BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

APPLICATION OF TRANSFORMERS FOR TECHNICAL DOCUMENTATION ANALYSIS

VYUŽITÍ TRANSFORMÁTORŮ KE ZPRACOVÁNÍ TECHNICKÉ DOKUMENTACE

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **DÁVID BARTUŠ**
AUTOR PRÁCE

SUPERVISOR **doc. Ing. PETR MATOUŠEK, Ph.D., M.A.**
VEDOUCÍ PRÁCE

BRNO 2025

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

164738

Institut:	Department of Information Systems (DIFS)
Student:	**Bartuš Dávid**
Programme:	Information Technology
Title:	**Application of Transformers for Technical Documentation Analysis**
Category:	Artificial Intelligence
Academic year:	2024/25

Assignment:

1. Review the state-of-the-art of Transformer Neural Networks used in natural language processing. Examine existing approaches, their applications and performance.
2. Investigate the format of technical documents. Propose essential pre-processing steps for these documents. Investigate and develop word embeddings and vector databases.
3. Based on the consultant's recommendation use prompt engineering to deploy deep learning models based on Transformer Neural Networks for specific natural language processing tasks, such as Q&A tasks and text summarization.
4. Create a web-based interface to the model and demonstrate application of the model to selected use cases.
5. Evaluate your results. Discuss contribution of your work and the potential real-world deployment.

Literature:

- Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D., “ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators”, Art. no. arXiv:2003.10555, 2020.
- Rothman, Denis, and Antonio Gulli. Transformers for Natural Language Processing: Build, train, and fine-tune deep neural network architectures for NLP with Python, PyTorch, TensorFlow, BERT, and GPT-3. Packt Publishing Ltd, 2022.
- Lewis Tunstall, Leandro von Werra, Thomas Wolf: Natural Language Processing with Transformers, Revised Edition, O'Reilly Media, Inc., 2022.
- Scotti, V., Carman, M.J. (2024). LLM Support for Real-Time Technical Assistance. In: Bifet, A., *et al.* Machine Learning and Knowledge Discovery in Databases. Research Track and Demo Track. ECML PKDD 2024.LNCS, vol 14948. Springer, Cham.
- Julien Breton, Mokhtar Boumedyen Billami, Max Chevalier, Cassia Trojahn: Leveraging Semantic Model and LLM for Bootstrapping a Legal Entity Extraction: An Industrial Use Case, Studies on the Semantic Web, vol. 60, 20 - 36.

Requirements for the semestral defence:
Points 1-3.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Matoušek Petr, doc. Ing., Ph.D., M.A.**
Consultant:	Dr. Martin Holkovič, Flowmon Networks
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	1.11.2024
Submission deadline:	14.5.2025
Approval date:	22.10.2024

Abstract

The bachelor's thesis explores the integration of Transformer Neural Network models and Retrieval-augmented generation into an application designed for efficient interaction with technical documentation. Its primary function lets users ask specific questions to gather relevant information from technical documentation. The tool makes use of large language models, such as Gemini 2.5 and Llama 4. The thesis describes how the application was developed, including how the vector database, ideal models and RAG parameters were chosen. It also describes how the frontend was built and how the application was deployed. It also includes evaluation and comparison of different large language models, and their parameters focusing on their effectiveness in processing technical documents and answering questions related to them.

Abstrakt

Bakalárska práca skúma integráciu modelov Transformerov a Generovanie rozšírené o vyhľadávanie do nástroja určeného na efektívnu interakciu s technickou dokumentáciou. Jeho hlavná funkcia umožňuje používateľom klásť špecifické otázky a získať relevantné informácie z technickej dokumentácie. Nástroj využíva veľké jazykové modely, ako sú Gemini 2.5 a Llama 4. V práci je opísaný spôsob vývoja nástroja vrátane spôsobu výberu vektorovej databázy a ideálnych modelov a parametrov pre generovanie rozšírené o vyhľadávanie. Opisuje tiež, ako bol zostavený frontend a ako bol nástroj nasadený. Zahŕňa aj hodnotenie a porovnanie rôznych modelov so zameraním na ich účinnosť pri spracovaní technických dokumentov a zodpovedaní otázok s nimi súvisiacich.

Keywords

Large Language Models, Natural Language Processing, Retrieval-augmented generation, RAGAs, Technical Documentation

Klíčová slova

Veľké jazykové modely, Spracovanie prirodzeného jazyka, Generovanie rozšírené o vyhľadávanie, RAGAs, Technická dokumentácia

Reference

BARTUŠ, Dávid. *Application of Transformers for Technical Documentation Analysis.* Brno, 2025. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor doc. Ing. Petr Matoušek, Ph.D., M.A.

