



BACHELORARBEIT

AI – USING A LARGE LANGUAGE MODEL
(LLM) TOGETHER WITH
RETRIEVAL-AUGMENTED GENERATION (RAG)
FOR ANSWERING QUESTIONS ON CUSTOM DATA

Ivan Khomich

angestrebter akademischer Grad
Bachelor of Science (BSc)

Wien, 2025

Studienkennzahl lt. Studienblatt: UA 033 521

Fachrichtung: Informatik Allgemein

Betreuer: Dipl.-Ing. Dr.techn. Marian Lux

Contents

1	Motivation	4
1.1	Motivation and problem statement	4
1.2	Goals	4
1.3	Process and Methods	5
1.4	Related Work	5
1.4.1	Smaller Individual Works (Guides/Tutorials)	5
1.4.2	Comparison of RAG Implementations	5
1.4.3	Identified challenges	6
2	Foundational Concepts, Algorithms, and Data Processing	6
2.1	Retrieval-Augmented Generation (RAG)	6
2.2	Large Language Models (LLMs)	7
2.3	Ollama	7
2.4	Embeddings	8
2.5	Vector Database	8
2.6	Data Preparation	8
3	Implementation	9
3.1	UI	10
3.2	LlamaIndex ReActAgent	11
3.3	Document Retrieval and Storage	12
3.4	Data Ingestion Pipeline	12
4	GitHub Repository and Tests	12
4.1	Repository	12
4.2	Test Coverage	12
5	Results and Evaluation	14
5.1	Performance Evaluation	15
6	Lessons Learned	17
6.1	Better embeddings lead to better results	17
6.2	RAG Works best for facts, not deep understanding	17
6.3	The impact of configurations	17
6.4	Multi-agent systems and their challenges	17
6.5	Potential hallucinations from chat history	18
6.6	Final thoughts	18
7	Future Improvements	18
8	Conclusion	19

Abstract

This bachelor's thesis focuses on the development of a chatbot application that integrates a large language model (LLM) with Retrieval-Augmented Generation (RAG)[11] to provide accurate and contextually appropriate responses based on custom datasets. The objective is to develop a chatbot that can efficiently retrieve relevant information from uploaded documents and websites and use this data to generate accurate responses.

To achieve this, the project uses a local LLM (LLaMA 3.1) in combination with a retrieval mechanism and a vector database together with a web search and a math solving tools. This setup allows the LLM to access external documents and use the information it finds to improve the accuracy and relevance of its responses. By integrating RAG and websearch, the model is not limited by its pre-trained knowledge and can dynamically access specific, up-to-date content relevant to the user's query.

The system is implemented through a web-based interface developed using Streamlit[18], which allows users to interact with the chatbot. The interface allows users to ask questions, upload data, and clean the database.

The results of this implementation demonstrate the effectiveness of combining LLMs with RAG for handling knowledge-intensive tasks, particularly in environments where the data is custom or frequently updated. The thesis concludes with a discussion of the performance evaluation of the chatbot and exploring potential for further optimization and enhancement.

1 Motivation

1.1 Motivation and problem statement

In recent times, the use of large language models has moved beyond generic question-answering to more specific fields. However, standard LLMs often struggle to accurately respond to queries involving custom, domain-specific, or newly updated information without significant modifications. This is where Retrieval-Augmented Generation (RAG) proves useful. RAG integrates the capabilities of LLMs with a retrieval mechanism, allowing models to extract relevant information from a particular dataset to generate precise and contextually appropriate responses. The aim of this project is to create a chatbot that uses a local LLM, such as LLaMA or Mistral, together with RAG to provide answers based on uploaded documents or web content. While fine-tuning LLMs on specific datasets can partially address this issue, it is a time-consuming process and doesn't allow the model to adapt to newly introduced information in real time.

This project concentrates on running the LLM locally, which offers two distinct advantages: first, it eliminates the need for cloud-based services, making the system **free of ongoing costs**; second, it provides **enhanced privacy** by keeping all data processing local, ensuring that sensitive or proprietary information remains secure.

1.2 Goals

The primary goal of this project is to overcome the mentioned limitations by integrating a ReActAgent[12] using a local LLM with Retrieval-Augmented Generation. RAG enhances the generative capabilities of the LLM by including a retrieval mechanism that fetches relevant information from external sources, such as locally uploaded documents, web searches, or even math-solving processes, before producing a response. As a result, the system can deliver answers grounded in up-to-date or specialized knowledge without the need for continual retraining.

The success of this project can be measured through several metrics:

- **Accuracy:** The chatbot should generate responses that are factually correct and aligned with the information provided in external documents or real-time web data.
- **Efficiency:** The system should handle large datasets and provide quick, relevant responses.
- **Adaptability:** The model should be able to handle new and changing datasets without needing retraining, making it suitable for cases where information is constantly being updated.

It should be noted that, in the context of a retrieval-augmented generation (RAG) system, the primary objective is to perform queries that target specific segments of documents. This approach focuses on extracting particular details

without considering the entire document and does not extend to executing more complex tasks.

