Se former autrement

2023 - 2024

GRADUATION PROJECT

NATIONAL ENGINEERING DEGREE

SPECIALTY : INFORMATION TECHNOLOGY

TITLE : AI Search Engine

By: ***Mohamed Amine TAIEB***

Academic supervisor: ***Chifaa GHANMI***

Corporate Internship Supervisor: ***Hazem RAOUAFI***

Standard Sharing Software

Abstract

With the recent progress in technology and Artificial Intelligence in particular, businesses became increasingly interested in leveraging the domains of application of these advancements in order to optimize their internal processes and the efficiency and productivity of their employees. The emergence of Generative AI chatbots (e.g., ChatGPT, Google Gemini, etc...) has opened up exciting new perspectives in this domain; It came up with an alternative approach for searching large volumes of information, delivering significantly more insightful results in less time and effort than traditional methods.

This report presents the design and implementation of an Entreprise Knowledge Base searching tool to facilitate the quick and precise retrieval of custom and proprietary information by employees, thereby enhancing knowledge-intensive workflows. This solution was conceived in fulfillment of an end-of-studies internship project at "Standard Sharing Software", specifically tailored to the company's requirements. It provides users with functionalities to augment and organize the knowledge base, enabling them to retrieve the most relevant contents thereof when needed.

To fulfill the purposes of this solution, various technologies have been explored and implemented: A Vector Store for persisting documents into the knowledge base, embeddings and similarity search algorithms for retrieving the most relevant passages corresponding to user queries, Large Language Models (LLMs) to synthesize these retrieved passages, condensing them into concise and understandable responses and Prompt Engineering for the optimal utilization of LLMs' capabilities, among other techniques. The synergistic orchestration of these diverse tools constitutes the Retrieval-augmented Generation (RAG) framework.

This report first details the context and the compelling need for such solution within the company. It subsequently introduces the various technical terms and concepts employed throughout the project. It then elucidates the rationale behind the selection and application of these technologies. Finally, it concludes by detailing the implementation process and the achieved results, while also acknowledging potential limitations and avenues for further improvement.

Keywords: *Retrieval-augmented Generation (RAG), Large Language Models (LLMs), Generative AI, Vector Store, Embeddings, Similarity Search, Prompt Engineering*

Acknowledgements

I would like to thank...

Contents

List of Figures

List of Tables

Keywords and Abbreviations

- 3S : Standard Sharing Software (The hosting company)
- RAG : Retrieval-Augmented Generation
- LLM : Large Language Model
- EKB : Entreprise Knowledge Base
- FM : Foundational Model
- AI : Artificial Intelligence
- GAI : Generative Artificial Intelligence
- GPT : Generative Pre-trained Transformer
- NLP : Natural Language Processing
- PE : Prompt Engineering
- XoT : Everything of Thought
- CoT : Chain of Thought
- ToT : Tree of Thoughts
- RL : Reinforcement Learning
- RLHF : Reinforcement Learning from Human Feedback
- SBERT : Sentence Bidirectional Encoder Representations from Transformers (or Sentence Transformers for short)
- HF : Hugging Face
- MTEB : Massive Text Embedding Benchmark

- ML : Machine Learning
- DL : Deep Learning
- URL : Uniform Resource Locator
- API : Application Programming Interface
- OS : Operating System
- CPU : Central Processing Unit
- GPU : Graphics Processing Unit
- RAM : Random-access memory
- CUDA : Compute Unified Device Architecture
- VM : Virtual Machine
- WSL : Windows Subsystem for Linux
- JS : JavaScript
- HTML : Hypertext Markup Language
- FOSS : Free and Open-Source Software
- TTM : Time To Market
- a.k.a. : also known as
- e.g. : for example
- i.e. : that is

Chapter 1

Introduction

This chapter is a preamble to the conceptualization of this end-of-studies project. It lays out the context and influences which made it relevant in current settings. First, it describes the background and sequence of developments that brought much of the employed technologies into existence. Next, it introduces the hosting company and discusses its necessity for a such novel solution. And finally, it walks through the pursued objectives and innovations which address the insufficiencies of existing solutions.

1.1 Background

Business Knowledge is both integral and proprietary for any enterprise. The contents of private documents are useful for internal employees who are constantly consuming it to accomplish their workstreams. It becomes evident when considering that modern software development and network infrastructure deployment (among many other fields) are often based on exploring exhaustive documentation and lengthy research papers. Paradoxically, this upfront research and learning introduce a sort of bottleneck requiring much time before a new project gets initiated. In most cases, a further looking up for previously reviewed information is often required. Accelerating delivery, however, remains a key objective for organizations seeking a competitive edge. This together emphasizes the need to consider and propose innovative solutions which would help reduce the preliminary requirements and achieve a faster time-to-market (TTM).

In line with this discussion, it is worth highlighting the evolving landscape of web search in the contemporary era. We used to input a query into a search engine (e.g. Google) and it would look through its indexed webpages and then return a list of the most relevant pages which we would then read until we have a satisfying answer. This paradigm has veered towards generative AI chatbot solutions, like ChatGPT, Copilot and Gemini, which leverage Machine Learning algorithms to mimic the natural language understanding capability of humans to generate short and accurate answers based on public information and the open web.

Auspiciously, the underlying technology that empower this kind of chatbots is discrete from their corresponding platforms, i.e., it can be integrated in other projects as a library or a software component rather than being exclusive to their native applications. This presents an interesting theme for an internship project which attempts to build upon these technologies to provide a much needed solution for an entreprise with many activities and projects to accomplish and for a data science student aspiring to continuously learn the newest trends in AI and Machine Learning.

1.2 Hosting Company

The hosting of this project was managed by Standard Sharing Software (3S).

Figure 1.1: Standard Sharing Software logo

3S Group is a leading company in Tunisia, specializing in the integration of IT infrastructures and the provisioning of advanced technological solutions. It was founded in 1988 with the mission of transforming the Tunisian technological landscape. The group brings together more than 650 employees spread across 16 separate affiliates, each specialized in different cutting-edge technologies covering a wide range of sectors, including the integration of IT infrastructures, the provisioning of internet services, the edition of ERP solutions, IT training, and the management of multichannel call centers.
It is based in Montplaisir, Tunis, Tunisia (headquarters), and in Charguia 1, Tunis (the office where this internship was conducted), with different teams specializing in Network and Telecommunication infrastructures, Cyber security, Cloud Computing and Software. This project was organized and executed in the Cloud and Software department of the company and close collaboration with these different units was required since they constitute the targeted users of this project.
3S Group's work culture helped the success of this project through its core values of commitment, excellence, innovation and teamwork, which were reflected during the whole duration of this internship, from project initialization until delivery, as evidenced by the organizers's relentless dedication to meeting project deadlines, the pursuit of using innovative tools, the collaborative problem-solving approach, and the unwavering commitment to delivering exceptional results.

1.3 Problem Statement

Traditional enterprise search engines, including the methods currently utilized by 3S employees, often have many limitations due to the complexity of user queries' context comprehension and human language understanding. This is true because traditional methods of content searching is based on exact text matching without semantic-based retrieval. Even with the emergence of AI tools like ChatGPT, which address Natural Language Processing/Analysis, access to large enterprise data is still a major hurdle for such systems. Also, we have witnessed in recent years the amount of pressure governments have put on AI products in order to evade potential privacy and copyright infringements. This is good overall as such regulations ensure publishers' copyright and security of private information. However, this also means that companies need to develop their own solutions to effectively harness the recent breakthroughs of this field in their toolkit.

In the table below, an identification and comparison between a few existing solutions has been undertaken to understand their limitations and how to build upon them.

Solution	**LLM Chatbots**	**NotebookLM**	**Verba**
Developer	Google, OpenAI and Microsoft	Google	Weaviate
RAG-based (with persistent storage)	✘	✔	✔ / ✘
Enterprise focused	✘	✘	✔ / ✘
Extensible	✔ / ✘	✘	✘
Multiple LLMs	✘	✘	✔
User-LLM feedback	✔	✘ / ✔	✘
Stable (Not experimental)	✔	✘	✘

Table 1.1: Comparison of existing solutions

In general, one can utilize any ready-to-use LLM-powered chatbot (e.g. ChatGPT, Microsoft Copilot, Google Gemini, and many others). But these solutions, even

though allowing document-answering, don't come with document persistence, which means by leaving the chat, the documents are gone unless re-uploaded.
Google is experimenting with a new solution (NotebookLM) which allows exactly that by connecting to cloud-hosted documents, in addition to methods allowing to upload local files and webpage content from URLs. Yet, this falls short for enterprise use as it is intended for casual use cases (e.g. note taking, document editing suggestions and personal project research) rather than question-answering and does not provide features to share knowledge between different teams. On the other hand, Weaviate, a company behind a vector store implementation, maintains a repository on GitHub which addresses the same objectives as this project. Their solution, called Verba, is a well-organized open-source project that can be easily deployed on-premises or in a cloud environment. However, several challenges need to be addressed before a company can adapt it to their specific needs: customization is difficult if not impossible given that the project depends on many packages they developed or contributed to (goldenverba, weaviate, ...), requiring many API keys from various providers (vector database, LLMs, cloud providers, etc...), in addition to the inability to create and manage different databases for different teams.

This scarcity and inadequacy of available solutions raises the need to design and develop a custom product suitable for the specific needs and requirements set up by individual enterprises, which are discussed in the following section.

1.4 Objectives

This project aims to address general-purpose large language models' limitations, which can be summed up in data confinement to training phase and private data access, by developing a novel search functionality that leverages retrieval-augmented generation techniques to deliver insightful results for enterprise data. The system will prioritize the following objectives:

- Enhance AI-generated responses with External Knowledge: Retrieve relevant passages from a curated enterprise knowledge base each time an LLM is prompted to improve its factual grounding and reduce the risk of hallucinating.
- Flexible Data Ingestion and Organization: Implement diverse methods for multi-source, format-agnostic file content and document indexing into manageable and domain-specific vector databases to tailor to enterprise needs and potential evolution.
- Diverse Answering Options: Allow the selection from a list of various Large Language Models for answering and just-in-time (of generating responses) data scraping to diversify and improve results each time.
- User-Driven Feedback: Integrate a mechanism for providing user feedback for LLM-generated responses which can be used to rank future results and further train and improve LLMs (when supported).
- Leverage RAG-based metrics for evaluation: Employ retrieval-augmented generation evaluation metrics to assess the quality of pipeline stages' results (retrieval quality/accuracy, context-generation consistency, answer relevancy)

1.5 Significance of the study

The proposed Generative AI solution would help employees find answers from enterprise knowledge without having to browse vast amounts of documents. This is enabled by augmenting LLM knowledge using techniques of retrieval-augmented generation and prompt engineering for context precision. It would also allow them to interfere in the steps of answer-generation by providing ways to customize the external knowledge base and conduct online researching, powered by traditional search engines and generative AI researching tools, through a unified interface.

From an educational perspective, this project presents a valuable opportunity to gain expertise on some of the most capable ML models and personalize/extend their functioning. Anticipated learning outcomes of accomplishing this project include an enhanced perception of large models' parameters and how they affect performance, picking up prompt engineering techniques, planning out and implementing RAG pipelines that tailor to specific needs, using both local and cloud components, in addition to developing various data scraping methods.

Chapter 2

Literature Review

For the purpose of this project, an extensive and investigative study into recently released Large Language Models has been conducted. This chapter delves into a comprehensive exploration of Text-Generative AI in general: it examines the concept and diverse applications of generative AI and a Retrieval-augmented Generation pipeline system, its elements and components alongside a comparison between some of key aspects, and its eventual limitations.

2.1 Text Generative AI

The term Text Generative AI refers to the artificial generation of textual content, it can allude to the tasks of next word suggestions, summarization, rewriting in different tones, cross-language translation, question answering, text or code generation etc... It is a type of Artificial Intelligence that utilizes models trained on massive natural language data (Large Language Models). Prominent examples of such models include GPT-3, Gemini, Llama-3. Even though these some of these LLMs are multi-modal (not only text-generative AI, can also analyse multimedia content such as pictures, audio...), they are in essence language models whose functionality provide a good foundation for natural language processing and generation tasks.

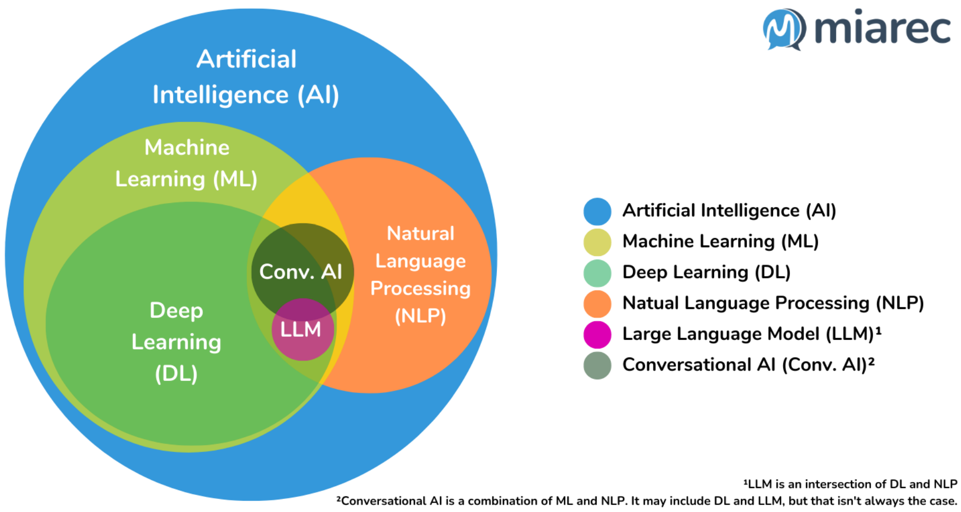

Figure 2.1: Relationship between AI, ML, DL, NLP, and Conversational AI terms. [miarec]

This figure illustrates the relationship between AI domains. In this project, our focus is more onto LLMs (we use the term "LLM" interchangeably to mean "Large Language Model" that may or may not be "Conversational AI"), which are ML models able to handle some NLP tasks, like translation, summarization in addition to their main task of text generation.

