Bachelorarbeit

Using Open Source Large Language Models together with Retrieval-Augmented Generation for Chatting with own Domain-Specific Data

Verfasser

Nikolaos Lithoxopoulos

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2024

Studienkennzahl lt. Studienblatt:	A 01129439
Fachrichtung:	Informatik - Data Science
Betreuer:	Ing. Dipl.-Ing. Dr.techn. Marian Lux

Acknowledgements

First and foremost, I would like to express my deepest gratitude to my family for their unwavering support throughout my academic journey. Their financial assistance, provision of everything I needed for my studies, and, most importantly, their understanding and emotional support have been invaluable. I am truly thankful for their belief in me and their constant encouragement.

I would also like to extend my sincere thanks to all my professors at the university who have guided and taught me over the years. Their dedication to education and their willingness to share their knowledge have played a crucial role in my academic development.

Lastly, I am profoundly grateful to my supervisor for his guidance and the patience he has shown me throughout the course of this thesis. His insights and support have been instrumental in the completion of this work, and I deeply appreciate the time and effort he invested in helping me achieve my goals.

Contents

1 Introduction 5

2 Background 6
2.1 Overview of Large Language Models (LLMs) 6
2.2 Retrieval-Augmented Generation (RAG) 6
2.3 Prompt Engineering 7

3 Related Work 8
3.1 Hallucination in Language Models 8
3.2 Mathematical Inaccuracies in Generative Models 8
3.3 Existing Conversational Applications 9

4 Implementation and Solution 9
4.1 Project Architecture 10
4.1.1 Introduction to the Project Structure 10
4.1.2 Overview of the Main Directories and Files 11
4.1.3 Detailed Description of Each Directory and File 11
4.2 Tools, Frameworks, and Design Decisions 14
4.3 Configuration and Settings 16
4.4 Domain Selection 18
4.5 Document Upload and Processing 19
4.6 Metadata Utilization 20
4.7 Query Handling and Response Generation 20
4.8 Chat History Preservation 22
4.9 Prompt Engineering 22
4.9.1 Structured Output Parsers and Response Schemas 23
4.9.2 Question Classifier 23
4.9.3 Mathematical Problem-Solving Prompt: 24
4.10 Unit Tests 26

5 Practical Outcomes 28

6 Limitations 28

7 Conclusion 29

Abstract

Large Language Models (LLMs) have seen significant development in recent years, showcasing their capabilities while also presenting new challenges and limitations. This thesis introduces a Retrieval-Augmented Generation (RAG) chatbot designed to operate strictly within a user-defined domain. By integrating external document retrieval systems within an open-source framework using tools like LangChain, Ollama, and NumExpr, the chatbot accesses relevant information from domain-specific documents uploaded by the user, as well as external sources such as DuckDuckGo, grounding its responses in verified data to reduce hallucinations. For mathematical queries, the system classifies and solves arithmetic reasoning tasks, enhancing accuracy. Structured prompt engineering and unit tests ensure reliable query classification, efficient document retrieval, and accurate response generation. A key feature of this system is its strict adherence to the predefined domain, ensuring that the generated responses are relevant and accurate. The RAG approach effectively addresses the limitations of LLMs, but challenges remain, including handling complex mathematical problems, occasional hallucinations, and response delays due to the multi-stage processing pipeline.

1 Introduction

At the end of 2022, OpenAI released a free web-based chatbot LLM called ChatGPT. The introduction of LLMs in the form of chatbots allowed millions of people to interact and experience human-like communication. However, the rapid availability of this new technology has led to a lack of user understanding of its weaknesses. As more users engage with these chatbots, they often place trust in large language models without fully understanding their limitations.

While LLMs can generate human-like responses, they can also produce inaccurate, misleading, or biased information. Many users may assume the system's outputs are entirely reliable, failing to recognize that the models lack true understanding. This gap between user expectations and the technology's actual capabilities highlights the need for greater awareness of LLM limitations.

One key limitation is hallucinations, where LLMs generate plausible-sounding but factually incorrect or entirely fabricated information. This occurs because the models rely on statistical patterns rather than true understanding of the data. Another issue is their struggle with mathematical problems, often producing incorrect results, particularly for complex calculations, due to a lack of reasoning capabilities. Finally, the outcome heavily depends on prompt engineering - the way a user frames their query. Small changes in wording can significantly affect the quality and relevance of the response, highlighting the importance of carefully crafting prompts.

To address these issues, the implemented system enhances accuracy by first determining whether a query is domain-specific. Users specify a domain of interest, and this selection directly impacts how both uploaded documents and user queries are handled. The system only accepts and processes documents that are relevant to the chosen domain, filtering out irrelevant data to ensure the chatbot accesses domain-specific information. Once domain validation is complete, the system retrieves relevant information and applies safeguards to minimize hallucinations. Additionally, prompt engineering is utilized to categorize questions and formulate appropriate responses, including mechanisms for handling mathematical queries. Input rephrasing reduces dependency on the quality of the user's initial request, enabling more structured and reliable interactions. By integrating domain selection, the system improves the relevance and accuracy of chatbot responses, ensuring they are grounded in the user's specified area of interest.

2 Background

2.1 Overview of Large Language Models (LLMs)

Large Language Models (LLMs), such as GPT[1] and BERT[2], are built on the transformer architecture[3], which has significantly advanced natural language processing. At the core of this architecture is the attention mechanism, which enables models to focus on the most relevant parts of a text, improving their ability to capture long-range dependencies and contextual relationships.

The first step in processing text with LLMs is tokenization, where text is divided into smaller units, such as words or subwords. These tokens are then transformed into embeddings dense vectors that represent their semantic meaning in a high-dimensional space. Embeddings allow models to better grasp the meaning of words in context, improving the overall coherence of the generated output.

The self-attention mechanism further enhances this process by assigning different importance to each token, depending on its relevance to other tokens in the sequence. This allows the model to generate more contextually appropriate responses. As a result, transformers excel in tasks such as machine translation, summarization, and question-answering, where understanding the relationships between tokens across long texts is crucial.

2.2 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG)[4] combines the strengths of language models with external retrieval systems to improve response accuracy. Unlike conventional language models, which rely solely on their internal knowledge, RAG retrieves relevant documents or data from external sources before generating an answer. This process enables the model to provide more factually grounded responses by accessing up-to-date information, as illustrated in Figure 1.

RAG is particularly effective in addressing hallucinations. By incorporating external data retrieval, RAG ensures that generated content is backed by verifiable sources, reducing the chances of misinformation.

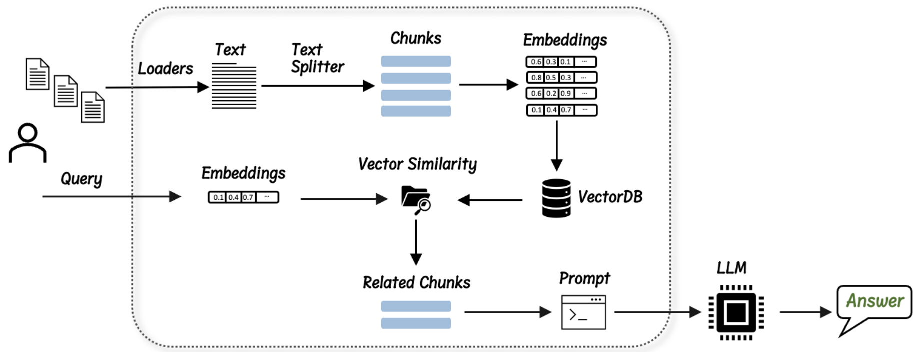

Figure 1: A schematic overview of a typical RAG system.