1.3 Process and Methods

To achieve these objectives, the chatbot integrates a local LLM with a retrieval mechanism that draws on a vector database, supported by an external web search tool and a math solver. The vector database facilitates fast and relevant extraction of text segments from uploaded files or online sources. Rather than relying solely on the LLM’s pre-existing knowledge, the system grounds its responses in the most pertinent external data it retrieves. A user-friendly Streamlit interface provides the user interface, enabling users to upload documents, ask questions, and obtain answers, creating a platform for both data ingestion and communication.

1.4 Related Work

Many public repositories and articles explore the concept of RAG, with the majority serving as introductory guides rather than providing advanced solutions.

Existing works can generally be categorized into smaller individual projects and more comprehensive RAG frameworks.

1.4.1 Smaller Individual Works (Guides/Tutorials)

Many open-source projects and blog articles focus on demonstrating the fundamental use of RAG with locally running LLMs. These implementations are often toy examples or tutorials aimed at familiarizing users with the concept rather than offering robust, production-ready solutions. Common limitations include:

- Lack of robust graphical user interfaces (GUIs).
- Restriction to a single dataset without scalable multi-source retrieval.
- Over-reliance on default prompts, leading to suboptimal retrieval accuracy.

1.4.2 Comparison of RAG Implementations

Several established RAG implementations attempt to address these issues to varying degrees. Table 1 provides a comparative analysis of widely used RAG systems.

As shown in the table, existing solutions present various trade-offs. **We-bGPT** [14] integrates web search but suffers from hallucinations and unreliable sources. **Haystack** [2] allows multi-dataset retrieval and scalability, yet its efficiency drops significantly on large-scale retrieval tasks. **LangChain** [5] is widely used due to its ease of implementation but lacks advanced text-processing mechanisms.

Approach	Strengths	Weaknesses
WebGPT [14]	High accuracy, integrates web search	Prone to hallucinations, retrieval of low-quality sources
Haystack [2]	Handles multiple datasets, scalable	Performance issues on large datasets
LangChain [5]	User-friendly, widely adopted	Poor text splitting, simplistic prompt engineering

Table 1: Comparison of Different RAG Implementations

1.4.3 Identified challenges

The common challenges in existing RAG implementations can be categorized as follows:

- **Insufficient retrieval accuracy:** Many implementations rely on default embeddings and naive similarity search algorithms, which impact retrieval precision.
- **Inability to handle large datasets efficiently:** Some frameworks support large-scale retrieval but suffer from performance bottlenecks.
- **Limited dataset integration:** Most existing solutions focus on a single dataset rather than providing a scalable multi-source retrieval pipeline.
- **Absence of user-friendly interfaces:** Many open-source implementations lack an intuitive GUI, making them less accessible for non-technical users.

The root causes of these problems often stem from suboptimal document parsing, poor-quality embeddings, and inefficient similarity search techniques. For instance, the choice of embedding model significantly influences retrieval performance, while the selection of the similarity search method affects both efficiency and relevance.

2 Foundational Concepts, Algorithms, and Data Processing

2.1 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) couples the text-generation capabilities of a large language model with a document-retrieval component. When a user submits a query, the system retrieves pertinent information from a data store and presents it as supporting context for the language model’s response. This strategy yields more accurate and domain-specific answers, as the model can directly reference relevant external materials.

In this project's updated approach, a ReAct agent from the LlamaIndex framework is introduced to enhance the RAG pipeline. The ReAct agent dynamically decides how to solve a query, employing tools such as a web search or RAG-search (for up-to-date information) and a specialized math solver (for computational tasks). This agent-based mechanism enables a step-by-step reasoning process to produce more robust answers.

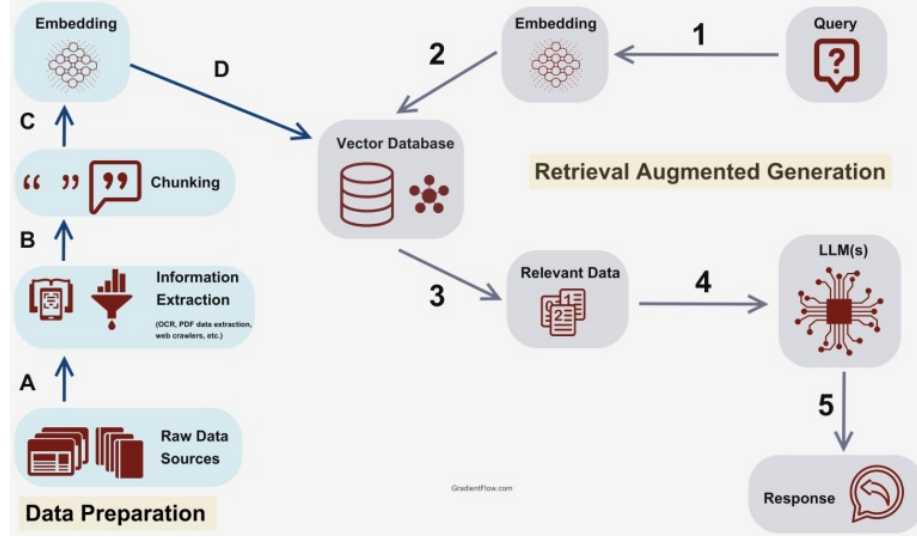


Figure 1: RAG workflow [6]