Recent trends in Large Language Models have shown how well they can perform as searching tools or assisting agents. They mimic humanistic thinking and actions by providing chat-like responses to questions and messages. It can achieve this through its ability to recognize whether the prompt is a question, a request to perform an action, or a casual natural language task.

2.2 Retrieval-augmented Generation

RAG refers to the process of augmenting LLM knowledge by automatically fetching and providing relevant context for the question. It is a technique used to address data confinement and hallucination of generative AI models. This is achieved without re-training the models, which provides a cost-effective approach to improving LLM output so it remains useful and accurate in various scenarios.

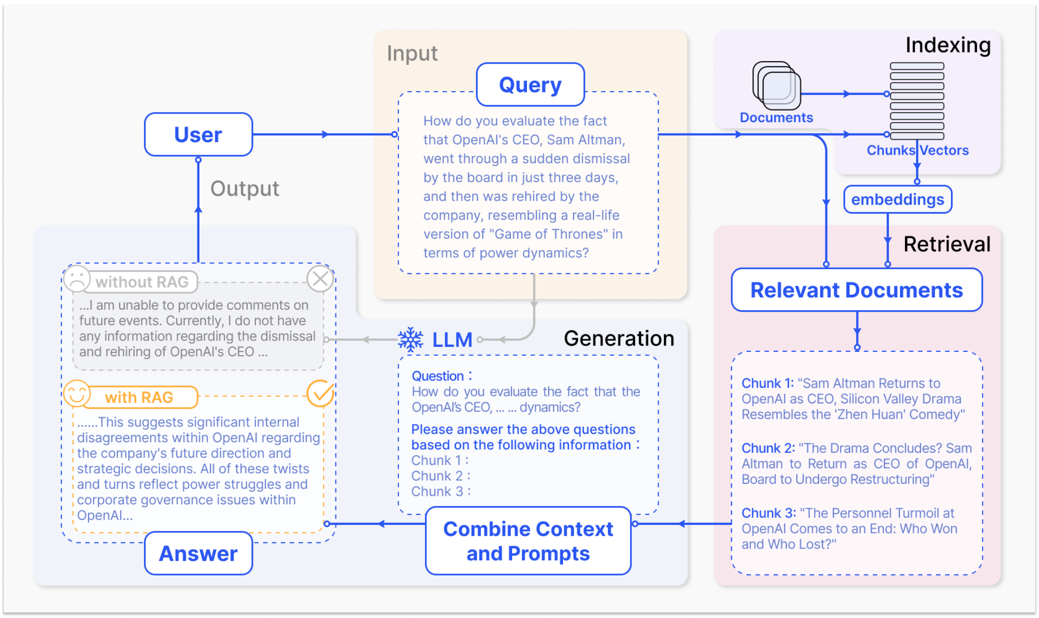

Figure 2.2: An illustration of a RAG pipeline in work. ([Retrieval-Augmented Generation for Large Language Models: A Survey], March 2024)

A comparative representation between a simple LLM and a RAG process applied to question answering. RAG mainly consists of 3 steps: 1. Indexing: Documents are split into chunks, encoded into vectors, and stored in a vector database. 2. Retrieval: Retrieve the Top k chunks most relevant to the question based on semantic similarity. 3. Generation: Input the original question and the retrieved chunks together into LLM to generate the final answer.

2.2.1 Retrieval-augmented Generation paradigms

Retrieval-augmented Generation process has been constantly evolving since the emergence of transformer-based LLMs (GPT models). It has significantly improved on the limitations of LLMs by developing their performance while being cost-effective. Even so, a simple RAG pipeline still exhibit various limitations such as 'retrieval-echoing generation' which means outputting the retrieved documents rather than synthesizing answers to add more insights to the generated text. Moreover, this naive RAG paradigm still introduce hallucinations into answers by confining the generative process to the retrieved documents, which in many cases may not be pertinent to the query (such as in case relevant information not present in the database). In this respect, as more research and engineering is continually being conducted, advanced spinoffs of RAG has emerged to reduce the mis-reckoning and limitations of naive RAG.

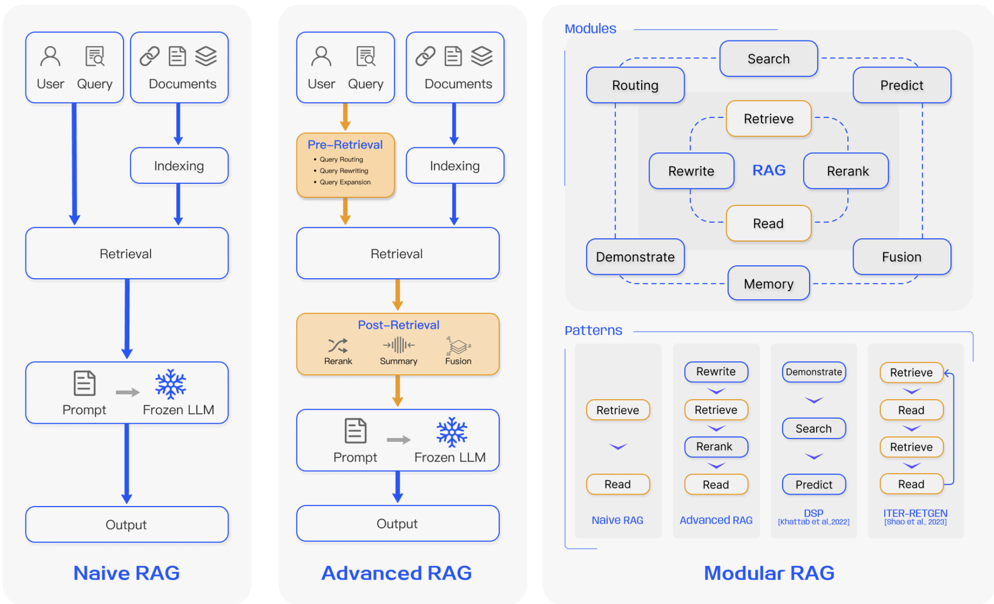

Figure 2.3: Comparison between the three paradigms of RAG: Naive, Advanced and Modular RAG. ([Retrieval-Augmented Generation for Large Language Models: A Survey], March 2024)

Naive RAG (LEFT) mainly consists of three parts: indexing, retrieval and generation. Advanced RAG (MIDDLE) proposes multiple optimization strategies around pre-retrieval and post-retrieval, with a process similar to the Naive RAG, still following a chain-like structure. Modular RAG (RIGHT) inherits and develops from the previous paradigm, showcasing greater flexibility overall. This is evident in the introduction of multiple specific functional modules and the replacement of existing modules. The overall process is not limited to sequential retrieval and generation; it includes methods such as iterative and adaptive retrieval

In the light of this, we focus on the advanced RAG pipeline structure, which addresses many of the shortcomings of a naive RAG chain without introducing too much complexity to the overall pipeline, (which may result in very long question-to-generation delay). This paradigm involves knowledge base augmentation by routing the input query to other processes and external knowledge sources such as web content, summarization, translation, generative research conducting and LLM instructing through prompt engineering in addition to the re-ranking of retrieval and generation phases' results. This is achieved through introducing two additional steps or processes into the overall pipeline: pre-retrieval and post-retrieval. The pre-retrieval phase focuses on selecting knowledge source from which the context can be retrieved (query routing), translating or summarizing the question if needed (query rewriting) and connecting it to external knowledge sources e.g. search engines and AI research generator (query expansion). The post-retrieval phase focuses on constructing the best prompt by interpolating more relevant context into it while eliminating unrelated parts (in case the knowledge base does not contain much pertinent information to the query). This phase also addresses content overload by filtering recurrent and repetitive information, compressing or summarizing context (if needed), and re-ranking generated answers (in our case of multiple LLMs).
These extra steps would result in better generative AI experience in case knowledge base augmentation is required just-in-time of answer-generation or a re-ranking/adjustment of either retrieval or generation phase output is required.

2.2.2 Significance of Retrieval-augmented Generation

RAG technology comes with many benefits for businesses:

- Private and up-to-date data: Even with the best performing LLM products, it is challenging to maintain relevancy in an enterprise environment. By allowing AI models to connect to external knowledge bases and personal data, much more relevant results can be achieved.
- Cost-effective: Even with the availability of open-source LLMs and cloud-based FMs (Foundational Models), the computational and financial costs of re-training such models on domain-specific or private data can be high. By circumventing the re-training phase of LLMs, a huge gain can be brought off using this technique.
- More control and customization: It is easy for developers to customize RAG pipelines: Adapting to changing requirements and sources of information, troubleshooting model performance and results, restricting access to some information for authorized users, etc...
- User trust: By enabling user choices and interference in the overall process of RAG, users can look up how their choices in information sources and LLM selections directly affect the system's performance and results.

In many ways, RAG techniques, when combined with an effective Prompt Engineering approach, provide an alternative framework to re-training or fine-tuning with better adaptability and performance on knowledge-intensive tasks. This is because it is difficult for LLMs to capture new factual information through unsupervised fine-tuning. RAG technique, on the other hand, is designed specifically to control and prioritize relevant data in addition to instructing the model behavior.

2.3 Components of a typical RAG pipeline

The RAG techniques often involves many stages and processes which take place between prompting and answer-generation (as seen in previous sections). This section introduce high-level definitions of the discrete components' concepts and their orchestrated functionality.

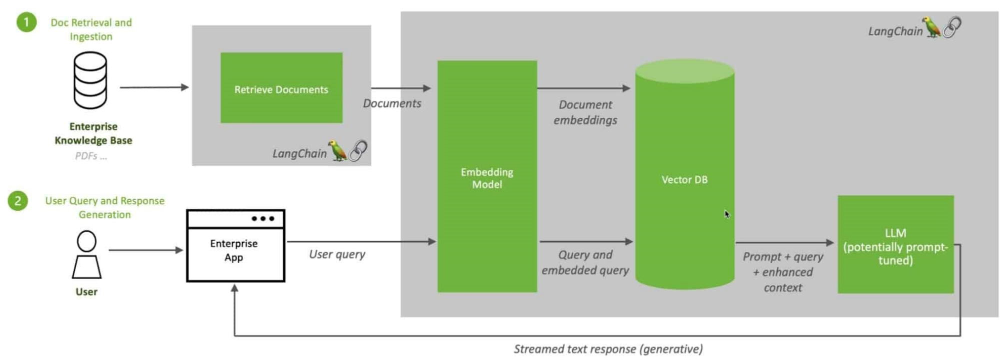

Figure 2.4: Retrieval-augmented Generation Sequence Diagram. [NVIDIA Blogs / What Is Retrieval-Augmented Generation, aka RAG?]

2.3.1 Vector Stores

'Vector Store' refers to a type of database responsible for the storage and indexation of documents / unstructured data in a numerical representation suitable for retrieving relevant parts from large volumes of data through similarity searching algorithms.

> "Traditional databases are made up of structured tables containing symbolic information. For example, an image collection would be represented as a table with one row per indexed photo. Each row contains information such as an image identifier and descriptive text. Rows can be linked to entries from other tables as well, such as an image with people in it being linked to a table of names.
>
> AI tools, like text embedding (word2vec) or convolutional neural network (CNN) descriptors trained with deep learning, generate high-dimensional vectors. These representations are much more powerful and flexible than a fixed symbolic representation, as we'll explain in this post. Yet traditional databases that can be queried with SQL are not adapted to these new representations. First, the huge inflow of new multimedia items creates billions of vectors. Second, and more importantly, finding similar entries means finding similar high-dimensional vectors, which is inefficient if not impossible with standard query languages." ([Faiss: A library for efficient similarity search], 2017)

This specific type of database provides many functionalities pertinent to the functioning of a RAG pipeline.

Embedding Models

Vector, as in vector store, refers to the embedding vectors of data (arrays of floating-point numbers) which are produced by AI algorithms called embedding models. The output of these models captures the semantics of what is being embedded, which is suitable for LLM use cases. As in LLMs' case, embedding models can be designed specifically to handle textual data or multimedia files.

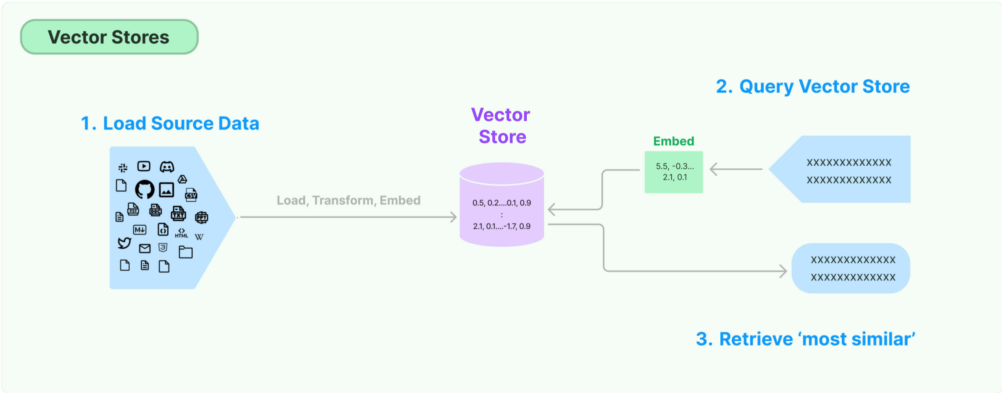

Figure 2.5: Vector Store Process Diagram. [LangChain Documentation]

This figure illustrates the functioning of an embedding model in a vector store environment, transforming source data to numerical representations, and embedding queries to retrieve the most similar vectors.

