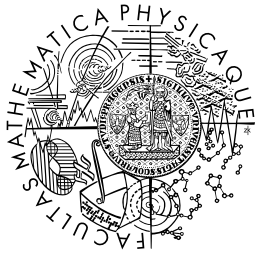

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Martin Lejko

Using LLM to create a knowledge base from documents

Department of Software Engineering

Supervisor of the bachelor thesis: Mgr. Petr Škoda, Ph.D.
Study programme: Computer Science

Prague 2025

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to sincerely thank my parents for their unwavering support throughout my studies and the writing of this thesis. My gratitude also goes to my supervisor, Mgr. Petr Škoda, Ph.D., for his valuable insights and guidance, as well as to Adam Ruzicka and Peter Moran from Deutsche Börse Group for their helpful consultations and support. Thank you all for making this journey possible.

Title: Using LLM to create a knowledge base from documents

Author: Martin Lejko

Department: Department of Software Engineering

Supervisor: Mgr. Petr Škoda, Ph.D., Department of Software Engineering

Abstract: Large language models (LLMs) face challenges when applied to enterprise documents, including outdated knowledge, hallucinations, and privacy risks associated with cloud-based services. This thesis addresses these problems by exploring a fully local implementation of retrieval-augmented generation on standard personal computer to ensure complete data privacy. The research focuses on analyzing document characteristics, particularly financial reports, and building an initial proof-of-concept pipeline using open-source tools such as Ollama and quantized LLMs. The work adopts an iterative development approach, refining the system through targeted modifications of individual pipeline components. Each iteration is systematically evaluated using an LLM as a judge to assess changes in performance and quality. The final outcome is a functional retrieval-augmented generation pipeline that demonstrates the practical feasibility of secure, privacy-preserving information retrieval on a personal computer. This thesis provides insights into optimizing local configurations, evaluates trade-offs under resource constraints, and offers a novel assessment framework for improving retrieval-augmented generation pipelines.

Keywords: fine-tuning, language model, natural language processing, AI

Název práce: Použití LLM k vytvoření znalostní databáze nad dokumenty

Autor: Martin Lejko

Katedra: Katedra softwarového inženýrství

Vedoucí bakalářské práce: Mgr. Petr Škoda, Ph.D., Katedra softwarového inženýrství

Abstrakt: Velké jazykové modely (LLM) čelí při použití v oblasti podnikových dokumentů řadě výzev, včetně zastaralých znalostí, halucinací a rizik spojených s ochranou soukromí při využívání cloudových služeb. Tato práce se těmito problémy zabývá prostřednictvím návrhu plně lokální implementace retrieval-augmented generation na běžném osobním počítači s cílem zajistit úplnou ochranu dat. Výzkum se zaměřuje na analýzu charakteristik dokumentů, zejména finančních zpráv, a na vytvoření počátečního proof-of-concept řešení pomocí open-source nástrojů, jako je Ollama a kvantované LLM. Práce využívá iterativní přístup k vývoji systému, kdy je systém postupně vylepšován cílenými úpravami jednotlivých komponent pipeline. Každá iterace je systematicky hodnocena pomocí LLM, který slouží jako hodnotitel změn výkonu a kvality. Výsledkem je funkční pipeline pro retrieval-augmented generation, která ukazuje praktickou proveditelnost bezpečného a soukromí chránícího vyhledávání informací na osobním počítači. Tato práce přináší poznatky o optimalizaci lokálních konfigurací, hodnotí kompromisy při omezených zdrojích a nabízí nový rámec pro hodnocení a zlepšování retrieval-augmented generation pipeline.

Klíčová slova: doladění, jazykový model, zpracování přirozeného jazyka, AI

Contents

Introduction

Large Language Models for short LLMs, are cutting-edge natural language processing (NLP)[1] development designed to understand and generate human language. LLMs are advanced AI models trained on vast amounts of text data, enabling them to recognize linguistic patterns, comprehend context, and produce coherent and contextually relevant responses [1]. As they have demonstrated remarkable capabilities in understanding and generating human-like text, they hold great potential for both individual users and large enterprises. This thesis, however, will concentrate on applications within corporate settings, where data privacy, regulatory compliance, and document scale pose distinct challenges and opportunities. LLMs could transform how large organizations and businesses manage and interact with their information, presenting opportunities to improve internal processes. Opportunities such as improved employee access to domain-specific expertise or accelerated on-boarding for new colleagues are becoming more realistic goals. [2] However, the practical application of these models faces significant hurdles. LLMs often operate with data from their last training, making them unaware of recent events or information. [3] This is not their only flaw. Generative LLMs have been observed to confidently assert claims of fact which do not seem to be justified by their training data, a phenomenon which has been termed "hallucination". [4] Specifically, hallucinations in the context of LLMs correspond to the generation of text or responses that seem syntactically sound, fluent, and natural but are factually incorrect, nonsensical, or unfaithful to the provided source input. [5] Perhaps most critical flaw, mainly in enterprise contexts is leveraging LLMs with private, company sensitive data as it poses substantial security and privacy risks when relying on external, cloud-bases services. [6]
Retrieval-Augmented Generation (RAG) has emerged as a relevant paradigm in this context. It is an AI framework that combines the strengths of traditional information retrieval systems, such as search and databases, with the capabilities of generative large language models. [7] An overview of the RAG framework is illustrated in Figure 1.

This framework has two main benefits. It ensures that the model has access to the most current, reliable facts, and that users have access to the model's sources, ensuring that its claims can be checked for accuracy and ultimately trusted. As we can cross-reference the answers of a model with the original content. RAG has additional benefits. By grounding an LLM on a set of external, verifiable facts, the model has fewer opportunities to pull information baked into its parameters. This reduces the chances that an LLM will leak sensitive data, or 'hallucinate' incorrect or misleading information. [9] At the time of the query, specific request for information, RAG instead of solely relying on the model's internalized which can be potentially outdated knowledge, it first searches a specified data source for example a collection of company documents, slack messages for facts that are relevant to the user's query. Once retrieved, the relevant information undergoes pre-processing; process in which data is prepared for analysis. The pre-processed retrieved information is then seamlessly incorporated

[1]Natural Language Processing (NLP) is an AI field focused on computer-human language interaction. See Wikipedia: Natural Language Processing for an overview.

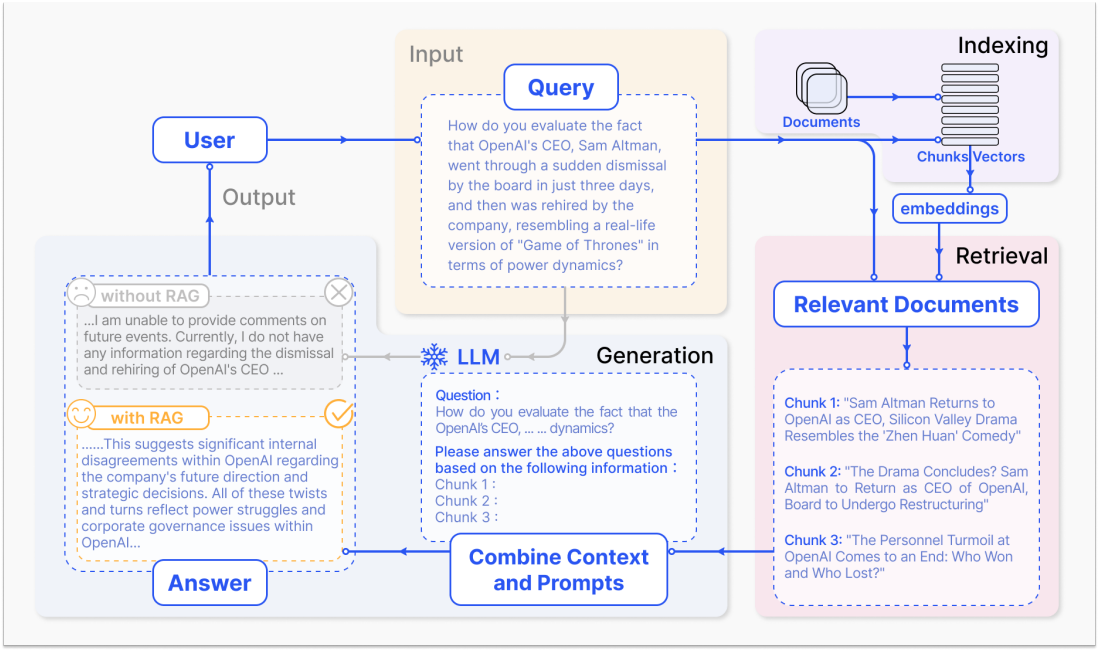

Figure 1 An example of typical RAG pipeline. Sourced from [8].

into the LLM. This integration enhances the LLM's context, providing it with a more comprehensive understanding of the topic. This augmented context enables the LLM to generate more precise, informative, and factually grounded responses. [7]

Implementing an effective RAG system involves several interconnected concepts and processes which work together to generate relevant answer; hereafter, we will refer to this as the RAG pipeline. This representative RAG pipeline mainly consists of 3 steps. If we are not counting the question to which the LLM system responds is referred to as the input. If no RAG is used, the LLM is directly used to respond to the question. [10]

1. **Indexing:** In this phase, the relevant documents or data sources are indexed so that they can be efficiently retrieved later. Indexing is a process where documents are broken into smaller chunks, making them easier to search and fit within a model's finite context window. [11]. The context window of an LLM is the amount of text, in tokens (a collection of characters that has semantic meaning for a model), that the model can consider or 'remember' at any one time. [12] We can be adding metadata fields here that the pipeline creates based on the content of the chunks. We can categorize the metadata into discrete fields, such as title, summary, and keywords. Followed by using an embedding model to vectorize the chunk and any other metadata fields that are used for vector searches. [13] Embedding models are models that are trained specifically to generate vector embeddings: long arrays of numbers that represent semantic meaning for a given sequence of text. [14] In the final step of indexing, we need to store the vector embeddings, which is often done using vector store. [11] Vector stores are specialized data stores that enable indexing and retrieving information based on vector representations. [15]

2. **Retrieval:** In this step, the LLM system searches the indexed documents

to retrieve relevant information that can aid in answering the question. The relevant information is obtained by comparing the query against the indexed vectors. [11] The vector store will embed the query, perform a similarity search [2] over the embedded documents, and return the most similar ones [15].

3. **Generation:** Finally, the retrieved relevant documents are combined with the original query as an additional context. The combined text and query are then passed to the model for response generation, which is then prepared as the final output of the system to the user. [10]

The main goal of this thesis is to systematically explore, implement, and evaluate RAG pipelines. While maintaining emphasis on creating solutions that uphold data privacy. This is achieved by local processing, while being able to operate efficiently on standard personal computers. For a standard personal computer we take Macbook Air M1, 2020 with 16 GB of RAM. Most often we meet with RAG implementations that rely on public APIs or cloud infrastructure. These are not suitable for scenarios involving private data or for users who do not have access to significant computational resources. This work aims to explore how achievable it is to build a functional RAG pipeline within these constraints. To validate this, the research adopts an iterative approach. It begins by creating Proof of Concept (PoC); also known as proof of principle, is an inchoate realization of a certain idea or method in order to demonstrate its feasibility or viability. [16] We continue to transform the initial PoC into a more abstract and software-engineered solution to enhance extensibility and reduce the effort required to swap out different sections of the pipeline. This transformation will represent the "0th iteration". Subsequent iterations are formed by targeted experiments, modifying individual sections or "building blocks". Building blocks are targeted sections of an RAG pipeline on which we experiment through iterations. By progressively combining these building blocks, a fully functioning RAG pipeline is assembled. With these iterative experiments, we examine and evaluate the impact on the performance and quality of the said pipeline. This iterative process can provide a deeper understanding of the impact produced by each section. Identify trade-offs and effective configurations for local RAG solution.

The local implementation of RAG is made possible by recent advances in open-source LLMs and their supporting frameworks. The key utilization in this thesis is Ollama [3]. Ollama is a tool designed to simplify the process of running open-source LLMs directly on your computer. It acts as a local model manager and runtime, handling everything from downloading the model files to setting up a local environment where you can interact with them. [17] These models, such as the Llama 3.1:8b, which we use throughout the thesis, are being quantized to make them runnable on standard personal computers. Quantization is a technique utilized within large language models to convert weights and activation values of high precision data, usually 32-bit floating point (FP32) or 16-bit floating point (FP16), to a lower-precision data, like 8-bit integer (INT8). [18]

This thesis is structured as follows: Following the Introduction, Chapter 1 presents the analysis of the problem, including the background, motivation,

[2] See Wikipedia: Similarity Search for more details.

[3] Ollama serves as the primary framework in this work for managing, and running LLMs locally. Project homepage: ollama.com

objectives, and scope. Chapter 2 introduces the core RAG pipeline concept, detailing its building blocks, the iterative approach adopted, and an initial proof of concept. Chapter 3 delves into the evaluation methodology, covering challenges, the LLM-as-a-Judge approach, selected metrics, limitations, the testing framework implementation, and the creation of the evaluation dataset. Chapters 4 and 5 present the core experimental work through an iterative development process: Chapter 4 focuses on experiments related to data preparation, cleaning strategies, chunking methods, and embedding models, while Chapter 5 addresses LLM selection and prompt optimization techniques, analyzing the results of each step. Chapter 6 provides practical user documentation for setting up and running the developed system. Finally, the Conclusion 6.3 summarizes the key findings and contributions.

During the writing process of this thesis, various tools were employed such as Writefull, Grammarly and LLMs to enhance clarity, correctness, and style of the text.

1 Problem analysis

This chapter lays the foundation for the thesis by analyzing the problem domain and defining concrete boundaries regarding system capabilities, hardware limitations, and the manageable complexity of input documents. Building on the discussion in Introduction, we then focus on how LLMs can be applied specifically in corporate environments, where private data, compliance, and scale introduce unique challenges, explaining the goals of this thesis and setting clear limits on what the work will cover. This prepares us for the following sections of this thesis, where we iterate upon RAG sections with aim to produce test results 3.3.

1.1 Background and Motivation

The advent of LLMs marks a significant technological shift, causing a potential shift in the way how large organizations manage and interact with information. As detailed in the Introduction, LLMs possess remarkable capabilities in understanding and generating human language. This can be leveraged to streamline internal processes and unlock valuable knowledge contained within vast repositories of corporate documents.

LLM usage is becoming increasingly relevant, with even large corporations like Shopify are now mandating the use of AI tools as part of daily work, requiring employees to use them for learning, problem-solving and discussing their solutions with them [19]. This shift highlights the growing importance of integrating AI into corporate workflows. By utilizing techniques such as RAG, we can further enhance this integration, unlocking more effective ways to access information and supporting a wide range of applications within business environments.

- **Importance of LLMs in modern business:** Potential applications within a corporate setting are numerous. As highlighted in the thesis assignment, LLMs can enhance better employee access to domain-specific expertise, which is often siloed within specific departments. They can also significantly accelerate the onboarding and training process for new colleagues by providing instant access to relevant historical project information or policies. Improve the speed of searching for relevant information within the corporation database. The main idea is to utilize LLMs to create intelligent knowledge bases, acting as gateways to corporate documentation. Which could potentially evolve into sophisticated systems for Intelligent Document Understanding. Intelligent Document Understanding is the ability for the technology to assist a subject matter expert in a document based process to come to some sort of decision or complete a process. [20] This promises substantial gains in operational efficiency and knowledge accessibility.

- **Challenges with handling private data:** Even though the potential is immense, the practical application of LLMs in enterprise environments faces significant obstacles, primarily concerning data privacy and security. As discussed in the Introduction, many advanced LLMs are cloud-based services. Utilizing these external services leads to processing internal, often confidential, company documents. If not careful, this could potentially

lead to substantial risks related to data breaches, unauthorized access, and compliance with data protection regulations [6]. This clash between the utility of LLMs and the drive to protect sensitive information is a key challenge in broader adoption. Furthermore, while LLM capabilities are rapidly advancing, the data aspect – ensuring data quality and preparing processes for its effective use – is often overlooked, potentially sabotaging successful implementation. This thesis tries to explore this challenge by experimenting with solutions that prioritize data privacy through local processing. While keeping in mind the inherent flaws and limitations of LLMs. Such as potential "hallucinations" and reliance on potentially outdated training data. [4]

1.2 Objectives

Based on the background and motivation outlined above, this thesis aims to systematically investigate the feasibility and effectiveness of using RAG pipelines in a private, locally-run environment. With the goal being to enable access to information from corporate documents while preserving data privacy and ensuring practical deployment on personal hardware. This investigation will follow an iterative process that gradually refines and evaluates the individual components of the RAG pipeline. And identify the most suitable configurations for the given constraints.