Rozšířený abstrakt

Táto bakalárska práca sa venuje využitiu modelov neurónových sietí Transformer a techniky Generovania Rozšíreného o Vyhľadávanie (RAG) na analýzu technickej dokumentácie. Hlavným cieľom práce bolo navrhnúť, implementovať a vyhodnotiť systém, ktorý umožňuje efektívnu interakciu používateľov s rozsiahlymi technickými dokumentmi, najmä prostredníctvom odpovedania na otázky a sumarizácie textu. Práca reaguje na potrebu rýchleho a presného prístupu k informáciám v technickej dokumentácii, kde manuálne vyhľadávanie môže byť časovo náročné a náchylné na chyby. Práca začína prehľadom súčasného stavu v oblasti spracovania prirodzeného jazyka (NLP), pričom sa zameriava na neurónové siete, konkrétne na architektúru Transformerov. Detailne popisuje kľúčové komponenty Transformerov, ako sú mechanizmy pozornosti (attention mechanisms), kódovače a dekodéry, ktoré umožňujú paralelné spracovanie dát a efektívne zachytávanie kontextových vzťahov v texte. Následne sa venuje veľkým jazykovým modelom (LLM), ktoré sú postavené na architektúre Transformerov a trénované na obrovskom množstve textových dát. Práca zdôrazňuje výhody LLM, ako napríklad schopnosť spracovať dlhé kontextové okná, čo je kľúčové pre analýzu rozsiahlej technickej dokumentácie. Kľúčovou technikou použitou v práci je Generovanie rozšírené o vyhľadávanie (RAG). Táto metóda umožňuje LLM prístup k špecifickým dátam používateľa, čím sa zvyšuje relevantnosť a presnosť generovaných odpovedí bez potreby pretrénovania celého modelu. Práca podrobne opisuje jednotlivé kroky RAG techniky. Dokumenty sú rozdelené na menšie, sémanticky koherentné časti. Skúmané sú rôzne metódy rozdelovania textu, na fixnú dĺžku, podľa semantiky alebo podľa štruktúry textu. Ďalej je popísané vytváranie vektorových reprezentácií dát z ich častí pomocou transfomrerov. Je popísaný aj spôsob vkladania vektorov a dát, ktoré k nim patria do databázy. Pri položení otázky sa jej vektorová reprezentácia porovná s vektormi v databáze a nájdu sa najrelevantnejšie časti dokumentov. Práca tiež skúma efekt zmeny poradia nájdených častí dokumentov, na generovanie odpovedí. Práca tiež opisuje ako LLM použije pôvodnú otázku a nájdený kontext na vygenerovanie odpovede, a porovnáva výkon rôznych modelov pri tejto činnosti. Ďalej práca opisuje dve dátové sady, ktoré používa na vyhodnotenie výsledkov. Hlavná dátová sada, poskytnutá spoločnosťou Flowmon Networks, obsahuje technickú dokumentáciu k systému Flowmon ADS vo formáte Markdown, ktorý zahŕňa rôzne štrukturálne prvky ako nadpisy, zoznamy, tabuľky a JSON bloky. Tento dataset pozostáva zo 107 súborov o veľkosti 382 KB. Pre porovnanie a testovanie robustnosti aplikácie sa využívajú aj RFC dokumenty (oficiálne internetové štandardy) v textovom formáte. Tento dataset obsahuje 37 súborov o veľkosti 408 KB. Ďalej je v práci popisovaný návrh a implementácia aplikácie. Vyvinutá webová aplikácia umožňuje používateľom nahrávať technickú dokumentáciu, vyberať modely LLM a klásť otázky alebo žiadať o sumarizáciu dokumentov. V implementácii sa pre komunikáciu s rozsiahlymi jazykovými modelmi (LLM) a získavanie odpovedí využívajú predovšetkým aplikačné programovacie rozhrania (API) poskytované spoločnosťou Google. Tieto rozhrania umožňujú efektívne dopytovanie modelov ako Gemini a prístup k ďalším službám, napríklad pre generovanie vektorových reprezentácií textu alebo pre triedenie nájdených častí kontextu pri RAG. Hodnotenie systému prebiehalo pomocou frameworku RAGAs a synteticky generovaných testovacích sád pre oba typy datasetov. Experimenty sa zamerali na porovnanie rôznych LLM, metód chunkingu, počtu nájdených kontextových častí a vplyvu zmeny poradia nájdených častí dokumentov.

Application of Transformers for Technical Documentation Analysis

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of doc. Ing. Petr Matoušek, Ph.D., M.A. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Dávid Bartuš
May 13, 2025

Acknowledgements

I would like to express my sincere gratitude to my supervisor, doc. Ing. Petr Matoušek, Ph.D., M.A., for his valuable and exceptionally professional guidance throughout my thesis. I would also like to thank Flowmon Networks for providing the dataset essential for this thesis.

Contents

Chapter 1

Introduction

For professionals in various technical fields, it is essential to access and understand important knowledge from technical documents quickly, given the speed with which projects have to be delivered, the huge size of some projects, and the large amount of documentation for them. Manual searching is a common approach used to do this, but it takes time and can cause errors. In addition, it can be difficult to locate accurate information effectively in technical documents due to their large size, complex nature, and the fact that they are often dispersed in various locations.

This thesis presents an application designed to accelerate and simplify the interaction of users with technical documentation. Its core functionality allows users to insert technical documents from projects in different formats, such as Markdown or plain text, and ask specific questions. To make finding information faster and more accurate, this application leverages the Retrieval—Augmented Generation (RAG) method combined with powerful transformer models. Large Language Models (LLMs) have significantly advanced Natural Language Processing (NLP), showing a remarkable ability to capture deeper contextual links within text. Their underlying structure allows them to grasp complex contextual details, which is particularly useful for interpreting technical documentation effectively.

This thesis is structured to logically guide the reader through the problem, proposed solution, and evaluation. Chapter 2 explains NLP, Neural Networks, Transformers, LLMs, and RAG, which are all important concepts to understand in order to understand how the application and tools it is built with work. Chapter 3 describes the input documents, their format, and details about them. Chapter 4 explains the design and implementation of the user interface, backend logic, document preprocessing, and the question answering parts of the web-based application for document analysis. The last part of the thesis will be dedicated to evaluating different approaches found through research about the topic.