Chunking

In addition to embedding documents, one should consider the size of these data (may be a single huge document) and the context window limitation of LLMs, as each model has some limit on the amount of information it can receive as context. In these conditions, a chunking of documents should be implemented before the embedding and storing phases. This technique allows to divide documents into smaller passages, while marking these with metadata (document id, order etc...) which would allow the vector store to structure and store these chunks as if they were a single monolithic record.

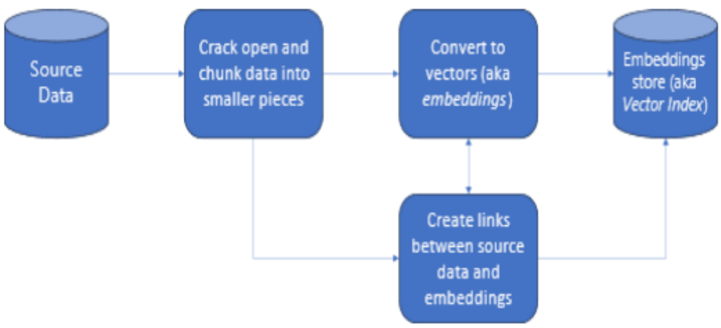

Figure 2.6: Embedding Document Chunks Diagram. (Microsoft, 2023)

Similarity Searching

In addition to the ability to store documents' sections in a suitable form, the vector store index should also be able to search these elements (and then decode them back into their original textual form). This is achieved through similarity searching algorithms often implemented as constituent functionalities with the vector store. These functions return a number of passages semantically similar to the search term. This is achieved due to the concept of distance calculation between vectors in mathematics.

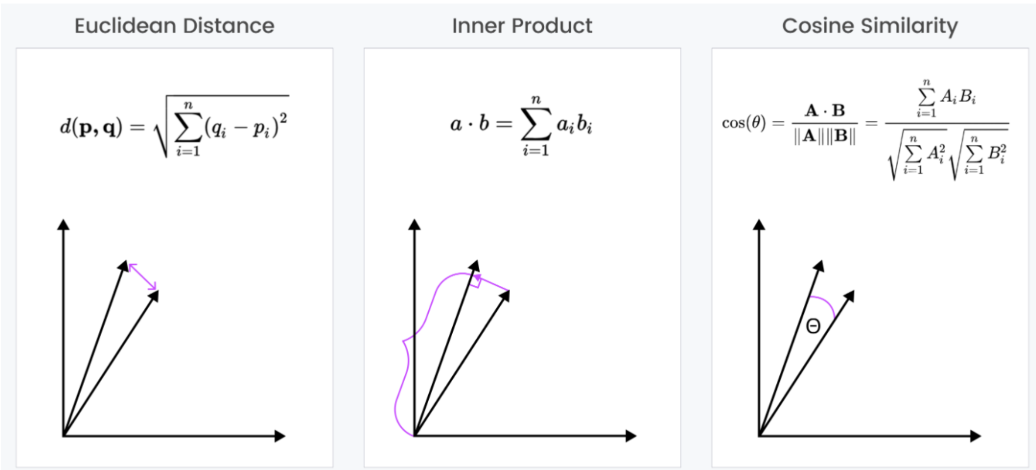

Figure 2.7: Similarity Metrics for Vector Search. [Zilliz Blog]

2.3.2 Large Language Model

This is the principal component of a RAG system. A textual GAI model acts as tool that understands user prompts' context and generate textual responses accordingly. Essentially, these models are transformer-based, which allows them to learn complex relationships between words in sentences thanks to their attention mechanisms.

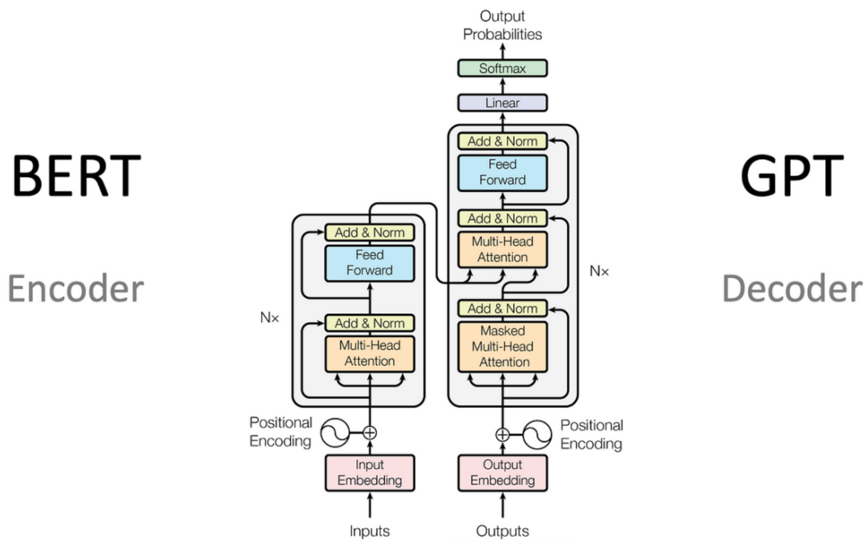

Figure 2.8: An illustration of main components of the transformer model from the original paper, where layer normalization was performed after multiheaded attention. [On Layer Normalization in the Transformer Architecture (2020)]

To provide some cursory understanding of the functioning of these models, the following are the main steps that such architecture involves:

- **Input Embedding:** The transformer first converts the input text into a series of numerical representations, which are then passed through a positional encoding layer. This layer adds information about the word's position in the sentence, which is important because word order matters in a language like English.

- **Encoder Layers:** The core building block of the transformer encoder is the "encoder layer". An encoder layer typically consists of two sub-layers: a multi-head attention layer and a feed-forward layer.

- **Multi-Head Attention Layer:** The multi-head attention layer allows the model to attend to different parts of the input sentence simultaneously. This is important for understanding the relationships between words in a sentence.

- **Feed Forward Layer:** The feed-forward layer is a simple neural network that further processes the information from the attention layer.
- **Decoder Layers:** After the encoder has processed the input text, the decoder generates the output text. The decoder also uses encoder layers, but with an additional masked multi-head attention layer. This layer prevents the decoder from attending to future words in the output sentence, which would allow it to "cheat" by looking ahead.
- **Softmax Layer:** The softmax layer converts the decoder's output into a probability distribution over all the words in its vocabulary. This allows the model to predict the next word in the sentence for instance.

2.3.3 Prompts and Prompt Engineering

Prompt Engineering (PE) plays a crucial rule in the context of RAG and LLMs in general. It refers to how the model is prompted, i.e. what does it receive as input. It may include several instructions to the LLM that guides it on how to perform the generation stage. It also can include different parts or steps, such as passing the retrieved context or chat history directly into the prompt, provide it with examples which it should consider when providing answers, or instruct it on how long the answer should be or in which tone it should respond.

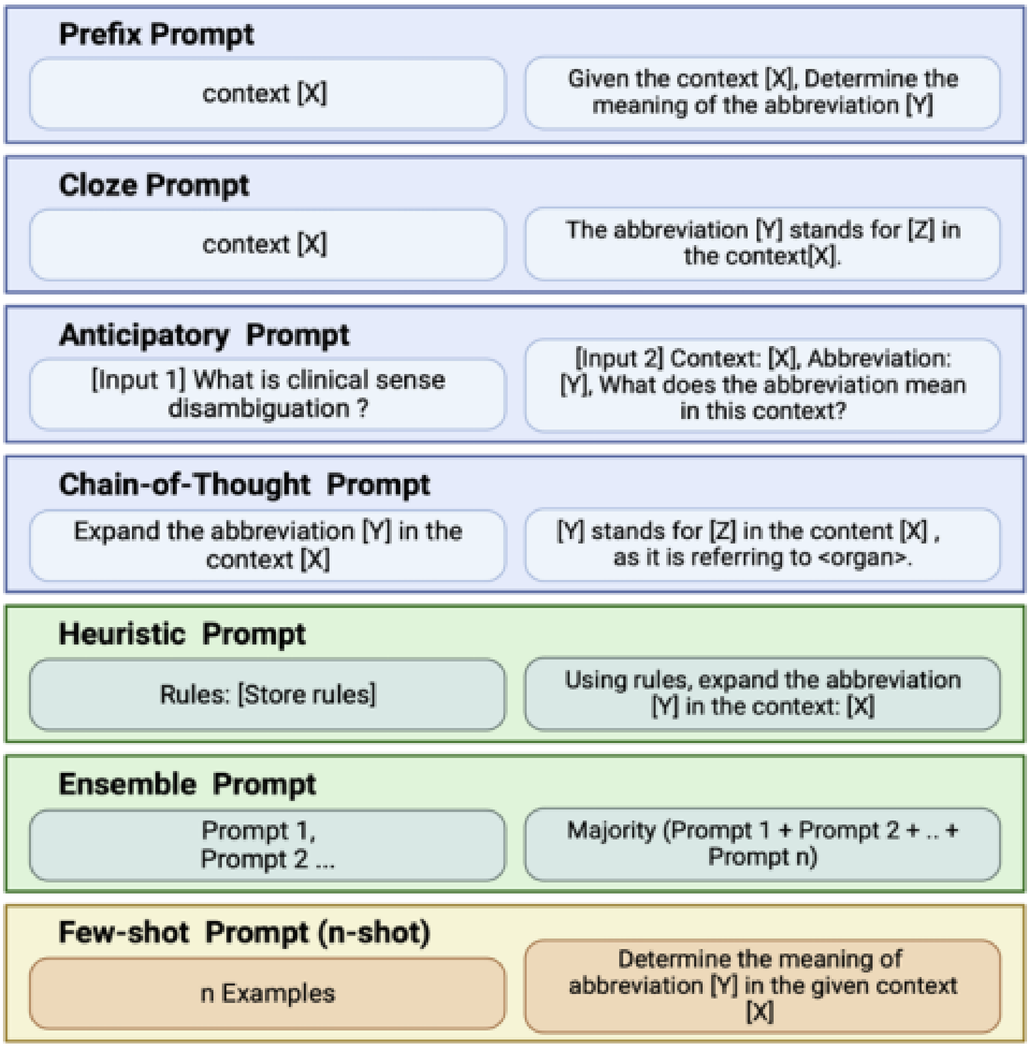

Figure 2.9: Types of Prompts: [X]: context, [Y]: Abbreviation, [Z]: Expanded Form ([A Study on the Implementation of Generative AI Services Using an Enterprise Data-Based LLM Application Architecture], 2023)

2.4 Conclusion

This chapter has identified the key components that make up our system. In summary, to implement a functional RAG pipeline, one needs to orchestrate a Vector Store with methods for knowledge augmentation, similarity search methods to retrieve relevant information, LLMs to generate responses, and prompt engineering to control LLMs' behavior. Having a clear understanding of these individual parts is essential, as the following chapter is about adapting these concepts to our requirements and laying out the system's architecture.

Chapter 3

Methodology

This chapter explores the implementation strategies of the system's components and explains the rationale behind the decisions made when different alternatives were available. It then details the project's schedule and anticipated time frames for tasks' completion.

3.1 Groundwork Analysis

This section is about researching and exploring the different approaches which can be pursued to implement the various components of the system and choosing the proper way that fits our situation.

3.1.1 Cloud providers vs local environment

The main components of a RAG system (LLMs, a Vector Store, Embedding Model) can be deployed and used both locally or through cloud providers. One should consider the benefits and drawbacks of each method and choose accordingly.

Vector Store and Embedding Models

These technologies are coupled together because they complement each other (a vector store depends on an embedding model) and it would be preferable to organize a single environment for both of them. For example, if one or both of these were hosted on the cloud, an internet failure or an access token expiration would result in the overall system outage. It would be wise to avert this downtime by choosing to implement those in a local environment. This would mean that uploading local files will always succeed and network overload or API key expiration risks would be mitigated.
On the other hand, choosing this method would result in an increased hardware utilization (RAM, GPU and Hard Disk) and execution time.
In these circumstances, the advantages of implementing the vector store and embedding model locally outweighs choosing a cloud provider; It minimizes system's downtime, provides more flexibility over choosing the vector store and embedding model (discussed in the next chapter) and minimizes the cost, as embedding and storing large documents is a pricey service.

Large Language Models

Contrary to embedding models, LLMs are much more heavier on the hardware and can take long times to generate outputs. This can result in long wait times when an a need for information may be crucial, especially that the system is designed to facilitate access to information. Moreover, combining a question with its retrieved context, in addition to running multiple instances of LLMs in parallel would aggravate these problems.
One instance where a smaller variant of a Large Language Model (Phi-3-mini-4k-instruct) was implemented locally, it was able to generate an answer to a sim-

ple question (without being integrated in a RAG pipeline) after a long delay of approximately 15 minutes and failed in a RAG pipeline (took forever until cancelled). In addition to this, another reason behind implementing LLMs through cloud providers was the plethora and competitiveness of these platforms. Many providers (OpenAI and Mistral for example) provide APIs to access their up-to-date models while others provide multiple LLMs through a single API key. These cloud providers also come with free plans, even though with some limits, but that automatically renew each month or so.