- **Goals of the thesis:**

  1. Conduct an analysis of selected corporate documents, with a particular emphasis on financial reports. Explore documents specific characteristics such as structure, tables, or graphs, and identify potential obstacles in their automated processing by LLMs.
  2. Prepare a portfolio of these documents based on the analysis.
  3. Identify suitable LLM models, specifically those runnable locally, and RAG approaches applicable for our context.
  4. Develop a Proof of Concept solution demonstrating the ability to access information from user-defined documents using a locally run RAG pipeline.1
  5. Define, explore, and implement a testing structure that enables systematic evaluation of RAG pipelines. This includes establishing scoring metrics and benchmarks. This enables us to compare iterations with one another, ensuring consistent progress and improvement. 3.3
  6. Define and explore potential levels of information access made possible with RAG, such as:
     - Responding to user queries with accurate information extracted from relevant documents, mainly financial documents.
     - Returning the specific documents identified as relevant to the query.

- Combining both approaches by providing relevant documents alongside generated responses.

7. Iterate on the individual components of the RAG pipeline and evaluate their combinations with the aim of refining the overall setup.

- **Expected outcomes:** This thesis is expected to deliver:

1. A functional PoC demonstrating a locally executable RAG pipeline for accessing information from a defined set of documents. With this in mind, we want to ensure that the thesis concept is possible to implement while remaining within the defined constraints 1.3.
2. A structured evaluation of different RAG configurations and approaches based on relevant benchmarking metrics.
3. An analysis of the specific challenges and considerations involved in processing financial reports, within a local RAG context.
4. Insights into the practical availability, trade-offs, and limitations of building and running private RAG solutions on standard personal computer.

1.3 Scope

The scope of this thesis is defined as follows:

- **Scope of Work:**
  - **Document Focus:** The analysis and implementation will concentrate on a curated set of document types, primarily financial reports. The goal is to understand the possibility of extracting information from such semi-structured documents locally. Semi-structured data is a form of structured data that does not obey the tabular structure of data models associated with relational databases or other forms of data tables, but nonetheless contains tags or other markers to separate semantic elements and enforce hierarchies of records and fields within the data. [21]
  - **Implementation Strategy:** The core effort involves constructing and evaluating RAG pipelines designed for entirely local execution with an iterative process. This constrains us to open-source, locally runnable tools—such as the Ollama ecosystem and frameworks like LangChain[1]—all of which are designed to run efficiently on standard consumer hardware.
  - **Testing and Evaluation:** Evaluation will assess the effectiveness of the implemented local RAG iteration. Key performance indicators (KPIs) will include qualitative assessments of retrieval relevance and correctness of response. Providing insights into practical usability and iteration impact.

[1]LangChain is an open-source framework designed to simplify the development of applications using LLMs, providing modular components relevant to RAG pipelines. See LangChain Documentation.

  - **Deliverable:** The primary outcome of this thesis is a refined RAG pipeline. The pipeline is representing the configuration that yielded the best results during the iterative testing. This pipeline integrates the most effective components that we explored within this work. Serving as a concrete demonstration of an optimized local RAG pipeline under the defined constraints.

- **Out of Scope:**
  - This work will *not* conduct an exhaustive analysis of all possible document types or tackle highly specialized formats. Such as complex CAD diagrams, deeply nested legal documents or documents specific for a certain organization.
  - This work does *not* involve the development, training, or fine-tuning of the underlying LLMs or embedding models themselves. Fine-tuning is an approach to transfer learning in which the parameters of a pre-trained model are trained on new data. [22] Pre-existing models will be used as is or in their readily available quantized forms.
  - Exploration or comparison of cloud-based RAG solutions or managed AI services is explicitly *excluded*. The focus remains strictly on local implementations. Without the use of APIs of said AI services or cloud bases LLM models. [2]
  - The development will *not* result in a production-grade, scalable system.
  - Large-scale benchmarking against a wide array of RAG pipelines is *not* planned. The evaluation is internal to the project's goals and constraints.
  - End-user experience is *outside the scope* of this technical study.

- **Limitations and Constraints:**
  - **Document Complexity Handling:** The PoC solution will likely face limitations in effectively processing highly complex or poorly structured documents. Particularly processing advanced tables / understanding or interpreting information solely conveyed through images or graphs. Reflecting the challenges identified in document analysis.
  - **Hardware Restrictions:** All development, testing, and benchmarks are constrained by the capabilities of the designated target hardware: a standard personal computer, in our case Macbook Air M1, 2020, 16 GB RAM. This directly limits the size and type of LLMs and embedding models that can be practically run locally. Potentially influencing the quality of the results.
  - **Tooling Choices:** The selection of LLMs, embedding models and other frameworks is primarily driven by their ability to run locally, their performance on standard personal computer, and their open-source nature. In addition to these technical criteria, particular emphasis

[2] An Application Programming Interface (API) defines how software components interact. See general definition at Wikipedia: API.

is placed on how each tool handles private data. Tools that process or transmit data externally are excluded. To mitigate risk of leaking private information.

– **Rapidly Evolving Landscape:** The field of LLMs and associated tooling is progressing at an extremely rapid pace.

  Consequently, by the time this thesis is read there may already be newer models or more advanced techniques available. Accordingly, the models and frameworks used here represent the snapshot as of December 2024, providing a fixed reference point for our experiments.

2 Retrieval-Augmented Generation Pipeline

Retrieval Augmented Generation is a powerful technique that enhances large-language models by allowing them to retrieve additional information. This allows an LLM to respond to user queries with an altered response based on the reference to the selection of documents. As mentioned in Introduction, this approach is mainly used to allow a large-language model to use data and include them into the models operations. To implement this technique, we created a pipeline which we discuss in this chapter. In Section 2.1, we introduce a simple RAG implementation and then, in Section 2.2, break it into discrete "building blocks". Here we will further explain the implementations of each building block and better our solution.

2.1 Proof of Concept

In this section, we will introduce a straightforward solution as our initial proof of concept. As stated in the Introduction, following this proof of concept, we will further investigate its workings that will allow us to build a good base of understanding how the technique behaves. This methodological procedure ensures that we do not proceed blindly, but instead have validated insights on which we further our implementation. This should help prevent any backtracking. Ultimately, the aim is to create the final PoC, which will provide us with the necessary information to guide the next steps of the thesis.

To effectively illustrate the RAG concept, we will utilize a local implementation that leverages the Lang Chain framework. It builds on the local RAG implementation approach described in this guide [23]. These resources, specifically the LangChain documentation and the described RAG implementation approach in the guide [23], provide the fundamental framework for our implementation, which we have adapted and enhanced for our specific use case 1.2.

2.1.1 0th iteration

Implementation of this pipeline was quite easy as there are many resources on the internet that are targeting RAG as it is quite popular topic for AI at the time of writing this thesis (2024/2025). We aimed for Lang Chain implementation as it is not that long and fits our goals. We chose LangChain because of its comprehensive set of tools and libraries. This simplifies implementations of complex language model functionalities in Python. For reference, in this 0th iteration we used:

- `langchain-text-splitters` 0.3.7
- `langchain-ollama` 0.3.0
- `langchain-chroma` 0.2.2
- `pypdf` 5.4.0

- `langchain-community` 0.3.20

The decision to use Python as the primary programming language was driven by its broad ecosystem of AI libraries, making it well-suited for fast-paced prototyping. Following the LangChain documentation and the RAG implementation approach described in the mentioned guide [23], and with some small tweaks, we got a pipeline. The main goal if to create a pipeline that takes a question, retrieves relevant context, and generates an answer using an LLM.

Listing 2.1 Creating the question–answering chain

```
def create_qa_chain(
    vectorstore: Chroma,
    model: ChatOllama,
    prompt: ChatPromptTemplate
) -> Runnable:
    """Create the -questionanswering chain."""
    # derive a Retriever from our VectorStore
    retriever = vectorstore.as_retriever()

    # assemble: retrieve → format → prompt → model → parse
    return (
        {"context": retriever | format_docs,
         "question": RunnablePassthrough()}
        | prompt
        | model
        | StrOutputParser()
    )
```

Listing 2.1 defines our core chain builder. It accepts three inputs—a Chroma vector store, a ChatOllama model, and a ChatPromptTemplate—and wires them into a single, sequential pipeline:

1. **Retriever derivation**: `vectorstore.as_retriever()` produces a Retriever that, when invoked, returns the most relevant `Document` chunks from the vector store.

2. **Formatting**: Retrieved chunks are passed through `format_docs` to clean and normalize them.

3. **Prompting**: The formatted context and the user's question are injected into our prompt template.

4. **Generation**: The prompt is sent to the LLM, and its raw output is parsed into a string.

Before this chain runs, we must populate the vector store with documents. We do this via our load_data() helper 2.2, which uses LangChain's `WebBaseLoader` to fetch HTML content over HTTP/HTTPS. Load_data() returns a list of `Document` objects—each containing the page text and metadata—that we then split, embed, and insert into the vector store. In general, we load raw data via our load_data helper 2.2, which returns a list of 'Document' objects ready for splitting. For this proof of concept, we passed in a set of HTTP/HTTPS URLs This specific 'WebBaseLoader' supports loading content only over HTTP/HTTPS. The retrieved context, after being identified through similarity search, that is

the default setting of VectorStoreRetriever[1], is formatted appropriately, in this specific case it is stripped of extra whitespaces, before being merged with our prompt template 2.5. This formatted context, along with the original question, is then passed sequentially through the prompt template and the language model within the defined chain. At the end we get output as a string. This is essentially the whole process behind the straightforward pipeline. Lets go over the smaller steps that have been taken in first iteration.

The first step in our Retrieval-Augmented Generation pipeline involves loading custom information, which the language model would not have access to otherwise. This is achieved using the load_data function as shown in Listing 2.2.

Listing 2.2 Loading data from Urls

```
def load_data(urls: List[str]) -> List[Document]:
    loader = WebBaseLoader(urls)
    return loader.load()
```

Leveraging the capabilities of libraries like LangChain in Python, these concise functions take a list of URLs and return a list of Document objects. Here, each document object contains all the text and metadata for the specific URL. This approach, using the WebBaseLoader, was initially adopted from a LangChain tutorial to establish a basic RAG pipeline. However, it's important to note that this method is primarily for loading generic web content over HTTP/HTTPS. For the remainder of this thesis, our focus will be on processing financial data, as outlined in the Objectives section. Therefore, subsequent implementations will be designed to handle financial documents and reports, rather than generic web pages. We have written more about the dataset in Section 3.7. The process typically involves sending an HTTP request to the specified URL, parsing the HTML content, extracting the relevant text, and potentially retrieving metadata such as the header or title, depending on the website's structure and the capabilities of the WebBaseLoader. [24]

After loading the Document objects, the next step is to pre-process it into manageable chunks using the function process_data 2.3. This step is essential to ensure that the model and the vector store can handle the data efficiently.

Listing 2.3 Processing data into chunks

```
def process_data(documents: List[Document], chunk_size: int = 500,
    chunk_overlap: int = 50) -> List[Document]:
    """Split documents into chunks for processing."""
    text_splitter = RecursiveCharacterTextSplitter(
        chunk_size=chunk_size,
        chunk_overlap=chunk_overlap
    )
    return text_splitter.split_documents(documents)
```

Here, the function takes the documents loaded earlier and splits them into smaller chunks using the RecursiveCharacterTextSplitter class. [2] This is also

[1]The `VectorStoreRetriever` in LangChain handles querying the vector store. By default, as used here, it performs a similarity search to find document chunks relevant to the input query. See VectorStoreRetriever API Reference.

[2]The `RecursiveCharacterTextSplitter` attempts to split text based on a prioritized list of separators aiming to keep semantically related text together within chunks. See RecursiveCharacterTextSplitter Documentation.

provided by the Lang Chain package. The chunk_size parameter controls the size of each chunk defaulting to 500 characters, while the chunk_overlap parameter with a default of 50 ensures that there is no or minimal overlap between chunks. As it is the 0th iteration, we will leave it to default settings and get to it later in our process. Next, we persist those chunks in a vector store—LangChain's in-memory implementation for our PoC—which is the simplest and most common approach for semantic retrieval. Vector stores are specialized data stores that enable indexing and retrieving information based on vector representations. These vectors, called embeddings, capture the semantic meaning of data that has been embedded. Vector stores are frequently used to search over unstructured data, such as text, images, and audio, to retrieve relevant information based on semantic similarity rather than exact keyword matches. [15] These vector stores enable efficient similarity searches by finding the closest vectors to a query vector using techniques like the approximate nearest neighbor search [25]. That enables the model to search fairly efficiently for our user provided data and append it together with our query. For this initial proof of concept, we will utilize an in-memory vector store provided by LangChain, which allows for quick experimentation. This is handled by the setup_model function shown below 2.4. As there is not much to this function, then setting up the model we will use, we can transition to the more import and harder parts of the PoC.

Listing 2.4 Initializing the language model

```
def setup_model(model_name: str = "llama3.1:8b") -> ChatOllama:
"""Initialize and return the language model."""
    return ChatOllama(model=model_name)
```

This function initializes our language model using the ChatOllama class[3]. This class is part of the LangChain library and acts as an interface to large language models run locally via the Ollama framework. Ollama enables users to easily download and run various open-source LLMs, such as the one specified by model_name, directly on their personal computers. [17]

Here, we specify the llama3.1:8b model identifier used by Ollama [4]. This refers to a quantized version of the Llama 3.1 model with 8 billion parameters. Quantization is a technique utilized within large language models to convert weights and activation values of high precision data, usually 32-bit floating point (FP32) or 16-bit floating point (FP16), to a lower-precision data, like 8-bit integer (INT8). [18] Given our goal of running the pipeline on personal devices as defined in Section 1.2, this 8-billion parameter model represents a suitable size for our use case. That is, prompting and querying our model. A crucial aspect of interacting with any LLM is the prompt: a translational bridge between human communication and the computational abilities of LLMs. It is comprised of natural language instructions that act as an intermediary language, translating human requests into machine-executable tasks [26].

The effectiveness of the LLM heavily depends on how this prompt is formulated – a practice known as prompt engineering. Prompt engineering is the strategic

[3]The `ChatOllama` class in LangChain provides a standardized interface to interact with various LLMs served locally through the Ollama framework. See ChatOllama Integration Documentation.

[4]This identifier refers to the specific version of the Llama model used in this work, as available on the Ollama model library. See Ollama: llama3.1:8b model card.

process of designing prompts that guide LLMs to generate specific, relevant outputs. For systematic applications like RAG, where we repeatedly query the model with different data, using a prompt template is essential. Templates provide a standardized structure with placeholders, like context and question in our case, that can be dynamically filled with relevant information at runtime. This ensures consistency and directs the model on how to process the provided context and user query effectively. [27] The create_prompt_template function, shown in Listing 2.5, defines the specific template used in our initial RAG pipeline. The prompt template used here is based on the initial RAG tutorial from LangChain [11] and serves as our 0th-iteration baseline. In later chapter, we will experiment with alternative prompt designs to optimize accuracy and reduce hallucinations 5.2.

Listing 2.5 Creating the RAG prompt template

```
def create_prompt_template() -> ChatPromptTemplate:
"""Create and return the RAG prompt template."""
template = """
You are an assistant for question-answering tasks. Use the following
pieces of retrieved context to answer the question. If you don't
    know
the answer, just say that you don't know.
Use three sentences maximum and keep the answer concise.
<context>
{context}
</context>
Answer the following question:
{question}
"""
return ChatPromptTemplate.from_template(template)
```

This specific template plays a crucial role in our RAG implementation by establishing a consistent and structured format for query processing. In our case, it includes two main components: a context section marked by XML-style tags, which will be populated with relevant information from our vector store, and a question section that contains the user's query. By using these placeholders, we get a flexible template that can be dynamically populated with information during run-time. In addition, the template includes specific instructions, such as limiting responses to a maximum of three sentences to encourage conciseness. It is important to view this prompt, including constraints like the sentence limit, as part of our 0th iteration - a baseline, which was inspired by the tutorial. These parameters are yet not optimized and will be experimented on in Chapter 5. The prompt template and other previous parts are then integrated into a comprehensive question-answering chain through the create_qa_chain function, as shown in Listing 2.1

2.2 Building Blocks of the RAG Pipeline

The following outlines the parts of the pipeline that we have introduced in the previous section. Along with breaking the big picture 2.1 into smaller parts. Basically, segmenting it for us to focus only on some area of the RAG pipeline. These parts we call building blocks as together they build the whole pipeline, and we can take some block out and substitute it quite easily for different block

with the same purpose. This modularity is available mainly because of the design principles of libraries like LangChain. Which often define common interfaces or base classes for component types. For example a WebBaseLoader which we used in the load_data function 2.2, conforms to a DocumentLoader. This allows us to swap the WebBaseLoader for a different document loader for example PyPDFLoader, if we wanted to use PDF files rather then HTTP urls. [5] This allows for experimentation and adaptation with relative ease. Not depending on what operations are performed on the inside of the block itself. This provides clarity and enables focusing only on one building block at once. Identifying these blocks is not complicated as it can be easily derived from the source code and functions we used in Chapter 2.1. We tried to visualize these blocks in Figure 2.1.