The goal is to develop an application that combines LLMs with a user-friendly interface that allows easy submission of documents to the knowledge base and a way to ask questions about these documents. This thesis details the process of designing and implementing the application, including the selection of appropriate LLMs, the creation of the vector database, and an intuitive user interface.

The research involves a thorough evaluation and comparison of multiple transformer models, along with various techniques for text chunking, information retrieval, and response generation within a RAG technique. Additionally, it investigates the importance of effective prompt engineering. The goal is to find the configuration of model, RAG strategies, and prompts that yield the most favorable balance of speed, accuracy, and cost efficiency for the specified use cases.

Chapter 2

State-of-the-art of NLP methods

This chapter provides an introduction to state-of-the-art natural language processing relevant to the thesis. It focuses on transformer neural networks, LLMs that utilize the transformer architecture, and RAG, a method used to enhance the output of LLMs. Understanding this is essential to understanding the parts of the implementation of this application and the tools and frameworks used.

2.1 Natural Language Processing

Natural language processing is a field of computer science focused on enabling computers to understand and interpret human language. There are many ways to do this. While historical approaches included models like Naive Bayes Classifiers and Hidden Markov Models, Neural Networks gained prominence in the 2010s and became the dominant methodology [1]. After natural language is processed so the computer can work with it, many natural language processing tasks can be done. Those involve translation, sentiment analysis, speech recognition, text summarization, text classification etc. Focus of this thesis will be on question answering.

2.2 Types of Neural Networks Used for Natural Language Processing

- Recurrent Neural Networks (RNNs)
- Long Short-Term Memory Networks (LSTMs)
- Gated Recurrent Units (GRUs)
- Transformer Networks

RNNs, LSTMs and GRUs are all used for natural language processing [1]. They handle sequential data, such as text. They process inputs sequentially, maintaining an internal memory of previous inputs.

Transformers do not process data sequentially. Rather, they process all of the input simultaneously as shown in Figure 2.1 (parallel processing) by using self-attention mechanisms to balance the importance of various input data components [2]. This architecture has shown

to be very effective for a lot of natural language processing tasks. For question-answering tasks, transformer-based models are currently considered the best approach [1].

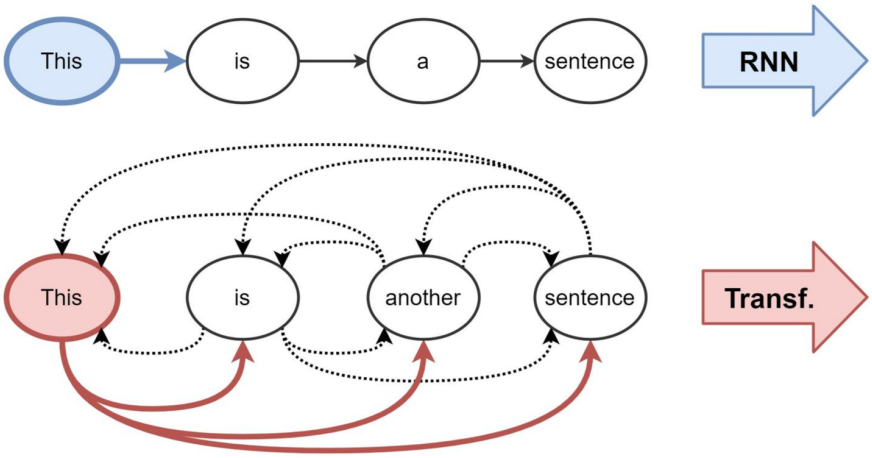


Figure 2.1: The transformer processing compared RNN processing [3]

2.3 Transformer Neural Networks

Transformer Neural Networks are a type of deep learning architecture that process sequential data, like natural language, by using a parallel multi-head attention mechanism. Since their initial proposal in the 2017 paper „Attention Is All You Need“ [2] by Vaswani et al., they have become the standard for a wide range of natural language processing tasks, including text generation, machine translation, question answering, text summarization, and more. The transformer model seen in Figure 2.3 consists of an encoder and a decoder.

The encoder processes the input sequence, while the decoder generates the output sequence. This architecture processes data in parallel, increasing the computational efficiency.

Let's look at how the input sentence goes through the transformer. Let's use the sentence „What is a WRQ packet in TFTP and what information does it contain?“ as an example. First, the sentence is broken down into smaller parts called tokens in a process called tokenization. This can be visualized using OpenAI platform tool for tokenization visualization, using their GPT-4o model[1]. This sentence, which consists of 13 words, would be divided into 16 tokens. Longer words or abbreviations can be broken down into multiple tokens as can be seen in Figure 2.2.