2.3 Prompt Engineering

Prompt engineering is the practice of carefully crafting inputs (prompts) to guide language models in generating accurate and relevant responses. The way a prompt is structured significantly influences the quality of the model's output, making prompt design crucial when interacting with large language models. Effective prompts provide clear context, specific instructions, or examples, helping to steer the model towards more reliable and coherent answers.

Prompt engineering offers a powerful solution to the challenges of explainability, interpretability, and reliability in AI based chatbot applications[5]. By developing methods for crafting prompts that allow users to understand how LLMs arrive at their responses, prompt engineering enhances the transparency of the model's decision-making process. This increased clarity is essential for building trust and ensuring reliability, especially in sensitive fields. Focusing on explainability and interpretability through prompt engineering helps make LLMs more accessible and trustworthy.

Several strategies[5] are used to guide models effectively: zero-shot[6] prompting involves giving the model no prior examples, relying on its general knowledge; few-shot prompting[6] provides a few examples to help steer the response; and chain-of-thought[7] prompting asks the model to explain its reasoning step by step, improving accuracy for complex tasks. The goal is to balance clarity and flexibility, ensuring the model generates relevant, coherent responses while minimizing errors, hallucinations, or biased outputs. Effective prompt design remains key to maximizing the potential of large language models.

3 Related Work

3.1 Hallucination in Language Models

Hallucinations, where language models generate text that is plausible but factually incorrect, have been a significant challenge in generative systems like GPT and BERT. Numerous studies have investigated the causes and potential solutions for this issue. Early research identified that these models rely heavily on patterns in training data without a deep understanding of factual accuracy, leading to frequent hallucinations in open-domain tasks.

To address hallucinations in language models, several approaches have been developed[8]. Beyond the use of RAG and Prompt Engineering, Self-Refinement techniques, such as the Chain of Verification (CoVe)[9], allow models to iteratively verify and correct their outputs by generating and answering fact-checking questions. Supervised Fine-Tuning has also proven effective, where models are trained on carefully curated datasets to enhance factual accuracy[10, 11]. Another important method is the use of Knowledge Graphs, like the RHO[12] framework, which integrates structured external knowledge to ground the model's responses in verifiable facts. Finally, innovative Decoding Strategies, such as Context-Aware Decoding (CAD)[13] and Decoding by Contrasting Layers (DoLa)[14], guide the generation process to focus on contextually accurate and consistent information, thereby reducing hallucinations.

3.2 Mathematical Inaccuracies in Generative Models

Generative language models also face limitations in performing accurate mathematical calculations. This issue arises because models are trained to predict the most likely next word or phrase based on language patterns, rather than solving mathematical problems using formal reasoning. Research addressing this problem has led to various solutions, including the development of specialized models for mathematical reasoning.

Recent approaches have focused on enhancing the ability of LLMs to handle arithmetic and symbolic reasoning tasks. Some methods incorporate external computational tools, such as Wolfram Alpha[15], to manage mathematical queries and address the limitations of purely generative models. Other strategies, like prompt engineering[16], explore step-by-step reasoning to guide the answering process and reduce computational errors, offering a more structured approach to improving mathematical accuracy in LLMs. These techniques aim to bridge the gap between language-based predictions and the formal reasoning required for accurate mathematical problem-solving.

3.3 Existing Conversational Applications

Recent advancements in conversational AI have led to the development of powerful models such as ChatGPT by OpenAI[17], Claude by Anthropic[18], and Gemini by Google DeepMind[19]. These models represent the state-of-the-art in LLMs, each offering unique capabilities and features. ChatGPT provides community-generated custom chats, enhancing versatility with user-driven prompts and scenarios. Claude focuses on AI safety and alignment with human values, ensuring ethical responses, while Gemini integrates multimodal capabilities, allowing it to process both text and images. However, a key concern across these models is privacy, as the underlying response generation process remains largely opaque to users, limiting transparency. Additionally, while these systems can accept files as input, it remains unclear how they handle or access user-uploaded documents, making it challenging to ensure responses are grounded in personalized or document-specific data.

While state-of-the-art conversational models like ChatGPT, Claude, and Gemini excel in general-purpose dialogue, they do not inherently employ a RAG approach. Instead, they rely on pre-trained knowledge and lack transparency in how they handle user-uploaded documents, often generating responses that are not explicitly grounded in specific data sources. In contrast, RAG-based tools like PaperChat[20], ChatDoc[21], and Sharly.ai[22] explicitly retrieve and reference information from uploaded files, ensuring more contextually grounded responses. However, these systems face their own limitations: they are often designed for general-purpose use rather than domain-specific applications, come with pricing models that may not be suitable for all users, and impose restrictions on the number of requests or the volume of documents that can be uploaded, limiting their scalability for extensive use.

4 Implementation and Solution

This section provides a detailed explanation of the implementation of a chatbot application that employs a RAG approach to mitigate hallucinations, mathematical inaccuracies, and prompt dependency. The system is designed to enhance user interaction by leveraging uploaded documents in formats such as `.txt`, `.docx`, and `.pdf`, as well as external `urls`. Additionally, the system integrates DuckDuckGo as a search engine to retrieve accurate, domain-specific answers when necessary, displaying relevant metadata such as filenames, page numbers, and URLs to ensure transparency.

To optimize its responses, the chatbot allows users to select a specific domain, and the system ensures that only relevant documents from the selected domain are used. Key settings, such as the language model, text chunk sizes, and chunk overlap, can be adjusted via the `config.py` file to tailor performance.

The overall system follows a structured flow, which includes steps such as checking query relevance to the defined domain, retrieving and grading documents, detecting hallucinations, and utilizing fallback mechanisms when necessary. This multi-stage process ensures that the chatbot delivers accurate, relevant, and contextually appropriate responses while minimizing errors. The system also efficiently handles mathematical queries by classifying questions and using specialized methods to generate answers.

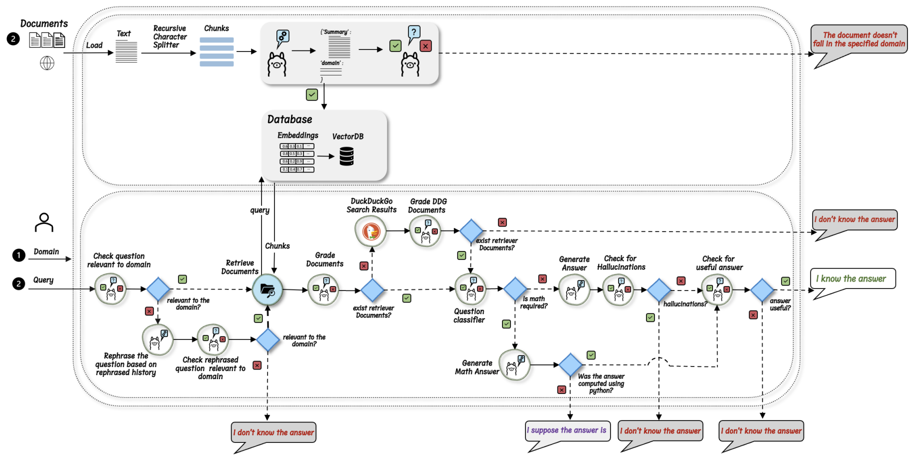

Figure 2: Overview of the chatbot system's architecture.

