
# UNIVERSITÀ DI PISA

MASTER'S DEGREE IN ARTIFICIAL INTELLIGENCE AND DATA ENGINEERING

## Business and Project Management

***LLM-Powered RAG system for Legal Document Assistant Using Legal Case Document Summarization Dataset***

Submitted To:

Prof. Andrea Bonaccorsi

By:

Tsegay Teklay Gebrelibanos
ID Number: 683925

- Source Code is available on: *my GitHub repository*

Academic Year 2024/2025

# Contents

**1 Introduction** **1**

1.1 Background: The Evolution and Limitations of LLMs . . . . . 1

1.2 Retrieval Augmented Generation (RAG) as a Solution . . . . . 1

1.3 Related work . . . . . 2

1.4 Project Objectives . . . . . 3

1.5 Use Case: Legal Document Analysis . . . . . 4

**2 Retrieval Augmented Generation Components** **5**

2.1 Core RAG Components . . . . . 5

2.1.1 Large Language Model (LLM) . . . . . 5

2.1.2 Embedding Models . . . . . 5

2.1.3 Vector Database or Search Engine with Vector Capabilities 5

2.2 RAG Pipeline . . . . . 6

**3 System Design and Implementation** **7**

3.1 Overall Microservices Architecture . . . . . 7

3.1.1 Role of Each Service . . . . . 8

3.1.2 Data Flow and Inter-Service Communication . . . . . 8

3.2 Technology Stack for Implementation . . . . . 10

3.2.1 Docker Compose Orchestration . . . . . 10

3.3 Setup and Deploy the RAG System Assistant . . . . . 10

**4 Result and Evaluation** **11**

4.1 Result . . . . . 11

4.1.1 ChatBot . . . . . 11

4.1.2 Dashboard For monitoring and Evaluation . . . . . 12

4.1.2.1 Operational Performance evaluation . . . . . 12

4.1.2.2 User Feedback evaluation . . . . . 12

5 Conclusion 14

References 15

# Section 1: Introduction

## 1.1 Background: The Evolution and Limitations of LLMs

The past few years have witnessed a transformative acceleration in the field of Artificial Intelligence, largely driven by the advent and widespread adoption of Large Language Models (LLMs). Models such as those developed by OpenAI (e.g., GPT series), Google (e.g., LaMDA, PaLM, Gemini), and Anthropic (e.g., Claude) have demonstrated remarkable capabilities across a wide spectrum of natural language processing tasks, including text generation, summarization, translation, and question answering. Trained on colossal datasets comprising trillions of words from the internet and various digital texts, these models exhibit a sophisticated understanding of language patterns, grammar, and a vast amount of general knowledge embedded within their training data. Their ability to generate coherent, contextually relevant, and often creative text has paved the way for numerous innovative applications, from content creation assistance and code generation to sophisticated conversational agents.

**Despite their impressive capabilities,** traditional LLMs possess significant limitations, particularly when applied to use cases requiring access to specific, up-to-date, or proprietary information. A primary challenge is their static nature; their knowledge is limited to the data they were trained on. Consequently, they cannot provide information about recent events, internal company documents, or domain-specific details that were not part of their training corpus. Furthermore, while often fluent and convincing, LLMs can sometimes generate plausible-sounding but entirely false or misleading information, a phenomenon known as **"hallucination"**. This lack of guaranteed factual accuracy makes their direct application risky in domains where precision and verifiability are paramount, such as legal analysis, medical diagnosis, or financial advising. Addressing these limitations is crucial for extending the utility of LLMs into enterprise and specialized applications that rely on accurate, contextually grounded responses.

## 1.2 Retrieval Augmented Generation (RAG) as a Solution

Retrieval-Augmented Generation (RAG) has emerged as a powerful and effective paradigm to mitigate the limitations of traditional LLMs by enabling them to access and utilize external, up-to-date, or domain-specific knowledge. RAG systems combine the strengths of information retrieval systems with the generative capabilities of LLMs. Instead of solely relying on the knowledge encoded in their internal parameters, RAG models first retrieve relevant information from a designated external knowledge base based on the user's query. This retrieved information, often in the form of text snippets or documents, is then provided to the LLM as context alongside the original query. The LLM is instructed to generate a response that is grounded only in this provided context.