Figure 2.2: Example of tokenization

[1]OpenAI Tokenizer. Available at: https://platform.openai.com/tokenizer [Accessed: May 2025]

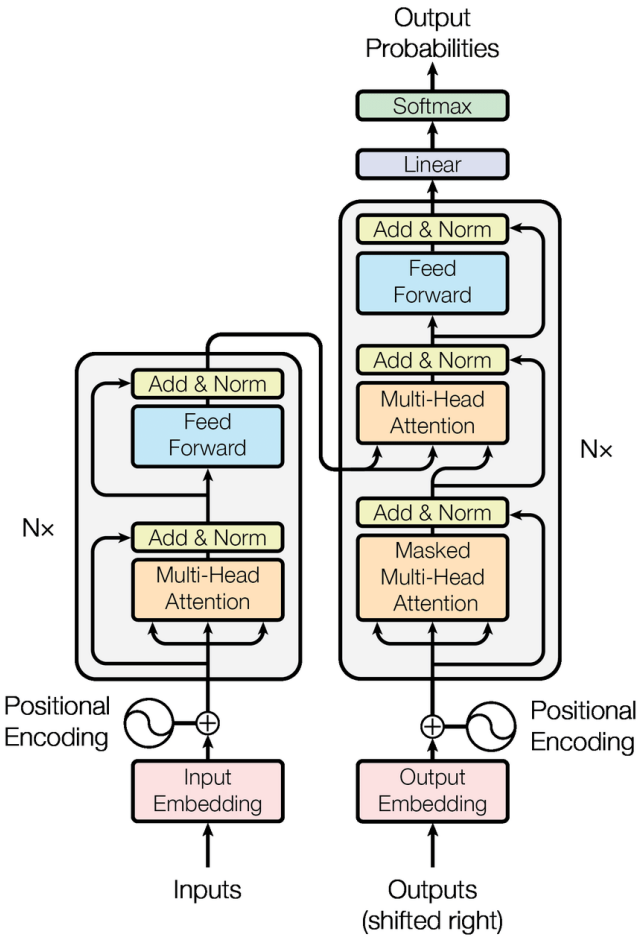


Figure 2.3: The transformer model architecture [2]

After tokenization, the process called embedding, maps each token to a high-dimensional (e.g., 512, 768, 1024) numerical vector. These vectors, called embeddings, capture the semantic meaning of the token or they can capture complex semantic relationships between words as geometric relationships within a high-dimensional vector space. Since the vector space is high-dimensional, and direct visualization is not possible, there are some techniques to reduce the dimensionality that help us to show some relations between the vectors. Examples[2] can be seen in Figure 2.4 and Figure 2.5.

[2]Linear substructures. Available at: https://nlp.stanford.edu/projects/glove/ [Accessed: May 2025]

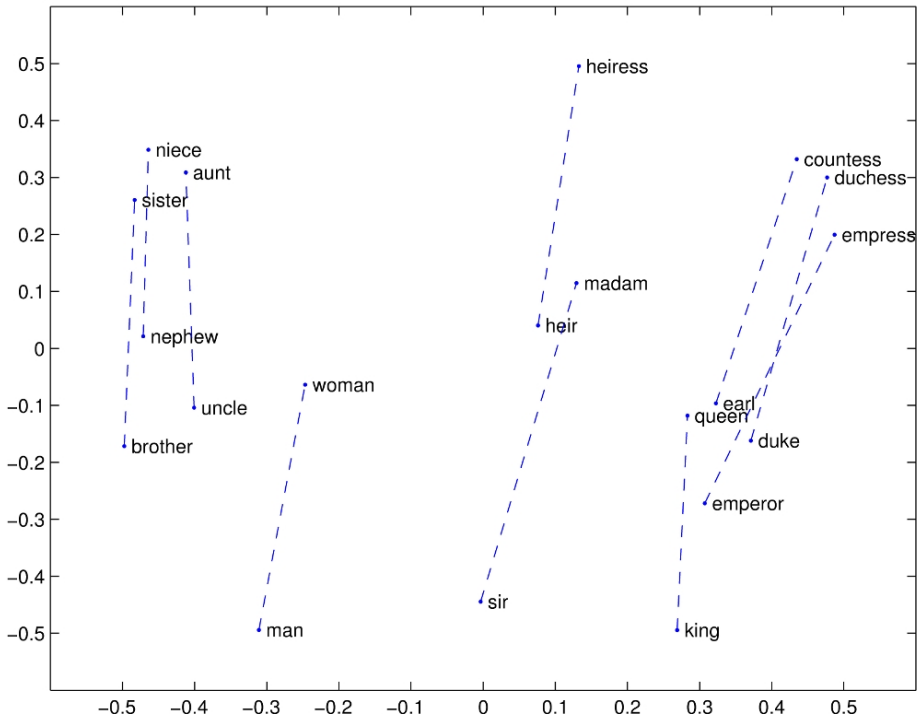


Figure 2.4: Word vector relationships [4]