2.2 Large Language Models (LLMs)

Large Language Models (LLMs), such as Llama, are high-capacity neural networks trained on extensive text corpora. They excel at generating human-like text and interpreting intricate queries. In this implementation, Llama serves as the primary model, augmented by the LlamaIndex ReAct agent and the retrieval mechanism, to deliver precise and context-rich responses. The agent-based integration allows the model to invoke external tools, including web lookups and math solvers, as needed.

2.3 Ollama

Ollama is an open-source platform designed to run LLMs locally, offering the following benefits [1]:

- Free and open-source, minimizing license constraints
- Ensures full control over local data

- Operates offline, enhancing data privacy and accessibility
- Streamlines integration with various software systems

2.4 Embeddings

Embeddings in Large Language Models (LLMs) are high-dimensional vectors that capture the semantic context and relationships between tokens, enabling models to understand and process data like text, images, and audio. These embeddings enrich tokenized data with meaning, allowing LLMs to comprehend nuances, context, and patterns, essential for tasks such as text generation, image recognition, and audio analysis.[3]

This project utilizes bge-base-en-v1.5 [7] as the embedding model.

2.5 Vector Database

A vector database is a type of database system designed to efficiently store, index, and query high-dimensional vector data. Unlike traditional relational databases, which organize data into tables based on rows and columns, vector databases focus on the geometric properties of the data they store. This allows them to perform operations like distance calculations and similarity searches much more efficiently, leveraging the mathematical properties of vector spaces.

ChromaDB, which is used in this project, is a Vector Database and plays a crucial role in applications that require handling high-dimensional vector data, such as large language models and semantic search engines operating on text data. Chroma DB stands out for its ability to store embeddings alongside associated metadata, facilitating advanced use cases beyond mere data storage. [15]

2.6 Data Preparation

Project data is drawn from uploaded documents (PDFs) and online sources. PDF files undergo text extraction via `PyPDFLoader`[8], and their contents are divided into smaller sections using the `RecursiveCharacterTextSplitter`[9]. This segmentation ensures that the model can efficiently handle the text chunks. Web content is retrieved through the `requests`[16] library, parsed with `BeautifulSoup`[17], and then similarly segmented.

Overall data ingestion is outlined in Figure 2. In the updated system, these prepared chunks are not only used for RAG-style retrieval but also made available to the ReAct agent, which can conduct targeted web searches or calculations where needed, resulting in accurate and comprehensive responses.

Ingestion

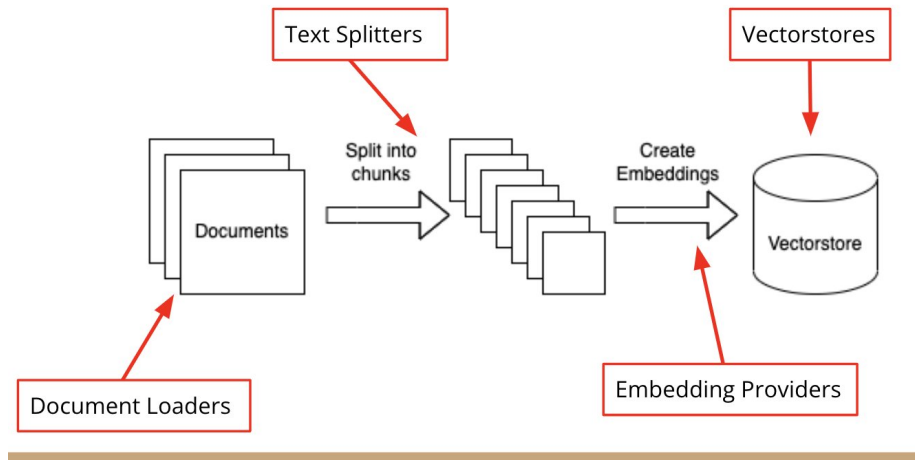


Figure 2: Data ingestion and storage process [10]

3 Implementation

The implementation consists of several core components, illustrated on the following diagram:

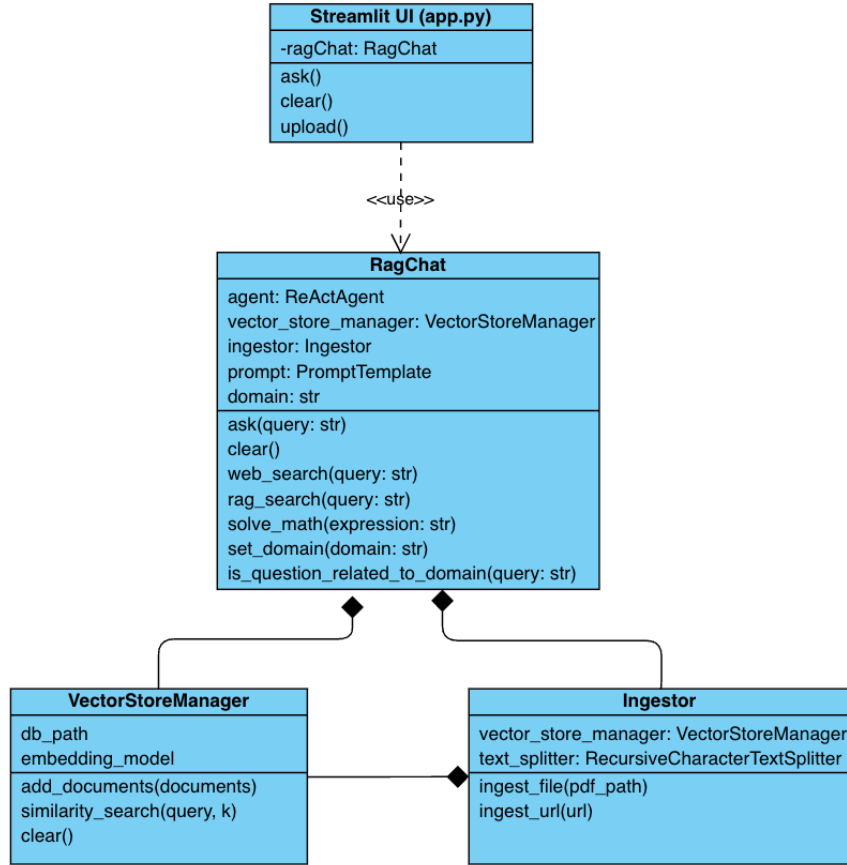


Figure 3: Class Diagram of the System

3.1 UI

The user interface is implemented using Streamlit, a framework for creating interactive web applications. Users can upload documents, input URLs for web content, and interact with the chatbot through a simple web-based interface and clear the database. The `RagChat` class processes these inputs, updates the vector store, and generates responses to user queries.

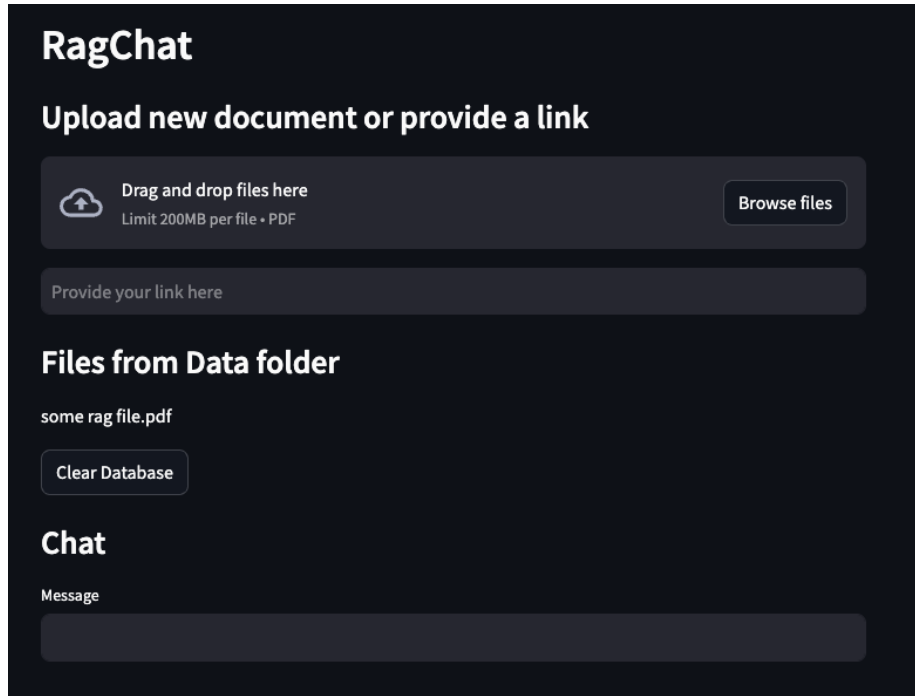


Figure 4: Streamlit UI

Upon starting, the app scans data folder and adds new files to its database.

3.2 LlamaIndex ReActAgent

In the revised architecture, the LlamaIndex ReActAgent plays the central role of orchestrating both retrieval operations and task-specific tool usage. The agent-based strategy enables dynamic querying of the knowledge base, real-time web searches, and mathematical computations. Here is the high-level process:

1. **User Query:** The user's question is passed to the LlamaIndex ReAct agent.
2. **Agent Reasoning:** The agent employs a step-by-step reasoning method:
 - If context from local documents is needed, the agent invokes retrieval from the underlying vector store.
 - If additional context is required, the agent leverages a web search tool.
 - If the query involves numerical analysis, the agent calls upon a dedicated math solver tool.