An overview of the project's architecture and the entire query-handling process, including document handling, question processing, and answer generation, is depicted in Figure 3. This diagram illustrates the chatbot's pipeline, which highlights the different stages of query processing, from loading documents to retrieving and grading potential answers, detecting hallucinations, and generating final responses.

4.1 Project Architecture

4.1.1 Introduction to the Project Structure

This section provides a comprehensive overview of the project's file structure, detailing the purpose and functionality of each directory and file. Understanding this structure is crucial for maintaining, extending, and debugging the system. By clearly defining the responsibilities of each component, this section aims to guide developers in navigating the codebase effectively, ensuring that any modifications or enhancements are made with a full understanding of how different parts of the system interact.

4.1.2 Overview of the Main Directories and Files

The project structure is visualized below Figure 2 to provide a clear overview of the organization of files and directories.

```
│   graph_img
├── qa_system
│   ├── lang_graph.py
│   ├── prompts.py
│   ├── qa_manager.py
│   └── structure_answer.py
├── rag
│   ├── rag_prompts.py
│   ├── rag.py
│   └── vectordb.py
├── tests
│   ├── data
│   ├── edges_test
│   └── path_test
├── utils.py
│   ├── __init__.py
│   ├── text_doc_processing.py
│   ├── ui_helpers.py
│   └── upload_source.py
├── chatbot.py
├── config.py
├── README.md
└── requirements.txt
```

Figure 3: Project File Structure

4.1.3 Detailed Description of Each Directory and File

Folder graph_img:

The graph_img directory contains images used for visualizing the project's structure in the README.md file, as well as graphs generated by the lang_graph.py, which illustrate the query process within the system.

Folder qa_system:

The qa_system folder contains key components responsible for managing the question-answering system. This includes the workflow initialization, prompt templates, and various utility functions that structure the query and response process.

- `lang_graph.py`: Manages the initialization of the workflow for query processing. It sets up the sequence of operations that the system follows to process a query. It also generates a visual graph of the query process.
- `prompts.py`: Contains all the prompt templates used for the query process when interacting with the language model.
- `qa_manager.py`: Implements the functions corresponding to each step in the workflow defined in `lang_graph.py`. These functions represent the nodes in the query processing graph, handling tasks such as domain relevance checks, document retrieval, grading, and response generation.
- `structure_answer.py`: Defines data models used to structure the answers returned by the system. This includes models for standard answers, arithmetic reasoning, and hallucination checks, ensuring that responses are well-organized and include necessary metadata.

Folder `rag`:

The `rag` folder contains components crucial for implementing the RAG approach, which integrates document retrieval with language model generation to enhance response accuracy within a specified domain.

- `rag_prompts.py`: Contains prompt templates that are crucial for processing and verifying document domains within the RAG system. The `domain_detection` prompt first summarizes the content of the documents and then identifies three possible domains they could belong to. The `domain_check` prompt subsequently verifies whether the identified domains align with the user-specified domain, ensuring that the documents are relevant to the defined scope.
- `rag.py`: This file orchestrates the core functionality of the RAG system. It handles the ingestion and processing of documents from various formats, ensuring they are clean, normalized, and split into manageable chunks. It performs domain detection and verification to ensure that the documents align with the user-defined domain before storing them in a vector database. The file also includes functionality to set the domain for processing queries and initiates the conversation process within the system.
- `vectordb.py`: This file manages the vector database that stores document embeddings. It utilizes a HuggingFace BGE model to generate embeddings for the documents, which are then stored in a Milvus-based vector database.

Folder `tests`:

The `tests` folder contains the necessary components for unit testing the system. It includes:

- `data`: A folder containing the data used for the unit tests, ensuring that the tests run with consistent and relevant inputs.
- `edges_test`: This folder contains unit tests for all the edges in the query processing workflow. It also includes a `README.md` file that explains each test in detail.
- `path_test`: This folder holds unit tests for the paths within the graph that define the query processing workflow. Similar to the `edges_test` folder, it also contains a `README.md` file that provides explanations for each test.

Folder `utils` :

The `utils` folder contains various utility functions that support different aspects of the system, providing essential tools for document processing, user interface enhancements, and managing uploads.

- `__init__.py`: Imports key functions from other utility modules.
- `text_doc_processing.py`: This file contains utility functions for processing and handling text documents within the system. These functions include:
  - `print_documents`: Formats and prints document metadata and content for easy readability.
  - `clean_text`: Cleans text in documents by removing unnecessary whitespace and formatting issues.
  - `convert_str_to_document`: Converts a string input into a list of `Document` objects, extracting snippets and associated links.
  - `normalize_documents`: Normalizes document metadata, particularly handling PDF files by adjusting page numbers.
  - `extract_limited_chat_history`: Extracts a portion of chat history based on a specified maximum length, ensuring the total length of messages does not exceed the limit.
  - `trim_url_to_domain`: Trims a full URL down to its base domain, providing a simplified version alongside the original URL.
- `ui_helpers.py`: Contains utility functions designed to enhance the user interface in a Streamlit application. These functions provide tools for styling and formatting the interface.
- `upload_source.py`: Manages the uploading of documents and URLs within the system. It ensures that uploaded files are processed correctly, preventing duplicates and validating the domain before ingestion. The `upload_document` function handles file uploads, checks for duplicates using file hashing, and processes the files if they are not already uploaded. The `upload_url`

function manages URL uploads, verifying that the URL is valid and associated with the correct domain before ingestion. Both functions return status codes to indicate the success or failure of the upload process.

File `chatbot.py`:

This file is the main script for the Streamlit application that runs the chatbot interface. It manages the overall user interaction, including the display of chat messages, processing of user inputs, and handling of file and URL uploads. The script is responsible for initializing the session state, setting the domain for the chat, and orchestrating the ingestion of documents and URLs. It also includes utility functions for displaying uploaded files and URLs, as well as customizing the user interface with headings, dividers, and other elements. The `page()` function renders the app layout and manages the dynamic interactions based on user inputs and the domain context.

File `config.py`:

This file defines configuration settings for the system. It uses a data class to store these settings, ensuring they are easily accessible and modifiable throughout the application.

File `README.md`:

This file serves as the primary guide for users to understand the purpose of the project, how to install and run it, and how to configure its settings. The `README` focuses on practical usage rather than going into technical details.

File `requirements.txt`:

This file lists all the Python dependencies required to run the project. It specifies the exact versions of the libraries needed, ensuring that the environment is consistent and compatible across different setups. Users can install these dependencies using `pip` to set up the project environment.