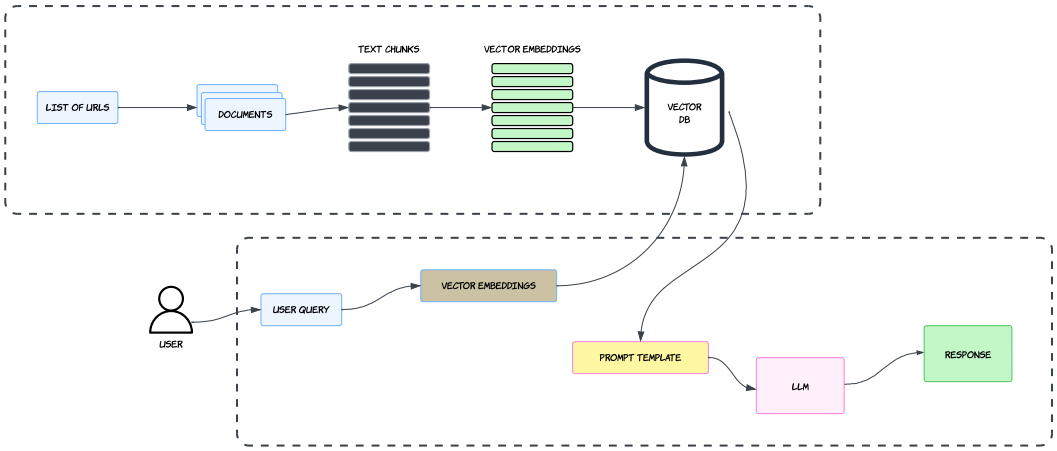

Figure 2.1 Visualization of RAG sections.

Each building block in our RAG pipeline serves a specific purpose and contributes to the overall functionality of the system. As seen from the graph, we can separate it into two big parts, the data processing chain and more of a model input/output part. We had a list of URLs in our proof of concept, which we will substitute for actual user provided data. We specify data getting as one of our building blocks, as without data we cant build RAG. This block is not as easily switchable as the others because the data is linked to an area we want to query about. These are mainly financial documents, such as the annual reports of the ETF fonds mentioned in Chapter 1. Rather then switching we can expand the block by getting additional data from different sources. We can follow this up with storing it, in the proof of concept we have used the vector store 2.1. As we progress further, we will experiment with different approaches and use evaluation metrics that we define in Chapter 3.3. Because indexing and retrieval operate on the same data structures, we treat them as a single, coupled block rather than splitting them into separate steps. Following the flow chart, we follow up with the prompt template. Many sources point out that we can increase accuracy and decrease hallucination of our model without the need for fine-tuning [28].

[5] LangChain defines a standard `DocumentLoader` interface, ensuring that different loaders like `WebBaseLoader` or `PyPDFLoader` can be used interchangeably within the pipeline if they adhere to this common structure. See LangChain Document Loaders Overview.

Same as with data storing we experiment with more approaches but this article points out "Large language models, they exhibit significant drawbacks, particularly when processing long contexts. Their inference cost scales quadratically with respect to sequence length, making it expensive for deployment in some real-world text processing applications, such as RAG." [28] Which for us as we are working with big amounts of data that we want to search trough, is a big problem. The article suggests that by implementing superposition prompting for their naive RAG "Their approach facilitates a 93× reduction in compute time while improving accuracy by 43% on the NaturalQuestions-Open dataset with the MPT-7B instruction-tuned model over naive RAG" [28] We do a deeper dive into promting and doing experiments with promting techniques in Section 4.11. Last in line is a model building block. We already discussed this matter in Section 1.3 and also in Section 2.4. Where we have some technological constrains, with what we work with as the pipeline must work on personal computer. This focus on local execution implies that our primary set of candidate models shrinks considerably. However, while optimizing for these constrained environments, it can still be valuable to benchmark performance against larger, state-of-the-art models. To conclude these are all the building blocks we have mentioned in this section.

- Extract data
- Data cleaning
- Data pre-processing
- The embeddings
- Initialize model
- Prompt templates

2.3 Structured Pipeline Approach

In this section, we outline the key building blocks of our RAG pipeline and how they contribute to the overall system. While there are many potential configurations and optimizations, we define a structured approach to ensure data integrity, efficient retrieval, and high-quality generation. The pipeline consists of the following essential components:

- **Extracting Data**: Collecting relevant financial reports and structured documents.
- **Data Cleaning**: Removing inconsistencies, ensuring tables and equations remain intact, and preventing corrupted retrieval.
- **Data Preprocessing**: Structuring the data using metadata, hierarchical indexing, and segmentation techniques.
- **Embeddings Generation**: Converting structured information into a searchable vector format.

- **Initializing the Model**: Selecting a lightweight model capable of running efficiently on personal hardware.
- **Prompt Engineering**: Designing optimized prompt templates to minimize hallucinations and improve response accuracy.

These building blocks form the backbone of our RAG pipeline. The pipeline is implemented in an iterative manner, with continuous refinements based on performance evaluations. The source code and current implementation of the 0th iteration can be found in: 0th implementation

A deeper exploration of experimental techniques—including data storage methods, retrieval optimizations, and prompting strategies—follows in Chapter 4.

3 Evaluation of RAG Pipeline

A successful RAG pipeline needs proper evaluation. While a pipeline might appear functional upon initial deployment, rigorous testing is crucial to ensure its reliability, accuracy, and effectiveness in the long run. Superficial success might hide underlying issues or limitations that only systematic evaluation can reveal. This is especially true when the goal is to create a system that helps users access complex corporate documents. In our case, these documents are financial reports. They often include graphs, tables, and semi-structured content. Because of this, robust and meaningful evaluation becomes even more important [29]. As outlined in the goals of this thesis, the RAG system aims to enhance information accessibility within a large company, demanding reliability and accuracy 1.2. This chapter details the methodology and framework established to evaluate the performance of the RAG pipeline developed in this work.

The evaluation process is crucial to the iterative development approach adopted in this thesis. After each experiment involving modifications to the pipeline's sections as discussed in Chapter 2.2, a systematic evaluation is conducted to measure the impact of these changes. This requires a well-defined set of evaluation criteria and a reliable testing framework to ensure that improvements are quantifiable and aligned with the thesis's objectives. This chapter will outline the challenges that we encounter in evaluating RAG systems, the chosen evaluation methodology, the specific metrics used, the implementation of the testing framework, and the limitations of the current setup.

3.1 Challenges in Evaluating RAG Systems

Evaluating generative AI systems like RAG pipelines is not straightforward. It involves challenges that differ significantly from traditional software testing. Many of these issues are specific to RAG systems and LLMs in general [30] These challenges shaped the way we designed our evaluation strategy:

- **Subjectivity aspect of Natural Language:** Judging the quality of generated text often depends on subjective opinions. As everyone has a different view on the topic and information given, terms like "relevance" or "coherence" can mean different things to different people. This makes it hard to measure output purely with objective metrics [31]. Additionally, the context in which the text is being evaluated, such as the background knowledge of the reader can significantly influence perceptions of quality. This can further complicate the process of developing standardized evaluation methods.

- **Randomness of LLMs:** Large Language Models produce outputs that include a degree of randomness. This is due to their parameter called temperature. Temperature is a parameter that controls the randomness and creativity of the model's output. It acts like a "dial" that adjusts the probability distribution of the model's next-word predictions, influencing how deterministic or varied the generated text will be. While setting temperature to 0 aims for deterministic outputs by always selecting the most probable

next word, it doesn't entirely eliminate evaluation challenges. Firstly, even at temperature 0, minor variations in model versions or environments can sometimes lead to different outputs. Secondly, strictly deterministic outputs might become overly repetitive, lack naturalness, or fail to explore potentially better phrasings. [32] Even with the same input, the output may vary. Similarly, if an LLM is used to evaluate the output, the scores can differ across runs. This makes reproducibility and consistent benchmarking more difficult.

- **Complexity of the RAG Pipeline:** A RAG system includes two separate parts — retrieval and generation as mentioned previously in the Introduction. Evaluation must cover both. That means we assess how well the system retrieves relevant context, how effectively it uses that context.

- **Inadequacy of Traditional Metrics:** Traditional evaluation metrics like BLEU (Bilingual Evaluation Understudy) and ROUGE (Recall-Oriented Understudy for Gisting Evaluation) are commonly used in NLP tasks such as machine translation and summarization. BLUE is an algorithm for evaluating the quality of text which has been machine-translated from one natural language to another. Quality is considered to be the correspondence between a machine's output and that of a human: "the closer a machine translation is to a professional human translation, the better it is" – this is the central idea behind BLUE. [33] Similarly, ROGUE is a set of metrics and a software package used for evaluating automatic summarization and machine translation software in NLP. The metrics compare an automatically produced summary or translation against a reference or a set of references summary or translation. ROUGE metrics range between 0 and 1, with higher scores indicating higher similarity between the automatically produced summary and the reference. [34] These metrics assess the quality of generated text based on how much overlap there is between the model's output and reference texts.

  However, these metrics have limitations when applied to RAG systems. They primarily focus on word overlap and fail to capture more complex qualities such as semantic meaning, fluency, coherence, factual accuracy, and how well the text aligns with the context. This is especially problematic for tasks like question-answering, where there can be many correct phrasings for the same answer. [35]

Because of these challenges, we simply cannot rely on simple word matching alone. We need better methods that evaluate the nuanced aspects of quality. This is especially important when the goal is to build a knowledge system for financial data.

3.2 Evaluation Methodology: LLM-as-a-Judge and G-Eval

To overcome the limitations of traditional metrics previously discussed, this thesis adopts the "LLM-as-a-judge" paradigm for evaluation. This approach

leverages the sophisticated language understanding and reasoning capabilities of LLMs to assess the outputs of other AI systems including the RAG pipeline developed in this thesis [36]. LLM judges can evaluate semantic meaning, contextual appropriateness, and alignment with more human-like preferences and also with more efficiency than surface-level metric calculators. This provides us with better and nuanced perspective on the correctness and utility of the generated answers, particularly crucial for complex domains like financial reporting. [36] This paradigm holds significant promise due to its potential for strong correlation with human judgments and scalability. [30]

3.2.1 G-Eval: Structured and Reasoning-Based LLM Judging

Among various LLM-as-a-judge approaches, this work specifically employs the G-Eval methodology. G-Eval is a framework designed for Natural Language Generation [1] evaluation, noted for its strong correlation with human judgment and its simulation of human evaluation patterns through structured, multi-criteria assessment. [37]

G-Eval operates by prompting a LLM with the necessary information: the input query, the generated answer from the RAG system, and the retrieved context. A defining characteristic of G-Eval is its use of Chain-of-Thought (CoT) reasoning. CoT is a technique that aims to improve language models' performance on tasks requiring logic, calculation and decision-making by structuring the input prompt in a way that mimics human reasoning. [38] Before assigning any scores, the evaluator LLM is instructed to generate intermediate reasoning steps, outlining why it will arrive at a specific evaluation for each criterion. This step-by-step breakdown significantly enhances the interpretability and reliability of the final assessment, making the evaluation process more transparent. [37]

Following the CoT reasoning, G-Eval utilizes a form-filling paradigm. Based on its articulated reasoning, the LLM judge completes a structured evaluation form, providing scores, typically on a 1-5 scale, representing quality levels, for predefined criteria relevant to the task. To get more detail, see Figure 3.1. While G-Eval supports general criteria like coherence, consistency, fluency, and relevance, its flexibility allows for customization. [39] For this thesis, the evaluation focuses specifically on the criteria defined in 3.3. Often, the LLM judge also provides brief natural language justifications alongside each score, further clarifying the basis for its assessment. [37]

This structured, reasoning-based approach allows G-Eval to capture nuances often missed by traditional metrics. Empirical studies have shown that G-Eval achieves a higher correlation with human judgments compared to metrics such as BLEU and ROUGE across various benchmarks [37]. See Figure 3.2 for benchmark table.

The implementation of G-Eval for this thesis leverages the open-source *Deepeval* Python library. [2] Deepeval facilitates the execution of G-Eval metrics. It

[1] Natural Language Generation (NLG) is a subfield of AI focused on producing natural language text from structured or non-structured data. G-Eval aims to evaluate the quality of such generated text. See Wikipedia: Natural Language Generation.

[2] The official documentation and getting started guide for the Deepeval library can be found

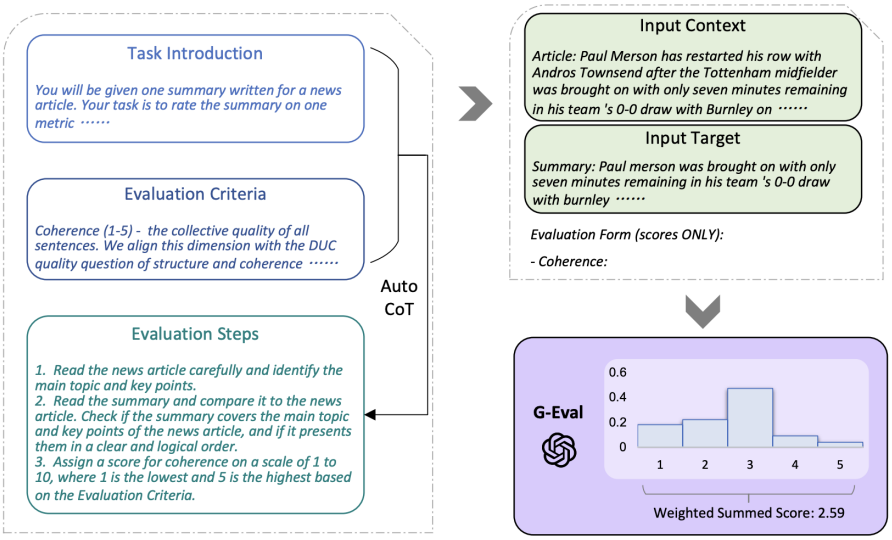

Figure 1: The overall framework of G-EVAL. We first input Task Introduction and Evaluation Criteria to the LLM, and ask it to generate a CoT of detailed Evaluation Steps. Then we use the prompt along with the generated CoT to evaluate the NLG outputs in a form-filling paradigm. Finally, we use the probability-weighted summation of the output scores as the final score.

Figure 3.1 The G-Eval process: The evaluator LLM receives the input, generated answer, and context, performs Chain-of-Thought reasoning, and then fills a structured evaluation form with scores and justifications. Sourced from [37]

Metrics	Coherence		Consistency		Fluency		Relevance		AVG	
	ρ	τ	ρ	τ	ρ	τ	ρ	τ	ρ	τ
ROUGE-1	0.167	0.126	0.160	0.130	0.115	0.094	0.326	0.252	0.192	0.150
ROUGE-2	0.184	0.139	0.187	0.155	0.159	0.128	0.290	0.219	0.205	0.161
ROUGE-L	0.128	0.099	0.115	0.092	0.105	0.084	0.311	0.237	0.165	0.128
BERTScore	0.284	0.211	0.110	0.090	0.193	0.158	0.312	0.243	0.225	0.175
MOVERSscore	0.159	0.118	0.157	0.127	0.129	0.105	0.318	0.244	0.191	0.148
BARTScore	0.448	0.342	0.382	0.315	0.356	0.292	0.356	0.273	0.385	0.305
UniEval	0.575	0.442	0.446	0.371	0.449	0.371	0.426	0.325	0.474	0.377
GPTScore	0.434	–	0.449	–	0.403	–	0.381	–	0.417	–
G-EVAL-3.5	0.440	0.335	0.386	0.318	0.424	0.347	0.385	0.293	0.401	0.320
- Probs	0.359	*0.313*	0.361	*0.344*	0.339	*0.323*	0.327	*0.288*	0.346	*0.317*
G-EVAL-4	**0.582**	**0.457**	**0.507**	**0.425**	**0.455**	**0.378**	**0.547**	**0.433**	**0.514**	**0.418**
- Probs	0.560	*0.472*	0.501	*0.459*	0.438	*0.408*	0.511	*0.444*	0.502	*0.446*
- CoT	0.564	0.454	0.493	0.413	0.403	0.334	0.538	0.427	0.500	0.407

Table 1: Summary-level Spearman (ρ) and Kendall-Tau (τ) correlations of different metrics on SummEval benchmark. G-EVAL without probabilities (*italicized*) should not be considered as a fair comparison to other metrics on τ, as it leads to many ties in the scores. This results in a higher Kendall-Tau correlation, but it does not fairly reflect the true evaluation ability. More details are in Section 4.

Figure 3.2 Comparison of evaluation metric correlations with human scores on benchmark datasets. G-Eval demonstrates superior alignment with human judgment over traditional metrics. Sourced from [37]

also provides supports for using locally hosted models like the quantized Llama 3.1 8B discussed previously in Introduction as judge, integrates smoothly into testing workflows, like the *pytest* framework used here, detailed in Section 3.5.1, and [3] provides tools for defining test cases and analyzing result. This makes it

at www.deepeval.com/docs/getting-started.