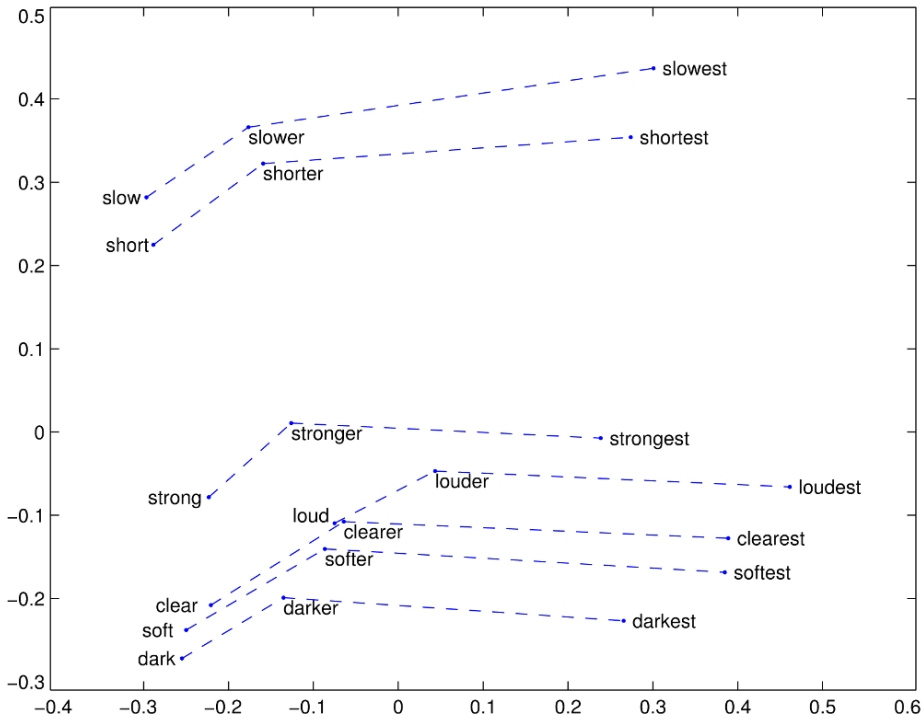


Figure 2.5: Word vector relationships [4]

As Transformer models process all tokens in parallel, they don't inherently know the original order of the tokens. Since word order is essential to the meaning of a sentence,

positional encodings are added to embeddings. The final vector for each token contains both semantic and positional information. Following the addition of positional encodings, these combined embedding vectors are fed into the first encoder layer of the transformer. A standard transformer architecture stacks multiple identical encoder layers sequentially. The output vectors from the first encoder layer become the input vectors for the second encoder layer, allowing the computational process within the layer to be repeated multiple times, progressively refining the representations. Each encoder layer has two main sub-layers, each followed by a residual connection and layer normalization.

The first sub-layer is a Multi-Head Self-Attention mechanism. For each token's vector, this mechanism allows the model to look at all other token vectors in the sequence (including itself) and determine which ones are most relevant or important for understanding this specific token in its current context. Learned weight matrices transform each token's vector into three specialized vectors: a 'Query', representing what the token seeks from others, a 'Key', representing how it might be relevant to others, and a 'Value', its actual content. By comparing a token's Query to other tokens' Keys, the mechanism determines the 'attention' or importance assigned to each corresponding Value. This focus is typically calculated using Scaled Dot-Product Attention as seen in Figure 2.6. Weight matrices contain numerical parameters that start with initial values (often random) and are iteratively updated using algorithms like backpropagation based on how well the model performs on the training data, allowing the model to learn the optimal transformations.

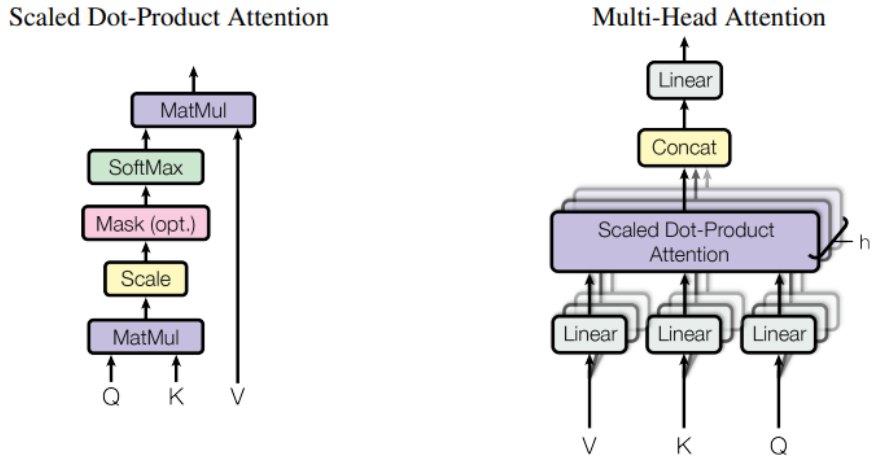


Figure 2.6: Multi-Head Attention [2]

Tokens deemed more relevant get higher weights. The result is a new vector for the token that is enriched with information from other relevant tokens. Following the multi-head attention calculation, the resulting vectors are added back to the original input vectors and the sum is then normalized.

The second sub-layer is a position-wise Feed Forward Network. The Feed Forward Network provides extra computational steps for the model to learn more intricate patterns and features within each token's context-aware representation. It looks at each token's vector one by one, completely separately from the others. The resulting vectors are again added back to the original input vectors and the sum is then normalized.

After passing through the final Encoder layer, we have a sequence of output vectors, one for each input token. The output vectors from this last layer represent the encoder's final, context-rich understanding of the input sequence. Each vector corresponds to an input token but now contains information influenced by the entire sequence through the stacked attention and processing layers.

The final output vectors are passed to the Decoder part of the model. The Decoder uses these vectors, for generating output sequence. Decoder also includes Masked Multi-Head Self-Attention mechanism. Here each token that is being generated can only attend to tokens before it and itself. Decoder generates the output sequence token by token, using the input from encoder and the output that decoder generated so far. Each decoder block contains one additional sub-layer compared to encoder called Multi-Head Cross-Attention where decoder integrates the encoder's output. The Queries come from the Decoder's output of the masked self-attention, and the Keys and Values come from the final Encoder output vectors.