4.2 Tools, Frameworks, and Design Decisions

The specific versions of the tools and frameworks used, are reported, to ensure consistency and reproducibility throughout the project (see Table 1). The table outlines the key components of the system, along with their versions, primary purposes, and additional notes relevant to them. The table provides an overview of the technology stack and ensures that future work can replicate the environment.

Ollama[23] simplifies the deployment of open-source LLMs by offering an easy

Table 1: Tools, Frameworks, and Versions Used in the Project

Framework/Tool	Version	Purpose	Notes
Ollama[23]	v0.3.11	LLM deployment and local model hosting	Used for running LLMs locally
LangChain[24]	v0.3.0	Framework for chaining LLM models and data retrieval	Core of the RAG implementation
Milvus[25]	v0.1.5	Vector database for embedding retrieval	Chosen for its seamless LangChain integration
HuggingFace-Hub	v0.25.1	`HuggingFaceBgeEmbeddings` Text embedding model for fast, low-latency retrieval	Selected for real-time performance
Streamlit[26]	v1.38.0	Web framework for UI development	Used to build the user interface
DuckDuckGo Search	v6.2.12	External search engine for real-time document retrieval	For external data retrieval
NumExpr	v2.10.1	Mathematical expression evaluation	Used for processing mathematical queries
LangGraph[24]	v0.2.24	State and context management in queries	Useful for graph-based processing and query tracking
Python	v3.12.03	Programming language for backend logic	Core programming environment

interface for running models locally, eliminating the need for external APIs or cloud services. It supports various models, allowing developers to experiment with a wide range of models that can be easily accessed and used in their workflows. This setup ensures that developers can maintain control over data locally, reducing operational costs and improving privacy.

Once Ollama is installed, various models can be downloaded using a simple terminal command. For example, the LLama 3.1 model can be pulled using the following command:

```
ollama pull llama3.1
```

Streamlit was utilized to build the chatbot's user interface. Streamlit offers a simple yet powerful framework for creating interactive web applications, making it ideal for rapid prototyping and deploying the chatbot in an intuitive and user-friendly environment. It allows seamless integration of key features like domain selection, document uploads, and query input, all while maintaining a clean and responsive interface. This approach simplifies user interaction and enhances the overall usability of the system.

LangChain was selected for this project because of its flexibility, extensive capabilities, and seamless integration with LLMs. One of its key strengths is its model-agnostic design, which allows developers to experiment with different models, such as those provided by Ollama, to find the best fit for specific tasks. This flexibility simplifies the process of working with various LLMs, enabling developers to test and use multiple offerings without significantly expanding the codebase for each new provider.

LangGraph, a part of the LangChain ecosystem, was chosen for this project due to its ability to handle complex workflow orchestration and graph-based reasoning, which are ideal for managing multi-step decision-making processes in a chatbot. By integrating LangGraph, the system gains the ability to track the flow of user queries across multiple interactions, ensuring that the chatbot can maintain context and logic over time. This structured approach enables more sophisticated, dynamic conversations, allowing the chatbot to route user queries through various processes, adapt responses based on previous interactions, and keep track of dependencies between different steps. One of the most significant advantages of LangGraph is its ability to create graph-based workflows, where different components of the system can interact in a flexible but organized manner. This feature is essential for managing complex reasoning tasks, making it ideal for applications that require deeper decision-making beyond simple question-answering. Moreover, LangGraph's integration allows for workflow visualization, helping to debug and optimize the chatbot's performance. By representing the decision-making process as a graph, LangGraph makes it easier to manage and maintain the chatbot's intricate logic paths. This makes it an indispensable tool for projects, where stateful interactions and multi-turn conversations are crucial for delivering accurate and contextually aware responses. In the project, LangGraph plays a crucial role due to the need for multi-step conditional workflows, where decisions are made at various stages based on the system's output. Each node in the workflow represents a task, such as checking the domain relevance of a query or retrieving documents, while the edges between nodes represent the conditions that determine the next step in the process.

In this project, Milvus[25] and HuggingFaceBgeEmbeddings were chosen for their specific strengths. Milvus is well-suited for RAG tasks due to its seamless integration with LangChain and efficient text embedding retrieval. HuggingFaceBgeEmbeddings provides low-latency performance, ensuring quick and efficient text retrieval, making it ideal for real-time LLM workflows. While alternatives such as FAISS[27], Pinecone[28], and Chroma[29] offer valuable features, such as FAISS's scalability, Pinecone's managed services, and Chroma's flexibility in handling both approximate and exact searches, Milvus and HuggingFaceBgeEmbeddings were selected for their balance of performance, ease of integration, and ability to meet the project's real-time requirements.

4.3 Configuration and Settings

The configuration of the system is centralized in the `config.py` file, which defines key parameters such as model settings, document splitter settings, retrieval settings, and external search settings. Table 2 summarizes the key configuration parameters.

Table 2: Configuration Parameters

Parameter	**Default Value**	**Description**
Model Parameters		
`MODEL`	`"llama3.1"`	Language model for generating answers.
`MODEL_TEMPERATURE`	`0.0`	Controls randomness; lower values make output deterministic.
`KEEP_IN_MEMORY`	`-1`	If `-1`, keeps the model loaded in memory.
Text Splitter Parameters		
`SPLITTER_CHUNK_SIZE`	`512`	Size of text chunks during ingestion.
`SPLITTER_CHUNK_OVERLAP`	`51`	Overlapping tokens between chunks to maintain context.
Database and Retriever Parameters		
`COLLECTION_NAME`	`"rag_chroma"`	Name of the document collection in the database.
`URI`	`"./vector.db"`	The path to the database used for storing and retrieving document vectors.
`N_DDG_TO_RETRIEVE`	`4`	Number of DuckDuckGo documents to retrieve when needed.

The configuration parameters outlined in Table 2 are chosen to ensure optimal system performance across various tasks. The used `MODEL "llama3.1"` is capable of handling both natural language and mathematical queries effectively, which simplifies the configuration by eliminating the need to switch to a math-specific model, such as `"mathstral"`. This approach leverages Llama 3.1's versatility in processing different types of queries without sacrificing performance. The `KEEP_IN_MEMORY` parameter, set to `-1`, ensures that the model remains loaded in memory indefinitely, which can be beneficial for performance by avoiding the overhead of reloading the model between queries. This setting is particularly useful in systems with frequent interactions, where maintaining the model in memory can reduce latency. The `KEEP_IN_MEMORY` parameter can also be configured to other values, such as a duration string (e.g., "10m" for 10 minutes) or a number in seconds, depending on the use case. However, the default value of `-1` is ideal for chat environments where uninterrupted access to the model is needed for continuous operations. A `MODEL_TEMPERATURE` of 0.0 is set to produce deterministic outputs, which is crucial for consistency, especially in critical applications like document retrieval or mathematical problem solving. The text splitter's chunk size and overlap are fine-tuned to ensure efficient processing of text while maintaining sufficient context across chunks. Finally, external search settings like `N_DDG_TO_RETRIEVE` help supplement the database with additional online resources when necessary, enhancing the system's overall information retrieval capability.