		CONTEXT	MODEL QUALITY	PRICE	OUTPUT TOKENS/S	LATENCY	
API PROVIDER	MODEL	CONTEXT WINDOW	INDEX Normalized avg	BLENDED USD/1M Tokens	MEDIAN Tokens/s	MEDIAN First Chunk (s)	FURTHER ANALYSIS
OpenAI	GPT-4o	128k	100	$7.50	86.8	0.59	Model Providers
OpenAI	GPT-4 Turbo	128k	94	$15.00	27.2	0.68	Model Providers
Microsoft Azure	GPT-4 Turbo	128k	94	$15.00	29.8	0.55	Model Providers
OpenAI	GPT-4	8k	84	$37.50	24.7	0.87	Model Providers
Microsoft Azure	GPT-4	8k	84	$37.50	20.2	0.54	Model Providers
OpenAI	GPT-3.5 Turbo	16k	59	$0.75	67.2	0.46	Model Providers
Microsoft Azure	GPT-3.5 Turbo	16k	59	$0.75	62.5	0.32	Model Providers
OpenAI	GPT-3.5 Turbo Instruct	4k	60	$1.63	65.3	0.37	Model Providers
Microsoft Azure	GPT-3.5 Turbo Instruct	4k	60	$1.63	136.5	0.61	Model Providers
Google	Gemini 1.5 Pro	1m	95	$5.25	62.4	1.14	Model Providers
Google	Gemini 1.5 Flash	1m	84	$0.53	155.2	1.17	Model Providers
Google	Gemini 1.0 Pro	33k	62	$0.75	88.2	2.18	Model Providers
Fireworks AI	Gemma 7B	8k	45	$0.20	234.1	0.26	Model Providers
deepinfra	Gemma 7B	8k	45	$0.07	65.9	0.29	Model Providers
groq	Gemma 7B	8k	45	$0.07	1,039.4	0.87	Model Providers
together.ai	Gemma 7B	8k	45	$0.20	141.4	0.30	Model Providers
replicate	Llama 3 (70B)	8k	83	$1.18	46.8	1.66	Model Providers
aws	Llama 3 (70B)	8k	83	$2.86	49.3	0.48	Model Providers
OctoAI	Llama 3 (70B)	8k	83	$0.90	62.0	0.29	Model Providers
Lepton AI	Llama 3 (70B)	8k	83	$0.80			Model Providers
Microsoft Azure	Llama 3 (70B)	8k	83	$5.67	17.4	2.91	Model Providers
Fireworks AI	Llama 3 (70B)	8k	83	$0.90	112.1	0.26	Model Providers
deepinfra	Llama 3 (70B)	8k	83	$0.61	20.5	0.35	Model Providers
groq	Llama 3 (70B)	8k	83	$0.64	350.8	0.37	Model Providers
databricks	Llama 3 (70B)	8k	83	$1.50	69.9	0.53	Model Providers
perplexity	Llama 3 (70B)	8k	83	$1.00	46.9	0.33	Model Providers
together.ai	Llama 3 (70B)	8k	83	$0.90	125.8	0.67	Model Providers
replicate	Llama 3 (8B)	8k	64	$0.10	78.7	1.78	Model Providers
aws	Llama 3 (8B)	8k	64	$0.38	79.2	0.29	Model Providers
OctoAI	Llama 3 (8B)	8k	64	$0.15	132.1	0.21	Model Providers
Lepton AI	Llama 3 (8B)	8k	64	$0.07			Model Providers
Microsoft Azure	Llama 3 (8B)	8k	64	$0.55	76.5	0.91	Model Providers

Figure 3.1: API Providers and LLM Performance Leaderboard [ArtificialAnalysis/LLM-Performance-Leaderboard space on Hugging Face]

.

3.1.2 Entreprise Knowledge Base Augmentation

The main issue with Large Language Models is that they are confined to the data they were trained on. This means they do not have access to information about recent events or private EKB, thus the need to implement methods to allow the continuous upload of documents into the vector store.
In our case, there is no need to collect data as this project does not involve training or fine-tuning models, but rather providing methods that allow scraping and uploading information from various sources.
For this purpose, an anticipatory study of the system's knowledge augmentation methods have resulted in identifying the following requirements :

- Offline files: Support for the different offline document formats (text files, pdfs, word documents, powerpoint presentations, e-book format/epub, markdown documentation)
- Web Scraping: A single web page from a URL, or recursively scraping its child pages
- Search Engine Results: By using some of Search Engine APIs, connect the user's query directly to search results and then scrape information for the returned URLs.
- AI-generated researches: LLMs can also automatize the traditional process of searching the web, reading webpages, and artificially generating a report based on those analyzed pages.

3.2 Project Management

After gaining a better understanding of the elements that would constitute the project it is now time to plan out expected tasks into an organized timetable. This section of the chapter addresses that. It is composed of two parts, one explaining the suitability of the employed methodology throughout the project, and another detailing the tasks to be undertaken.

3.2.1 Methodology

Given the nature of this project, the Scrum methodology was selected for the implementation of the system. This iterative approach allowed for the continuous development and refinement of tasks (Sprints) throughout this internship by allowing users' interference and feedback during the development process. This flexibility was particularly valuable in this end-of-studies project where the final requirements kept evolving.

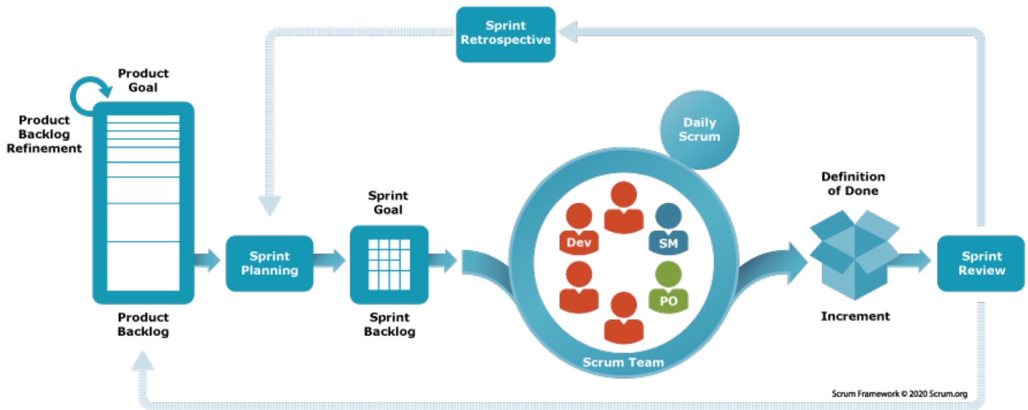

Figure 3.2: Scrum framework [What is Scrum (scrum.org)]

.

This methodology defines the steps to implement a project as follows :

1. Project Planning : Defining the project's overall objectives and values to be gained by users. And on a smaller scale, the specific Sprint objectives.

2. Product Backlog : Identifying and re-prioritizing the customer's requirements into an organized list of Sprints.

3. Sprints : Identifying and organizing tasks into a list of Sprints
4. Scrum Meetings : Daily short meetings to discuss progress and challenges to complete a Sprint's tasks.
5. Increment development : Iteratively complete Sprint cycles (planning, design, coding, testing and review) to complete increments.
6. Sprint Review : Review of the completed work with the entreprise supervisor
7. Sprint Retrospective : Discussing what went good and what went bad during the Sprint to gain better insight on what can be improved in the future.

The following section demonstrates the influence of this agile methodology, by showcasing a planned timetable in compliance with the iterative approach of the Scrum methodology.

3.2.2 Planning

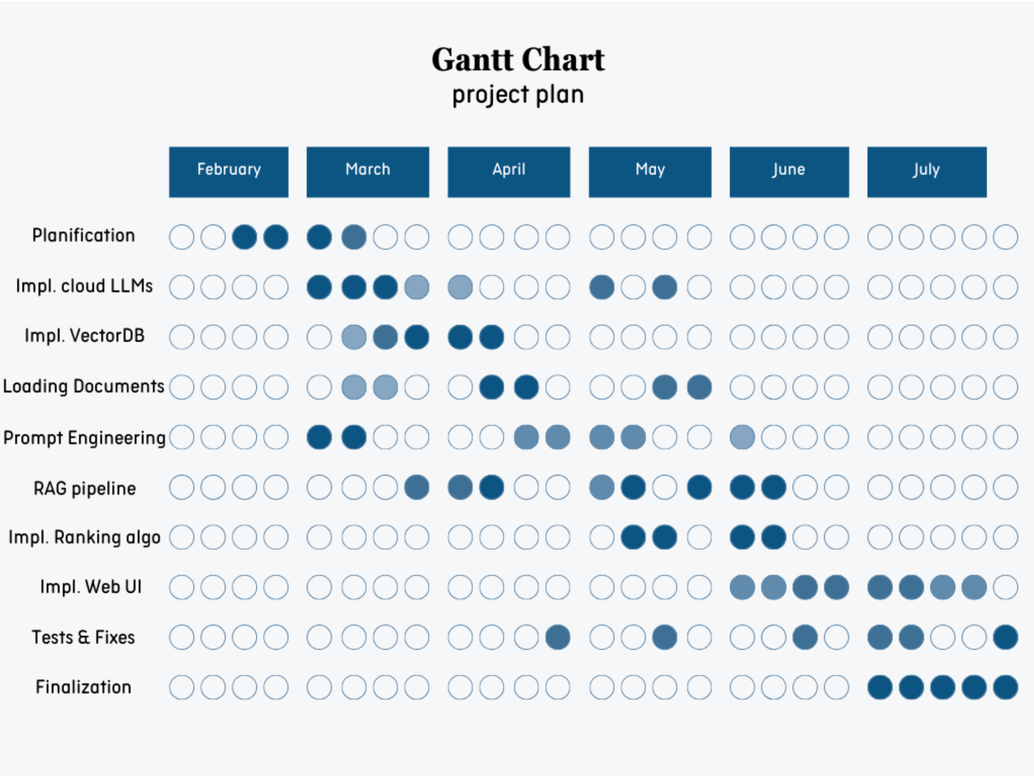

Figure 3.3: The planned Gantt Chart

This Gantt Chart illustrates the timetable of planned tasks and how they were split over smaller chunks to accomplish them and iterate the development process over different segments. This method allowed to accomplish Increments, which represent tangible steps towards the final project implementation: The first iteration (second week of March) was planned to implement Large Language Models standalone answering functionality (without RAG or a vector store), combining cloud-based LLMs with prompt engineering techniques. The next iteration focused on initiating the RAG pipeline by implementing a vector store index with static content, modifying Prompts to control the LLMs' factual grounding, after which some testing and further development was conducted to tailor to the different LLMs and make the vector store more dynamic by implementing data augmentation methods. After these two iterations which were completed by the start of May, further enhancement was undertaken to customize the indexing methods to tailor to different teams and the retrieval phase to minimize dissimilar results, in addition to supplying more data augmentation knowledge sources through APIs and the exploration of how a ranking algorithm could be implemented. The final iteration focused on integrating all the functionalities into a web interface, implementing a user feedback option and enhance answer ranking and validating the results with the corporate supervisor.

3.3 Conclusion

This chapter, along with the previous one, clarified the project's roadmap and helped breaking down the problem statement into a list of tasks to be implemented. These tasks are now explained in more detail in the following chapter, which discusses the implementation phase of this project.

Chapter 4

Implementation

This chapter details the implementation process of the solution discussed in the previous chapter: The work environment, tools and libraries employed to accomplish the tasks, the technical aspects of design choices execution, in addition to some of the major results accomplished and how to build upon them.

4.1 Workstation

This section identifies the characteristics, both hardware and software, of the computer system on which this project was implemented.

4.1.1 Hardware

- Device : DELL Inspiron 3593
- Processor : Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
- RAM : 24 GB
- GPU : NVIDIA GeForce MX230

4.1.2 OS and Software

- OS : Debian trixie inside a Windows Subsystem for Linux (WSL2) VM
- Source-code Editor : Visual Studio Code (with extensions to enable interacting with WSL, Jupyter Notebooks and Python virtual environments) + NeoVim
- Web Browser : Microsoft Edge (for web application testing and troubleshooting)
- Drivers and Toolkits : NVIDIA GPU driver + CUDA Toolkit among others

4.2 Components

In this section, we research available tools making it possible to develop the different functionalities, provide comparison between alternatives when choices were made, go through how they were implemented and end up with testing the effectiveness of the developed solution in addition to laying out its unfortunate limitations.

4.2.1 Libraries and Frameworks

Web interface and services

There is a plethora of frameworks of this kind, from customary JavaScript frameworks (React, Angular, Vue) to others designed to enable faster delivery of interactive web apps (such as Streamlit and Chainlit).
The choice was made to use Streamlit as it provides a much faster way to develop LLM chat interfaces than JS frameworks, while also being more advanced than Chainlit in terms of flexibility and building custom interfaces by supplying developers with many customizable and ready-to-use interface components like chat and message containers.

Figure 4.1: Streamlit logo.

We can look through its app gallery to find an abundance of templates that provide many examples of built apps which interact with LLMs, LangChain and other frameworks.

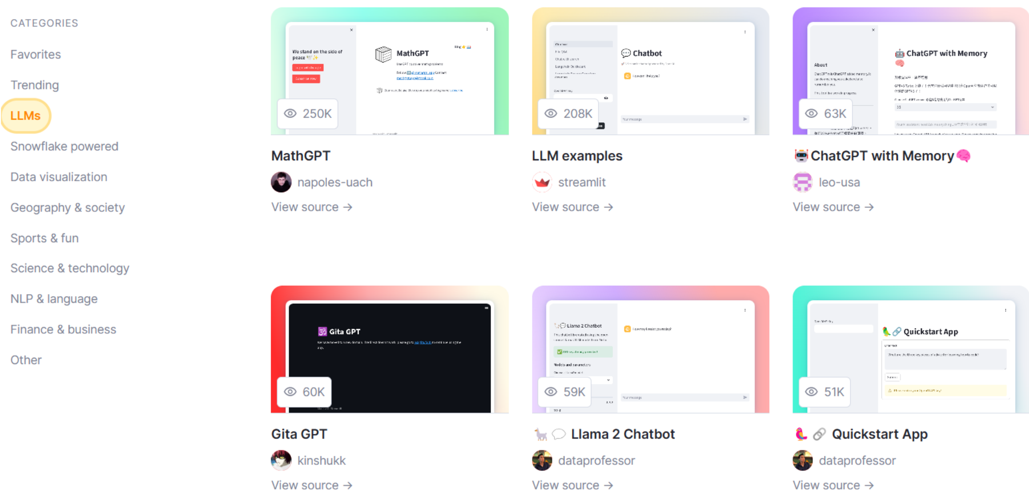

Figure 4.2: Streamlit App Gallery.