2.4 Large Language Models

LLMs are specifically designed to understand and generate human-like text. They are trained on an enormous amount of text and have a huge number of parameters acquired during training [5]. Transformers architecture is what enabled the scaling required to create modern LLMs, as before architectures like RNNs/LSTMs sequential processing was a bottleneck, making the training speed significantly slower than with transformers parallel processing [2]. Another problem was that, while LSTMs improved memory compared to basic RNNs, they still struggled to maintain relevant information across very long sequences. Information from the beginning of a long paragraph or document could face degradation or loss by the time the model reaches the end. The latest state-of-the-art models like Gemini 2.0 Flash have a 1 million token context window[3]. Such an extended context window is particularly advantageous for tasks involving large volumes of information, including technical documentation associated with complex projects.

2.5 Retrieval-augmented generation

This section introduces Retrieval-augmented generation, which is a method used for the development of this application, and explains more about essential parts such as chunking, embedding, similarity search, context retrieval, and vector databases. Retrieval-augmented generation is a technique used to give LLM access to the data the user wants it to have knowledge of. It is ideal for applications designed to answer questions about technical documentation, as we can just insert files into a vector database and retrieve context from them for LLM. Files then can be dynamically added and accessed at any time. There is no need for lot of data to be stored, just the data user will want to ask about so the search and generation is faster. For example, model named RETRO matched the performance of models 25 times larger using Retrieval augmented [6]. This also offers good generalization as files can be fed from lot of different projects at any time that will serve as context for the LLM.

[3]Google, Long context | Gemini API. Available at: https://ai.google.dev/gemini-api/docs/long-context [Accessed: May 2025]

2.5.1 Chunking

Chunking involves breaking down large documents into smaller parts, so the LLM doesn't have to process whole files every time it needs to produce output, just the relevant parts. Chunking is done in many different ways, the next part describes the most relevant ones for this thesis [7].

Fixed-Size Chunking

Fixed-Size Chunking is one of the simplest methods. Splitting text into segments of a predetermined number of characters, words, or tokens is easy to implement and computationally efficient. It does not take semantic boundaries into consideration, potentially splitting sentences, paragraphs, or code blocks in the middle, leading to context fragmentation. To make this less of a problem, an overlap between consecutive chunks is typically recommended to preserve some semantic context across chunks. However, this adds redundancy and is often not as efficient as other methods.

Sentence/Paragraph Based Chunking

The sentence-based chunking method splits text based on sentence boundaries (e.g., using periods, question marks, exclamation points), often grouping a fixed number of sentences per chunk. It keeps the integrity of individual sentences, which can be crucial for semantic meaning. Frameworks like LangChain integrate with NLP libraries like NLTK for robust sentence segmentation. This approach gives smaller, potentially more precise chunks but may lack broader context compared to the paragraph-based method. The paragraph-based chunking method is similar to sentence splitting but uses paragraph delimiters (e.g., double newlines `\n\n`) as the primary splitting point. This preserves more context within a chunk compared to sentence splitting. However, paragraphs can vary significantly in length, potentially leading to chunks that are too large or too small.

Recursive Chunking

Recursive Chunking approach offers more adaptability by attempting to split text using a predefined list of separators in a hierarchical order (e.g., paragraphs `\n\n`, then sentences `\n` or ., then words). It recursively splits by these separators until the chunks reach the desired size. LangChain's RecursiveCharacterTextSplitter is a popular implementation, using `["\n\n", "\n", " ", ""]` as default separators. This method aims to keep semantically related units (paragraphs, sentences) together better than fixed-size chunking.

Structural Chunking

Structural Chunking methodologies are specifically designed to leverage the structure of the documents, especially when they are in structured formats like Markdown or HTML. Markdown Chunking is a structure-aware text splitting strategy specifically designed for documents formatted using Markdown. Popular implementations of this are LangChain's MarkdownHeaderTextSplitter and MarkdownTextSplitter. It splits text based on Markdown-specific characters, and it also adds in relevant information about where that chunk came from based on the Markdown formatting [8].

Semantic Chunking

This technique moves beyond structural or size-based splitting to group text based on semantic meaning. It typically involves embedding sentences or paragraphs and then identifying breakpoints where the semantic similarity between adjacent units drops below a certain threshold. The goal is to create chunks that are internally similar in topic or meaning, ensuring the integrity of information during retrieval. By focusing on the text's meaning, semantic chunking can enhance retrieval quality, especially when maintaining semantic integrity is vital. However, this method is computationally more intensive and slower than structural or recursive methods due to the need to generate embeddings for all sentences during the chunking process itself. Popular implementation is LangChain's SemanticChunker.

2.5.2 Comparison of chunking methods

Graph in Figure 2.7 shows how different approaches worked for financial documents. Different methods can also be improved by using LLM to generate LLM-based document-level metadata and LLM-based chunk-level metadata. Embeddings can be made with the chunk content and metadata that describe the content. This requires additional computation with LLMs. Adding metadata made great improvements for the trials described in the blog the graph is from, and processing them with a markdown approach yielded the best results.