[3]For details on the Pytest framework, see the official documentation:

highly suitable for the iterative development that we are using for this thesis.

By incorporating G-Eval via Deepeval into our testing pipeline, we gain the ability to iteratively assess the RAG output quality using evaluations that align more closely with human expectations.

3.3 Selected Evaluation Metrics

Based on the goals of the thesis 1.2, the following key metrics, provided and implemented by the Deepeval library using its G-Eval capability [4], are used to evaluate the RAG pipeline's performance. Each metric is defined by the following components:

- **Criteria:** Describes the specific aspect of quality the metric assesses.
- **Motivation:** Explains why this metric is important for evaluating the RAG pipeline in this context.
- **Evaluation Parameters:** Lists the specific inputs, such as the user's question, the generated answer, retrieved context and more. Which are required by the LLM judge to compute the metric score.
- **Threshold:** Specifies the minimum acceptable score, on a 0-1 scale after normalization by Deepeval for the metric to be considered 'passed' in our tests.

The selected metrics are:

- **Answer Correctness**
  - **Criteria:** Determine if the actual output correctly answers the question based on the expected output.
  - **Motivation:** Measures the factual accuracy and faithfulness of the generated answer compared to the ground truth information potentially present in the retrieved context or expected output.
  - **Evaluation Parameters:** Input, Actual Output, Expected Output, Context
  - **Threshold:** 0.5
- **Answer Relevancy**
  - **Criteria:** Evaluate if the response is directly relevant to the question asked.
  - **Motivation:** Assesses whether the generated answer directly addresses the user's query without including unnecessary or tangential information. Ensures the system provides focused and useful responses.
  - **Evaluation Parameters:** Input, Actual Output.

docs.pytest.org/en/stable/.

[4]Detailed descriptions of the G-Eval metrics as implemented in Deepeval are available in their documentation: www.deepeval.com/docs/metrics-llm-evals.

  - **Threshold:** 0.5

- **Conciseness**
  - **Criteria:** Check if the response is within three sentences and provides a clear answer. This criteria changes depending on the prompt given to the RAG pipeline.
  - **Motivation:** Measures the brevity and clarity of the answer.
  - **Evaluation Parameters:** Actual Output.
  - **Threshold:** 0.5

- **Context Relevancy**
  - **Criteria:** Evaluate if the retrieved context contains information relevant to answering the query. The context should contain facts or information directly related to what the question is asking about.
  - **Motivation:** Measures the quality of the retrieval step. Assesses whether the documents or chunks passed to the generator LLM are pertinent to the user's query. Irrelevant context can lead to incorrect or unfocused answers. [23]
  - **Evaluation Parameters:** Input, Retrieval Context.
  - **Threshold:** 0.5

These quantitative metrics provide objective scores that allow for comparison across different iterations of the RAG pipeline. The defined thresholds set a minimum standard for acceptable performance for each criterion. A metric is only successful if the evaluation score is equal to or greater than threshold. These threshold has been chosen on a fact that it is a default standard of the Deepeval library. [39] We can tweak this thresholds to make the testing be more focused on some parts of the metrics.

3.4 Limitations of the Evaluation Setup

While the LLM-as-a-judge approach with G-Eval offers significant advantages, the specific implementation in this thesis has limitations. Mostly related to the LLM used for test evaluation:

- **Evaluator Model:** The evaluations are performed using a quantized version of the Llama 3.1:8B model, running locally via Ollama. This choice was made by the need to run evaluations efficiently on available personal computer. Providing us with rapid iterative testing without reliance on costly API calls or high-end AI cloud solutions 1.3. .

- **Potential Accuracy Trade-off:** Smaller and quantized models, while efficient, may possess less nuanced language understanding and reasoning capabilities compared to larger ones. Which was used in the original G-Eval paper. [37]

- **Impact on Metric Suitability:** This reduced capability means the quantized Llama 3.1 8B model might struggle to reliably evaluate highly complex or subtle aspects of text quality. While it is deemed sufficient for the core GEval metrics defined above. Its performance on other, potentially more demanding, metrics available in Deepeval is lacking. We experimented with metrics such as contextual precision and contextual recall. [5] Those requiring deeper semantic analysis or nuanced coherence checks. In that way we were unable to use these as metrics since the framework was asking for bigger and stronger LLM model.

These limitations are taken into account, and the evaluation results are interpreted within this context. The focus remains on observing relative improvements across iterations using a consistent evaluation setup.

3.5 Testing Framework Implementation using Pytest and Deepeval

To put the evaluation methodology into use described in Section 3.2.1 and support the iterative development cycle central to this thesis, a robust testing framework was implemented. This framework leverages the strengths of two key technologies: Pytest, a widely-used Python testing framework for test organization and execution, and Deepeval, a specialized library for evaluating LLM-based applications, particularly suited for RAG systems. [39] With this combination we are able to automate and get detailed assessments of the RAG pipeline's performance across different configurations.

3.5.1 Testing Architecture

The foundation of the testing framework relies on structured test data and a clear workflow managed by Pytest and Deepeval.

Test Case Definition. Test cases are defined externally in a structured JSON format. [6] This approach allows for easy modification and expansion of the test suite without changing the testing framework code. Each entry in the JSON file represents a distinct test case, containing the core elements needed for evaluation:

- `question`: The input query posed to the RAG system.
- `expected_output`: A manually curated answer serving as the benchmark for correctness.
- `context`: Reference information or key facts expected to be relevant for answering the question correctly.

An example snippet of the test data structure is shown below:

[5] Further details on these advanced metrics are available in the Deepeval documentation for Contextual Precision and Contextual Recall.

[6] The JSON (JavaScript Object Notation) standard is specified at www.json.org/json-en.html.

Listing 3.1 Example test case

```
{
  "market_trends_2024": {
    "question": "What were the key market trends in 2023?",
    "expected_output": "The stock market saw a strong rebound
    driven by technology stocks and easing inflation.",
    "context": ["Investors regained confidence as inflation cooled
       ..."]
  },
  // ... more test cases
}
```

Dataset Creation. During test execution deepeval.EvaluationDataset is created. By taking these static JSON definitions and trasnforming them into dynamic deepeval's *LLMTestCase* objects. An LLMTestCase in deepeval can be used to unit test LLM application outputs, which includes use cases such as RAG and LLM agents. It contains the necessary information as for example retrieval_context for RAG to evaluate your LLM application for a given input. [40] This is handled by a custom EvaluationDatasetFactory class.

Listing 3.2 EvaluationDatasetFactory

```
class EvaluationDatasetFactory:
    @staticmethod
    def create_from_dict_with_invocation(
        test_cases_config: Dict[str, Dict[str, Any]], qa_pipeline
    ) -> EvaluationDataset:
        test_cases = []
        for name, config in test_cases_config.items():
            question = config.get("question", "")
            result = qa_pipeline(question)

            if isinstance(result, dict):
                answer = result["answer"]
                retrieval_context = result["retrieval_context"]
            else:
                answer = result.answer
                retrieval_context = [doc.content for doc in result.
                    retrieved_documents]

            test_case = LLMTestCase(
                input=question,
                context=config.get("context", []),
                retrieval_context=retrieval_context,
                actual_output=answer,
                expected_output=config.get("expected_output", ""),
                name=name,
            )
            test_cases.append(test_case)
        return EvaluationDataset(test_cases=test_cases)
```

Our create_from_dict_with_invocation method iterates through the JSON test cases, invokes the specific RAG pipeline iteration being tested with the question, captures the actual_output which is the generated answer and the retrieval_context, the document chunks retrieved by the RAG system, and then packages all these elements - input, actual_output, expected_output, retrieval_context,

context - into LLMTestCase objects. These are collected into a deepeval's EvaluationDataset ready for evaluation. An evaluation dataset, or just dataset, is a collection of LLMTestCases. [41]

3.5.2 Evaluation Execution Workflow

The testing workflow proceeds as follows within the Pytest execution environment:

1. **Setup:** Test functions import necessary modules, load the JSON test data, and instantiate the specific G-Eval metrics defined in Section 3.3 using a helper functions.

2. **Pipeline Instantiation:** Pytest fixtures dynamically load the appropriate RAG pipeline module based on the iteration being tested.

3. **Test Case Processing:** The EvaluationDatasetFactory processes each JSON test case, running the instantiated RAG pipeline to generate the actual_output and retrieval_context.

4. **Evaluation Assertion:** The core evaluation is performed using deepevals evaluate(). This function takes the generated EvaluationDataset and the list of G-Eval metrics. Deepeval handles the interaction with the configured LLM judge which is Llama 3.1 8B via Ollama in our case. And computes scores for each metric against their defined thresholds. [41]

5. **Results Storage and Reporting:** Upon completion, detailed results are automatically saved. The save_test_results function stores raw metric scores, inputs, outputs, and pass/fail status into timestamped JSON files. Subsequently, the save_report function processes this JSON data to generate interactive HTML reports. This provides us with a visual representation of the results. While still having a copy in a data format that is well suited to be processed by machines. You can find an example of such an HTML report at this Github link.

3.5.3 Cross-Iteration Testing Capability

A critical feature of this framework is its ability to systematically test multiple RAG pipeline iterations, enabling direct comparison. This is achieved through:

- **Directory Structure:** Each pipeline iteration resides in a dedicated directory such as src/iterations/iter_X/, each containing the specific implementation for that variant. For example different chunking strategy, embedding model and so on.

- **Parametrized Fixtures:** Pytest fixtures are parametrized using helper functions like get_iteration_modules() to yield the name of each iteration directory.

- **Dynamic Pipeline Loading:** A fixture like qa_pipeline_(iteration_name) uses importlib.import_module to dynamically load and instantiate the create_pipeline function from the correct iteration's module path like this one src.iterations.iter_1.pipeline.

This ensures the exact same set of JSON test cases and evaluation metrics are applied consistently across every pipeline variant, allowing for objective comparison of their performance based on the generated results.

3.5.4 Running the Tests and Visualizing Results

The entire evaluation suite can be executed via a simple command, potentially wrapped in a task runner. This command triggers pytest to discover and run all tests, iterating through each pipeline configuration identified by the fixtures.

The framework generates two key outputs per iteration run:

- **JSON Results Files:** Contain comprehensive, machine-readable data on each test case, including inputs, outputs, context, metric scores, thresholds, pass/fail status, and overall success rates. These are stored in the test_results/ directory.
- **HTML Reports:** Provide an interactive, human-readable summary of the results, featuring overall statistics, progress bars, detailed views for each test case, comparing expected vs. actual outputs, and clear pass/fail indicators for each metric. These reports facilitate the analysis and comparison of performance across different iterations.

This testing framework, therefore, provides a structured, automated mechanism for evaluating the RAG pipeline. It ensures that performance assessments are objective, reproducible. They are mainly comparable across all the iterations.

3.6 Integrating Evaluation into the Iterative Process

The testing framework described above is used systematically throughout the development of the RAG pipeline. Whenever any significant change to a component – such as modifying the document chunking strategy, swapping the embedding model or adjusting retrieval parameters – the full test suite is executed.

The results, comprising the scores for Answer Correctness, Answer Relevancy, Context Relevancy, and Conciseness, are recorded. By comparing the scores against the thresholds or the results from the previous iteration, the impact of the change can be assessed. This approach allows for informed decisions about whether a modification leads to improvement in performance according to the prioritized metrics. This helps us guide subsequent development efforts towards optimizing the pipeline.

3.7 Test Case Dataset Creation

The foundation of any evaluation framework is a dataset representative of the target domain and use case. This section details the creation of such dataset used throughout this thesis for evaluating the various RAG pipeline iterations. The primary goal was to create a realistic set of questions pertaining to corporate financial knowledge, grounded in actual financial documents.

3.7.1 Source Material and Initial Processing

The process began with the selection of source documents. We gathered a collection of annual reports from financial entities, such as iShares. These documents were chosen because they are publicly available, represent the complex nature of financial reporting, and contain a mixture of unstructured text, tables, and potentially graphs. The dataset used for experimentation in this thesis consists of approximately 300 pages of PDF documents. This collection is intended to simulate a realistic set of internal documents that might be found within an enterprise environment, such as a team's confluence pages, project reports, and other informational materials. This dataset is publicly accessible for review in this Github folder. The textual content extracted from these reports was then segmented into smaller, manageable chunks. This chunking step is crucial as it defines the atomic units of information that the retrieval component of the RAG pipeline will operate on. In this particular case we have used the RecursiveCharacterTextSplitter class used in the Proof of Concept 2.3.

3.7.2 Synthetic Data Generation for Jumpstarting

In scenarios where pre-existing user interaction data is unavailable. Evaluation of retrieval effectiveness presents a challenge. Without knowing the "correct" chunk for a given query, measuring whether the system retrieved it is impossible. To overcome this "cold start" problem, we adopted a synthetic data generation approach, as advocated by Liu [42].

Utilizing a LLM, we processed each text chunk generated in the previous step. The LLM was prompted with instructions akin to: "Generate relevant questions that can be answered using only the information present in the following text chunk: [text chunk]". This procedure yielded an initial set of question $\rightarrow$ chunk pairs. These pairs provide a provisional ground truth and essentialy creating our base dataset.

3.7.3 Domain Expert Curation and Final Dataset

While LLMs can generate syntactically valid questions related to a text chunk, these questions may lack real-world relevance. Also they could contain subtle inaccuracies, or fail to capture some thoughts expected by users in the financial domain. Therefore, the synthetically generated question-chunk pairs underwent a critical manual review and curation phase performed by a domain expert.

The expert examined each question for:

- **Relevance:** Does the question address a topic of genuine interest within the financial context of the source document?
- **Clarity:** Is the question phrased well?
- **Answerability:** Can the question be answered using only the paired text chunk?
- **Accuracy:** Does the question accurately reflect the information without any misinterpretations?

Questions that failed this scrutiny were either corrected, or discarded. The expert also formulated the 'expected_output' for each validated question. Final set of the tests cases that we have iterated on can be reviewed in this Github file.

The result of this iterative refinement process is the final test case dataset. This collection of questions, expected answers, and associated context chunks forms the benchmark against which all RAG pipeline iterations are evaluated in this thesis. This dataset is then structured into the JSON format detailed in Section 3.5.1, enabling its direct use within the Pytest and Deepeval testing framework.

3.8 Summary

Evaluating the RAG pipeline is a critical and essential task for validating its effectiveness in making corporate financial knowledge accessible. This chapter has outlined the challenges inherent in this evaluation. By justifying the adoption of the LLM-as-a-judge methodology, specifically G-Eval, implemented via the deepeval library. Key metrics – Answer Correctness, Answer Relevancy, Context Relevancy, and Conciseness – were selected to align with the thesis goals and the demands of the financial domain. The implementation of an automated testing framework using pytest and deepeval enables systematic and repeatable evaluation, driving the iterative improvement process. Which we are using throughout the thesis to see how are the iterations performing. Despite limitations related to the use of a smaller evaluator LLM, this framework provides valuable insights into the pipeline's performance. This ensures that development is guided by measurable progress.

4 Data Preparation and Embeddings

This chapter documents the iteration process of developing and optimizing the RAG pipeline focused on the previously defined building blocks 2.2. The primary objective is to iteratively enhance the pipeline's performance based on improving the pipelines process of data cleaning / processing. Later in the chapter, we are experimenting with the embedding generation.

We leverage the evaluation framework, metrics and the curated dataset established in Chapter 3. This is allowing for an objective assessment of changes made to the pipeline architecture.

The starting point for this iterative process is the PoC pipeline detailed and established in Chapter 2, Section 2.1. This PoC, initially demonstrated using generic web data, serves as our baseline also addressed as Iteration 0 when applied to the target financial domain. Subsequent sections in this chapter will describe the motivation, implementation, results, and analysis for each modification introduced to improve upon this baseline. This will help us track the evolution of the pipeline's effectiveness.

4.1 Iteration 0 Performance: Evaluating the PoC on Financial Data

Before introducing modifications, it is essential to establish the baseline performance of the initial Proof of Concept pipeline when applied to the intended use case. This involves running the PoC architecture described in Section 2.1, adapted for PDF processing, against the financial test dataset created in Section 3.7 and evaluating it using our framework.

4.1.1 Baseline Configuration Recap

The baseline configuration of Iteration 0 adapts the PoC implementation from Chapter 2 for PDF documents and uses the following specific parameters:

- **Data Loading**: Handled by a PdfFolderSource class, utilizing PdfLoader to scan a directory, identify PDF files, track metadata, and extract text content page by page. This replaces the WebBaseLoader used in the generic PoC example. This operation is possible by using the Document Loaders provided by Langchain which we already mentioned in Section 2.2
- **Chunking Strategy**: Utilized RecursiveCharacterTextSplitter with parameters chunk_size=500 and chunk_overlap=50.
- **Embedding Model**: We utilize LangChain's high-performing open embedding model, `nomic-embed-text`, which supports a large token context window [43].