3. **Contextual Answer Generation:** The agent synthesizes the results into a concise and coherent response.

3.3 Document Retrieval and Storage

To facilitate retrieval, document embeddings are stored in a vector database (e.g., **Chroma**). Whenever the agent determines it needs local context to answer a question, it queries this store for the top matching chunks. The retrieved segments are then passed back into the LLM’s prompt, helping the model generate a context-aware reply.

3.4 Data Ingestion Pipeline

Instead of a standalone **Ingestor** class, the system now relies on a streamlined ingestion pipeline. This pipeline:

1. **Parses Documents:** Uses **PyPDFLoader** for PDFs and **requests** plus **BeautifulSoup** for webpages.
2. **Splits Text:** Breaks down large text blocks into smaller segments through **RecursiveCharacterTextSplitter**.
3. **Stores Embeddings:** Converts each text chunk into embeddings and stores them in the vector database.

This modular design allows the LlamaIndex ReAct agent to transparently access all relevant data—whether from PDFs, websites, or its own reasoning tools—during the query-response cycle.

4 GitHub Repository and Tests

4.1 Repository

The full implementation of this project is available in the following repository: <https://github.com/grippvh/ragChat>. The repository includes a detailed README file that provides step-by-step instructions on how to set up and use the application.

4.2 Test Coverage

The core functionality of this project revolves around Retrieval-Augmented Generation (RAG), which has been tested using a set of examples derived from the U.S. Constitution. Testing the outputs of a Large Language Model presents unique challenges due to the inherent variability in natural language responses. For instance, consider a simple query: "How many cents are there in a dollar?" Although the expected answer is "100," the model, being a language-based system, might generate a variety of correct responses, such as "a hundred" or "one

American dollar consists of 100 cents.”

To manage this variability, a two-step approach is employed:

1. The model is asked the target question.
2. The model is then prompted to validate its response using a pre-defined evaluation format. The validation is guided by the following prompt:

```
EVAL_PROMPT = """
Question: {question}
Expected Response: {expected_response}
Actual Response: {actual_response}
---
(Answer strictly with 'true' or 'false') Does the actual response
match the expected response?
"""
```

The function designed to perform this validation is as follows:

```
def query_and_validate(self, question: str, expected_response: str,
expect_success: bool = True):
    response_text = self.rag_chat.ask(question)

    print("\nQuestion:", question)
    print("Expected Response:", expected_response)

    prompt = EVAL_PROMPT.format(
        question=question,
        expected_response=expected_response,
        actual_response=response_text
    )

    evaluation_result = self.rag_chat.model.invoke(prompt)
    evaluation_result_cleaned = evaluation_result.content.strip().lower()

    is_success = "true" in evaluation_result_cleaned

    if is_success:
        print("\033[92m" + f"Response: {response_text}" + "\033[0m")
    else:
        print("\033[91m" + f"Response: {response_text}" + "\033[0m")

    if is_success:
        return True, response_text
    else:
        return False, response_text
```

To assess whether the model is overly lenient in evaluating its own responses, negative test cases have also been introduced. These tests are designed to determine if the model correctly identifies and flags incorrect answers. An example of such a test is provided below:

```
def test_incorrect_us_constitution_article_i_section_1(self):
    return self.query_and_validate(
        question="What powers are granted by Article I,
                 Section 1 of the U.S. Constitution?",
        expected_response="No power is granted to the President
                           of the United States."
    )
```

When dealing with Retrieval-Augmented Generation combined with a Large Language Model, it is important to note that there is no guarantee that the tests will yield the same result 100 % of the time. This inconsistency arises not only because LLMs can provide different responses to the same question depending on subtle changes in context, but also because the model's responses are influenced by the underlying probabilistic mechanisms. These mechanisms introduce a level of randomness, meaning that even identical queries can sometimes lead to different outputs.

5 Results and Evaluation

The performance of the chatbot was assessed based on its ability to accurately retrieve relevant context and use it to answer questions. Two primary observations arose during this evaluation:

- Initially, the task appeared straightforward.
- However, when diving into more advanced solutions and articles, it became evident that the implementation and optimization challenges were complex.

These challenges will be explained in more detail below. Initially, a simpler approach by focusing on implementing a basic version of RAG, often referred to as "naive RAG" [13], was implemented. The goal was to develop a functional solution to the problem without delving into more advanced techniques. Throughout this process, varying levels of success were encountered. When the chatbot produced satisfactory results for shorter documents, its performance tended to degrade when handling longer ones, and vice versa.