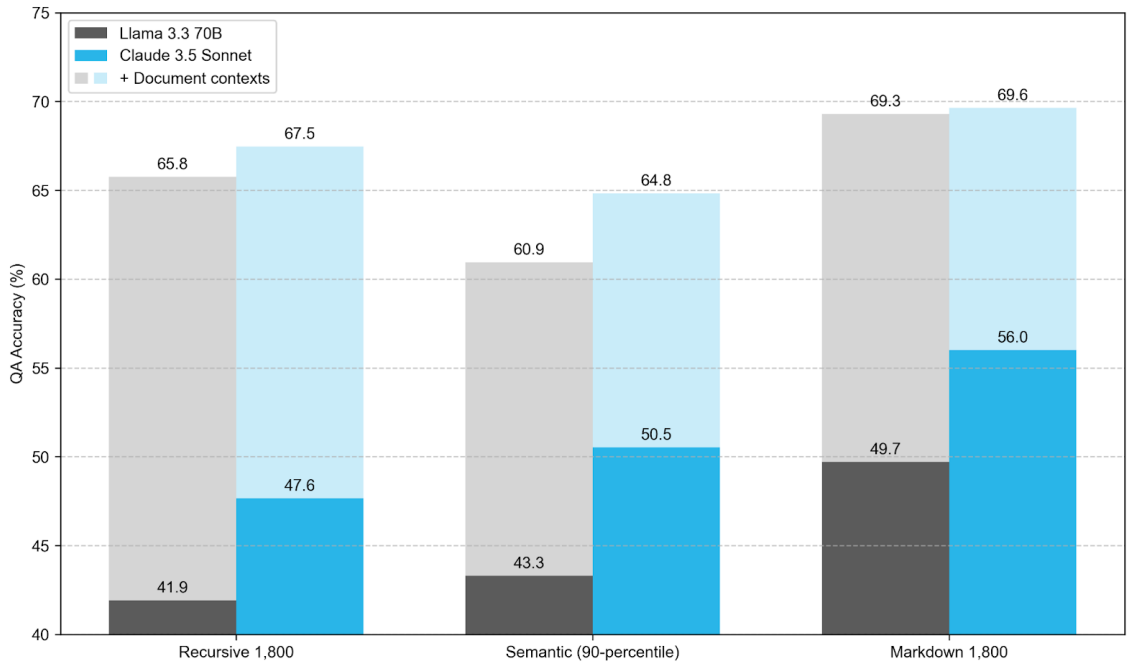


Figure 2.7: Chunking approaches and how LLM-based metadata injection enhances them [8]

2.5.3 Chunk Embeddings

Sentence embeddings are high-dimensional vectors that represent features or attributes of chunks of text. They are made by transformer models to later find similar ones. Vector

dimensions depend on model. These vectors can be generated from semantics, context etc. It's a complex process and it's hard to tell what all the things are that contribute to generating the final vector.

Popular framework for this is Python framework SentenceTransformers. It can use a lot of different models to generate embeddings, one of the most popular ones being all-MiniLM-L12-v2. In the figure 2.8 is visual representation of how embeddings are made, with model deBERTa that makes 768 dimensional vectors.

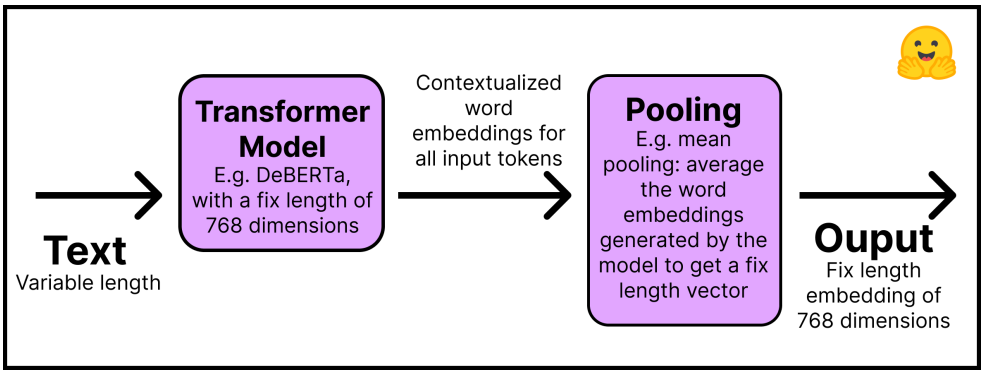


Figure 2.8: Making embeddings from text with SentenceTransformers[4]

An alternative method involves utilizing Application Programming Interfaces (APIs) offered by companies like Google. Although this approach offloads the computational requirements from the user, the user is often rate limited on the free tier or the use of API is paid, typically based on the number of input and output tokens processed. After the embeddings are made, they are stored in the vector database with the chunk of text it was made from, and possibly more valuable metadata for later retrieval of the right vectors.

2.5.4 Vector Database

Vector database is used for efficient storing of high-dimensional vectors. Main reason for using vector databases is that they offer ways to do efficient similarity search [9]. Main purpose is to find chunks of context that are semantically similar to the embedding of the user's query. There are plenty of databases where the user can do hybrid search, SQL queries that filter depending on both standard relational criteria and vector similarity. One of those is Postgres with the extension called pgvector, which makes Postgres hybrid database. This means the user can input vectors into database with other metadata like name of the file, chapter or some metadata generated by LLMs, and then can do hybrid search leveraging both vector similarity search and SQL queries.

Figure 2.9 shows a visual representation of how vector databases are used to store and retrieve context.

