen": "qwen3:1.7b",
    "llama": "llama3.2:latest",
    "gemma": "gemma:2b",
    "phi3": "phi3:3.8b",
    "tinyllama": "tinyllama:1.1b",
}
```

A.6 | Data Processing and Synthetic Data Generation

A.6.1 | Synthetic Data Generation for Public Deployment

```
# synthetic_data/synthetic_data_generator.py (key methods)
class SyntheticDataGenerator:
    """Generate realistic synthetic clinical data matching MIMIC-IV structure"""

    def __init__(self, num_patients=100, num_admissions=150):
        self.num_patients = num_patients
        self.num_admissions = num_admissions
        self.fake = Faker()

    def generate_complete_dataset(self):
        """Generate complete synthetic dataset with all clinical tables"""
        # Generate base patient population
        patients = self._generate_patients()
        admissions = self._generate_admissions(patients)

        # Generate clinical data tables
        diagnoses = self._generate_diagnoses(admissions)
        procedures = self._generate_procedures(admissions)
        lab_events = self._generate_lab_events(admissions)
        prescriptions = self._generate_prescriptions(admissions)
        microbiology = self._generate_microbiology_events(admissions)

        return {
            'patients': patients,
            'admissions': admissions,
            'diagnoses': diagnoses,
            'procedures': procedures,
            'lab_events': lab_events,
            'prescriptions': prescriptions,
            'microbiology': microbiology
        }

    def _generate_realistic_lab_values(self, admission_type):
        """Generate realistic lab values based on admission context"""

        normal_ranges = {
            'Hemoglobin': (12.0, 16.0),
            'White Blood Cell Count': (4.5, 11.0),
            'Platelet Count': (150, 450),
            'Creatinine': (0.6, 1.2),
            'Glucose': (70, 110),
            'Sodium': (136, 145),
            'Potassium': (3.5, 5.0)
        }

        labs = []
        for lab_name, (min_val, max_val) in normal_ranges.items():
            # Introduce abnormalities based on admission type
            if admission_type == 'EMERGENCY':
                # Higher chance of abnormal values
                if random.random() < 0.4:
                    if random.random() < 0.5:
                        value = random.uniform(min_val * 0.7, min_val * 0.9)
                    else:
                        value = random.uniform(max_val * 1.1, max_val * 1.3)
                else:
                    value = random.uniform(min_val, max_val)
            else:
                # Mostly normal values
                value = random.uniform(min_val, max_val)

            labs.append({
                'itemid': hash(lab_name) % 10000,
                'label': lab_name,
                'value': round(value, 2),
                'valueuom': ('mg/dL' if 'Glucose' in lab_name or
                           'Creatinine' in lab_name else 'count'),
                'flag': ('abnormal' if value < min_val or value > max_val
                        else 'normal')
            })

        return labs
```

This comprehensive code appendix provides implementation details for all major components of the Clinical RAG system, demonstrating the technical depth and architectural decisions that enable the system's clinical capabilities while maintaining code quality and safety standards.

B

Dataset Samples and Clinical Data

This appendix presents sample data from the MIMIC-IV clinical database and demonstrates the structure and characteristics of the clinical text used in the RAG system.

B.1 | MIMIC-IV Database Overview

The Medical Information Mart for Intensive Care IV (MIMIC-IV) database provides the clinical foundation for this research. This section describes the database structure and the specific tables utilized.

B.1.1 | Database Schema

B.2 | Sample Clinical Records

B.2.1 | Admission Records

B.2.2 | Diagnostic Information

B.2.3 | Procedure Documentation

B.3 | Text Preprocessing Examples

B.3.1 | Raw Clinical Text

B.3.2 | Processed and Chunked Documents

B.4 | Data Statistics

B.4.1 | Dataset Characteristics

Statistics about the clinical text corpus used in the RAG system:

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

C

Evaluation Metrics and Results

This appendix provides detailed evaluation results, performance metrics, and statistical analysis of the Clinical RAG Chatbot system.

C.1 | Evaluation Framework

C.1.1 | Metrics Definition

Comprehensive evaluation metrics used to assess the RAG system:

C.1.2 | Test Dataset

C.2 | Quantitative Results

C.2.1 | Retrieval Performance

C.2.2 | Response Quality Metrics

C.3 | Qualitative Analysis

C.3.1 | Expert Evaluation

C.3.2 | User Study Results

C.4 | Comparative Analysis

C.4.1 | Baseline Comparisons

C.4.2 | Ablation Studies

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

D

User Interface Screenshots

This appendix contains screenshots and visual examples of the Clinical RAG Chatbot user interface, demonstrating the system's functionality and user experience.

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like "Huardest gefburn"? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

E

Installation and Setup Guide

This appendix provides detailed instructions for installing, configuring, and running the Clinical RAG Chatbot system.

E.1 | System Requirements

E.2 | Installation Steps

E.3 | Configuration

E.4 | Running the System

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.