*This two-step process—Retrieval followed by Generation—offers several key advantages:*

- ***Reduced Hallucination:*** By forcing the LLM to base its answer on provided facts, RAG significantly reduces the likelihood of generating false information.
- ***Access to Up-to-Date Information:*** The external knowledge base can be continuously updated, allowing the RAG system to answer questions about recent events or newly added documents, which a static LLM cannot do.
- ***Domain Specificity:*** The knowledge base can be populated with domain-specific data (e.g., legal documents, medical literature, company reports), enabling the LLM to provide expert-level answers within that domain.
- ***Interpretability and Citation:*** By citing the retrieved documents used as context, RAG systems can provide transparency into the source of the information, allowing users to verify the response.
- ***Reduced Reliance on LLM Parameters:*** Less need to encode all world knowledge into the LLM's weights, making models smaller and potentially more efficient for specific tasks.

RAG has become a standard and effective architecture for building reliable and accurate question-answering systems over large, dynamic, or proprietary datasets.

## 1.3 Related work

The integration of Artificial Intelligence (AI) in the legal domain has accelerated with the emergence of Large Language Models (LLMs), which exhibit remarkable capabilities in natural language understanding and generation. Legal AI applications span document summarization, legal question answering, judgment prediction, and contract analysis[1]. Despite their utility, traditional LLMs suffer from limitations such as hallucination, static knowledge, and lack of domain-specific context—issues that are particularly critical in legal settings.

To overcome these limitations, the **Retrieval-Augmented Generation (RAG)** paradigm was introduced by Lewis et al. (2020), combining the reasoning abilities of LLMs with external document retrieval mechanisms. This architecture significantly enhances output accuracy and factuality by grounding the generated text in retrieved, up-to-date information. RAG has proven especially effective in tasks requiring long-context understanding and legal document referencing[2].

Recent studies have validated RAG's effectiveness in **legal applications**. Schumann (2023) demonstrated that integrating RAG with LLMs like LLaMA 2, along with advanced prompting and classification techniques, substantially improves performance in legal document analysis, clause detection, and chatbot interactions[1].Similarly, Amato et al. (2024) emphasized the role of RAG in Federated Search (FS) systems to securely retrieve legal information from distributed data sources while preserving data privacy[3].

Building on the RAG paradigm, Wang (2024) proposed a **Knowledge Graph-enhanced RAG (KG-RAG)** framework for contract review. By embedding structured legal knowledge, the system achieved improved accuracy in identifying contractual entities, relationships, and cross-references. This approach enables deeper contextual understanding and better generalization, especially in complex legal documents like agreements and compliance reports[2].

While promising, these RAG-based systems face several challenges, including legal data heterogeneity, jurisdictional variations, and maintaining compliance with data protection regulations like GDPR. Additionally, system scalability, response interpretability, and continuous performance monitoring are areas requiring further research. Approaches like real-time feedback collection, microservices orchestration (e.g., via Docker Compose), and continuous learning loops are being adopted to address these issues and enhance system robustness.

This project builds upon this body of work by focusing on the design and implementation of a robust, modular RAG system for legal documents using a microservices architecture, integrating real-time monitoring and user feedback to provide a practical prototype and insights into operational performance and user-perceived quality.

## 1.4 Project Objectives

The primary objectives of this project is to design and implement a functional Retrieval-Augmented Generation (RAG) system specifically tailored for assisting with legal document analysis. The goals encompassed both architectural design principles and the realization of a working prototype with essential features. The key objectives are:

- To design a scalable and maintainable RAG system architecture utilizing a microservices approach.
- To implement a functional prototype of the designed RAG system, integrating various components for data ingestion, document search and retrieval, and language model-based text generation.
- To leverage specific technologies, including Elasticsearch for efficient document indexing and retrieval, and the Google Gemini API for advanced text generation.
- To develop a user-friendly interface that allows users to easily interact with the RAG system by submitting queries and viewing the generated answers and their sources.
- To implement a system for real-time monitoring of the RAG system's operational performance and health, utilizing tools like Prometheus and Grafana.
- To incorporate a mechanism for collecting explicit user feedback on the quality of the generated responses as a method for evaluating perceived performance.
- To containerize the entire system using Docker Compose to facilitate ease of setup, deployment, and management of the individual microservices.

## 1.5 Use Case: Legal Document Analysis

This project focuses on developing an LLM-Powered RAG system designed to function as a Legal Document Assistant. The core task this assistant aims to facilitate is providing accurate and contextually relevant answers to user queries based on a defined set of legal documents.