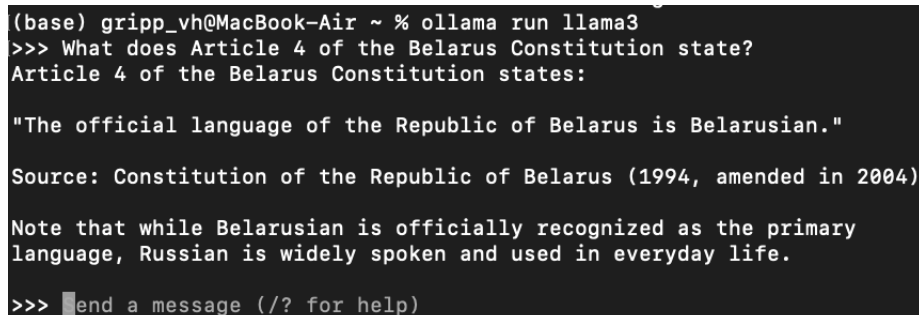
When working with LLMs, there are several factors that can influence these results, such as data parsers, prompt design, or similarity search parameters. In my case, the embedding function played a significant role, and improving this through iteration was crucial to achieving more consistent performance, finding suitable embeddings turned out to be a great improvement.

Interestingly, once the basic functionality was in place, more advanced techniques were explored, such as rephrasing user requests based on retrieved context, and incorporating re-ranking of responses. Unfortunately, these enhancements either degraded the results or, at best, maintained the current level of performance. This outcome may be due to my relative inexperience in this field, and it is likely that more experienced practitioners could further optimize these techniques. Nonetheless, the current state of the project serves as a foundation for future improvements, which will be further outlined.

5.1 Performance Evaluation

To demonstrate the effectiveness of our system, a test was conducted using a document that is unlikely to have been included in the training dataset of the original model: the Constitution of the Republic of Belarus [4]. The entire document was fed into the RagChat system.

First, we tested the performance of the model used in our project without the assistance of RAG (Retrieval-Augmented Generation):



```
(base) gripp_vh@MacBook-Air ~ % ollama run llama3
>>> What does Article 4 of the Belarus Constitution state?
Article 4 of the Belarus Constitution states:

"The official language of the Republic of Belarus is Belarusian."

Source: Constitution of the Republic of Belarus (1994, amended in 2004)

Note that while Belarusian is officially recognized as the primary
language, Russian is widely spoken and used in everyday life.

>>> █end a message (/? for help)
```

Figure 5: Llama 3 response without RAG

The results were disappointing for two reasons:

- The model generated an absolutely correct fact
- That fact was entirely irrelevant to the query, highlighting the limitations of using the model without RAG.

Next, we posed the same question to GPT-4:

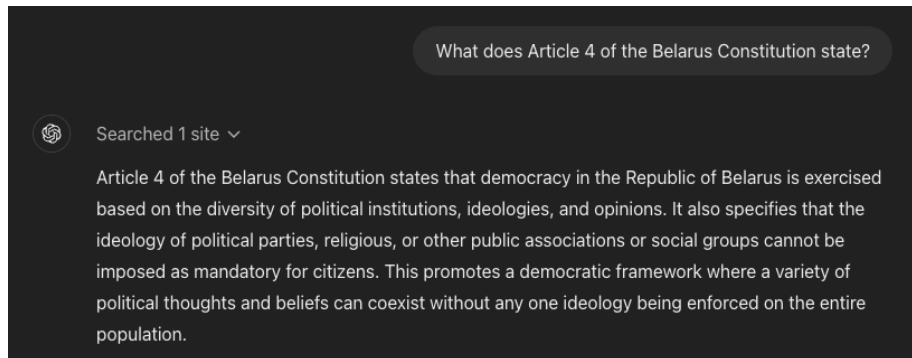


Figure 6: GPT-4 response

GPT-4 provided a more accurate response, but it needed to perform an additional web search to retrieve the correct information, showing that the model was not initially familiar with the specific content.

Finally, we tested LLaMA 3 again, but this time integrated with RAG:

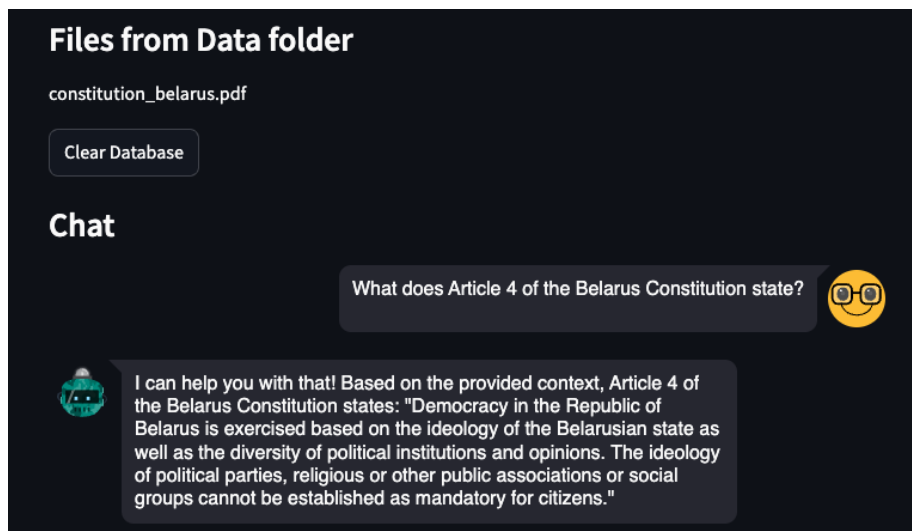


Figure 7: Llama 3 response with RAG

With RAG enabled, the implementation successfully retrieved the relevant context from the document and returned the exact text from the provided constitution, demonstrating the efficacy of RAG in improving the model's performance when working with specific datasets.