- **Vector Store**: Used Chroma as the vector store, configured for persistence to disk.
- **Retrieval**: Retrieved the top-k = 4 chunks as it is the default setting. Used the default similarity search mechanism of the vector store retriever. [15]
- **Generation Model**: Utilized llama3.1:8b model, served locally via Ollama.
- **Prompt Template**: Used the basic prompt template defined in Chapter 2, Listing 2.5. Instructing the model to use the context and answer concisely.
- **Pipeline Orchestration**: Conceptually similar to that defined in Chapter 2, Listing 2.1.
- **Text Cleaning**: No explicit text cleaning steps were applied beyond the basic text extraction from PDFs.

4.1.2 PDF Handling and Caching Mechanisms

Handling PDF documents efficiently and avoiding redundant processing is crucial for many systems. The baseline implementation incorporates several mechanisms:

- **PDF Processing**: The PdfFolderSource class discovers PDFs, extracts text using pypdf, handles potential errors gracefully, and assigns unique IDs based on file metadata. Where metadata is data that provides information about other data, but not the content of the data itself, such as the text of a message or the image itself. [44]
- **Source Caching**: Metadata about processed PDFs is cached, into a JSON file. Before processing, the system checks the current file metadata against the cache. Cache is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere [45]. If a file hasn't changed based on a hash of its metadata, the expensive text extraction step is skipped.
- **Document Processing Caching**: We do the same process as above also for the chunks that are created by the RecursiveCharacterTextSplitter. These processed chunks, along with their metadata, are cached. This prevents against expensive operations. In this particular case, we prevent multiple re-chunking if the document content and splitting parameters remain unchanged.
- **Vector Store Persistence**: The generated vector embeddings are persisted to disk by the Chroma vector store [15]. This allows the system to load existing embeddings upon restart. In that way we avoid the need to re-embed the entire dataset, unless needed.

These caching layers significantly speed up repeated runs and development iterations by minimizing redundant computations.

4.1.3 Baseline Analysis

This Iteration 0 configuration was evaluated against our dataset 3.7 using the framework we created in Section 3.5.1 with Llama 3.1:8B as the judge. The performance scores for the key metrics were:

Metric	Average	Std. Dev.
Answer Correctness	0.49	0.42
Answer Relevancy	0.68	0.38
Context Relevancy	0.76	0.30
Conciseness	0.80	0.32

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 4.1 Iteration 0 Evaluation Results

Evaluating the initial PoC pipeline on the financial dataset yielded the scores listed above. The score for Context Relevancy (0.68) indicates that the basic chunking and embedding strategy struggles significantly with the financial reports. Failing to retrieve the most relevant segments. This directly impacts Answer Correctness (0.48) and Answer Relevancy (0.76), as the generator receives suboptimal context. While Conciseness (0.80) might be acceptable due to the prompt's constraint, the core retrieval performance is not good enough. This is reflected in the scores, with metrics like Answer Correctness failing to meet the 0.5 threshold defined in Section 3.3.

Furthermore, the baseline lacks any text cleaning. Extracted PDF text is often susceptible to noise such as potential OCR errors, inconsistent spacing, unwanted headers/footers, and stray special characters resulting from the conversion process. Optical character recognition (OCR) is the electronic conversion of images of typed, handwritten or printed text into machine-encoded text, whether from a scanned document, a photo of a document, a scene photo, for example the text on signs and billboards in a landscape photo or from subtitle text superimposed on an image. [46] Such potential artifacts can interfere with the embedding process and the LLM's interpretation. As suggested by related research [29]. Even though modern models might benefit from nuances in text, some level of normalization and cleaning is often necessary. Especially for noisy sources like extracted PDFs. As we do not do any cleaning of the text, this potential noise likely contributes to the suboptimal retrieval performance observed in the baseline. Therefore, the subsequent iterations will focus on implementing and evaluating different text cleaning strategies to determine their impact on improving context relevance and answer quality.

4.2 Iteration 1: Radical Text Cleaning

Based on the baseline analysis suggesting that raw PDF text quality might be a noisy source. The first iteration introduced an aggressive text cleaning step. The hypothesis was that removing as much perceived "noise" as possible—including special characters, common words such as stop words, and variations in word forms—would allow the embedding model and LLM to focus on the core semantic

content. This approach involved leveraging the NLTK library [1] for advanced text processing tasks.

The following clean_text function was integrated into the document processing pipeline immediately after text extraction and before chunking:

Listing 4.1 Aggressive cleaning

```
def clean_text(self, text: str) -> str:
    """
    Clean and normalize text.
    - Convert to lowercase
    - Remove special characters (keep alphanumeric and
      whitespace)
    - Consolidate whitespace
    - Tokenize
    - Remove stop words
    - Lemmatize

    Args:
        text: Raw text

    Returns:
        str: Cleaned text
    """
    cleaned = text.lower()
    cleaned = re.sub(r"[^a-z0-9\s€%£$±=()/-]", "", text)
    cleaned = re.sub(r"\s+", " ", cleaned).strip()

    if not self.lemmatizer or not self.stop_words:
        logger.warning("Skipping NLTK processing due to
            initialization error.")
        return cleaned

    try:
        logger.info("Starting NLTK processing...")

        tokens = word_tokenize(cleaned)

        processed_tokens = [
            self.lemmatizer.lemmatize(word) for word in tokens
                if word not in self.stop_words and len(word) > 1
        ]

        cleaned = " ".join(processed_tokens)
        logger.info("NLTK processing completed successfully.")
    except Exception as e:
        logger.error(f"Error during NLTK processing: {e}.
            Returning partially cleaned text.")
        return re.sub(r"\s+", " ", text.lower()).strip()

    return cleaned
```

All other parameters, embedding model, LLM, chunking, etc., remained the same as in Iteration 0.

[1] The official website for the Natural Language Toolkit (NLTK) library, providing downloads and documentation, is available at www.nltk.org.

4.2.1 Iteration 1 Analysis

This Iteration 1 configuration was evaluated against the same dataset and the performance scores were:

Metric	Baseline Average	Baseline Std. Dev.	Iter 1 Average	Iter 1 Std. Dev.
Answer Correctness	0.49	0.42	0.60	0.36
Answer Relevancy	0.68	0.38	0.77	0.37
Context Relevancy	0.76	0.30	0.66	0.38
Conciseness	0.80	0.32	0.69	0.36

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 4.2 Iteration 1 Evaluation Results

Contrary to the initial hypothesis that aggressive cleaning would improve performance, the evaluation results for Iteration 1 in Table 4.2 showed a degradation in performance compared to some of the baseline metrics. The aggressive cleaning, particularly the removal of stop words and extensive special characters, proved to worsen the overall performance involving the context part.

Research suggests that models like nomic-embed-text and generative models like Llama 3.1 often utilize stop words for understanding context and generating coherent text [8]. Removing them, along with lemmatization altering word forms, likely stripped away crucial semantic information. Which is needed for effective retrieval and generation.

Furthermore, the regex pattern we used, [2] which was `[^a-z0-9\s%€£$±=()/-]`, eliminated vital punctuation like commas in numbers or periods and potentially other symbols important in financial contexts. This likely further degraded the quality of the text chunks. This iteration demonstrated that overly aggressive, generic text cleaning significantly harms performance.

4.3 Iteration 2: Minimal Text Cleaning

Given the negative impact of the radical cleaning in Iteration 1, the second iteration adopted the opposite extreme: minimal cleaning. The hypothesis was that perhaps only the most basic normalization—lowercasing and consolidating whitespace—was necessary. Aligning partially with research findings that lowercasing is generally beneficial [42] while avoiding the significant information loss observed in Iteration 1. This approach aimed to establish if a very light touch is sufficient. Or if further more moderate cleaning is required.

The clean_text function from Iteration 1 was replaced with the following clean_text_minimal function:

Listing 4.2 Minimal cleaning

```
def clean_text_minimal(self, text: str) -> str:
    """Minimal cleaning: lowercase, consolidate whitespace,
       basic unicode normalization.
    Args:
        text: Raw text
```

[2] A regular expression or also known as regex is a sequence of characters that specifies a match pattern in text, commonly used for searching and replacing within strings. [47]

```
    Returns:
        str: Cleaned text
    """
    cleaned = text.lower()
    cleaned = re.sub(r"\s+", " ", cleaned).strip()

    return cleaned
```

All other parameters remained identical to Iteration 0.

4.3.1 Iteration 2 Analysis

This Iteration 2 configuration was evaluated against the same dataset as above and the performance scores were:

Metric	Baseline Average	Baseline Std. Dev.	Iter 2 Average	Iter 2 Std. Dev.
Answer Correctness	0.49	0.42	0.36	0.36
Answer Relevancy	0.68	0.38	0.82	0.32
Context Relevancy	0.76	0.30	0.82	0.32
Conciseness	0.80	0.32	0.86	0.19

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 4.3 Iteration 2 Evaluation Results

On average, the minimal cleaning in Iteration 2, see Table 4.3, surpassed the aggressive approach of Iteration 1 in several metrics, notably boosting Context Relevancy (average score 0.82 vs 0.66). When compared to the baseline, see Table 4.3, this iteration presented a slight change: Context Relevancy (0.82 vs 0.76) and Answer Relevancy (0.82 vs 0.68) improved, indicating better retrieval of generally relevant passages. However, Answer Correctness significantly declined (0.36 vs 0.49).

This suggests that while retaining most text elements, unlike Iteration 1, aids relevance, the minimal cleaning was insufficient to handle underlying noise often present in PDFs. This noise likely effected the model's ability to extract precise facts. Which could leading to less correct answers despite seemingly better context. Therefore, Iteration 2 confirms that minimal cleaning alone is inadequate. Maybe more balanced, moderate cleaning strategy is needed to improve correctness without sacrificing the gains in relevancy. This will effectively guide our next iteration.

4.4 Iteration 3: Moderate Financial Text Cleaning

Iterations 1 and 2 demonstrated the pitfalls of cleaning too much or too little. Iteration 3 aimed for a "moderate" approach specifically designed for financial documents. The goal was to remove common sources of noise, like inconsistent unicode characters, irrelevant symbols, excessive whitespace, while carefully preserving characters and structure. That are essential for financial context such as key symbols like %, $, punctuation for numbers, sentence structure. A key

step introduced in this iteration was the use of `unicodedata.normalize("NFKC", text)`[3] to standardize visually identical but semantically distinct Unicode characters. This ensures, for example, that full-width digits, ligatures, and circled symbols are converted into their ASCII equivalents. Enabling more consistent embeddings.

The clean_text_minimal function from Iteration 2 was replaced with the following clean_text_moderate function:

Listing 4.3 Moderate cleaning

```
def clean_text_moderate(self, text: str) -> str:
    """Moderate cleaning for financial documents: normalize
       Unicode, preserve key symbols,
    remove unnecessary special characters, and consolidate
       whitespace.

    Args:
        text (str): Raw financial text.
    Returns:
        str: Cleaned financial text.
    """

    # Normalize Unicode characters
    text = unicodedata.normalize("NFKC", text)

    # Standardize common typographical characters
    text = text.replace("’", "'").replace("“", '"').replace("”",
       '"')
    text = text.replace("\u2013", "-").replace("\u00a0", " ")

    cleaned = text.lower()
    cleaned = re.sub(r"[^a-z0-9\s€.,?!:%&$±=()/-]", "", text)

    # Normalize spaces and blank lines
    cleaned = re.sub(r"\s+", " ", cleaned).strip()
    cleaned = re.sub(r"\n\s*\n(\s*\n)+", "\n\n", cleaned)

    return cleaned
```

All other pipeline parameters remained the same as Iteration 0.

4.4.1 Iteration 3 Analysis

This Iteration 3 configuration was evaluated against the financial test dataset. The performance scores were:

Iteration 3 implemented a more nuanced cleaning strategy tailored for financial text, incorporating Unicode normalization and preservation of key symbols. The results, see Table 4.4, show an improvement in Answer Correctness (0.50) compared to both Iteration 2 (0.36) and the baseline (0.49). However, contrary to expectations, this moderate cleaning led to a decrease in Context Relevancy (0.67) compared to both the baseline (0.76) and especially Iteration 2 (0.82). This unexpected outcome suggests that while some aspects of the cleaning were beneficial for improving correctness. This specific implementation might have

[3] See the official Python documentation for details on the `unicodedata.normalize` function: docs.python.org/3/library/unicodedata.html.

Metric	Baseline Average	Baseline Std. Dev.	Iter 3 Average	Iter 3 Std. Dev.
Answer Correctness	0.49	0.42	0.50	0.40
Answer Relevancy	0.68	0.38	0.76	0.43
Context Relevancy	0.76	0.30	0.67	0.44
Conciseness	0.80	0.32	0.78	0.36

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 4.4 Iteration 3 Evaluation Results

inadvertently removed or altered subtle textual features or semantic signals that are important for the retriever. Which could negatively impact context relevance. This iteration concludes that this specific moderate approach did not yield the desired overall improvement.

4.5 Iteration 4: Moderate Cleaning with Regex Pattern Removal

While Iteration 3 aimed for a balanced cleaning approach, it still didn't explicitly target common structural noise found in PDFs like repeating headers, footers, or page numbers. This might contribute to the suboptimal Context Relevancy observed. Iteration 4 builds upon the moderate cleaning function with clean_text_moderate from Listing 4.3 by adding a subsequent step that applies specific regex patterns designed to remove these types of structured noise elements.

The hypothesis is that combining moderate normalization with targeted removal of structural artifacts will further enhance the quality of the text chunks. All other parameters remained identical to Iteration 3.

4.5.1 Iteration 4 Analysis

This Iteration 4 configuration, incorporating moderate cleaning plus regex-based pattern removal, was evaluated against the same dataset as before. The performance scores were:

Metric	Baseline Average	Baseline Std. Dev.	Iter 4 Average	Iter 4 Std. Dev.
Answer Correctness	0.49	0.42	0.43	0.40
Answer Relevancy	0.68	0.38	0.72	0.43
Context Relevancy	0.76	0.30	0.83	0.44
Conciseness	0.80	0.32	0.66	0.36

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 4.5 Iteration 4 Evaluation Results

Iteration 4 attempted to refine the moderate cleaning approach which was introduced in Iteration 3 4.4 by adding regex patterns to remove specific structural noise like headers or page numbers. Evaluating the results, see Table 4.5, this strategy did not yield the expected improvements. Compared to Iteration 3, performance slightly decreased across Answer Correctness (0.43 vs 0.50), Answer Relevancy (0.72 vs 0.76).

Furthermore, when compared to the simpler minimal cleaning of Iteration 2, Iteration 4 performed significantly worse on both Answer Relevancy (0.72 vs 0.82) but it did achieve better Answer Correctness (0.43 vs 0.36) together with Context Relevancy (0.82 vs 0.83). It also underperformed the original baseline in terms of Context Relevancy (0.66 vs 0.76) and Answer Correctness (0.43 vs 0.49).

The conclusion for this iteration is that adding these specific regex patterns on top of the moderate cleaning function was detrimental to overall performance, particularly hurting relevancy metrics. It failed to improve upon Iteration 3 and represents a step back compared to the high relevancy achieved with minimal cleaning in Iteration 2. This suggests the added regex rules might have been overly aggressive or interacted negatively with the preceding moderate cleaning steps.

4.6 Conclusions on Data Cleaning Iterations

This initial phase of iterative development explored the impact of different text cleaning strategies on RAG pipeline performance. We evaluated five approaches: no cleaning in Iteration 0 4.1, aggressive NLTK-based cleaning in Iteration 1 4.2, minimal cleaning in Iteration 2 4.3, moderate cleaning with Unicode normalization in Iteration 3 4.4, and moderate cleaning plus regex pattern removal in Iteration 4 4.5.

The results highlight the complex and sometimes non-intuitive effects of text cleaning in our case. Aggressive cleaning from Iteration 1 4.2 significantly harmed context relevance (0.66) but surprisingly yielded the highest Answer Correctness (0.60) among all iterations. Conversely, minimal cleaning from Iteration 2 4.3 achieved the best performance by far on relevancy metrics, maximizing both Context Relevancy (0.82) and Answer Relevancy (0.82), yet it suffered from very low Answer Correctness (0.36).

The more targeted moderate cleaning in Iteration 3 4.4 and the subsequent addition of regex patterns in Iteration 4 4.5 failed to find a better balance. Iteration 3 4.4 improved correctness over the baseline but unexpectedly reduced context relevancy compared to minimal cleaning. Iteration 4 4.5 further degraded performance, showing that adding more cleaning steps is not necessarily better.

No single cleaning strategy emerged as universally optimal across all metrics. A clear trade-off was observed: the strategy maximizing correctness hurt relevancy, while the strategy maximizing relevancy hurt correctness. Since the minimal cleaning in Iteration 2 4.3 already performed well for relevancy, adding more complex cleaning might not help much and could even cause problems. The optimal approach may depend on prioritizing specific metrics such as relevancy vs. correctness. We plan to revisit combining insights from these experiments later, as discussed in Section 5.2

Having explored various text cleaning techniques, and identifying minimal cleaning from Iteration 2 4.3 as the best for relevancy. We proceeded with baseline approach as the default preprocessing step. And later on combine the best ones to see if it leads to and overall improvement. Now we will shift our focus to optimizing other components of the RAG pipeline: embedding models and chunking strategies.