For the implementation and testing of this RAG system prototype, a dataset specifically curated for legal document analysis was utilized. The dataset, titled "*Legal Case Document Summarization Dataset*[1]", comprises case documents and their corresponding summaries from both the Indian and United Kingdom Supreme Courts. This dataset includes different types of summaries[2]:

- *Abstractive Summaries:* These summaries, obtained from public online sources (e.g., liiofindia.org for Indian cases, supremecourt.uk for UK cases), represent condensed versions of the full case documents, capturing the main points in a new narrative.
- *Extractive Summaries:* For a subset of Indian Supreme Court cases, expert-written extractive summaries are provided. These summaries consist of sentences or phrases directly extracted from the original case document, representing key segments identified by legal experts. The dataset includes both full extractive summaries and segment-wise summaries broken down by categories such as 'analysis', 'argument', 'facts', 'judgement', and 'statute'.

The dataset's focus on legal case documents provides a realistic and challenging basis for building a RAG system. The cases cover various legal aspects, and the availability of both full case texts and different types of summaries is valuable. For the purpose of indexing in this RAG system, the **full case documents** themselves serve as the primary corpus from which relevant information is retrieved to answer user queries. The characteristics of this dataset, including the length and complexity of the case documents, directly influence the design choices for document processing (chunking), indexing, and retrieval within the RAG pipeline. The goal is for the Legal Document Assistant to effectively leverage this rich source of legal information to provide grounded responses.

[1] The dataset of Legal Case Document Summarization available here
[2] Find the detailed decription here

# Section 2: Retrieval Augmented Generation Components

Building a RAG application involves the orchestration of several key components that work together to retrieve relevant information and augment the LLM's generation process.

## 2.1 Core RAG Components

A typical RAG system is comprised of several interconnected core components, each with a distinct role in facilitating the retrieval and generation of grounded responses.

### 2.1.1 Large Language Model (LLM)

The Large Language Model serves as the generative engine within the RAG framework. While LLMs possess vast internal knowledge acquired during their training on massive datasets, their role in a RAG system is specifically focused on the generation phase. The LLM receives a user query that has been augmented with relevant context retrieved from an external knowledge source. Its task is to synthesize this information, combining the retrieved facts with its language capabilities to produce a coherent, natural-sounding, and contextually relevant response.

### 2.1.2 Embedding Models

Embedding models are fundamental to the retrieval process in RAG systems. Their function is to convert text, whether it's a document chunk from the knowledge base or a user's query, into a numerical representation called a vector embedding. These embeddings are high-dimensional vectors that capture the semantic meaning and contextual relationships of the text. The choice of an appropriate embedding model is crucial as it directly impacts the accuracy and relevance of the retrieval process – if the embeddings don't effectively capture semantic similarity, the retrieval stage will fail to find the most relevant information.

### 2.1.3 Vector Database or Search Engine with Vector Capabilities

The vector database, or a search engine with integrated vector search capabilities, serves as the knowledge source for the RAG system. Its primary role is to store the vector embeddings generated by the embedding model for the entire corpus of documents. Alongside the embeddings, it also stores the original text chunks and any associated metadata (like source file name, page number, section). Scalability, search speed, relevance ranking algorithms, and the ability to handle metadata filtering are important factors when choosing a vector database or search engine for a RAG system.

## 2.2 RAG Pipeline

The components described above are orchestrated within a multi-stage pipeline that defines the flow of data and processes from the initial documents to the final generated response (Figure 2.1). The core RAG pipeline typically involves the following conceptual stages:

**Data Ingestion:**This is the initial stage where the external knowledge source is prepared. It involves loading the raw documents from a corpus, processing them by splitting them into smaller, manageable chunks (to fit within LLM context windows and improve retrieval granularity), generating vector embeddings for each of these chunks using an embedding model, and indexing the chunks and their embeddings into the vector database or search engine. This process effectively transforms unstructured text data into a structured, searchable knowledge base. The efficiency and accuracy of the data ingestion pipeline significantly impact the overall performance of the RAG system.

**Retrieval:** This stage is triggered by a user query. Upon receiving a query, it is first processed and converted into a vector embedding using the same embedding model used during data ingestion. This query embedding is then used to perform a similarity search in the vector database or search engine. The system retrieves a set of the top K[1] most relevant document chunks whose embeddings are closest to the query embedding.

**Augmentation and Generation:** In this final stage, the retrieved document chunks are combined with the user's original query to create an augmented prompt. This augmented prompt provides the LLM with the necessary context to formulate a grounded response. The LLM then processes this augmented prompt and generates a natural language answer. The quality of the generated answer is evaluated based on its accuracy, relevance, coherence, and how well it addresses the user's query using the provided context.