RAG pipelines development

There are a few frameworks enabling developers to interact with LLMs and RAG pipelines, each of which provide different integrations with external APIs and tools: LangChain, LlamaIndex, Haystack, Langroid.
The choice was made by 3S project coordinators to use LangChain for their solution. It is a great choice given that this platform provides all the tools to build complex RAG pipelines and personalize the different steps of these pipelines. Also, since its emergence in October 2022, this library has gained remarkable prominence with courses available on DeepLearning.ai, tutorials from NVIDIA, OpenAI, Google and others in addition to the exhaustive documentation available online.

Figure 4.3: LangChain logo

This framework has all the required components to build the most advanced RAG pipelines: Data loading from various sources, LLM and vector store integrations from different providers (both locally and on the cloud), prompt templates for different LLMs and tasks, etc...

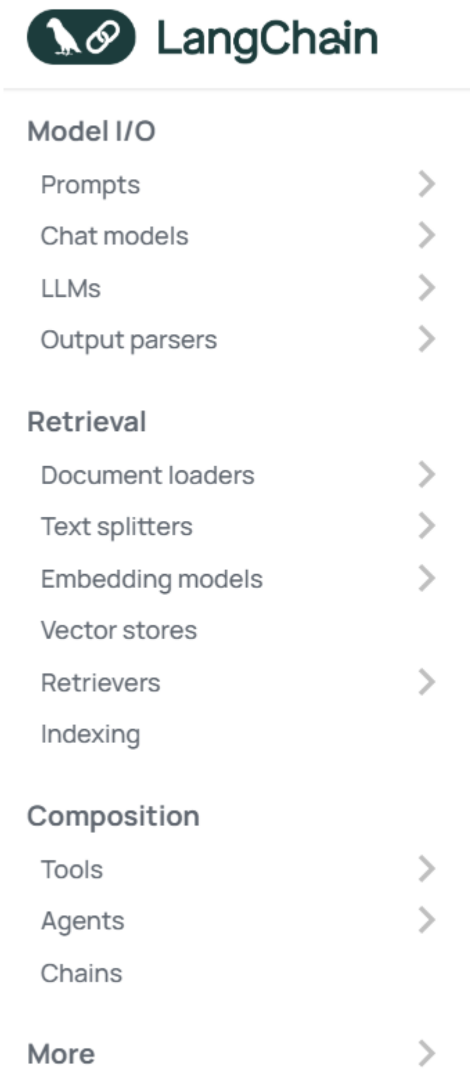

Figure 4.4: LangChain Components. (LangChain Documentation)

4.2.2 LLMs

There is a plethora of models of this type, both closed and open source, with many ways to access and use them.

Locally through Hugging Face Hub

Many open-source LLM variants are available on Hugging Face Models Hub, as it is the main developer of the 'transformers' Python library.

Figure 4.5: Hugging Face logo. [Hugging Face models Hub]

> "The Hugging Face Hub hosts many models for a variety of machine learning tasks. Models are stored in repositories, so they benefit from all the features possessed by every repo on the Hugging Face Hub. Additionally, model repos have attributes that make exploring and using models as easy as possible." ([Hugging Face Models Hub documentation], 2024)

We can find an abundance of pre-trained LLMs downloadable from the HF Hub.

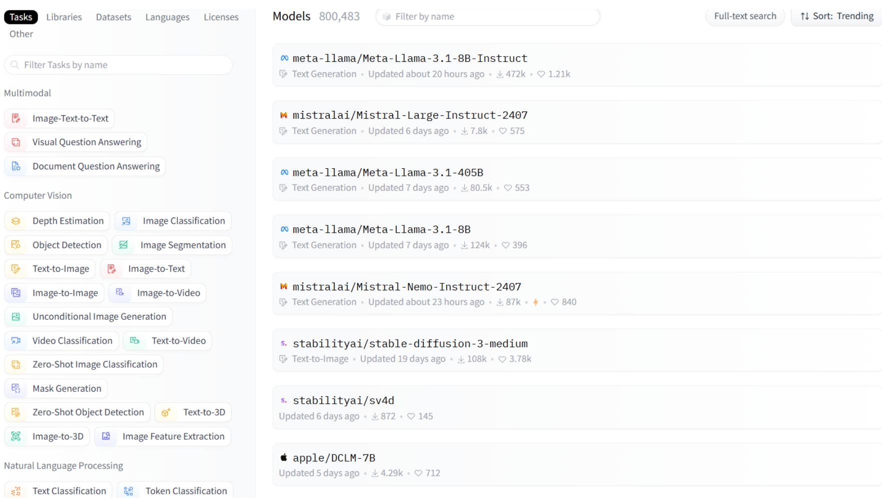

Figure 4.6: A short list of popular models hosted on HF Hub, which can be explored by their categories/tasks. [Models - Hugging Face]

This method allows the free utilization of a pre-trained model without being confined to cloud-platforms and their plans. As discussed in the previous chapter, this method will be used for the embedding model.

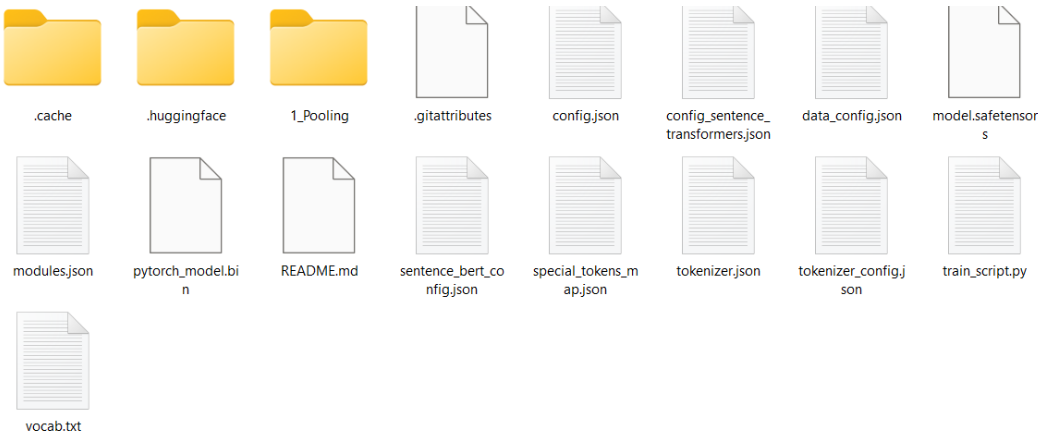

Figure 4.7: The contents of a model repository downloaded locally (all-mpnet-base-v2) on Hugging Face Hub

.

API-enabled services and Cloud-based solutions

Many companies behind Large Language Models development provide APIs or cloud-based environments to access their models. Some of the most popular options include OpenAI API, Google Cloud Vertex AI, Anthropic, Cohere, FireworksAI, MistralAI, TogetherAI, GroqCloud among others.
This method, in contrast to running models locally, does not come with the prerequisites of expensive hardware or delayed answer generation. Even though these solutions are paid services, most provide free trials and some of the available free plans only has a limit on daily and monthly usage, and is usually enough for personal usage. In addition to this, some of these cloud environments provide many models to use. For instance, FireworksAI allows to use Llama-3, Yi-Large, Mixtral, while TogetherAI provides models such as Qwen-2, Gemma (open-source version of Google's model, Gemini), Phi-2, Nous Capybara, and many others, all from within a single platform.

Model	Invoke	Async invoke	Stream	Async stream	Batch	Async batch
Anthropic	✅	✅	✅	✅	❌	❌
CTransformers	✅	✅	❌	❌	❌	❌
Cohere	✅	✅	❌	❌	❌	❌
Fireworks	✅	✅	✅	✅	✅	✅
HuggingFacePipeline	✅	❌	❌	❌	✅	❌
OpenAI	✅	✅	✅	✅	✅	✅
Together	✅	✅	❌	❌	❌	❌
VertexAI	✅	✅	✅	❌	✅	✅

Figure 4.8: LangChain Integrations with LLMs cloud providers. [Langchain Documentation - LLMs]

The previous list presents a subset of the integrations provided by Langchain. In addition to the aforementioned items, MistralAI and GrokCloud were integrated in the system to provide a plethora of Large Language Models suitable for Retrieval-augmented Generation. This solution also allows for some customization options through their web interfaces and API function parameters, even though limited when compared to local implementation.

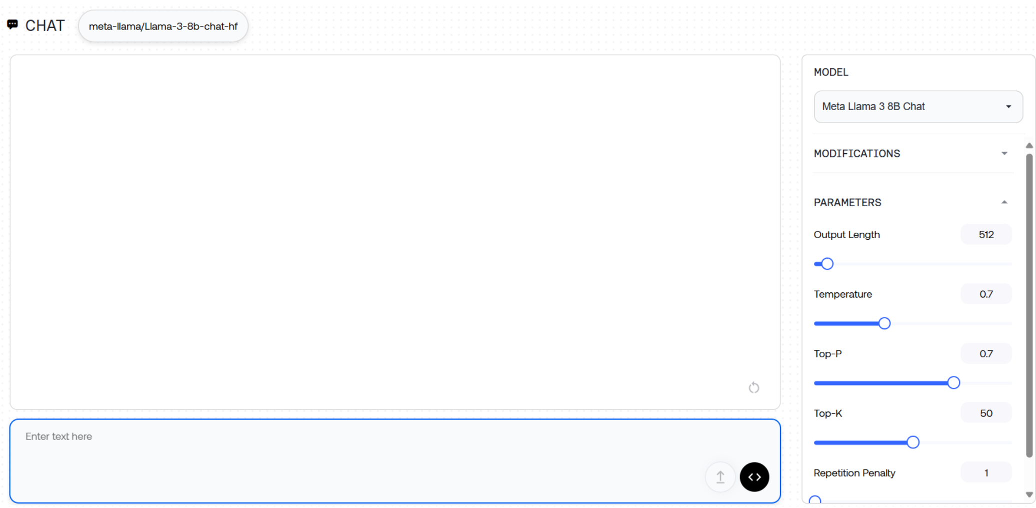

Figure 4.9: TogetherAI playground web interface, an example of a cloud-based environment allowing multiple LLM's customization options and parameters. TogetherAI Playground service

4.2.3 Vector Stores

There are many vector stores and embedding models to choose from which we will look through their differences.
The choice of the most suitable vector store solution was based on four criteria mainly:

- Self-hosting: This means that the vector store will be managed locally on the same computing infrastructure as the web server. This is opposed to a cloud-based deployment, which comes with the drawbacks of higher latency, possible network errors and high-costs.

- Latency: This refers to the performance and speed of similarity searching algorithms which are provided by the vector store and its ability to index and handle large volumes of data with the utilization of GPU parallel computing features.

- Accuracy: The relevance of retrieved data to the actual searched query. Often, this has a reverse relation with latency, as more accurate results take longer to be achieved.

- Documentation: Online Documentation and community forums that can guide on how to use the database efficiently.

Vector Store	Self-hosting	Latency	Accuracy	Documentation
FAISS	✔	✔	✔	✘
Pinceone	✘	✔	✘	✔
Chroma	✔ / ✘	✔	✘	✔
Lance	✔	✘	✘	✘

Table 4.1: Comparison of popular vector databases

After careful consideration, FAISS vector store was selected due to its high performance and accuracy in comparison to alternatives. It leverages the GPU-enabled CUDA toolkit, and provides a state-of-the-art implementation of similarity searching algorithms based on this structure.

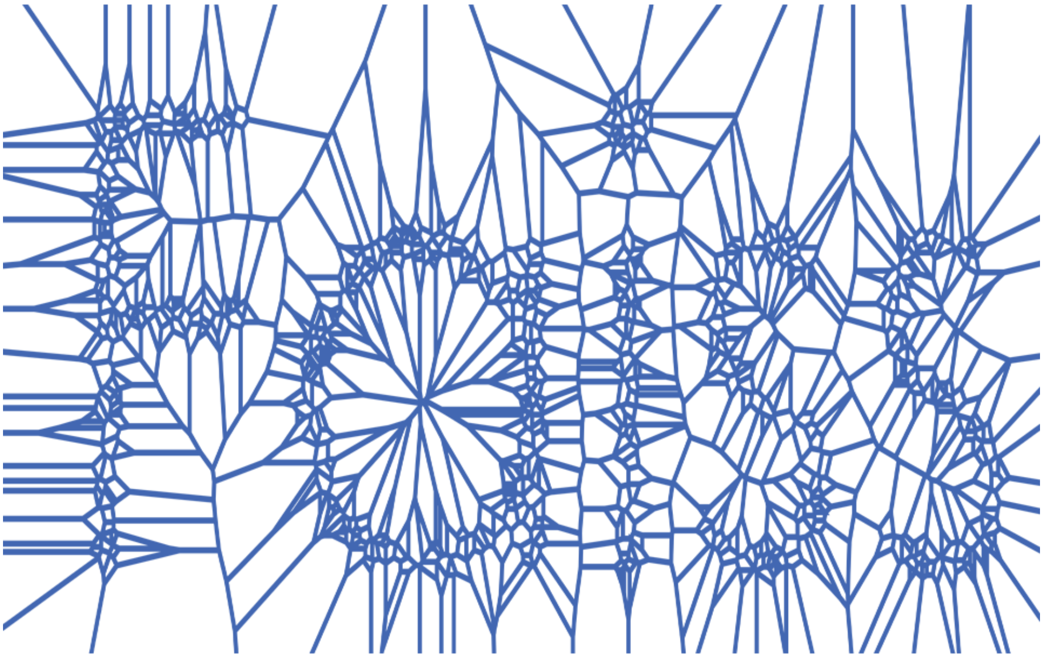

Figure 4.10: FAISS logo.