4.4 Domain Selection

The system is designed to allow users to specify a domain of interest, which plays a crucial role in determining how both uploaded documents and queries are processed. By selecting a specific domain, the user ensures that the chatbot focuses on retrieving and generating answers based only on documents relevant to that domain. This domain-driven approach improves the relevance and accuracy of the chatbot's responses.

When a user uploads documents, the system verifies whether the content aligns with the user-defined domain. Only documents that fit within the selected domain are accepted into the system for further use. This ensures that the chatbot does not reference irrelevant material when generating responses, maintaining domain-specific accuracy.

Upon receiving a user query, the system first checks whether the question falls within the defined domain. If the question clearly matches the domain, the system proceeds with normal processing. However, if the domain fit is ambiguous or cannot be determined solely based on the user's question, the system evaluates the entire conversational flow for additional context. This is particularly important to avoid mistakenly rejecting relevant queries or proceeding with irrelevant ones.

The integration of conversational flow context is a safeguard against situations where a user's question may not explicitly reference the domain, but is implicitly connected based on prior interactions. By examining the broader conversation, the system ensures that queries indirectly related to the domain are appropriately processed, enhancing user experience and maintaining continuity.

Once a question is confirmed to be within the defined domain, the system moves forward with processing the request. This includes retrieving relevant document sections, performing additional searches if necessary, and generating a response grounded in domain-specific information.

This domain selection mechanism enhances the system's focus and accuracy by ensuring that both the documents and the queries are aligned with the user's specified area of interest, while leveraging the conversational context when needed.

In the system, domain selection is managed through the `ChatPDF` class in `rag.py` and the `KnowledgeBaseSystem` class in `qa_manager.py`. Document ingestion and domain verification occur via the `ingest` method in `ChatPDF`, using the `summary_domain_chain` and `domain_checking` functions from `rag.py` to ensure only domain-relevant documents are accepted. For queries, the `query_domain_check` function in `qa_manager.py` assesses whether they align with the specified domain.

4.5 Document Upload and Processing

The uploaded documents or provided URLs are then processed and indexed for accurate, domain-specific responses, as shown in Figure 4. Upon upload, the system checks for duplicates using file hashes to prevent redundant processing. This operation is handled in the `chatbot.py` file, specifically within the `read_and_save_file` function, which leverages the `upload_document` and `upload_url` functions from the `upload_source.py` file to manage document and URL ingestion.

Supported file types include PDFs, Word documents, text files, and web pages via URLs, each handled by appropriate loaders (`PyMuPDFLoader`, `Docx2txtLoader`, `TextLoader` with encoding detection, and `WebBaseLoader`, respectively).

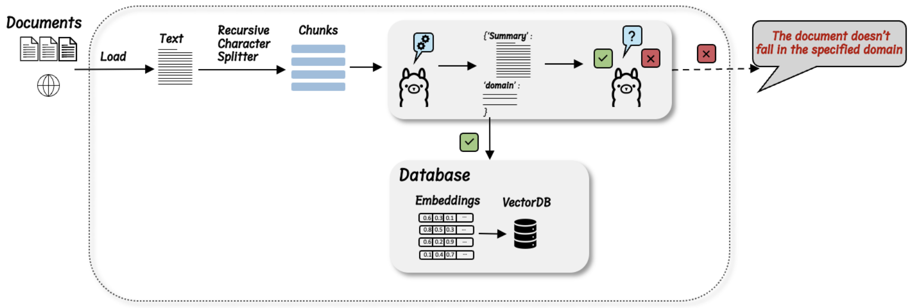

Figure 4: The process of internal sources handling.

These loaders are implemented in the rag.py file, within the ingest method of the `ChatPDF` class. Extracted text is cleaned and split into chunks using the `RecursiveCharacterTextSplitter`, with chunk size set to `SPLITTER_CHUNK_SIZE` (512 tokens) and overlap set to `SPLITTER_CHUNK_OVERLAP` (51 tokens). These constants are chosen to balance efficiency and context preservation: a chunk size of 512 tokens is manageable for processing while large enough to contain meaningful content, and an overlap of 51 tokens ensures continuity between chunks without excessive redundancy. Each chunk is annotated with metadata to help trace the original source of the answer, a process also managed within the `ingest` method.

The system summarizes the document and checks its relevance to the specified domain using a language model pipeline; irrelevant documents are rejected to maintain domain specificity. Relevant chunks are embedded using from `HuggingFaceBgeEmbeddings` the `BAAI/bge-large-en` model and stored in a Milvus vector database for efficient retrieval. Temporary files are deleted after processing to conserve resources, a process handled in the `upload_document` function.

4.6 Metadata Utilization

Metadata plays a crucial role in improving response quality by providing transparency, credibility, and relevance. Users can see the source of information, such as the filename, page number, or URL, fostering trust and enabling users to verify information sources.This is particularly important for allowing users to navigate to the original source for more details.

The process of attaching metadata to the document chunks occurs in the `ingest` function in the `rag.py` file. The `clean_text` function is responsible for cleaning the document text and assigning the file name as metadata to each chunk of text. The `normalize_documents` function in the same file further processes the metadata for PDF documents by extracting the page number and incrementing it by one. This function adjusts the source metadata to include the page number, making it easier to trace back the content to its exact location in the original document.

4.7 Query Handling and Response Generation

The system processes user queries through a multi-stage pipeline, as illustrated in Figure 5, designed to provide accurate and relevant answers while minimizing hallucinations. When a user poses a question, the system first evaluates its relevance to the specified domain (see the `Check question relevant to domain` stage in Figure 5). If the question is not directly relevant, it attempts to rephrase it using previous conversation context to align it with the domain; if relevance cannot be established, the system informs the user accordingly, as shown in the `Rephrase the question based on rephrased history` flow in the diagram. Throughout this pipeline, the whole flow and decision-making process is supported by LangGraph, a framework that helps guide key decisions, such as whether to retrieve documents internally or perform an external search, or whether a mathematical computation is required instead of a standard textual response. By dynamically adapting to the type of query and available data, LangGraph ensures that the system provides accurate, relevant, and contextually appropriate answers, while reducing the likelihood of hallucinations.

For questions deemed relevant, the system retrieves potential answers from the knowledge base. Retrieved documents are graded for quality, and only those meeting the relevance threshold are used. If insufficient relevant documents are found internally, the system performs an external search using sources such as DuckDuckGo, retrieving and grading additional documents (as depicted in the `DuckDuckGO Search Results"` and `"grade_ddg_documents"` stages).