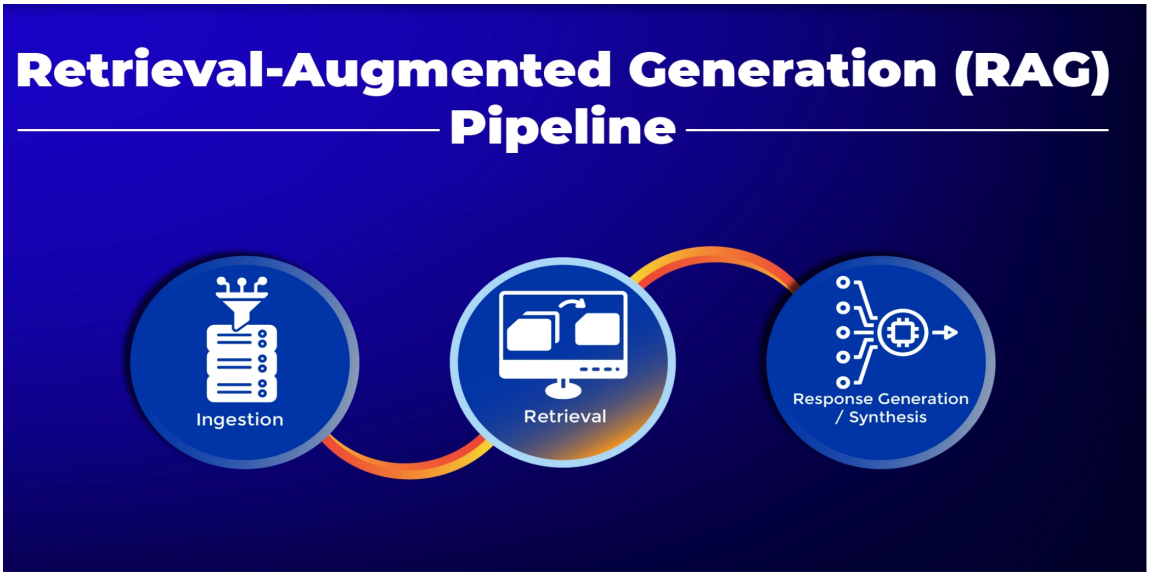


Figure 2.1
Source: www.projectpro.io

[1]In this project, it is designed to retrieve five relevant documents for each query, which serve as references to the Generated answer

# Section 3: System Design and Implementation

This section outlines the overall microservices architecture, describes the role of each service, explains the data flow and inter-service communication, and details the technology stack and orchestration using Docker Compose.

## 3.1 Overall Microservices Architecture

To achieve a modular, scalable, and maintainable RAG system, a microservices architecture was designed. This approach breaks down the complex system into smaller, independent services, each responsible for a specific function. The rationale for choosing microservices is to decouple the components involved in the RAG pipeline and supporting functions, allowing for independent development, deployment, and scaling of each service based on demand. For example, the LLM inference service can be scaled separately from the document indexing service, or the user interface can be updated without affecting the core RAG logic.

The designed architecture consists of several distinct microservices that interact to fulfill the RAG process and provide monitoring capabilities. These services are containerized using Docker, and their orchestration is managed by Docker Compose. The key services and their flow are depicted in Figure 3.1:

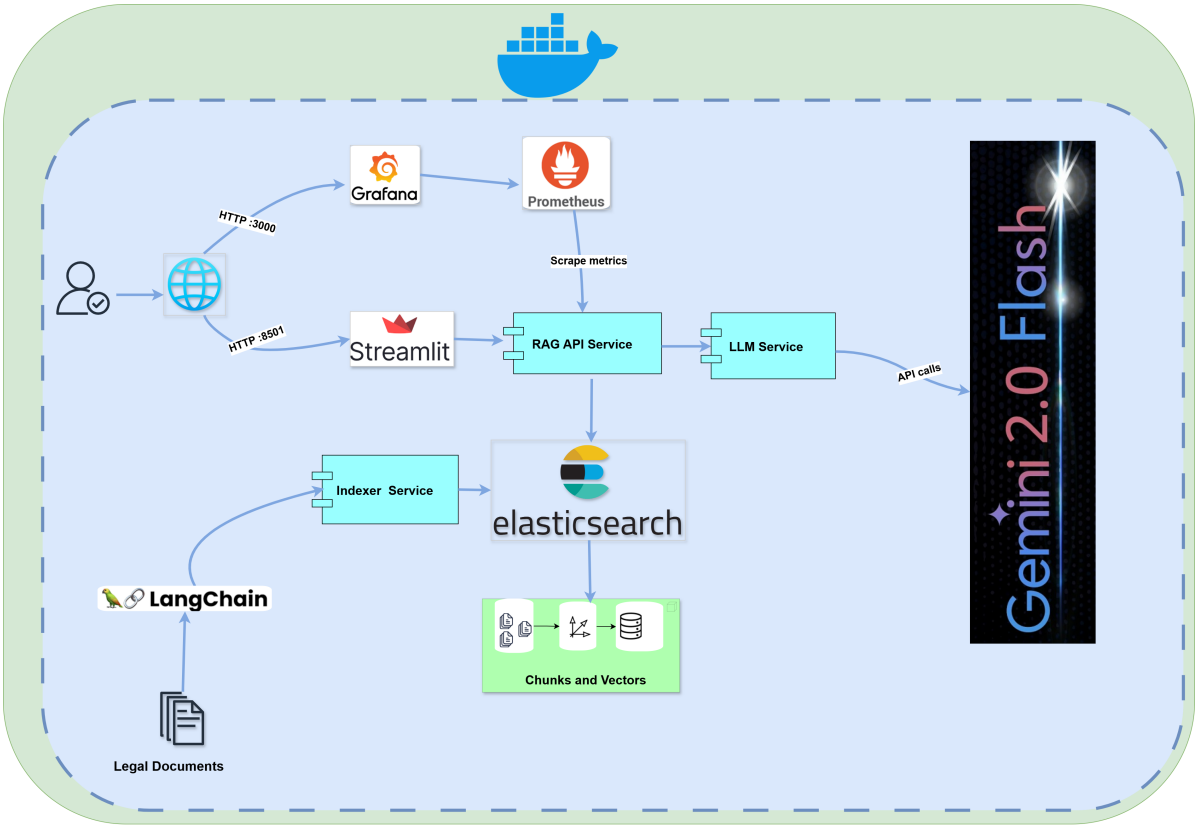


Figure 3.1: Proposed microservices architecture for RAG system

### 3.1.1 Role of Each Service

Within the designed microservices architecture, each service plays a crucial and well-defined role:

*The Indexer Service*: This service is designed as a batch processing job[1]. Its primary responsibility is to take the raw legal documents from a designated source (./docs folder), perform the necessary preprocessing (loading, splitting), generate vector embeddings for the document chunks, and load this data into the search engine (Elasticsearch).

*Elasticsearch:* This service serves as the persistent knowledge base and the core retrieval engine. It stores the indexed document chunks and their vector embeddings. It is responsible for efficiently executing similarity search queries based on the vector representations of user questions and returning the most relevant document chunks.

*The LLM Service:* This service acts as a dedicated wrapper around the external Large Language Model API (Google Gemini[2]). Its function is to receive text generation requests (containing the augmented prompt) from the RAG API service, interface with the external LLM API, and return the generated text response. This service isolates the RAG API from the specifics of the LLM provider.

*The RAG API Service:* This is the central orchestration layer of the system. It receives user queries from the Streamlit UI, coordinates the retrieval process by querying Elasticsearch, formats the prompt by augmenting the user query with the retrieved document chunks, calls the LLM Service to get the generated answer, and returns the final response (answer and sources[3]) back to the UI. This service also collects and exposes operational metrics for monitoring.

*The Streamlit UI:* This service provides the user-facing web application. It is responsible for presenting the interface for users to input their legal questions, submit queries to the RAG API service, display the generated answers and sources, and collect user feedback.

*Prometheus:* This service is the time-series database used for monitoring. It is configured to periodically scrape (pull) operational metrics (like query counts, latency, error rates) exposed by the RAG API Service. It stores this historical data for analysis.

*Grafana:* This service is the visualization tool. It connects to Prometheus as a data source and allows for the creation of interactive dashboards to display and analyze the collected operational metrics and user feedback data.

### 3.1.2 Data Flow and Inter-Service Communication

The services in the architecture communicate and exchange data to execute the RAG pipeline and monitoring processes. The primary communication method between most services is HTTP, facilitated by Docker's internal network.