FAISS is a FOSS library, it stands for "Facebook AI Similarity Search" with an implementation of nearest-neighbor search and k-selection algorithms designed specifically to efficiently handle large data sets, 8.5x faster than previous methods.

4.2.4 Embedding Model

We have the choice to select the most suitable embeddings model.
We have highlighted in the second chapter the importance of an embedding algorithm that minimize the loss of semantics when converting textual data to embedding vectors. In addition to this, considering that a vector store once initiated with an embedding model, can no longer swap it with another (unless re-initialized from zero). For this purpose, it would be a bad choice to consider cloud solutions as these may be unavailable in some cases (network or provider failure, expiration of tokens...).
The best choice in this case, as in vector store's choice, is to select a suitable model which is available offline (as in self-hosting). This will avoid foreseeable failures and limit problems.
This redirects us to the Hugging Face Hub where we can find many embedding models, which are available through the sentence transformers-repository.

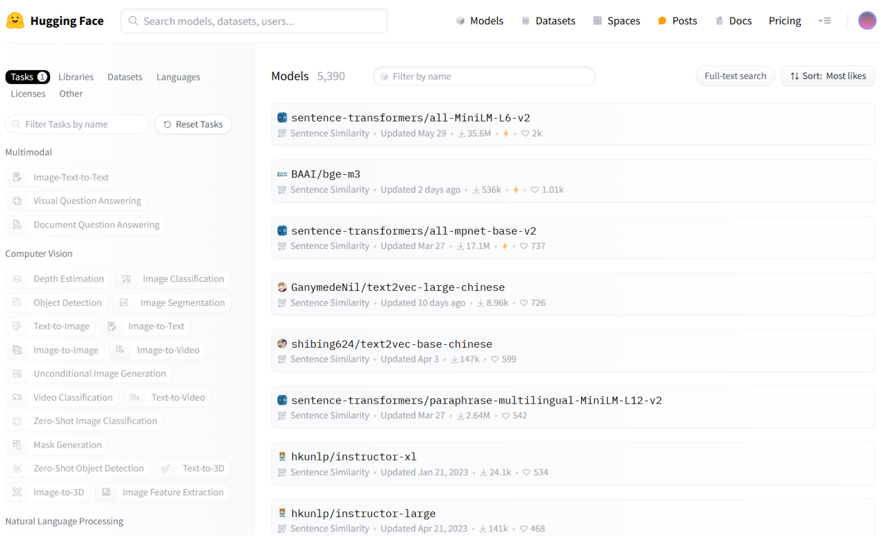

Figure 4.11: A short list of popular embedding models available on HF Hub. HF models filtered by 'sentence-similarity' task

Two of the most popular choices are "all-MiniLM-L12-v2" and "all-mpnet-base-v2", the first of which makes a better choice when working in a limited environment while not paying much attention to retaining semantics (faster embedding generation and smaller size), while the second one, "all-mpnet-base-v2", is the more suitable choice for our case due to, even with its larger footprint, its capability to capture most of the semantics and meanings of sentences.

Model Name	Performance Sentence Embeddings (14 Datasets)	Performance Semantic Search (6 Datasets)	Avg. Performance	Speed	Model Size
all-mpnet-base-v2	69.57	57.02	63.30	2800	420 MB
all-distilroberta-v1	68.73	50.94	59.84	4000	290 MB
all-MiniLM-L12-v2	68.70	50.82	59.76	7500	120 MB
all-MiniLM-L6-v2	68.06	49.54	58.80	14200	80 MB
multi-qa-mpnet-base-dot-v1	66.76	57.60	62.18	2800	420 MB
multi-qa-distilbert-cos-v1	65.98	52.83	59.41	4000	250 MB
paraphrase-multilingual-mpnet-base-v2	65.83	41.68	53.75	2500	970 MB
paraphrase-albert-small-v2	64.46	40.04	52.25	5000	43 MB
multi-qa-MiniLM-L6-cos-v1	64.33	51.83	58.08	14200	80 MB
paraphrase-multilingual-MiniLM-L12-v2	64.25	39.19	51.72	7500	420 MB
paraphrase-MiniLM-L3-v2	62.29	39.19	50.74	19000	61 MB
distiluse-base-multilingual-cased-v1	61.30	29.87	45.59	4000	480 MB
distiluse-base-multilingual-cased-v2	60.18	27.35	43.77	4000	480 MB

Figure 4.12: Sentence-Tansformers model performance comparison [Pre-Trained Sentence Transformers models' performance]

> ”The all-* models were trained on all available training data (more than 1 billion training pairs) and are designed as general purpose models. The all-mpnet-base-v2 model provides the best quality, while all-MiniLM-L6-v2 is 5 times faster and still offers good quality.” ([Sentence Transformers documentation])

4.3 Incorporating into a functional RAG system

After introducing the frameworks and tools which would allow us to build a RAG system, it is necessary to discuss the implementation details of these elements and how they were incorporated together.
This section is dedicated to showcasing how these components interact together in the implemented solution.

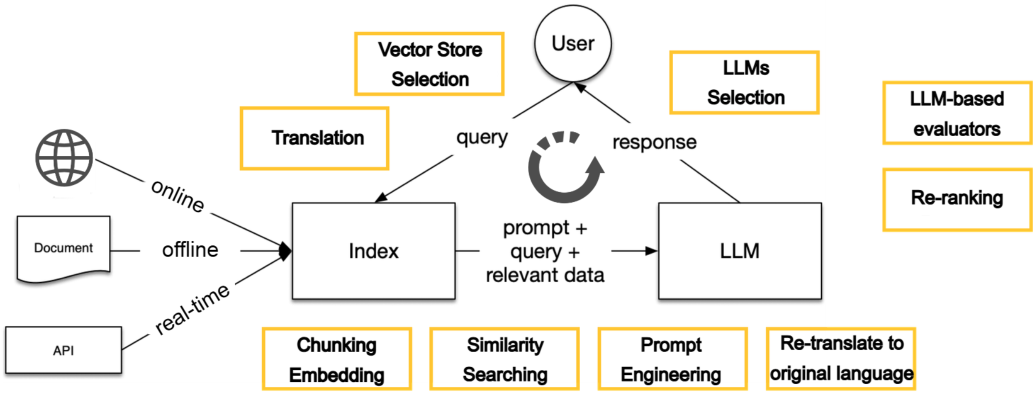

Figure 4.13: The overall pipeline of the system to be implemented, showcasing the interaction between the different components, from knowledge augmentation methods (document/data loading and chunking etc...), vector database (index), RAG pipeline (User-Index-LLM interaction), re-ranking and evaluation processes

This figure demonstrates the overall process that are executed in the system:

- Loading data from online web content, offline documents and files, or real-time data from APIs (Search engines and GPT Researcher) after which the chunking and embedding transformation occur.
- User preference selections (vector stores, LLMs, enable translation or not)
- Typical RAG processes: retrieval (similarity searching), prompt engineering and transformation, and finally generation
- Reiterate the RAG pipeline for every LLM selection (automatically retranslate the answer to the user's language if needed).
- Re-Ranking of answers based on RAG metrics and user feedback.

4.3.1 Data Ingestion Methods

The first essential part of the project is to allow the on-demand ingestion of data and new information into the vector store.
For this purpose, various file formats and online web scraping and fetching methods have been implemented; LangChain, as seen in the previous section, provides a ”Document loaders” section in its ”Retrieval” toolbox. These tools provide many tutorials and helpful functions to implement document loading and chunking from various sources, which has allowed to accomplish the required methods to satisfy the coordinating teams at 3S.

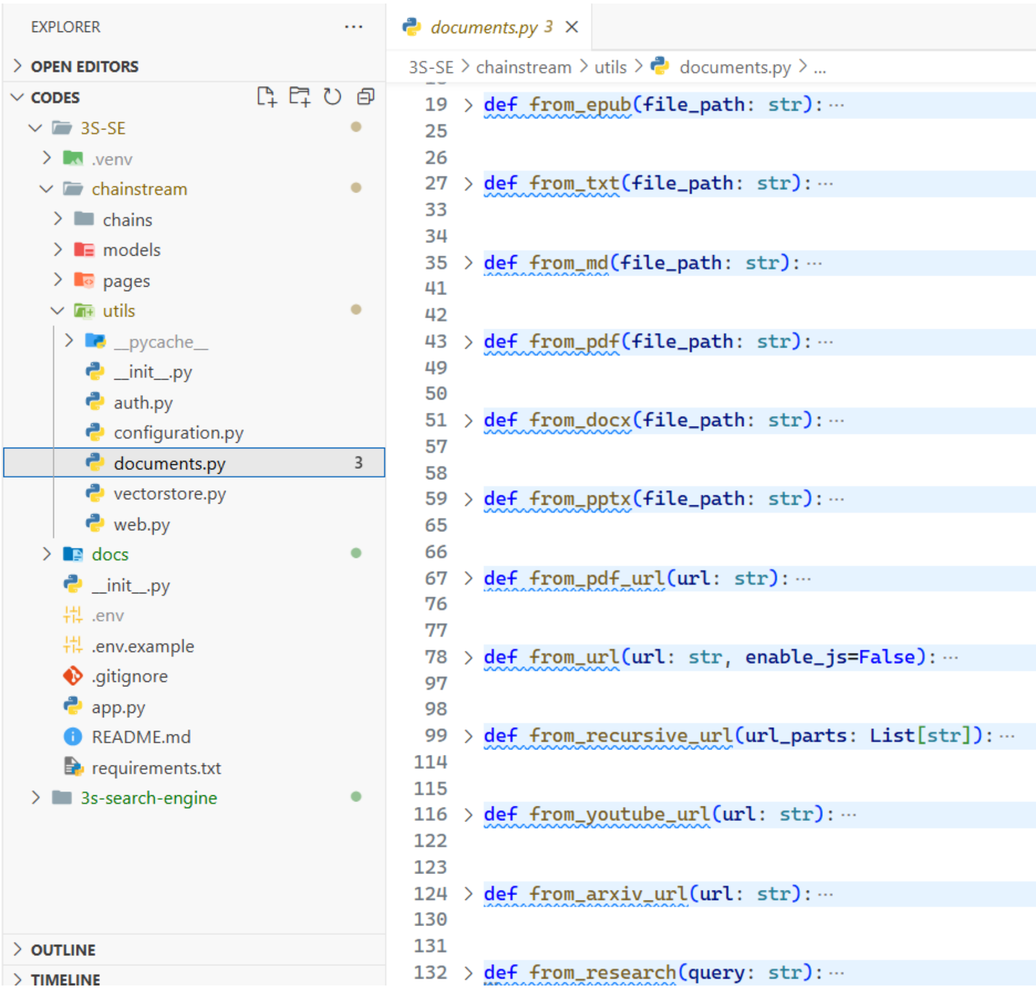

Figure 4.14: Various data ingestion methods

To make it more comprehensible, here is a short description of what each of these methods do:

- Web Pages (from_url, from_recursive_url): Allowing to read a single page from a given URL, or a page and its child pages recursively, supporting static and dynamic web content.
- Arxiv Research Papers (from_arxiv_url): Allows to read content directly from an arxiv URL, conserving metadata like author, dates, etc...
- Youtube Content (from_youtube_url): Transcribes a video uploaded on YouTube and constructs a document from a given URL.
- Generative Research (from_research): Utilizes GPT Researcher, a tool that allows to, given a question, generate multiple queries to send to search engines and then generating an artificial report.
- EPUB files (from_epub): This is an ebook format suitable for storing and distributing large volumes of information.
- Text files (from_txt): A file format for storing texts and notes.
- Markdown files (from_md): An easy to use syntax to write documentation and notes
- PDF files (from_pdf): Allowing to read PDF files.
- Same for other file formats...

Toolkits: NLTK, BeautifulSoup, Playwright, Unstructured, PyMuPDF, Pandoc, GPT Researcher, Tavily, LLMs

4.3.2 Retrieval

Vector Store

As outlined in the project objectives, the implementation of a vector database should allow different teams to upload their documents into separate vector databases. To achieve this, two Python classes have been implemented: one to hold the vector store functionalities: loading and saving to local storage in addition to listing available databases, and another to manage access credentials to vector store through a name and a passphrase.
A vector store can be initialized with ”vs_name” alone, which gives the read access to its contents. To modify it however, one should provide it with a passphrase (”vs_passphrase”), which would give users who have gained access the ability to load new content through the various data ingestion methods previously mentioned.