4.7 Optimizing Chunking Strategies

Following the exploration of text cleaning methods, where minimal cleaning in Iteration 2 4.3 was identified as the most effective for relevancy metrics, we now turn our attention to the chunking strategy. The baseline 4.1 used a chunk size of 500 with an overlap of 50. Based on research suggesting chunk size impacts the balance between context completeness and noise [8], we test two variations using the baseline cleaning approach preprocessing step.

4.8 Iteration 5: Larger Chunk Size

The first experiment explores whether larger chunks can improve performance by capturing more surrounding context within each chunk, potentially helping the LLM understand the broader narrative. We doubled the baseline chunk size and overlap, setting chunk_size=1000 and chunk_overlap=100. All other parameters remained the same as baseline 4.1.1.

4.8.1 Iteration 5 Analysis

This Iteration 5 configuration, incorporating larger chunks plus larger overlap, was evaluated against the same dataset as before. The performance scores were:

Metric	Baseline Average	Baseline Std. Dev.	Iter 5 Average	Iter 5 Std. Dev.
Answer Correctness	0.49	0.42	0.43	0.38
Answer Relevancy	0.68	0.38	0.64	0.49
Context Relevancy	0.76	0.30	0.91	0.15
Conciseness	0.80	0.32	0.63	0.31

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 4.6 Iteration 5 Evaluation Results

Comparing Iteration 5, see Table 4.6, to baseline results, 500/50 chunks, the larger chunk size significantly boosted Context Relevancy (0.91 vs 0.76), suggesting the retriever benefited from the broader context. Answer Correctness did not see an improvement (0.43 vs 0.49). However, this came at a substantial cost to Conciseness (0.63 vs 0.80). Answer Relevancy (0.64 vs 0.68) also saw a lost in performance. This aligns with the trade-offs mentioned in [8]: larger chunks captured more context, aiding retrieval relevance, but likely introduced more noise or diluted the key information within the chunk, hindering the LLM's ability to generate relevant and concise answers.

4.9 Iteration 6: Smaller Chunk Size

Given the mixed results with larger chunks, we explored the opposite: smaller chunks. The hypothesis is that smaller chunks might provide more focused, less noisy context, potentially improving correctness and answer relevancy, even if individual chunks contain less narrative context [8]. We approximately halved the baseline chunk size. Having now chunk size of 256 with overlap 32. Other parameters staid the same as the baseline solution 4.1.1.

4.9.1 Iteration 6 Analysis

The performance scores for iteration 6 were:

Metric	Baseline Average	Baseline Std. Dev.	Iter 6 Average	Iter 6 Std. Dev.
Answer Correctness	0.49	0.42	0.52	0.33
Answer Relevancy	0.68	0.38	0.72	0.34
Context Relevancy	0.76	0.30	0.92	0.09
Conciseness	0.80	0.32	0.74	0.19

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 4.7 Iteration 6 Evaluation Results

Iteration 6, see Table 4.7, showed interesting results compared to both previous chunking strategies. It achieved the highest Context Relevancy score yet (0.92), slightly surpassing even the large chunks of Iteration 5 (0.91) and significantly improving over the default chunks in Iteration 0 (0.80). Answer Correctness (0.52) was also the best among these three iterations, substantially higher than Iteration 2 (0.36) and Iteration 5 (0.43). However, Answer Relevancy (0.72) and Conciseness (0.74) were lower than in Iteration 0 (0.76 and 0.80 respectively), though better than Iteration 5 4.8. This suggests that smaller chunks were highly effective for retrieval precision and factual correctness in this setup, possibly due to reduced noise per chunk. Although they might lack sufficient surrounding context for the LLM to achieve peak answer relevancy and conciseness compared to the default 500-size chunks used in Iteration 2. Overall, the small chunk strategy (256/32) appears to offer the best balance found so far, particularly excelling in retrieval.

4.10 Exploring Embedding Models

Having experimented with cleaning and chunking, we now investigate the impact of the embedding model itself. While the ‘nomic-embed-text‘ model used thus far is considered high-performing, the choice of embedding model can significantly influence RAG performance, with no single model being universally best [8].

4.11 Iteration 7: Trying mxbai-embed-large

We decided to test an alternative open-source embedding model, mxbai-embed-large, available via Ollama. [4] his model is slightly larger than nomic-embed-text with approx. 334M parameters vs. 137M for nomic-embed-text v1.5.

External benchmarks, such as the one presented by Timescale [48], showed ‘mxbai-embed-large‘ performing marginally better, around 2 percentage points, than ‘nomic-embed-text‘ on the retrieval benchmark (59.25% vs 57.25%). While acknowledging that benchmark differences might not always translate to specific applications and that exhaustive model testing is beyond the scope of this thesis 1.3, we wanted to perform a limited comparison. We used the baseline configuration 4.1.1 to evaluate the results. Only changing the embedding model.

[4]The mxbai-embed-large embedding model is available via Ollama. Further details and usage instructions can be found on its library page: ollama.com/library/mxbai-embed-large.

4.11.1 Iteration 7 Analysis

The pipeline using the alternative embedding model was evaluated:

Metric	Baseline Average	Baseline Std. Dev.	Iter 7 Average	Iter 7 Std. Dev.
Answer Correctness	0.49	0.42	0.40	0.32
Answer Relevancy	0.68	0.38	0.90	0.14
Context Relevancy	0.76	0.30	0.86	0.38
Conciseness	0.80	0.32	0.76	0.25

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 4.8 Iteration 7 Evaluation Results

Comparing the results of Iteration 7, see Table 4.8, with those of Iteration 0, which used the exact same configuration except for the 'nomic-embed-text' model, reveals in some cases better score. Making the biggest difference in the Answer Relevancy section (0.90 vs 0.68). Although it is not the case across all four metrics. But if we take into consideration also the standard deviation, we can conclude that it is providing better results. Based on 'mxbai-embed-large' showing slightly higher performance in some external benchmarks [48] and overall measurable difference in our specific RAG pipeline evaluation. Switching to it produced positive results.

4.12 Summary on Chunking and Embeddings

Following the data cleaning experiments, this section focused on optimizing chunking strategies and exploring an alternative embedding model, using the baseline configuration.

The chunking experiments concluded in Iterations 5 4.8 and 6 4.9 demonstrated the expected trade-offs. Larger chunks (1000/100) improved context relevancy but degraded answer relevancy and conciseness compared to the default, 500/50 used in Iteration 0 4.1.1). Smaller chunks (256/32) achieved the highest context relevancy (0.92) and significantly improved correctness (0.52) compared to other chunk sizes tested, although with slightly lower conciseness than the default size. Based on these findings, the smaller chunk size (256/32) proved to be the most effective strategy identified for our dataset and task, particularly strong in retrieval.

The embedding model experiment in Iteration 7 4.11 showed that replacing nomic-embed-text with the slightly larger mxbai-embed-large model yielded positive performance difference for most of the metrics in our evaluation framework, further supporting minor advantages reported in some external benchmarks.

Now with these insights we transition into next chapter. The focus of the next chapter will shift to the subsequent stages of the RAG pipeline: the generative Large Language Model and the prompt templates used to guide its responses.

5 LLM Selection and Prompt Optimization

Following the exploration of data preparation and chunking strategies in Chapter 4, this chapter shifts focus towards the Large Language Model itself and the prompt used to instruct it.

First, we justify the selection of the specific LLM used throughout these experiments, Llama 3.1:8B. Given the constraints outlined in the thesis scope 1.3, an exhaustive comparison of multiple LLMs was not feasible. Instead, we rely on established third-party benchmarks to select a capable model suitable for our hardware.

Second, building upon the baseline configuration from Section 4.1, we conduct an iteration focused on prompt. We investigate whether a more detailed, domain-specific prompt can elicit better performance from the LLM compared to the generic baseline prompt, using the evaluation framework established earlier.

Finally, we explore the synergy of combining previously identified 'optimal' settings from different iterations: minimal text cleaning from Iteration 2 4.3, smaller chunk size from Iteration 6 4.9, and the improved prompt template developed in this chapter. This 'combined' iteration aims to assess whether simply merging the best-performing individual components leads to the overall best pipeline configuration.

5.1 Language Model Selection: Llama 3.1:8B

The choice of the Large Language Model is critical for the RAG pipeline's generation quality. For this thesis, constrained by the need for local execution on available hardware in our case Macbook Air M1. The selection focused on models known for strong performance within a feasible size range, which is approximately 7-8 billion parameters.

We selected Meta AI's *Llama 3.1 8B* model. This decision was based on publicly available benchmarks and analyses rather than exhaustive internal testing. Llama 3.1: 8B is presented by Meta as a significant improvement over previous iterations [49].

Crucially, comparative benchmarks indicate that Llama 3.1: 8B performs favorably against other prominent open-source models in its size class.

As illustrated in Figure 5.1, Llama 3.1: 8B demonstrates superior performance compared to models like Mistral 7B and Google's Gemma 7B on various standard NLP benchmarks. [49].

Given these strong third-party results, Llama 3.1 8B was chosen as the generative model for our experiments.

5.1.1 Hardware Constraints and Model Scalability

While Llama 3.1:8B offers a good balance of performance and resource requirements for local execution on our test hardware, it's important to note the

Meta Llama 3 Instruct model performance

	Meta Llama 3 8B	Gemma 7B - It Measured	Mistral 7B Instruct Measured
MMLU 5-shot	**68.4**	53.3	58.4
GPQA 0-shot	**34.2**	21.4	26.3
HumanEval 0-shot	**62.2**	30.5	36.6
GSM-8K 8-shot, CoT	**79.6**	30.6	39.9
MATH 4-shot, CoT	**30.0**	12.2	11.0

	Meta Llama 3 70B	Gemini Pro 1.5 Published	Claude 3 Sonnet Published
MMLU 5-shot	**82.0**	81.9	79.0
GPQA 0-shot	39.5	**41.5** CoT	38.5 CoT
HumanEval 0-shot	**81.7**	71.9	73.0
GSM-8K 8-shot, CoT	**93.0**	91.7 11-shot	92.3 0-shot
MATH 4-shot, CoT	50.4	**58.5** Minerva prompt	40.5

Figure 5.1 Performance Comparison of 8B Parameter LLMs
Note: This figure should visually compare Llama 3.1 8B against models like Mistral 7B and Gemma 7B on relevant benchmarks. Sourced from [49]

modularity of the implemented pipeline. The code is designed such that the LLM can be relatively easily swapped.

Should more powerful hardware become available, researchers could readily adapt the pipeline to utilize larger, potentially more capable models, such as Llama 3.1:70B. However, exploring these larger models falls outside the hardware limitations and defined scope of this thesis. Our focus remains on optimizing the RAG process around the selected Llama 3.1:8B model within the given constraints.

5.2 Iteration 8: Enhanced Prompt Engineering

The prompt template acts as the direct instruction to the LLM, guiding its behavior during generation. The baseline prompt in Listing 5.1, adapted from Chapter 2.5 was generic, providing only basic instructions. Effective prompt engineering, however, involves crafting prompts that are specific to the task and domain, potentially leveraging techniques like role-playing and providing structured output requirements [50].

This iteration tests the hypothesis that a more detailed prompt, specifically designed for analyzing financial documents. We replaced the baseline prompt with a new template that assigns the LLM the role of a financial analyst and specifies the types of information to extract and the desired output format.

All other parameters remained identical to the baseline configuration from Iteration 0, Section 4.1.1.

Listing 5.1 Baseline Prompt Template

```
template = """
    You are an assistant for question-answering tasks. Use the
       following pieces of retrieved
```

```
    context to answer the question. If you don't know the answer,
       just say that you don't know.
    Use three sentences maximum and keep the answer concise.
    <context>
    {context}
    </context>

Question: {question}
"""
```

Listing 5.2 Enhanced Financial Analyst Prompt Template

```
template = """
You are a specialized ETF and investment fund analyst with expertise
     in analyzing financial documents including annual reports,
   prospectuses, fact sheets, earnings calls, balance sheets, and
   performance summaries.
<context>
{context}
</context>

Extract and synthesize key financial insights from the provided
   context to answer the question. Key information includes:
    - Performance metrics (returns, expense ratios, tracking error)
    - Portfolio composition and allocation percentages
    - Risk indicators and benchmarks
    - Management changes or strategy shifts
    - Fee structures and expense trends

When answering:
    1. Only use information explicitly stated in the context
    2. Cite specific figures with their exact values and time
       periods
    3. If the answer is not clearly contained in the context, state
       "Based on the provided information, I cannot determine [
       specific aspect]" rather than speculating
    4. Distinguish between factual data points and forward-looking
       statements
    5. Preserve the precision of numerical data (do not round unless
        specified)

Keep your response to three sentences maximum unless additional
   detail is essential for accuracy.

Question: {question}
"""
```

5.2.1 Iteration 8 Analysis

This Iteration 8 configuration, using the enhanced prompt template, was evaluated against the test dataset. The performance scores were:

Comparing the results of Iteration 8 in Table 5.1 with the baseline Iteration 0, the enhanced prompt yielded mixed results.

Answer Correctness saw a marginal improvement (0.50 vs 0.49), and Answer Relevancy increased slightly (0.72 vs 0.68). Notably, Context Relevancy showed a significant improvement (0.87 vs 0.76). Conciseness remained unchanged (0.80).

Metric	Baseline Average	Baseline Std. Dev.	Iter 8 Average	Iter 8 Std. Dev.
Answer Correctness	0.49	0.42	0.50	0.27
Answer Relevancy	0.68	0.38	0.72	0.36
Context Relevancy	0.76	0.30	0.87	0.26
Conciseness	0.80	0.32	0.80	0.21

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 5.1 Iteration 8 Evaluation Results

The substantial gain in Context Relevancy is interesting, as the prompt primarily influences the generation step, not retrieval. This might suggest an indirect effect: the judge LLM might perceive the context as more relevant when the generated answer, shaped by the better prompt, aligns more closely with the expected financial analysis style. Alternatively, it could be noise in the evaluation process.

The improvements in Answer Correctness and Relevancy, while present, are modest. This indicates that while the enhanced prompt provides better guidance, its impact might be limited by the quality of the retrieved context or the inherent capabilities of the 8B parameter model for complex financial reasoning. However, given the improvement in Context Relevancy and slight gains elsewhere without sacrificing conciseness. We consider this enhanced prompt to be superior to the baseline for this specific task.

5.3 Iteration 9: Combined Optimizations

Previous iterations identified potentially optimal settings for individual components:

- **Text Cleaning**: Minimal cleaning from Iteration 2 Section 4.3, yielded the highest relevancy scores.
- **Chunking**: Smaller chunks in Iteration 6 Section 4.9, improved correctness over baseline, despite slightly lower relevancy than minimal cleaning alone.
- **Embedding Model**: Kept as nomic-embed-text.
- **Prompt**: Enhanced financial prompt from Iteration 8 Section 5.2, showed modest gains.

This iteration investigates whether combining these individually promising adjustments leads to a cumulatively superior pipeline. The configuration used:

- Text Cleaning: Minimal
- Chunking Strategy: chunk_size=250, chunk_overlap=25
- Embedding Model: nomic-embed-text
- Prompt Template: Enhanced Financial Analyst Prompt
- Other parameters staid as per baseline.

5.3.1 Iteration 9 Analysis

This Iteration 9 configuration, combining minimal cleaning, smaller chunks, and the enhanced prompt, was evaluated. The performance scores were:

Metric	Baseline Average	Baseline Std. Dev.	Iter 9 Average	Iter 9 Std. Dev.
Answer Correctness	0.49	0.42	0.40	0.38
Answer Relevancy	0.68	0.38	0.62	0.40
Context Relevancy	0.76	0.30	0.72	0.41
Conciseness	0.80	0.32	0.54	0.34

Average scores and standard deviations were calculated for each metric by aggregating the individual metric scores over all test cases.

Table 5.2 Iteration 9 Evaluation Results

The results for the combined configuration in Iteration 8, Table 5.2 are counterintuitive. Instead of aggregating the benefits of the individual changes, the performance degraded significantly compared to several previous iterations, including the baseline or some other iterations.

Specifically:

- Answer Correctness (0.40) is lower than the baseline (0.49).
- Answer Relevancy (0.62) is worse than the baseline (0.68).
- Context Relevancy (0.72) is substantially lower than achieved with minimal cleaning alone (0.82 in Iteration 2) or the enhanced prompt alone (0.87 in Iteration 7), and even below the baseline (0.76).
- Conciseness (0.54) dropped dramatically compared to most other iterations, which generally scored around 0.80 or higher except for Iteration 1 and 5.