**Indexing Data Flow:** Raw documents from the ./docs[4] are processed by the Indexer Service. The processed data (chunks, embeddings, metadata) is sent to Elasticsearch via

[1] this is job is done by the Elasticsearch client, and it is a one-time job
[2] Gemini-flash-2.0
[3] five relevant documents from the corpus as references
[4] the corpus/legal documents that stored inside the root directory ./docs/dataset

its client API for indexing as shown in Figure 3.2

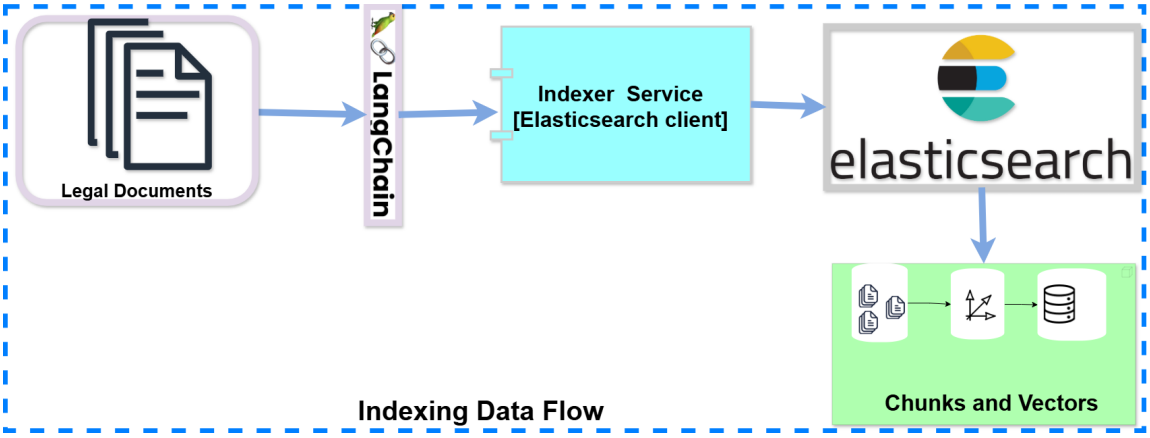


Figure 3.2

**Query Data Flow:** A user query originates from the Streamlit UI. It is sent via HTTP to the RAG API Service. The RAG API Service sends a search request to Elasticsearch using its client API. Elasticsearch returns relevant data. The RAG API Service then sends an HTTP request with the augmented prompt to the LLM Service. The LLM Service communicates with the external Google Gemini API. The generated response is returned from LLM Service to RAG API Service via HTTP, and finally from RAG API Service back to Streamlit UI via HTTP as shown in Figure 3.3

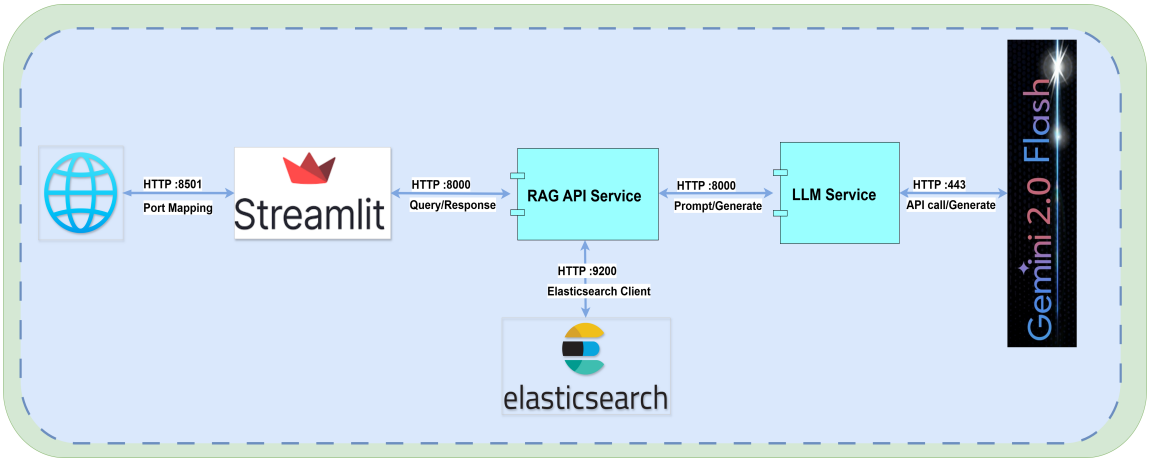


Figure 3.3: Query Data Flow

**User Feedback Flow:** User feedback from the Streamlit UI is sent via HTTP to a dedicated endpoint on the RAG API Service.

**Monitoring Data Flow:** The RAG API Service exposes operational metrics via an HTTP endpoint (/metrics). Prometheus periodically scrapes this endpoint via HTTP. Grafana queries the data stored in Prometheus via HTTP using the Prometheus data source.

## 3.2 Technology Stack for Implementation

The designed microservices architecture has been implemented using a specific set of technologies listed in Table 3.1 has chosen for their capabilities, compatibility, and suitability for the system.