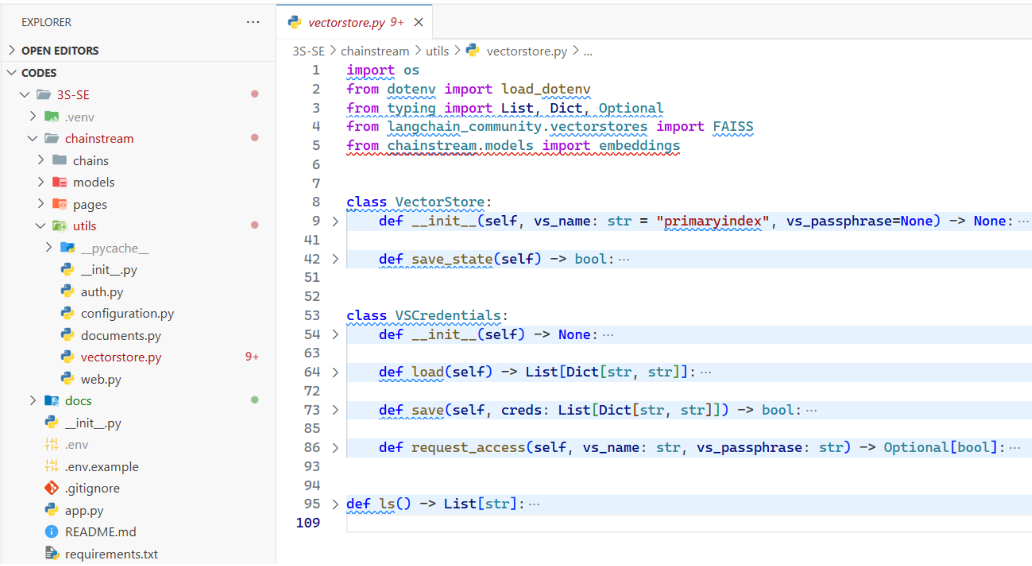

Figure 4.15: Vector Store implementation classes and methods

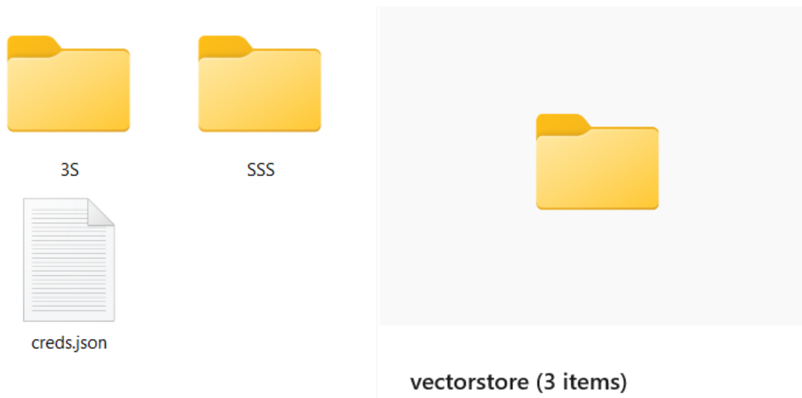

Figure 4.16: Local Directory for storing vector stores.

The ”creds.json” file store access credentials to the available vector stores, while other folders (”3S” and ”SSS” in this example) holds the vector store’s data.

Chunking and Embedding

A vector store initialization typically includes an embedding model initialization, which vectorizes the documents’ chunks. For our case, as we chose to select a model from Hugging Face Hub (”all-mpnet-base-v2”), we need to ensure that the model is downloaded and stays up-to-date.

chainstream > models > utils.py > ...

```python
import os
from huggingface_hub import snapshot_download
from dotenv import load_dotenv


load_dotenv()

def localize(model_name):
    try:
        model_path = os.environ.get(
            "PROJECT_DATA_DIR",
            os.path.expanduser("~/Downloads/3S-SE-AI/")
        ) + model_name
        snapshot_download(
            repo_id=model_name,
            local_dir=model_path
        )
        return model_path
    except BaseException as e:
        print(e)
```

(a) Downloading

3S-SE > chainstream > models > embeddings.py

```python
def load(model_name="sentence-transformers/all-mpnet-base-v2",
prefer_cuda=True):
    load_dotenv()
    # freeze_support()
    model_path = localize(os.environ.get("EMBEDDINGS_MODEL_NAME",
    model_name))
    try:
        return HuggingFaceEmbeddings(
            model_name=model_path,
            # multi_process=True,
            model_kwargs={
                "device": f"cuda:{cuda.current_device()}" if
                prefer_cuda and cuda.is_available() else "cpu"
            },
            encode_kwargs={"normalize_embeddings": True},
        )
    except cuda.OutOfMemoryError as e:
        return HuggingFaceEmbeddings(
            model_name=model_path,
            # multi_process=True,
            model_kwargs={
                "device": "cpu",
            },
            encode_kwargs={"normalize_embeddings": True},
        )
```

(b) Loading

Figure 4.17: Embedding Model Implementation

The ”snapshot_download” function imported from ”huggingface_hub” ensures that the latest version of the model is available locally and can be loaded, while the implemented ‘load‘ function leverages it by loading the model on the GPU and falls back to the CPU when needed.

Retrieval and Similarity Search

In some cases where the knowledge base does not contain relevant information to the user query, naive RAG would retrieve the passages with the highest similarity score, even if its contents are not pertinent to the question. This issue is addressed by only including documents which scores attain a certain threshold, eliminating unnecessary passages thus reducing context size when prompting LLMs, and providing trustworthiness when no content can be found so the LLMs inform the user rather than introducing hallucination in generated answers.

```
chain = RunnableParallel(
    {
        "context": vectorstore.as_retriever(
            search_type="similarity_score_threshold",
            search_kwargs={
                "k": 5,
                "score_threshold": 0.2,
            },
        ),
        "question": RunnablePassthrough(),
    }
```

Figure 4.18: Similarity Search Tuning

The score_threshold parameter allows to set a value which only passage that has a higher score are passed to the LLM's context, eliminating irrelevant information and setting a limit on the maximum number of passages (k parameter set to 5). The choice of these values was based on experimentation and testing with different values, where some relevant information could be dismissed if the threshold was set to a higher value, and the risk of irrelevancy was significant if set to lower value.

4.3.3 LLMs and Prompts

As new products and cloud platforms are constantly emerging and changing rapidly, the models and tools (web searching APIs), which allow the LLMs to connect to external web searching APIs, were implemented in an extendible manner, where adding, removing or customizing LLMs and their behavior is an easy change in code. The following code snippet showcases the tool loading process when the environment is started.

```
chainstream > utils > configuration.py > load
def load(args={}):
    config = {
        "llms": {},
        "web_tools": {},
    }

    if "OPENAI_API_KEY" in args:
        try:
            config["llms"]["gpt"] = {
                "name": "GPT 3.5 Turbo",
                "model": ChatOpenAI(
                    model="gpt-3.5-turbo", openai_api_key=args["OPENAI_API_KEY"]
                ),
            }
        except BaseException as e:
            print(e)
    if "ANTHROPIC_API_KEY" in args:
        try:
            config["llms"]["claude"] = {
                "name": "Claude 3 Sonnet",
                "model": ChatAnthropic(
                    model="claude-3-sonnet-20240229",
                    anthropic_api_key=args["ANTHROPIC_API_KEY"],
                ),
            }
        except BaseException as e:
            print(e)
    if "GOOGLE_API_KEY" in args:
        try:
            config["llms"]["gemini"] = {
                "name": "Gemini Pro",
                "model": ChatVertexAI(
                    model="gemini-pro", google_api_key=args["GOOGLE_API_KEY"]
```

Figure 4.19: Implementation of LLMs and Prompts, focusing on extensibility

The lists of Large Language Models ("llms" key in the "config" variable) and tools ("web_tools") grow dynamically when provided with valid API keys.

The list is much longer than this illustration, still it is easy to figure out how to add other LLMs and APIs as needed.
In addition to these APIs, this loading mechanism allows for flexible Prompt modification when different models require different prompts.

```
config["llms"]["llama"] = {
    "name": "LlaMa 3 70B Instruct",
    "model": ChatFireworks(
        model="accounts/fireworks/models/llama-v3-70b-instruct",
        fireworks_api_key=args["FIREWORKS_API_KEY"],
    ),
    "prompt": hub.pull("rlm/rag-prompt-llama3"),
}
```

Figure 4.20: Incorporating prompts into model definition.

This approach allows to define a custom prompt template suitable for a specific model (such as in our case of Llama 3) from within a unified source code file. In theory, the ”prompt” field can hold any f-string format (a concept in Python used to interpolate variables dynamically into a string), but the LangChain Hub provides many example prompts for different tasks.

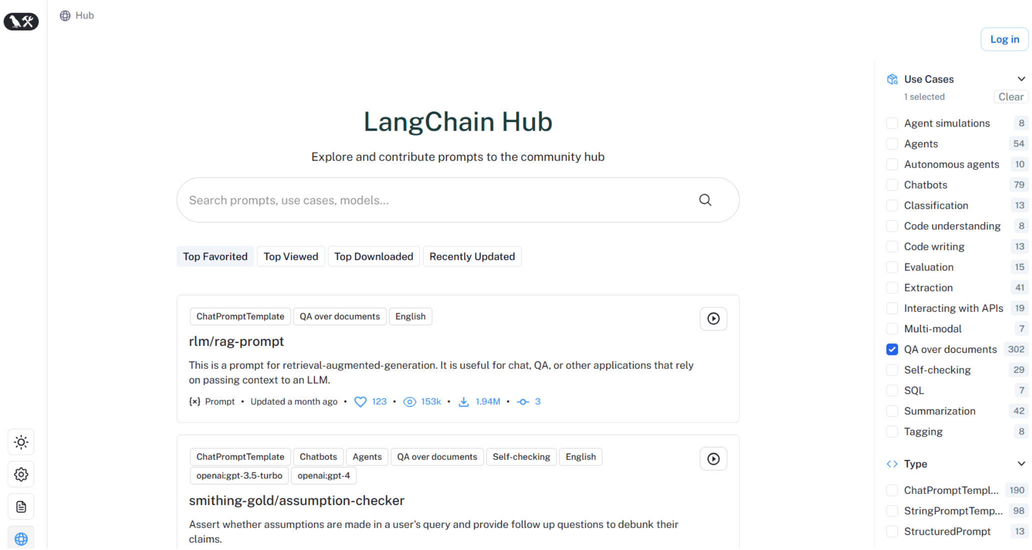

Figure 4.21: LangChain Hub web interface.

This Hub allows to find ready-to-use prompt templates suitable for various LLMs for Retrieval-augmented generation purposes.

The resulting list of implemented LLMs is as follows.

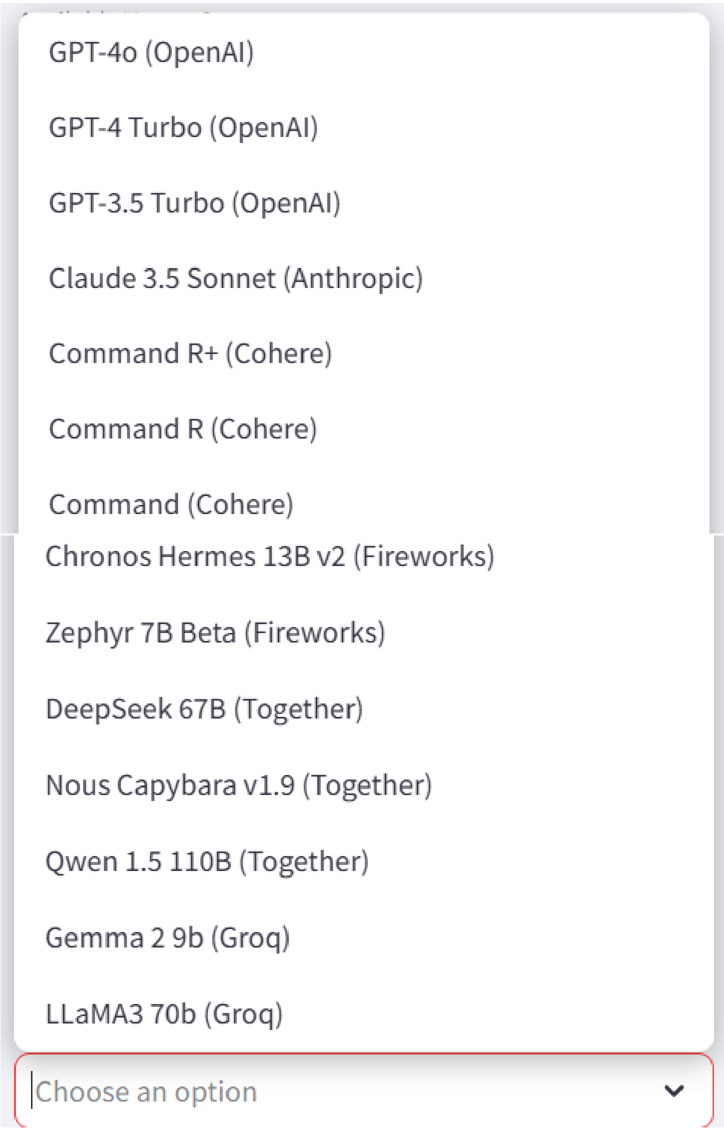

Figure 4.22: List of implemented LLMs

.

4.3.4 Evaluation and Ranking

As multiple answer generations occur in parallel, it is essential to provide the better quality responses first. For this purpose, RAG-based metrics were implemented with the help of the Ragas library. It stands for Rag Assessment and provides various metrics to evaluate RAG pipeline performance.