This outcome strongly suggests that RAG pipeline components interact in complex ways. Optimizing one component in isolation does not guarantee that combining these 'optimal' components will yield the best overall system. In this case, the combination had created negative interference, across all of the four specified metrics. This was possibly caused by retrieving less coherent context snippets which the detailed prompt then struggled to synthesize effectively, leading to less correct, less relevant, and much less concise answers.

5.4 Analysis of Iteration Performance Distributions

While the previous sections analyzed performance primarily through average scores and standard deviations, examining the full distribution of scores for each iteration provides deeper insights into the consistency and reliability of different configurations. Box plots are particularly useful for this, visualizing overall spread of the scores for each metric across the test cases. This allows us to assess the variability and presence of outliers in performance.

5.4.1 Answer Correctness Distribution

Figure 5.2 shows the distribution of Answer Correctness across all iterations, illustrating medians, interquartile ranges [1], and whiskers capturing the rest of the score spread.

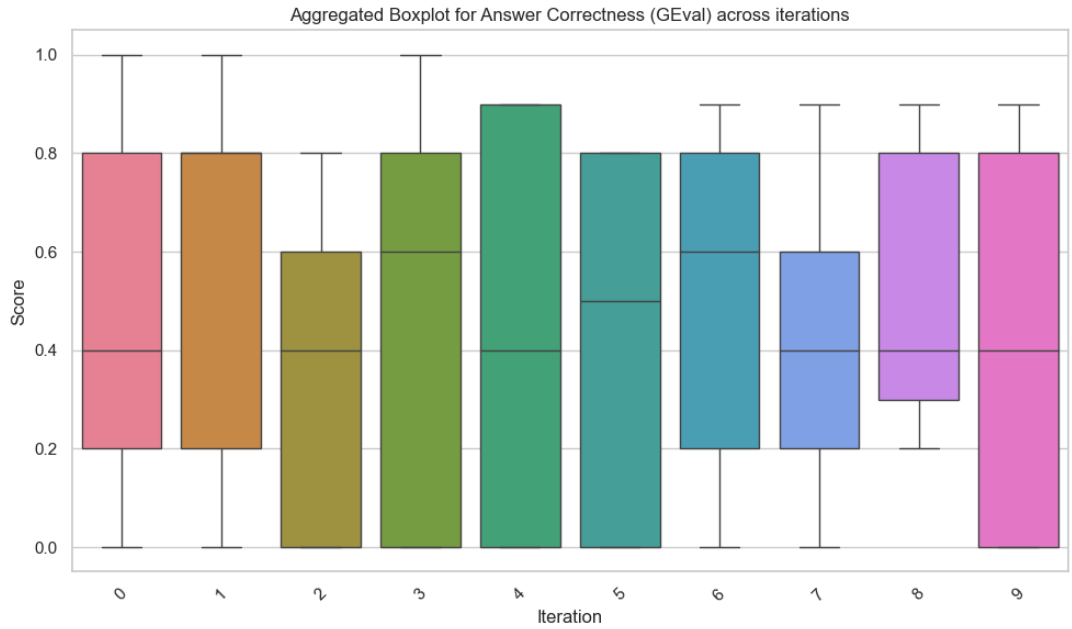

Figure 5.2 Aggregated Boxplot for Answer Correctness across iterations

- **Baseline – Iteration 0:** Scores span the full range (0.0–1.0). The median is 0.4 and the IQR runs from 0.2 to 0.8, indicating that half of the runs fall within this wide band and performance is highly inconsistent.
- **Enhanced Prompt – Iteration 8:** IQR narrows to 0.3–0.9, reflecting the reduced standard deviation noted in Section 5.2.1. Whiskers still capture lower outliers, but central performance is noticeably more consistent.
- **Combined Optimizations – Iteration 9:** Median falls to 0.4 with an IQR of 0.0–0.8, mirroring the baseline's variability and confirming that merging all "best" settings introduced negative interference.
- **Other Iterations:** Iteration 6 stands out with one of the tightest IQR and highest median overall, underscoring that reduced chunk size alone produces the most consistent answer correctness.

5.4.2 Answer Relevancy Distribution

Figure 5.3 presents the box plot for the Answer Relevancy metric across all iterations.

[1]The interquartile range (IQR) is the span between the 75th and 25th percentiles of a dataset, representing the middle 50 % of values; see `https://en.wikipedia.org/wiki/Interquartile_range` for more details.

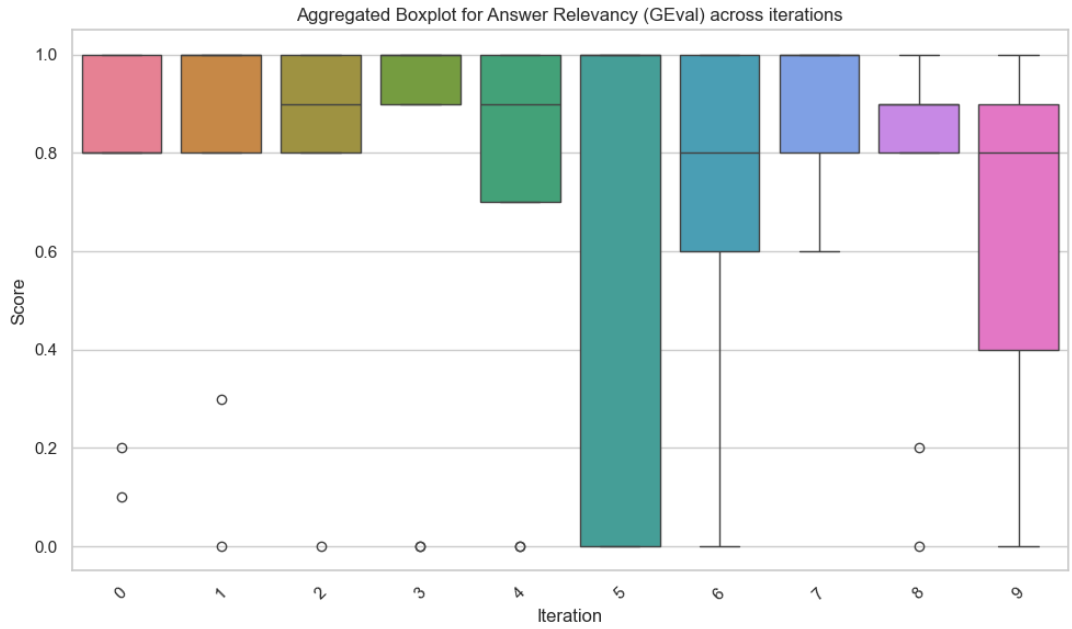

Figure 5.3 Aggregated Boxplot for Answer Relevancy across iterations

- **Baseline – Iteration 0:** Exhibits moderate variability with an IQR roughly from 0.8 to 1.0 but a few low outliers down to 0.1 and 0.2, indicating that while most answers were on-topic, a minority were off-target.
- **Early Prompt Tweaks – Iterations 1–3:** Show gradually tighter IQRs near the top of the scale (0.8–1.0), but also introduce zero-score outliers in Iterations 2 and 3, suggesting that more complex prompts occasionally led the model astray on a handful of test cases.
- **Chunking Experiments – Iterations 4–5:** Both maintain very high medians (0.9–1.0) but exhibit noticeably wider IQRs than earlier runs, indicating increased variability in relevancy. Iteration 4's box spans roughly 0.7–1.0—broader than prior iterations—showing that while many answers remain on-topic, some fall short. Iteration 5 is all over the place, suggesting that smaller chunks can boost peak relevancy but at the cost of consistency.
- **Combined Changes – Iterations 6–7:** Maintain high medians around 0.8–1.0 but introduce occasional outliers down to 0.0 or 0.2, indicating that mixing certain techniques can reintroduce some variance.
- **Combined Optimization – Iteration 9:** Sees a notable drop: median falls to 0.8 with IQR stretching down to 0.4, and low-end whiskers at 0.0, confirming that merging all "best" settings degraded relevancy consistency.

5.4.3 Context Relevancy Distribution

Figure 5.4 displays the box plot for Context Relevancy across iterations.

- **Baseline – Iteration 0:** Narrow IQR (0.7–0.8), with one perfect (1.0) and one low (0.6) outlier, indicating generally on-target context but occasional misses.

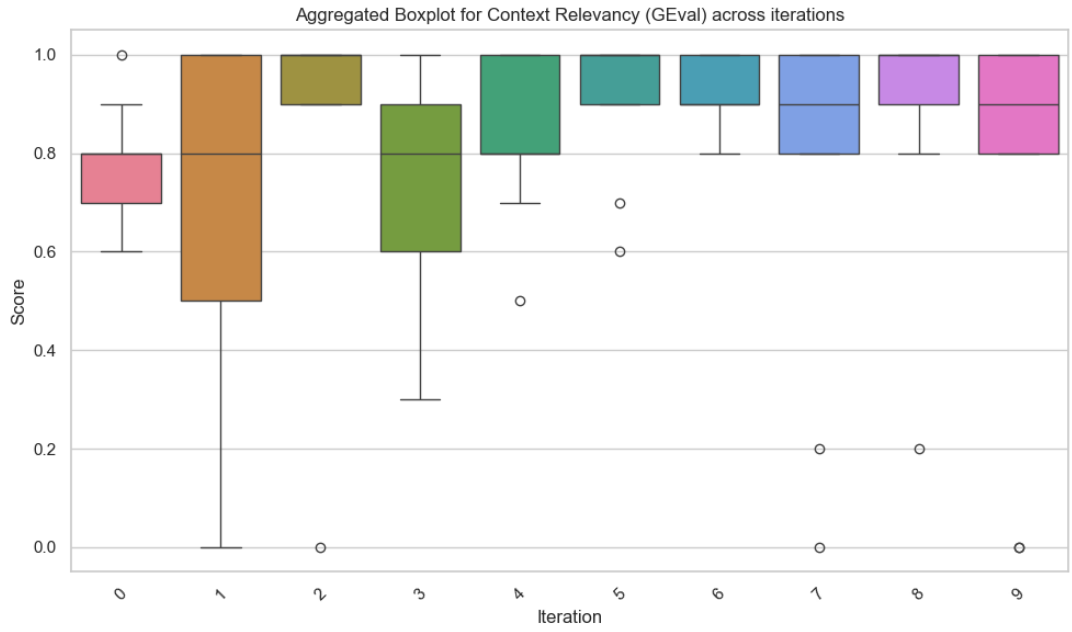

Figure 5.4 Aggregated Boxplot for Context Relevancy across iterations

- **Iteration 1:** Very high variability, IQR (0.5–1.0), whisker down to 0.0, reflecting that aggressive text transformations sometimes broke retrieval relevance entirely.
- **Iteration 2:** Tight top-end IQR (0.9–1.0) but a zero outlier, showing minimal cleaning boosted relevancy for almost all cases, with a couple of context-poor retrievals.
- **Iterations 3–4:** Both improve—Iteration 3 IQR (0.6–0.9), Iteration 4 (0.8–1.0)—demonstrating that measured cleaning and moderate chunk sizes help.
- **Iteration 5:** Very consistent high relevancy (IQR 0.9–1.0), suggesting that the retrieval model was most effective here, with small amount of outliers. Which did not perform terribly also.
- **Iterations 6–8:** Maintain high medians (0.9–1.0) and narrow IQRs, with occasional low-end outliers (0.0–0.2) in Iterations 6 and 7, indicating that some combined tweaks can still miss the mark.
- **Iteration 9:** Strong central tendency (IQR 0.8–1.0) but a zero-score outlier, showing that even the "combined best" pipeline did not fully guarantee relevant context in every case.

5.4.4 Conciseness Distribution

Figure 5.5 shows how conciseness varied across iterations.

- **Baseline – Iteration 0:** High median (0.8) and narrow IQR (0.7–1.0), but a low outlier at 0.2, indicating mostly concise replies with occasional verbosity.

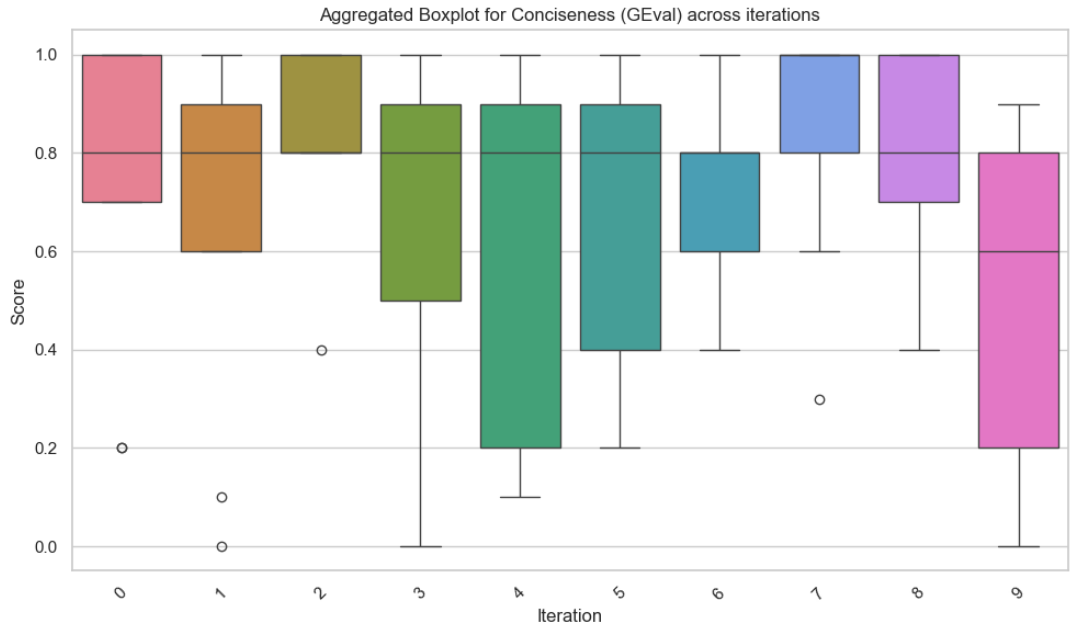

Figure 5.5 Aggregated Boxplot for Conciseness across iterations

- **Iteration 1:** Wider spread (IQR 0.6–0.9) plus a zero-score case, showing that the first prompt tweak introduced some overly long or unfocused responses.
- **Iteration 2:** Achieves perfect consistency (IQR 0.8–1.0, no whisker below 0.8), marking the most uniformly concise iteration—likely due to minimal cleaning letting the model stick closely to instructions.
- **Iterations 3–5:** Medians remain high (0.8) but with broader IQRs extending down to 0.2–0.4 in Iterations 3 and 4, and to 0.4 in Iteration 5, reflecting occasional over-elaboration when context was too fragmented or too dense.
- **Iteration 6:** IQR (0.6–0.8), few outliers, showing a good balance between brevity and detail when chunk size was reduced.
- **Iteration 7:** Very tight IQR at the top end (0.8–1.0) with one low outlier (0.3), indicating the enhanced prompt kept the model mostly concise.
- **Iteration 8:** Strong performance (IQR 0.7–1.0) with no extreme lows, matching its clean, focused prompt.
- **Iteration 9:** Largest drop: median around 0.6 and IQR 0.2–0.8, plus a zero-score case, showing the combined pipeline hampered the model's ability to be concise.

5.5 Future Enhancements for the Financial RAG Pipeline

Beyond the iterations covered in this work, three complementary improvements could further strengthen retrieval-augmented generation on financial documents.

5.5.1 Post-Retrieval Reranking

Introduce a lightweight reranking stage after the initial vector retrieval. An LLM can rescore the top-k candidates by directly comparing each snippet to the query. By promoting truly on-topic passages and filtering out tangential ones, reranking sharpens the context and reduces noisy or misleading inputs, which is critical for precision.

5.5.2 Structured Data Extraction

Financial reports are rich in tables, charts as we know. And generic text chunking often overlooks this part. A more structure-aware preprocessing step could detect and isolate tables or figure captions as dedicated chunks. This can be then later used to generate brief summaries of their key data points, which would be better processable for an LLM. Thereby improving answers that depend on the information stored in these objects.

5.5.3 Iterative Retrieval with Feedback

Adding a feedback loop enabling the LLM to self-assess whether the retrieved context sufficiently answers the question. After initial generation, the model decides or flags "insufficient context" if key facts are missing. The pipeline then re-triggers a retrieval for a second pass with broader scope or a different reranking strategy. This method could prevent hallucinations and better ensure that the pipeline only finalizes answers when it has collected sufficient context.

These three enhancements would make the pipeline more complex and could be integrated modularly into the existing pipeline to boost precision and distance us from answers with incomplete context in the future.

6 User Documentation

This chapter provides essential guidance on setting up the project environment and running the RAG system developed in this thesis. The goal is to enable users to replicate the experiments, run the PoC, or utilize the developed pipeline with their own data. For exhaustive, step-by-step commands and troubleshooting, please refer to the projects README. Task execution is streamlined using Poe the Poet [1], and dependencies are managed with Poetry [2]

6.1 Setup Prerequisites

Before running the system, several prerequisites must be met to ensure all components function correctly.