Table 3.1: Technology Stack per Service

| Service | Role | Key Technologies Used |
|---|---|---|
| `indexer_service` | Data Ingestion | Python, Sentence Transformers, Elasticsearch Client, pypdf |
| `elasticsearch` | Search Engine/Vector Store | Elasticsearch (Docker Image) |
| `llm_service` | LLM Wrapper (Gemini) | Python, FastAPI, google-generativeai, Uvicorn, Requests |
| `rag_api_service` | RAG Orchestrator API | Python, FastAPI, Elasticsearch Client, Sentence Transformers, Requests, prometheus_client, Uvicorn |
| `streamlit_ui` | User Interface | Python, Streamlit, Requests |
| `prometheus` | Metrics Collection | Prometheus (Docker Image) |
| `grafana` | Metrics Visualization | Grafana (Docker Image) |

### 3.2.1 Docker Compose Orchestration

Docker Compose is utilized as the primary tool for orchestrating the microservices. The docker-compose.yml[5] file defines each service as a container, specifies the Dockerfile or image to use, configures port mappings for external access (UI, monitoring), sets up volume mounts for persistence (Elasticsearch, Prometheus, Grafana data) and data access (source documents), passes environment variables to containers (for configuration and API keys), and manages service dependencies to ensure containers start in the correct order.

## 3.3 Setup and Deploy the RAG System Assistant

The instructions for setting up and deploying the implemented RAG system from the source code repository need step-by-step instructions for cloning the repo, setting up the .env file, building the images, starting the services with docker compose up -d, and running the indexer_service job. This is similar to the Setup and Running sections in the repository's README[6] file.

[5] Check Docker Compose configurations here
[6] README: the setup and deployment guidance is available here

# Section 4: Result and Evaluation

This section presents the outcome of the project's implementation – a functional LLM-Powered RAG system for legal document analysis – and details the methodology employed for its evaluation.

## 4.1 Result

The implementation of the microservices architecture orchestrated by Docker Compose resulted in a working prototype of the Legal Document RAG Assistant. This prototype is accessible via a web-based user interface and includes an integrated monitoring dashboard.

### 4.1.1 ChatBot

The core functionality of the implemented system is presented through a web-based chatbot interface built using Streamlit. This interface serves as the primary point of interaction for users to engage with the Legal Document Assistant. Users can input their legal questions into a text area and submit them for processing. Upon submission, the system processes the query, retrieves relevant information from the indexed legal documents, and generates a natural language answer. The interface then displays the generated answer along with the sources (document snippets with source file and page information) that were used to construct the response. Furthermore, the interface includes a mechanism for users to provide explicit feedback on the helpfulness of the received answer through "Satisfied" and "Unsatisfied" buttons.

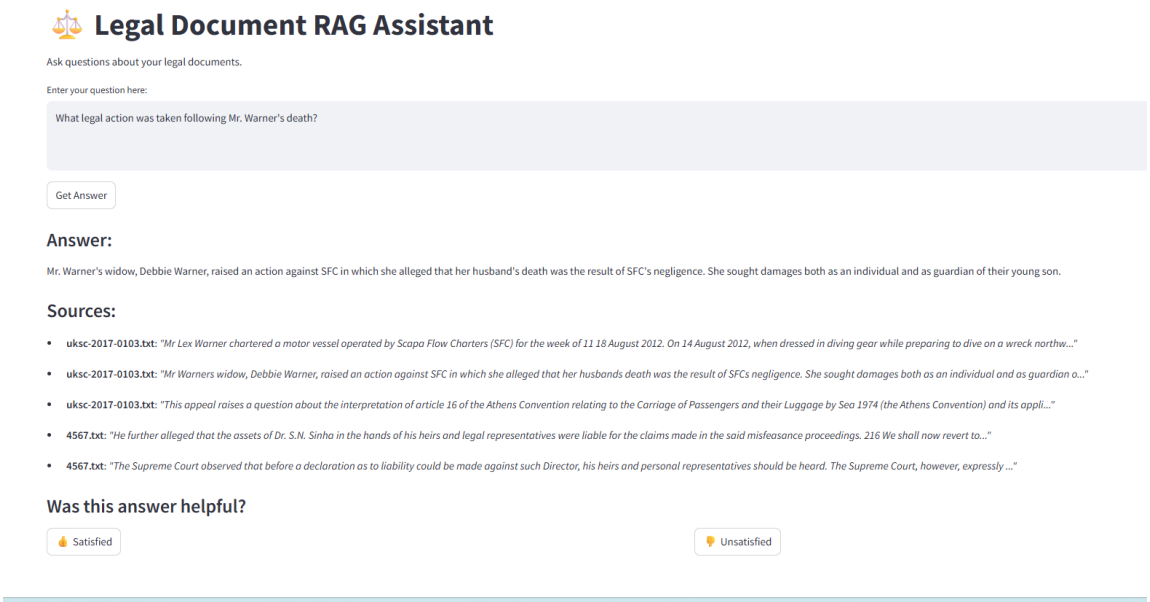


Figure 4.1: Streamlit User Interface (ChatBot Component)