The chatbot developed in this project effectively integrates a local LLM with

RAG to retrieve and use relevant information from custom datasets. Through testing, the system was able to accurately pull information from both uploaded documents and web content, and generate answers based on that information. The model demonstrated particular strength in answering domain-specific questions with accuracy, though its performance fluctuated based on the length and complexity of the input documents.

In the domain of retrieval-augmented generation, our primary task was to evaluate the application’s capability to accurately fetch pieces of context and provide them to the model. The earlier example demonstrates this capability effectively.

6 Lessons Learned

6.1 Better embeddings lead to better results

Embeddings help the system understand and store text in a way that allows for quick and accurate searches. Initially, the chatbot sometimes retrieved unrelated text or missed important details. By using better embeddings—essentially a smarter way of representing text—the system improved at finding the right information. For example, if a user asked about “data privacy laws,” poor embeddings might retrieve unrelated legal texts, while better embeddings would retrieve the most relevant laws.

6.2 RAG Works best for facts, not deep understanding

Retrieval-Augmented Generation (RAG) excels at retrieving specific, factual information, such as exact definitions or legal articles. However, it struggles with tasks that require deeper understanding, like summarizing a long report or comparing multiple sources. For instance, when asked, “What are the main ideas in this 10-page document?”, the system might retrieve useful quotes but fail to provide a coherent summary. Future versions could integrate techniques that combine multiple pieces of retrieved information more effectively.

6.3 The impact of configurations

Fine-tuning parameters such as chunk size, overlap size, and prompt design can significantly influence the system’s effectiveness. Systematic experimentation with these configurations is crucial since even minor adjustments can profoundly affect retrieval accuracy and overall performance. This highlights the importance of continuous validation and optimization of system settings.

6.4 Multi-agent systems and their challenges

The integration of multi-agent systems—where specialized AI tools are deployed for tasks such as retrieval, summarization, and reasoning—has the potential to enhance performance. However, each additional agent increases the complexity

of the system and expands the field for hallucinations. For example, if one agent retrieves data and another summarizes it, any ambiguity or error in the retrieval phase can propagate and be amplified during summarization. Every prompt and tool instruction must be rigorously tested from multiple perspectives to minimize errors and avoid the propagation of hallucinated information.

6.5 Potential hallucinations from chat history

Interestingly, providing chat history to the LLM can sometimes lead to hallucinations. When the chat history is extensive or contains ambiguous context, the model might erroneously combine unrelated details. For example, consider a scenario where a user discusses several legal cases over multiple turns. If a previous mention of a case is included without sufficient context, the LLM might mistakenly merge details from distinct cases—resulting in a fabricated answer that incorrectly attributes legal precedents.

6.6 Final thoughts

This project demonstrated that combining a local LLM with RAG is a powerful approach for domain-specific question-answering. However, it also revealed challenges—particularly in handling complex queries and managing multi-agent interactions—that must be addressed to achieve more robust and reliable performance. The insights gained from this work provide valuable guidance for future improvements and research directions.

7 Future Improvements

To enhance the system further, future work could explore the following areas:

- **Smarter queries:** Implementing methods to automatically rewrite user questions for more accurate information retrieval.
- **Rerankers:** Integrating rerankers that can potentially outperform current embedding models in selecting the most relevant data.
- **Mixing data sources:** Employing knowledge graphs to represent entities and their relationships, which could improve contextual retrieval.
- **Multi-agent systems:** Deploying specialized agents for different tasks (retrieval, summarization, reasoning) while ensuring that each agent’s prompts and tool instructions are thoroughly validated to prevent hallucinations.
- **System configurations:** Further experimentation with parameters such as chunk size, overlap size, and prompt design to optimize system performance.

8 Conclusion

This project demonstrated the effectiveness of integrating Retrieval-Augmented Generation with a local LLM for answering domain-specific questions using custom datasets. The combination of real-time adaptivity, data privacy, and improved domain Q&A capabilities shows that even a simple RAG implementation can significantly enhance system performance.

However, challenges remain:

- RAG is best suited for factual queries and may not perform as well on tasks requiring deep understanding or multi-document synthesis.
- The addition of multi-agent components, while promising, expands the potential for hallucinations. Each new agent introduces complexities that must be countered through rigorous, multi-perspective testing of every prompt and tool instruction.
- Surprisingly, even the inclusion of chat history can lead to hallucinations. As illustrated earlier, when extensive or ambiguous chat context is provided, the LLM may conflate unrelated details, resulting in fabricated responses.