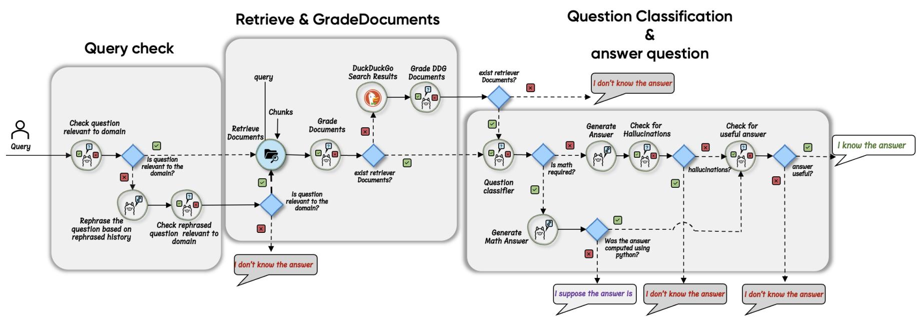
Query check
Retrieve & GradeDocuments
Question Classification & answer question
Query
Check question relevant to domain
Is question relevant to the domain?
Rephrase the question based on rephrased history
Check rephrased question relevant to domain
query
Chunks
Retrieve Documents
Is question relevant to the domain?
I don't know the answer
Grade Documents
DuckDuckGo Search Results
Grade DDG Documents
exist retriever Documents?
exist retriever Documents?
I don't know the answer
Question classifier
Is math required?
Generate Answer
Check for Hallucinations
hallucinations?
Check for useful answer
answer useful?
I know the answer
Generate Math Answer
Was the answer computed using python?
I suppose the answer is
I don't know the answer
I don't know the answer

Figure 5: The flow of the system's query handling process.

The system classifies the question to determine whether it requires a mathematical computation or a standard textual response. For mathematical queries, the system first retrieves relevant documents and generates an arithmetic reasoning answer using Python's `ne.evaluate()` for direct computation, as shown in the `Generate Math Answer"` path in the figure. If the Python-based computation fails, the system employs a web-based fallback mechanism, generating an answer based on external sources. Although the fallback web-based solution may not always be as accurate, as indicated by a warning in the final answer, it provides an alternative when local computation is not feasible. Both math or text-based questions utilize the same language model for generating answers. Once an answer is generated, it undergoes a hallucination check to ensure it does not contain unsupported or fabricated information, as illustrated in the `Check for Hallucinations` step.

Finally, the system checks whether the generated answer adequately addresses the user's question. This check is performed using an evaluation chain that compares the answer with the question. If the evaluation determines that the answer is useful, it is returned to the user, along with metadata such as source filenames, page numbers, or URLs (see the `Check for useful answer` step in Figure 5). However, if the system determines that the answer is not useful, it defaults to a fallback message, such as "I don't know the answer to that question," and does not include any metadata. This fallback mechanism ensures that the system maintains transparency and trust with the user, especially when it cannot generate a reliable response.

4.8 Chat History Preservation

The system maintains two types of histories: chat_history and chat_rephrased_history. The chat_history logs direct interactions between the user and the system, capturing user questions and LLM-generated responses. This helps maintain context across conversations and aids in retrieving relevant information. The chat_rephrased_history stores reworded versions of user questions to improve the user's prompt and provide more context. Both histories are used in query rephrasing, document retrieval, and domain detection processes, enhancing the accuracy and relevance of the system's responses.

When a query is made, the create_history_aware_retriever function combines the current question with past chat history, retrieving relevant documents based on the accumulated context. This ensures that the chatbot can understand the progression of the conversation, leading to more accurate and contextually appropriate responses. By maintaining a coherent interaction flow, the system improves user experience, especially in long or complex conversations.

4.9 Prompt Engineering

Prompt engineering is an essential component of guiding the language model to perform specific tasks accurately. Through carefully designed prompts and structured formats, the system directs the model through various stages like domain relevance checking, question rephrasing, and mathematical problem-solving. In this project, LangChain's structured output parsers and response schemas play a key role in generating consistent, well-formatted responses from the model. Each of these components is designed to ensure clarity and reduce variability in the model's output, depending on the task.

Additionally, Chain-of-Thought (CoT)[7] prompting is employed to guide the model to explain its reasoning step by step, which is particularly useful for complex reasoning and problem-solving tasks. This approach improves the model's logical structuring, especially in cases where a series of intermediate steps is required before reaching a final answer. CoT prompting ensures that the model breaks down the problem into manageable parts, leading to more accurate and interpretable responses.

Alongside CoT, zero-shot prompting[6] is also utilized in this system. In zero-shot prompting, the model is given no prior examples or specific training for the task at hand but is expected to perform based solely on its pre-trained knowledge. This technique allows the system to handle a wide variety of tasks efficiently without the need for task-specific fine-tuning or examples.

By incorporating both Chain-of-Thought prompting and zero-shot prompting, the system is able to enhance the model's ability to reason through problems and generate relevant answers, even when faced with tasks that it has not encountered before. This combination of techniques, along with structured output parsers and response schemas, provides a robust framework for handling a diverse range of user queries with clarity, consistency, and accuracy.

4.9.1 Structured Output Parsers and Response Schemas

Structured output parsers are employed for tasks that require multi-field responses, such as generating arithmetic solutions with step-by-step reasoning. These parsers often leverage Pydantic models for validation, ensuring that the outputs adhere to a strict predefined structure. For instance, when solving mathematical problems, structured output parsers ensure that the response includes fields such as `"step_wise_reasoning"`, `"expr"`, and `"sources"`, thus maintaining clarity and avoiding ambiguity.

In contrast, response schemas are more suited for tasks that require simpler outputs, such as binary classification. In these cases, the output typically consists of a single field, which reduces the need for the complexity of a full Pydantic model. For example, when checking whether a user's query is relevant to a specified domain, a response schema can effectively generate a straightforward ”yes” or ”no” answer.

4.9.2 Question Classifier

This prompt classifies whether a user's question is mathematical or not, leveraging the zero-shot prompting technique. In zero-shot prompting, the model is given no prior examples but is expected to perform the task based on its pre-trained knowledge. The model responds with ”yes” or ”no” depending on whether the question requires a mathematical solution.

```
response_schemas_question_classifier = [
    ResponseSchema(
        name="score",
        description="Answer with 'yes' if the question is a math
                            question, 'no' otherwise."),
]
output_parser_question_classifier =
              StructuredOutputParser.from_response_schemas(
                    response_schemas_question_classifier
                    )
format_instructions_question_classifier =
              output_parser_question_classifier.
    get_format_instructions(
                    only_json=True
                    )

question_classifier_prompt = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|
    end_header_id|>
    Classify the question as a math question or not.
    <|eot_id|><|start_header_id|>user<|end_header_id|>
    question: {question}
    <|eot_id|><|start_header_id|>assistant<|end_header_id|>
    format instructions: {format_instructions}
    """,
    input_variables=["question"],
    partial_variables={"format_instructions":
                    format_instructions_question_classifier}
)
```

Listing 1: Question classifier prompt

This prompt sets up a binary classification task, where the model is expected to determine if a question is mathematical. Since no specific examples are provided to the model beforehand, this task operates in a zero-shot fashion. The structured output schema ensures that the response is restricted to a ”yes” or ”no” answer, maintaining consistency and reducing ambiguity. As with other tasks, system tags like <begin_of_text> and <eot_id> delineate the sections of the task, ensuring compatibility with models designed for conversational or task-based outputs.