### 4.1.2 Dashboard For monitoring and Evaluation

In addition to the chatbot interface, the implemented system includes a monitoring dashboard that provides real-time visibility into the system's operational performance and user engagement. This dashboard is built using Grafana and visualizes metrics collected by Prometheus from the RAG API service. The dashboard provides insights into aspects such as query volume, error rates, and user feedback submissions.


Figure 4.2: Monitoring Dashboard (Grafana)

#### 4.1.2.1 Operational Performance evaluation

Operational performance and health monitoring were central to the evaluation of the implemented system. Using Prometheus for time-series data collection and Grafana for visualization, key operational metrics were tracked in real-time. The RAG API service is instrumented with the prometheus_client library to expose these metrics. Prometheus is configured to scrape these metrics periodically, storing the time-series data. Grafana dashboards were created to visualize these metrics in real-time, providing insights into the system's behavior under load and its overall stability. The primary operational metrics monitored include Query Throughput (Rate), Query Latency (Percentiles - e.g., 95th, 99th percentile), Error Rates (Overall Server, LLM Service Call, Elasticsearch Call), and Active Requests (Concurrency). By analyzing these metrics visualized in the Grafana dashboard Figure 4.2, the evaluation allows for a quantitative assessment of the system's performance, helping to identify areas for optimization and verify its stability.

#### 4.1.2.2 User Feedback evaluation

To gather insights into the user's perception of the RAG system's output quality, a user feedback mechanism was implemented in the Streamlit UI. After receiving an answer, users can indicate whether they were "Satisfied" or "Unsatisfied" with the response. This feedback is captured as Prometheus metrics, specifically incrementing the

rag_feedback_total counter with a label indicating the feedback type ("satisfied" or "unsatisfied"). The collected feedback metrics are visualized in the Grafana dashboard (Figure 4.2), allowing for tracking the total number of feedback submissions, the distribution of feedback types (satisfied vs. unsatisfied), and the calculation of an overall satisfaction rate over time. This provides a direct, albeit subjective, measure of the system's effectiveness in providing helpful answers from the end-user's perspective and helps guide future improvements.

# Section 5: Conclusion

In conclusion, this project successfully addressed the critical need for accurate and grounded Large Language Model (LLM) responses in the specialized domain of legal document analysis by designing and implementing a functional Retrieval-Augmented Generation (RAG) system. A key achievement of this work is the adoption of a robust microservices architecture, orchestrated seamlessly with Docker Compose, which effectively decouples the complex RAG pipeline into independent, manageable services. This prototype demonstrates the power of integrating specialized components – including Elasticsearch for efficient vector-based retrieval and the Google Gemini API, accessed via a dedicated service, for sophisticated text generation – to deliver contextually relevant answers grounded in a specific legal document corpus presented through a user-friendly Streamlit interface. Furthermore, the implementation successfully incorporated a comprehensive real-time operational monitoring system using Prometheus and Grafana, providing crucial visibility into the system's performance, health, and usage, complemented by a user feedback mechanism that adds a vital dimension to the evaluation of perceived output quality. This project stands as a tangible proof of concept, validating the feasibility and benefits of a microservices approach to RAG in a demanding domain and establishing a strong foundation for future expansion and development towards more advanced and practical legal AI solutions.

# References

[1] Lorenzo Schumann. Employing retrieval augmented generation to optimize llms for the legal domain: Evaluating methods to improve chatbot performance. Master's thesis, Universidade NOVA de Lisboa (Portugal), 2024.

[2] Jiaqi Wang. Rainbow rag: An llm-powered rag system for contract review. 2024.

[3] Flora Amato, Egidia Cirillo, Mattia Fonisto, and Alberto Moccardi. Optimizing legal information access: Federated search and rag for secure ai-powered legal solutions. In *2024 IEEE International Conference on Big Data (BigData)*, pages 7632–7639. IEEE, 2024.

## Quick Links


- Go to Home
- Go to Introduction
- Go to RAG Components
- Go to Design & Implementation
- Go to Result and Evaluation
- Go to conclusions