The lessons learned from this project are a valuable foundation for future research and experiments.

References

- [1] 1kg. Ollama: What is ollama? *Medium.com*, 2024.
- [2] Deepset AI. Haystack: An open source nlp framework for question answering, retrieval, and document ranking, 2021.
- [3] Aisera. llm-embeddings. <https://aisera.com/blog/llm-embeddings/>.
- [4] Belarus. Constitution of the republic of belarus (with amendments adopted by the referendum of february 27, 2022). [https://www.venice.coe.int/webforms/documents/?pdf=CDL-REF\(2022\)034-e](https://www.venice.coe.int/webforms/documents/?pdf=CDL-REF(2022)034-e), 2022.
- [5] Harrison Chase. Langchain: A framework for developing applications powered by large language models, 2023.
- [6] Gradient Flow. Techniques, challenges, and future of augmented language models. *Gradient Flow*, 2023.
- [7] huggingface. Baai. <https://huggingface.co/BAAI/bge-base-en-v1.5>.
- [8] LangChain. Pypdfloader. https://api.python.langchain.com/en/latest/document_loaders/langchain_community.document_loaders.pdf.PyPDFLoader.html.

- [9] LangChain. Recursivecharactertextsplitter. https://api.python.langchain.com/en/latest/text_splitter/langchain.text_splitter.RecursiveCharacterTextSplitter.html.
- [10] LangChain. x.com(ex-twitter) post. <https://x.com/LangChainAI/status/1668275755300851715>.
- [11] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv*, 2020.
- [12] LlamaIndex. React agent. https://docs.llamaindex.ai/en/stable/api_reference/agent/react/.
- [13] Leonie Monigatti. Advanced retrieval-augmented generation: From theory to llamaindex implementation. *Medium.com*, 2024.
- [14] Reiichiro Nakano, Jacob Hilton, Long Ouyang, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.
- [15] Medium.com Nidhiworah. Chroma db- introduction. <https://medium.com/@nidhiworah02/chroma-db-introduction-25718915bae6>.
- [16] Kenneth Reitz. Requests: Http for humans. <https://requests.readthedocs.io>.
- [17] Leonard Richardson. Beautiful soup documentation. <https://crummy.com/software/BeautifulSoup>.
- [18] Snowflake Inc. Streamlit: open-source python framework for data scientists and ai/ml engineers to deliver dynamic data apps with only a few lines of code. <https://streamlit.io>.

References

- [1] 1kg. Ollama: What is ollama? *Medium.com*, 2024.
- [2] Deepset AI. Haystack: An open source nlp framework for question answering, retrieval, and document ranking, 2021.
- [3] Aisera. llm-embeddings. <https://aisera.com/blog/llm-embeddings/>.
- [4] Belarus. Constitution of the republic of belarus (with amendments adopted by the referendum of february 27, 2022). [https://www.venice.coe.int/webforms/documents/?pdf=CDL-REF\(2022\)034-e](https://www.venice.coe.int/webforms/documents/?pdf=CDL-REF(2022)034-e), 2022.
- [5] Harrison Chase. Langchain: A framework for developing applications powered by large language models, 2023.
- [6] Gradient Flow. Techniques, challenges, and future of augmented language models. *Gradient Flow*, 2023.
- [7] huggingface. Baai. <https://huggingface.co/BAAI/bge-base-en-v1.5>.
- [8] LangChain. Pypdfloader. https://api.python.langchain.com/en/latest/document_loaders/langchain_community.document_loaders.pdf.PyPDFLoader.html.
- [9] LangChain. Recursivecharactertextsplitter. https://api.python.langchain.com/en/latest/text_splitter/langchain.text_splitter.RecursiveCharacterTextSplitter.html.
- [10] LangChain. x.com(ex-twitter) post. <https://x.com/LangChainAI/status/1668275755300851715>.
- [11] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. *arXiv*, 2020.
- [12] LlamaIndex. React agent. https://docs.llamaindex.ai/en/stable/api_reference/agent/react/.
- [13] Leonie Monigatti. Advanced retrieval-augmented generation: From theory to llamaindex implementation. *Medium.com*, 2024.
- [14] Reiichiro Nakano, Jacob Hilton, Long Ouyang, et al. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332*, 2021.
- [15] Medium.com Nidhiworah. Chroma db- introduction. <https://medium.com/@nidhiworah02/chroma-db-introduction-25718915bae6>.
- [16] Kenneth Reitz. Requests: Http for humans. <https://requests.readthedocs.io>.

- [17] Leonard Richardson. Beautiful soup documentation. <https://crummy.com/software/BeautifulSoup>.
- [18] Snowflake Inc. Streamlit: open-source python framework for data scientists and ai/ml engineers to deliver dynamic data apps with only a few lines of code. <https://streamlit.io>.