4.9.3 Mathematical Problem-Solving Prompt:

This prompt leverages the CoT[7] prompting technique to guide the model through step-by-step reasoning for arithmetic tasks. By instructing the model to break down arithmetic problems into logical steps, the prompt ensures that the solution process is both transparent and interpretable. The final output consists of a detailed reasoning structure followed by a mathematical expression that can be evaluated directly using the NumExpr library.

```
math_solver = PromptTemplate(
    template="""<|begin_of_text|><|start_header_id|>system<|
    end_header_id|>
    You are an expert AI specialized in generating expressions for
    ne.evaluate(.)
    to solve arithmetic tasks.
    Given specific numbers, write a NumExpr-compatible expression
    that
    directly calculates the result.
    The final expression must contain only numbers and operators,
    with no variables.

    Related Documents: {documents}\n

    Example Task:
    Related Documents: [Cappuccinos cost $2, iced teas cost $3,
    cafe lattes cost $1.5 and espressos cost $1 each.]
    Task: Sandy orders some drinks for herself and some friends.
    She orders three cappuccinos,
    two iced teas, two cafe lattes, and two espressos.
    How much change does she receive back for a twenty-dollar bill?
    Answer: {{
    'step-wise reasoning': [
        'Calculate the total cost of three cappuccinos: 3 * 2',
        'Calculate the total cost of two iced teas: 2 * 3',
        'Calculate the total cost of two cafe lattes: 2 * 1.5',
        'Calculate the total cost of two espressos: 2 * 1',
        'Sum these costs to get the total expenditure',
        'Subtract the total expenditure from 20 to find the change'
    ],
    'expr': '20 - (3 * 2 + 2 * 3 + 2 * 1.5 + 2 * 1)'
    }}
    <|eot_id|><|begin_of_text|><|start_header_id|>user<|
    end_header_id|>
    Task: {question}
    <|eot_id|><|start_header_id|>assistant<|end_header_id|>""",
    input_variables=["question", "documents"],
)
```

Listing 2: Arithmetic problem-solving prompt

In this case, the model is prompted to generate a NumExpr-compatible expression that performs arithmetic tasks directly, without the use of variables. By providing step-by-step reasoning, the model explains how each component of the problem is solved, ensuring that the process is easy to follow and validate. This prompt is particularly suited for complex arithmetic problems, where multiple operations need to be performed in a sequential manner.

The model's output is structured using the following Pydantic class to ensure consistency in response:

```
class AnswerWithSourcesMath(BaseModel):
    """An Arithmetic Reasoning step wise reasoning
    to the question, with keys 'step_wise_reasoning',
    'expr' and 'sources'.
        'step_wise_reasoning' : The step wise reasoning to the user
    's question.
        'expr' : Generate only parameter expressions (like 3 + 1)
        to be evaluated by ne.evaluate(.) for solving arithmetic
    tasks.
        'sources' : ONLY the key 'sources' of metadata of the
        Documents that are used to generate the answer."""
    step_wise_reasoning: List[str]
    expr: str
    sources: Annotated[
        Set[str],
        ...,
        "The key 'sources' of the Metadata from the provided
    documents",
    ]
```

Listing 3: Structured output class

The structured output ensures that the model generates both a step-by-step reasoning sequence and the final arithmetic expression, which can be directly evaluated using NumExpr. By using this approach, the system leverages both CoT[7] prompting and structured output validation, ensuring that each problem is solved with clear reasoning and accurate calculations.

In summary, these prompts utilize various prompt engineering techniques such as role assignment, explicit instructions, zero-shot[6] and chain-of-thought[7]. The tags used (<begin_of_text>, <eot_id>, <start_header_id>) are compatible with LLaMA 3 and LLaMA 3.1 models capable of handling conversational structures and task-based dialogue, ensuring clear segmentation of tasks and minimizing output errors. Each prompt is optimized for specific functions: domain relevance classification, question rephrasing, and mathematical problem-solving.

4.10 Unit Tests

The unit tests, summarized in Table 3, are designed to verify the functionality and reliability of the system across various scenarios. They cover both individual components—such as domain relevance checks, question rephrasing, and document retrieval—and complete execution paths that mimic user interactions. These tests improve the likelihood that the system correctly classifies questions, retrieves and grades documents appropriately, handles mathematical computations, detects hallucinations in generated answers, and provides useful responses. By testing these aspects, we enhance the chatbot's robustness and user trust.

Table 3: Unit Tests Summary

Test File	**Purpose**
Graph Edges Tests	
`test_1_domain_relevant.py`	Tests whether a question is correctly classified as relevant or irrelevant to the domain.
`test_2_rephrase_domain_relevant.py`	Tests if a rephrased question maintains domain relevance.
`test_3_exist_retriever_docs.py`	Tests the retrieval and grading of documents for relevance.
`test_4_question_classifier.py`	Tests the classification of questions as math-based or text-based.
`test_5_computation_method.py`	Tests whether the system computes answers using Python or falls back to web-based solutions.
`test_6_answer_hallucination.py`	Tests the system's ability to detect hallucinations in generated answers.
`test_7_useful_answer.py`	Tests whether the generated answer is useful based on the question's context.
Path Execution Tests	
`test_path_1.py`	Tests the execution path for a relevant question leading to a useful answer without hallucinations.
`test_path_2.py`	Tests rephrased questions becoming relevant and leading to a useful answer.
`test_path_3.py`	Tests handling of irrelevant questions resulting in a fallback response.
`test_path_4.py`	Tests the scenario where no relevant documents are found, even after a web search, leading to a fallback response.
`test_path_5.py`	Tests mathematical questions computed locally using Python.
`test_path_6.py`	Tests mathematical questions computed using web-based solutions.
`test_path_7.py`	Tests the system generating an answer that is not useful, resulting in a fallback response.

5 Practical Outcomes

The implemented system effectively addresses the weaknesses of LLM-based chatbots by incorporating document retrieval to minimize hallucinations. By retrieving relevant documents before generating responses, the system provides fact-based answers, reducing the likelihood of incorrect or fabricated information. Users are also provided with metadata, including source filenames and URLs, to verify the information, which further strengthens the reliability of the system.

The integration of prompt engineering techniques and a specialized model for mathematical reasoning reduces the dependency on precise query phrasing and improves accuracy in solving arithmetical problems. By refining the prompt structure, the system ensures that responses remain relevant even with less structured or ambiguous input. For mathematical queries, the system uses a dedicated model to process and deliver more accurate calculations, overcoming a key limitation of conventional LLMs in handling mathematical tasks.

6 Limitations

While the RAG system improves chatbot performance across multiple dimensions, it still faces several limitations:

The system utilizes the NumExpr module for mathematical reasoning and computations. NumExpr primarily supports basic arithmetic, logical operations, and simple mathematical functions, which means it may not handle advanced calculus, symbolic manipulations, or higher-order algebraic equations effectively. As a result, certain mathematical queries that require deeper symbolic reasoning or specialized mathematical knowledge may still produce incorrect or incomplete results.

Although the RAG system significantly reduces hallucinations by retrieving factual information from external sources, hallucinations may still occur in cases involving highly complex or ambiguous queries. If the retrieved documents are not sufficiently relevant or if the system cannot find any external data, the LLM may default to generating plausible-sounding but incorrect information. This can be particularly problematic for queries that span multiple domains or require nuanced understanding, where retrieval alone may not fully address the context of the question.