1. **Ollama Installation and Setup:** The core of the local LLM interaction relies on Ollama. It must be installed on the user's machine following the official instructions available at https://ollama.com/. Once installed, the specific LLM in our case `llama3.1:8b` and embedding model `nomic-embed-text` used in the project configuration found in `src/common/config.py` need to be downloaded using the `ollama pull <model_name>` command. Crucially, the Ollama service must be running in the background for the RAG pipeline to access the local models. User achieves that by running the (`ollama serve`) command.

2. **Project Code:** The project source code needs to be obtained, by cloning the Git repository:

   ```
   git clone https://github.com/martinlejko/bachelors_thesis.
       git
   cd bachelors_thesis/rag_project
   ```

3. **Dependency Management with Poetry:** This project uses Poetry for managing Python dependencies and virtual environments. Poetry needs to be installed first, see https://python-poetry.org/docs/#installation. After navigating into the project directory, dependencies are installed using:

   ```
   poetry install
   ```

   This command creates an isolated virtual environment and installs all required packages specified in the `pyproject.toml` file. User should then activate this created environment. By executing command:

   ```
   eval $(poetry env activate)
   ```

4. **NLTK Data:** Certain text processing steps rely on data from the NLTK library. A helper command is provided via Poe the Poet to download the necessary datasets:

[1] Poe the Poet is a task runner, which simplifies running commands. For more information check out Poe the Poet

[2] Poetry is used for packaging and dependency management. For more information checkout Poetry Documentation.

```
poetry run nltk-setup
```

Note: Ensure you are within the Poetry environment.

5. **Data Placement:** The system ingests documents from specific directories. Publicly accessible files such as PDFs should be placed in `src/data/public/`. Private data, such as Confluence exports, should be placed in `src/data/private/`.

6.2 Running the System

With the environment set up and the Ollama service running, the RAG system can be executed using predefined Poe the Poet tasks. Make sure you are operating within the project's virtual environment.

1. **Running the PoC:** A script inside proof_of_concept.py demonstrates the basic RAG functionality shown in Section 2.1. It can be run using:

   ```
   poe run-proof
   ```

   This provides a quick way to verify the core pipeline is working.

2. **Running a Specific Pipeline Iteration:** The project is structured into iterations `src/iterations/iter_X`, each representing a different RAG configuration. To run a specific iteration for querying, modify the import path in the src/testing/run_pipeline.py script to point to the desired iteration's pipeline module. Then execute:

   ```
   poe run-pipeline
   ```

   This script initializes the chosen pipeline and runs predefined questions.

3. **Running Evaluation Tests:** A key part of this thesis involves evaluating different RAG iterations. The evaluation suite uses DeepEval and Pytest.

   - **Testing the PoC:** A separate test suite exists for the initial PoC, which can be executed by:

     ```
     poe test-proof
     ```

   - **Testing Iterations:** To run the full evaluation across all defined iterations:

     ```
     poe test-iterations
     ```

   Evaluation results are saved in the `test_results/` directory, including JSON files and an aggregated HTML report. To see more detailed information about what is going on during the processes you can view logs in the `debug/` directory. These directories are created upon running the program.

4. **Viewing Documentation:** Project documentation, written using MkDocs[3], can be viewed locally:

[3]MkDocs is static site generator that's geared towards building project documentation. MkDocs user documentation.

```
mkdocs serve
```

Access the documentation via the address that is produced by MkDocs.

6.3 Summary

This chapter outlined the necessary steps to set up the project environment and execute the RAG system, including running the PoC, project iterations, and the evaluation framework. Users should now have a foundational understanding of how to interact with the codebase. For detailed command references, specific configurations, and advanced usage, consulting the projects README is highly recommended.

Conclusion

This thesis presented the design, implementation, and evaluation of a retrieval-augmented generation (RAG) pipeline using local large language models. Through a series of structured iterations and evaluations, key components of the pipeline-data cleaning, chunking, embeddings, and prompt engineering—were systematically refined under the constraint of running on personal hardware (a MacBook Air M1 with 16 GB RAM). While an exhaustive exploration of every possible configuration was beyond the scope of this work, the iterative approach provided valuable insights into how each component affects overall system performance. The resulting RAG system demonstrates that useful capabilities can be achieved even with limited computational resources. The final pipeline configuration delivers performance that is sufficient for use as an assistive tool in corporate workflows, providing relevant and contextually grounded answers to user queries. It is not intended as a replacement for human expertise, but rather as a support system that can efficiently retrieve and synthesize information from private knowledge sources. The evaluations revealed clear trade-offs between the relevance of retrieved content, the factual correctness of the generated responses, and the conciseness of the output. Balancing these objectives is crucial: prioritizing maximum relevance and detail can lead to longer, more informative answers at the cost of brevity, whereas enforcing strict conciseness may omit important context. These findings underscore the importance of tuning the pipeline to align with specific application needs and user expectations. There remains considerable room for further improving the RAG system beyond its current capabilities. More extensive experimentation and tuning—or alternatively, employing larger models on more powerful hardware—could yield significant gains in accuracy and robustness. Importantly, the pipeline's modular design ensures that such upgrades are feasible: individual components (from the vector store to the language model) can be refined or replaced independently as better techniques become available. In summary, this project demonstrates that an effective, privacy-preserving RAG pipeline can be constructed on modest hardware with careful design and iteration, leveraging open-source tools like Ollama and LangChain. It provides a solid foundation for future enhancements as both hardware and algorithms continue to evolve.

Bibliography

1. Palo Alto Networks. *What Are Large Language Models (LLMs)?* 2024. Available also from: https://www.paloaltonetworks.com/cyberpedia/large-language-models-llm. Accessed: 2025-04-06.
2. Zhang, Jeniffer. *Applying rag in enterprise*. 2024. Available also from: https://www.wiz.ai/applying-retrieval-augmented-generation-rag-in-enterprise-search-first-hand-insights/.
3. Wikipedia contributors. *Large language model*. 2025. Available also from: https://en.wikipedia.org/wiki/Large_language_model. Accessed: 2025-04-06.
4. Ji, Ziwei; Lee, Nayeon; Frieske, Rita; Yu, Tiezheng; Su, Dan; Xu, Yan; Ishii, Etsuko; Bang, Yejin; Dai, Wenliang; Madotto, Andrea; Fung, Pascale. Survey of Hallucination in Natural Language Generation. *ACM Computing Surveys*. 2022, vol. 55, no. 12, pp. 1–38. Available from doi: 10.1145/3571730. Retrieved 15 January 2023.
5. Varshney, Neeraj; Yao, Wenlin; Zhang, Hongming; Chen, Jianshu; Yu, Dong. A Stitch in Time Saves Nine: Detecting and Mitigating Hallucinations of LLMs by Validating Low-Confidence Generation. *arXiv preprint arXiv:2307.03987*. 2023. Available from arXiv: 2307.03987 [cs.CL].
6. Yan, Biwei; Li, Kun; Xu, Minghui; Dong, Yueyan; Zhang, Yue; Ren, Zhaochun; Cheng, Xiuzhen. On protecting the data privacy of Large Language Models (LLMs) and LLM agents: A literature review. *High-Confidence Computing*. 2025, p. 100300. issn 2667-2952. Available from doi: https://doi.org/10.1016/j.hcc.2025.100300.
7. Google Cloud. *What is Retrieval-Augmented Generation (RAG)?* 2025. Available also from: https://cloud.google.com/use-cases/retrieval-augmented-generation. Accessed: 2025-04-06.
8. Gao, Yunfan; Xiong, Yun; Gao, Xinyu; Jia, Kangxiang; Pan, Jinliu; Bi, Yuxi; Dai, Yi; Sun, Jiawei; Wang, Meng; Wang, Haofen. Retrieval-Augmented Generation for Large Language Models: A Survey. *arXiv preprint arXiv:2312.10997*. 2023.
9. IBM Research. *What is retrieval-augmented generation?* 2023. Available also from: https://research.ibm.com/blog/retrieval-augmented-generation-RAG. Accessed: 2025-04-06.
10. Prompt Engineering Guide. *Retrieval Augmented Generation (RAG) for LLMs*. 2024. Available also from: https://www.promptingguide.ai/research/rag. Accessed: 2025-04-06.
11. LangChain. *Build a Retrieval Augmented Generation (RAG) App: Part 1*. 2024. Available also from: https://python.langchain.com/docs/tutorials/rag/. Accessed: 2025-04-07.
12. Bergmann, Dave. *What is a context window?* 2024. Available also from: https://www.ibm.com/think/topics/context-window. Accessed: 2025-04-07.

13. Microsoft. *Design and develop a RAG solution*. 2025. Available also from: https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/rag/rag-solution-design-and-evaluation-guide. Accessed: 2025-04-07.

14. Ollama. *Embedding models*. 2024. Available also from: https://ollama.com/blog/embedding-models. Accessed: 2025-04-07.

15. LangChain. *Vectorstores*. 2024. Available also from: https://python.langchain.com/docs/concepts/vectorstores/. Accessed: 2025-04-05.

16. Wikipedia contributors. *Proof of concept*. 2025. Available also from: https://en.wikipedia.org/wiki/Proof_of_concept. Accessed: 2025-04-07.

17. Ghosh, Krishna Sarathi. *How to Run Open Source LLMs on Your Own Computer Using Ollama*. 2024. Available also from: https://www.freecodecamp.org/news/how-to-run-open-source-llms-on-your-own-computer-using-ollama/. Accessed: 2025-04-05.

18. IBM Corporation, Bryan Clark. *What is Quantization?* 2024. Available also from: https://www.ibm.com/think/topics/quantization. Accessed: 2025-04-06.

19. Lütke, Tobi. *Reflexive AI usage is now a baseline expectation at Shopify*. 2024. Available also from: https://x.com/tobi/status/1909251946235437514. Posted on X (formerly Twitter), Accessed: 2025-04-07.

20. IBM Technology. *Intelligent Document Understanding Explained*. 2021. Available also from: https://www.youtube.com/watch?v=FpfhY-_0uCw. Accessed: 2025-04-07.

21. Wikipedia contributors. *Semi-structured data*. 2025. Available also from: https://en.wikipedia.org/wiki/Semi-structured_data. Accessed: 2025-04-07.

22. Wikipedia contributors. *Fine-tuning (deep learning)*. 2025. Available also from: https://en.wikipedia.org/wiki/Fine-tuning_(deep_learning). Accessed: 2025-04-07.

23. LangChain. *Local RAG Tutorial* [online]. 2024. [visited on 2024-12-20]. Available from: https://python.langchain.com/v0.2/docs/tutorials/local_rag/.

24. LangChain. *WebBaseLoader — LangChain Documentation* [online]. 2024. [visited on 2024-12-20]. Available from: https://python.langchain.com/api_reference/community/document_loaders/langchain_community.document_loaders.web_base.WebBaseLoader.html.

25. Wikipedia contributors. *Nearest neighbor search — Wikipedia, The Free Encyclopedia*. 2024. Available also from: https://en.wikipedia.org/wiki/Nearest_neighbor_search. Accessed: 2025-04-05.

26. Lyu, Qing; Havaldar, Shreya; Stein, Adam; Zhang, Li; Rao, Delip; Wong, Eric; Apidianaki, Marianna; Callison-Burch, Chris. Faithful Chain-of-Thought Reasoning. *arXiv preprint arXiv:2301.13379*. 2023. Available also from: https://doi.org/10.48550/arXiv.2301.13379. IJCNLP-AACL 2023 camera-ready version.

27. Fagbohun, Oluwole; Harrison, Rachel M.; Dereventsov, Anton. An Empirical Categorization of Prompting Techniques for Large Language Models: A Practitioner's Guide. *arXiv preprint arXiv:2402.14837*. 2024. Available also from: https://doi.org/10.48550/arXiv.2402.14837.

28. Merth, Thomas; Fu, Qichen; Rastegari, Mohammad; Najibi, Mahyar. Superposition Prompting: Improving and Accelerating Retrieval-Augmented Generation. *arXiv preprint arXiv:2404.06910v2*. 2024. Available also from: https://doi.org/10.48550/arXiv.2404.06910. Submitted on 10 Apr 2024, last revised 19 Jul 2024.

29. Lakatos, Robert; Pollner, Peter; Hajdu, Andras; Joo, Tamas. *Investigating the performance of Retrieval-Augmented Generation and fine-tuning for the development of AI-driven knowledge-based systems*. 2024. Available also from: https://arxiv.org/pdf/2403.09727. Accessed: 2025-04-07.

30. Pratik, Bhavsar; Bogdan, Gheorghe. *LLM-as-a-Judge vs Human Evaluation*. 2024. Available also from: https://www.galileo.ai/blog/llm-as-a-judge-vs-human-evaluation.

31. Herreros, Quentin; Veasey, Thomas; Papaoikonomou, Thanos. *RAG Evaluation Metrics: A Journey Through Metrics*. 2023. Available also from: https://www.elastic.co/search-labs/blog/evaluating-rag-metrics. Accessed: 2025-04-16.

32. Hellstrom, Erich. *Temperature Setting in LLMs: A Comprehensive Guide*. 2025. Available also from: https://blog.promptlayer.com/temperature-setting-in-llms/. Accessed: 2025-04-16.

33. Wikipedia contributors. *BLEU*. 2025. Available also from: https://en.wikipedia.org/wiki/BLEU. Accessed: 2025-04-16.

34. Wikipedia contributors. *ROUGE (metric)*. 2024. Available also from: https://en.wikipedia.org/wiki/ROUGE_(metric). Accessed: 2025-04-16.

35. Roucher, Aymeric. *Using LLM-as-a-judge for an automated and versatile evaluation*. 2024. Available also from: https://huggingface.co/learn/cookbook/llm_judge. Accessed: 2025-04-16.

36. Krumdick, Michael; Lovering, Charles; Reddy, Varshini; Ebner, Seth; Tanner, Chris. *No Free Labels: Limitations of LLM-as-a-Judge Without Human Grounding*. 2025. Available also from: https://arxiv.org/abs/2503.05061. Accessed: 2025-04-16.

37. Liu, Yang; Iter, Dan; Xu, Yichong; Wang, Shuohang; Xu, Ruochen; Zhu, Chenguang. G-EVAL: NLG Evaluation using GPT-4 with Better Human Alignment. *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*. 2023, pp. 2511–2522.

38. Hashemi-Pour, Cameron; Craig, Lev. *What is Chain-of-Thought Prompting (CoT)? Examples and Benefits*. 2025. Available also from: https://www.techtarget.com/searchenterpriseai/definition/chain-of-thought-prompting. Accessed: 2025-04-16.

39. DeepEval contributors. *Metrics for LLM Evaluations*. 2025. Available also from: https://www.deepeval.com/docs/metrics-llm-evals. Accessed: 2025-04-16.

40. DeepEval contributors. *Test Cases | DeepEval - The Open-Source LLM Evaluation Framework*. 2025. Available also from: https://www.deepeval.com/docs/evaluation-test-cases#llm-test-case. Accessed: 2025-04-16.

41. DeepEval contributors. *Evaluation Datasets*. 2025. Available also from: https://www.deepeval.com/docs/evaluation-datasets. Accessed: 2025-04-16.

42. Liu, Jason. *Systematically Improving RAG Applications*. 2025. Available also from: https://jxnl.co/writing/2025/01/24/systematically-improving-rag-applications/#31-why-evaluate-retrieval-first. Accessed: 2025-04-16.

43. Ollama contributors. *nomic-embed-text: A High-Performing Open Embedding Model*. 2025. Available also from: https://ollama.com/library/nomic-embed-text. Accessed: 2025-04-16.

44. Wikipedia contributors. *Metadata*. 2025. Available also from: https://en.wikipedia.org/wiki/Metadata. Accessed: 2025-04-16.

45. Wikipedia contributors. *Cache (computing)*. 2025. Available also from: https://en.wikipedia.org/wiki/Cache_(computing). Accessed: 2025-04-16.

46. Wikipedia contributors. *Optical character recognition*. 2025. Available also from: https://en.wikipedia.org/wiki/Optical_character_recognition. Accessed: 2025-04-16.

47. Wikipedia contributors. *Regular expression*. 2025. Available also from: https://en.wikipedia.org/wiki/Regular_expression. Accessed: 2025-04-16.

48. Ishimwe, Hervé. *Finding the Best Open-Source Embedding Model for RAG*. 2024. Available also from: https://www.timescale.com/blog/finding-the-best-open-source-embedding-model-for-rag. Accessed: 2025-04-21.

49. Meta AI. *Introducing Meta Llama 3: The Most Capable Openly Available LLM to Date*. 2024. Available also from: https://ai.meta.com/blog/meta-llama-3/. Accessed: 2025-04-21.

50. Shah, Drishti. *The Complete Guide to Prompt Engineering*. 2024. Available also from: https://portkey.ai/blog/the-complete-guide-to-prompt-engineering. Accessed: 2025-04-21.

List of Figures

List of Tables