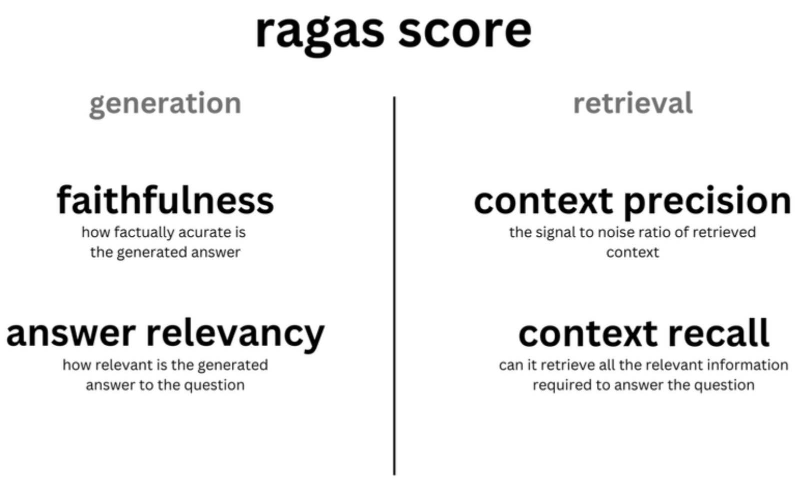

Figure 4.23: RAG pipeline evaluation metrics and description

When combined with the similarity score returned by the vector store when retrieving documents, this library allowed to assess the retrieval phase context relevancy and generation phase consistency against the given context and the prompt. In addition to these metrics, a user-driven feedback was implemented and integrated into the ranking algorithm to assess the overall satisfaction over a specific Large Language Model.

```python
def reranked(answers: List[Dict], embeddings=None, config=None) -> List[Dict]:
    if embeddings is not None and config is not None:
        scores = [
            calc_score(
                question=answer["answer"]["question"],
                answer=answer["answer"]["answer"],
                contexts=[doc.page_content for doc in answer["answer"]["context"]],
                embeddings=embeddings,
                llm=config["llms"][answer["llm_key"]]["model"],
            )
            * calc_score_from_feedback(answer["llm_key"])
            for answer in answers
        ]
    else:
        scores = [calc_score_from_feedback(answer["llm_key"]) for answer in answers]
    return sorted(answers, key=lambda obj: scores[answers.index(obj)], reverse=False)
```

Figure 4.24: Evaluation metrics for the ranking algorithm

This function sorts the generated responses based on the evaluation scores it obtained. It calculates the score based on consistency of the generated answer against the given feedback and its relevancy against the question it received, in addition to the similarity score and the feedback given by users. The formula to calculate the score is as follows : score = feedback score * similarity score * faithfulness score * answer relevance score

4.4 Testing and Validation

4.4.1 Response without Retrieval-augmented Generation

```python
from langchain_groq import ChatGroq

chat = ChatGroq(temperature=0, model_name="gemma2-9b-it")
```
Python

```python
from langchain_core.prompts import ChatPromptTemplate

system = "You are a helpful assistant."
human = "{text}"
prompt = ChatPromptTemplate.from_messages([("system", system), ("human", human)])

chain = prompt | chat
for chunk in chain.stream({"text": "what are the different types of retrieval augmentation processes"}):
    print(chunk.content, end="", flush=True)
```
Python

Figure 4.25: Implementation of a Large Language Model (Gemma 2) without RAG functionality

The execution of the previous code resulted in the following output.

```
Retrieval augmentation is a powerful technique used in information retrieval to improve the accuracy and relevance of search results. It involves enriching the original query with additional information to guide the retrieval process more effectively.

Here are some common types of retrieval augmentation processes:

**1. Query Expansion:**

* **Synonym Replacement:** Replacing words in the query with their synonyms to capture different meanings and related concepts.
* **Word Sense Disambiguation:** Identifying the intended meaning of ambiguous words in the query and using the appropriate sense for retrieval.
* **Concept Expansion:** Adding related concepts or terms to the query based on semantic relationships, ontologies, or knowledge graphs.
* **Negation Handling:** Identifying and incorporating negation terms (e.g., "not," "without") to refine the search scope.

**2. Document Augmentation:**

* **Passage Retrieval:** Instead of retrieving entire documents, focus on finding relevant passages within documents that directly address the query.
* **Document Summarization:** Generating concise summaries of documents to provide a quick overview and highlight key information.
* **Document Clustering:** Grouping similar documents together to identify topical clusters and retrieve more focused results.
```

Figure 4.26: A Large Language model response without RAG

The example demonstrates the inconvenience of using an LLM when looking for a short and accurate response to a given query ("what are the different types of retrieval augmentation processes"). The generated answer is very long and clearly exhibits hallucination, as it is not accurate enough.
The next section is about how this inconvenience can be effectively rectified and changed to an acceptable behavior.

4.4.2 Retrieval-augmented Generation results

Data: Question and Answer

We will need to load some data into the knowledge base. This information represents the ground truth on which we desire LLMs to base their answers.
For this purpose, a PDF file from a research parper on Arxiv will be uploaded as target data. Page 11 of this document contain the answer to our previous question.

11

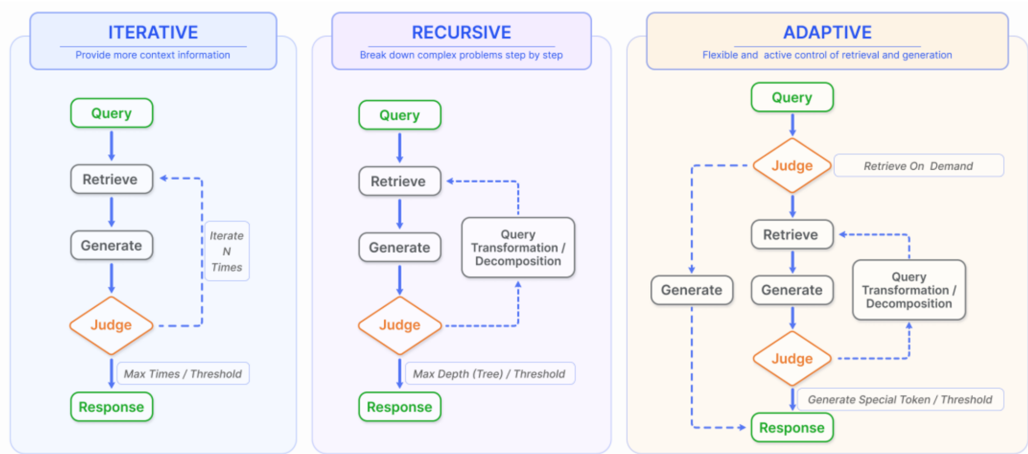

Fig. 5. In addition to the most common once retrieval, RAG also includes three types of retrieval augmentation processes. (left) Iterative retrieval involves alternating between retrieval and generation, allowing for richer and more targeted context from the knowledge base at each step. (Middle) Recursive retrieval involves gradually refining the user query and breaking down the problem into sub-problems, then continuously solving complex problems through retrieval and generation. (Right) Adaptive retrieval focuses on enabling the RAG system to autonomously determine whether external knowledge retrieval is necessary and when to stop retrieval and generation, often utilizing LLM-generated special tokens for control.

Figure 4.27: The content of the document containing the desired answer to the question ("what are the different types of retrieval augmentation processes")

This page from the PDF file contain clear answer to our question (3 types of retrieval augmentation processes: iterative, recursive, adaptive)

After having established a query to be asked to the model and its corresponding desired answer, we will now implement a simple RAG pipeline with the same previous model (Gemma 2) to see the difference in generated answers.

Knowledge Base initialization

The first step is to load the PDF file into the knowledge base. For this purpose, we need to initialize a FAISS index with an embedding model (all-mpnet-base-v2), which will convert the information read from the file through PyMyPDFLoader to a numerical representation suitable for similarity search afterwards.

```python
from langchain_community.vectorstores import FAISS
from langchain_huggingface import HuggingFaceEmbeddings
from langchain_community.document_loaders import PyMuPDFLoader

db = FAISS.from_documents(
    PyMuPDFLoader("./2312.10997v5.pdf").load_and_split(),
    HuggingFaceEmbeddings(
        model_name="sentence-transformers/all-mpnet-base-v2"
    )
)
```

✓ 20.8s Python

```python
db.index.ntotal
```

✓ 0.0s Python

38

Figure 4.28: Vector Store initialization with data

This sample vector store initialization included reading and loading the required PDF file from local storage, resulting in 38 chunks, which simplifies the process of retrieving and passing the most relevant chunks to the large language model later on.

Similarity Search dry run

The following code imitates the retrieval phase in a complete RAG pipeline, providing an overview of the contents of the vector store and the results when some passages get retrieved.

```python
db.similarity_search_with_relevance_scores(
    "what are the different types of retrieval augmentation processes"
)
```

Python

```
[(Document(metadata={'source': './2312.10997v5.pdf', 'file_path': './2312.109
  0.37452614314189936),
 (Document(metadata={'source': './2312.10997v5.pdf', 'file_path': './2312.109
  0.2994688858244252),
 (Document(metadata={'source': './2312.10997v5.pdf', 'file_path': './2312.109
  0.28909966527268394),
 (Document(metadata={'source': './2312.10997v5.pdf', 'file_path': './2312.109
  0.2674217709158937)]
```

Figure 4.29: Retrieving relevant documents from the database

```
[(Document(metadata={'source': './2312.10997v5.pdf', 'file_path':
 (Document(metadata={'source': './2312.10997v5.pdf', 'file_path':
 (Document(metadata={'source': './2312.10997v5.pdf', 'file_path':
 './2312.10997v5.pdf', 'page': 10, 'total_pages': 21, 'format':
 'PDF 1.5', 'title': '', 'author': '', 'subject': '', 'keywords':
 '', 'creator': 'LaTeX with hyperref', 'producer': 'pdfTeX-1.40.
 25', 'creationDate': 'D:20240328005445Z', 'modDate':
 'D:20240328005445Z', 'trapped': ''}, page_content='11\nFig. 5.
 In addition to the most common once retrieval, RAG also includes
 three types of retrieval augmentation processes. (left)
 Iterative retrieval involves\nalternating between retrieval and
 generation, allowing for richer and more targeted context from
 the knowledge base at each step. (Middle) Recursive
 retrieval\ninvolves gradually refining the user query and
 breaking down the problem into sub-problems, then continuously
 solving complex problems through retrieval\nand generation.
 (Right) Adaptive retrieval focuses on enabling the RAG system to
 autonomously determine whether external knowledge retrieval is
 necessary\nand when to stop retrieval and generation, often
 utilizing LLM-generated special tokens for control.\nbase for
 LLMs. This approach has been shown to enhance\nthe robustness of
 subsequent answer generation by offering\nadditional contextual
 references through multiple retrieval\niterations. However, it
 may be affected by semantic discon-\ntinuity and the
```

Figure 4.30: Output from similarity search

The returned passages, as seen above, contain parsed text from the PDF file. As the vector store contain separate chunks, the result of running the previous code is a list containing document chunks, each with the textual content and some metadata that helps representing the whole documents through its chunks in a vector store. These chunks are sorted by their relevance score, which is calculated based on the cosine similarity measure between the query and the actual passage's embeddings vectors. The highlighted text is the desired data that we want to pass to the LLM afterwards. This passage is returned 3rd on the list, with a score of 0.28.

The results of retrieval, even though still optimizable, is satisfactory. Given a query, it is almost impossible to not retrieve the relevant passage outside the first 5 elements. This is because of the indexing strategy implemented in the vector store which allows the calculation of the similarity between every chunk and the query to ensure most similar passages are retrieved effectively.
These retrieved documents can now be integrated in a prompt and passed to the LLM to control the context of its generation process.

Response with Retrieval-augmented Generation

```python
from langchain_core.prompts import PromptTemplate

template = """Use the following pieces of context to answer the question at the end.
If you don't know the answer, just say that you don't know, don't try to make up an answer.
Use three sentences maximum and keep the answer as concise as possible.
Always say "thanks for asking!" at the end of the answer.

{context}

Question: {question}

Helpful Answer:"""
rag_prompt = PromptTemplate.from_template(template)

rag_chain = (
    {"context": db.as_retriever() | format_docs, "question": RunnablePassthrough()}
    | rag_prompt
    | chat
    | StrOutputParser()
)

rag_chain.invoke("what are the different types of retrieval augmentation processes")
```
Python

Figure 4.31: Initialization of a simple RAG pipeline for testing

The steps to initialize the RAG pipeline are: 1. set the prompt to support passing the context (retrieved documents) and question, along with instructing the model to only base its answer on the given context, 2. reconstructing the prompt by filling the variables (retrieving and formatting the context and the question), 3. passing the constructed prompt to the LLM (chat), 4. and finally transforming the LLM's response into a human readable format.

Executing the previous code resulted in the desired output.

```
'There are three main types of retrieval augmentation processes:
iterative retrieval, recursive retrieval, and adaptive
retrieval.  Thanks for asking! \n\n\n'
```

Figure 4.32: The generated answer from a RAG pipeline

The response demonstrates how effective was the pipeline from transforming a long answer into a very accurate direct answer.

Chapter 5

Conclusion and Future Work

5.1 Summary of Findings

This project has resulted in the successful design and implementation of a RAG pipeline. By effectively combining the strengths of LLMs and vector stores, the tool demonstrates the potential to revolutionize how the organization manage and access their information assets.

The RAG pipeline architecture, incorporating data ingestion, preprocessing, embedding generation, vector store indexing, and query processing, has been meticulously designed to ensure optimal performance and accuracy. The employed LLMs, with their advanced language understanding and generation capabilities, have proven to be instrumental in extracting valuable insights from the data. The vector store, efficiently storing and retrieving embeddings, has significantly enhanced the search capabilities of the tool by retrieving relevant contexts.

5.2 Future Research Directions

While the project has achieved its primary objectives, there are opportunities for further exploration and improvement. These include employing local Large Language Models in order to make it possible to fine-tune them based on company data, or to employ Reinforcement Learning from Human Feedback techniques by allowing user interaction with generated responses, or to explore other retrieval techniques, like leveraging some type of LLMs (document-answering task models from Hugging Face Models Hub) which can give more accurate retrieval results. Additionally, a new approach of RAG, called GraphRAG, is currently being developed by Microsoft, which allows to build a graph knowledge base that make the LLMs more capable to extract and understand the relationships between entities from the knowledge base

In conclusion, the implemented RAG-based enterprise knowledge searching tool represents a significant step forward in harnessing the power of AI for information retrieval. It has the potential to streamline workflows, improve decision-making, and unlock new opportunities for the company.

References

ESPRIT SCHOOL OF ENGINEERING

www.esprit.tn - E-mail : contact@esprit.tn

Siége Social : 18 rue de l'Usine - Charguia II - 2035 - Tél. : +216 71 941 541 - Fax. : +216 71 941 889

Annexe : Z.I. Chotrana II - B.P. 160 - 2083 - Pôle Technologique - El Ghazala - Tél. : +216 70 685 685 - Fax. : +216 70 685 454