The multi-stage pipeline implemented in the RAG system introduces potential delays in response times, not only due to document retrieval but also the multiple prompts embedded within the system. Each stage of the pipeline, including document retrieval from large internal databases or external sources (e.g., via DuckDuckGo API), question classification, document grading, and prompt-based operations, adds to the overall processing time. This multi-step process, while improving accuracy, can be time-consuming, particularly when handling complex queries. In real-time applications, these delays may degrade the user experience, especially when fast response times are expected.

The current implementation of the RAG system primarily targets English language. Extending the system to support multiple languages may require significant modifications, reconfiguring retrieval mechanisms for multilingual documents, and adapting prompt engineering strategies. Without a robust multilingual model or pipeline, the system's effectiveness is constrained in multilingual environments.

Another limitation arises from the token constraints of large language models, which prevent the system from processing large documents in full. To work around this, the document is first split into smaller chunks, and all chunks are processed by the LLM for summarization. While techniques like map-reduce[30] summarization could help manage this issue, they significantly increase processing time, making it difficult to balance efficiency and accuracy.

7 Conclusion

In this work, a RAG chatbot was developed to address key limitations of large language models, including hallucinations, mathematical inaccuracies, and dependency on precise prompt engineering. By integrating external document retrieval systems and specialized mathematical reasoning modules, the chatbot delivers more accurate and reliable responses. The use of open-source frameworks ensures the system's accessibility and adaptability, allowing for broader application across various domains. Additionally, structured prompt engineering enhances the chatbot's ability to handle complex queries and maintain contextual relevance, thereby improving overall user trust and experience.

A significant feature of the system is its strict adherence to a predefined domain, which significantly enhances the relevance and accuracy of the chatbot's responses. Furthermore, the history aware retriever function that combines the current question with past chat history, retrieving relevant documents based on the accumulated context. This capability ensures that the chatbot can understand the progression of the conversation, leading to more accurate and contextually appropriate flow.

However, the system is not without its challenges. It currently struggles with advanced mathematical tasks, may still produce occasional hallucinations, and experiences response delays due to its multi-stage processing pipeline. Furthermore, its functionality is primarily limited to the English language, restricting its usability in multilingual environments. Future work could focus on integrating more sophisticated computational tools, optimizing retrieval and processing efficiency to reduce latency, and expanding language support to enhance the chatbot's versatility. Overall, the RAG-based approach demonstrated in this work represents an advancement toward creating more dependable and versatile conversational AI systems.

References

[1] JDMCK Lee and K Toutanova. Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 3 (8), 2018.

[2] Alec Radford. Improving language understanding by generative pre-training. 2018.

[3] A Vaswani. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017.

[4] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems*, 33:9459–9474, 2020.

[5] Ggaliwango Marvin, Nakayiza Hellen, Daudi Jjingo, and Joyce Nakatumba-Nabende. Prompt engineering in large language models. In *International conference on data intelligence and cognitive informatics*, pages 387–402. Springer, 2023.

[6] Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

[7] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed H. Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. URL `https://arxiv.org/abs/2201.11903`.

[8] SM Towhidul Islam Tonmoy, SM Mehedi Zaman, Vinija Jain, Anku Rani, Vipula Rawte, Aman Chadha, and Amitava Das. A comprehensive survey of hallucination mitigation techniques in large language models.

[9] Shehzaad Dhuliawala, Mojtaba Komeili, Jing Xu, Roberta Raileanu, Xian Li, Asli Celikyilmaz, and Jason E Weston. Chain-of-verification reduces hallucination in large language models. In *ICLR 2024 Workshop on Reliable and Responsible Foundation Models*.

[10] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In *ACL*, 2023.

[11] Hyung Won Chung, Le Hou, Shayne Longpre, Barret Zoph, Yi Tay, William Fedus, Yunxuan Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, et al. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

[12] Ziwei Ji, Zihan Liu, Nayeon Lee, Tiezheng Yu, Bryan Wilie, Min Zeng, and Pascale Fung. Rho: Reducing hallucination in open-domain dialogues with knowledge grounding. In *Findings of the Association for Computational Linguistics: ACL 2023*, pages 4504–4522, 2023.

[13] Weijia Shi, Xiaochuang Han, Mike Lewis, Yulia Tsvetkov, Luke Zettlemoyer, and Scott Wen-tau Yih. Trusting your evidence: Hallucinate less with context-aware decoding. *arXiv preprint arXiv:2305.14739*, 2023.

[14] Yung-Sung Chuang, Yujia Xie, Hongyin Luo, Yoon Kim, James R Glass, and Pengcheng He. Dola: Decoding by contrasting layers improves factuality in large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

[15] Debrup Das, Debopriyo Banerjee, Somak Aditya, and Ashish Kulkarni. Mathsensei: A tool-augmented large language model for mathematical reasoning. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 942–966, 2024.

[16] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[17] OpenAI. Chatgpt, 2023. URL `https://openai.com/chatgpt/`. Accessed: 2024-09-24.

[18] Anthropic. Claude ai, 2023. URL `https://www.anthropic.com/claude`. Accessed: 2024-09-24.

[19] DeepMind. Gemini ai, 2023. URL `https://deepmind.google/technologies/gemini/`. Accessed: 2024-09-24.

[20] PaperChat. Paperchat: Ai chat with documents, 2023. URL `https://www.paperchat.io/`. Accessed: 2024-09-24.

[21] ChatDoc. Chatdoc: Ai chat with pdf documents, 2023. URL `https://chatdoc.com/`. Accessed: 2024-09-24.

[22] Sharly. Sharly ai: Chat with any document and pdf, 2023. URL `https://sharly.ai/`. Accessed: 2024-09-24.

[23] Ollama. Ollama, 2024. URL `https://github.com/ollama/ollama`. Accessed: 2024-09-24.

[24] Harrison Chase. LangChain, October 2022. URL `https://github.com/langchain-ai/langchain`.

[25] Jianguo Wang, Xiaomeng Yi, Rentong Guo, Hai Jin, Peng Xu, Shengjun Li, Xiangyu Wang, Xiangzhou Guo, Chengming Li, Xiaohai Xu, Kun Yu, Yuxing Yuan, Yinghao Zou, Jiquan Long, Yudong Cai, Zhenxiang Li, Zhifeng Zhang, Yihua Mo, Jun Gu, Ruiyi Jiang, Yi Wei, and Charles Xie. Milvus: A purpose-built vector data management system. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2614–2627, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457550. URL `https://doi.org/10.1145/3448016.3457550`.

[26] Streamlit Inc. Streamlit: The fastest way to build and share data apps. `https://streamlit.io/`, 2024. Accessed: 2024-09-24.

[27] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. *arXiv preprint arXiv:2401.08281*, 2024.

[28] Pinecone Systems Inc. Pinecone: Vector database documentation. `https://docs.pinecone.io/`, 2024. Accessed: 2024-09-24.

[29] Chroma. Chroma: The open-source ai application database. `https://www.trychroma.com/`, 2024. Accessed: 2024-09-24.

[30] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.