[4]Hugging Face Documentation How to Train Sentence Transformers. Hugging Face Blog, 2022. Available at: https://huggingface.co/blog/how-to-train-sentence-transformers. [Accessed: May 2025]

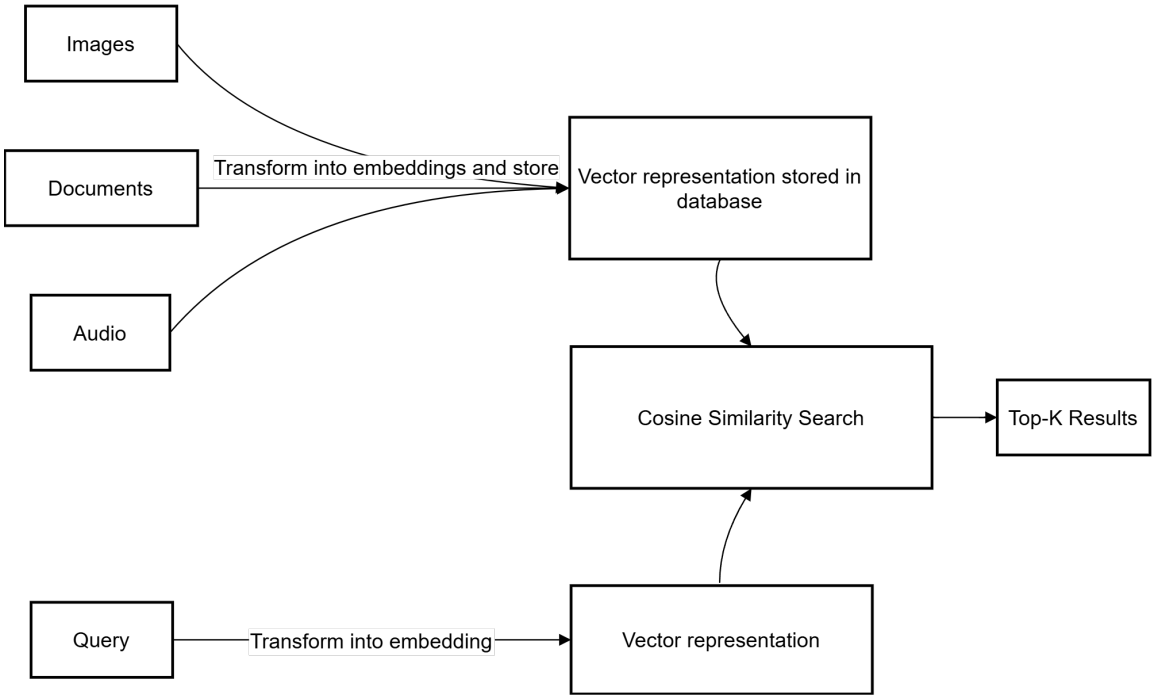


Figure 2.9: Vector database workflow[5]

2.5.5 Retrieval

When the user queries the model, the prompt is converted into a high-dimensional vector with the same embedding model as the chunk embeddings. The prompt vector is taken, and a similarity search is performed to find the most similar vectors, which should be the embeddings of chunks with the most similar meaning. Similarity is typically measured using a distance metric in a vector space, with the most common metric being cosine similarity. Then the chunks that are saved there with the most similar vectors are retrieved. Usually, it is more chunks, the number depending on what works best for that particular application. Cosine similarity measures the cosine of the angle between two vectors in a multi-dimensional space. If the vectors point in the exact same direction, the angle between them is 0°, and the cosine of the angle is 1. This indicates maximum similarity. If the vectors point in opposite directions, the angle is 180°, and the cosine is -1. This indicates maximum dissimilarity or opposition. If the vectors are orthogonal, the angle is 90°, and the cosine is 0. This indicates no correlation or similarity [10]. All of these are visualized in the Figure 2.10.

[5]Elastic, Vector database definition, 2024. Available at: https://www.elastic.co/what-is/vector-database [Accessed: May 2025]

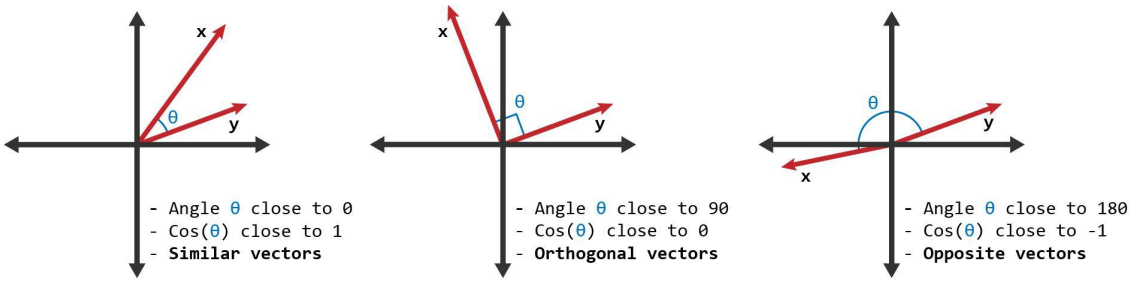


Figure 2.10: Cosine similarity[6]

Reranking can also be part of retrieval. It is a process of reranking the retrieved results and providing them to LLM in the next step in an order that would yield a better result. Mostly, it involves having the most relevant chunks of context given to the LLM first. This aims to reduces the hallucinations by selecting more relevant entries from the knowledge-base and and improves relevancy of output[11]. There are many models to do this, one of them being `semantic-ranker-default-004` from Google.[7]

2.5.6 Generation

In the last step of RAG, the prompt and the retrieved context for the prompt are given to the model that is trained to generate the output. It is typically a generative language model, often a LLM, but for some tasks like question answering, different models, like extractive models, can be used. Generative models can create new sequences of text from scratch. They learn patterns, grammar, and style from their training data and can produce new sentences and paragraphs. Extractive models identify and extract the most relevant parts of existing text. They select parts of text directly from the source document.

2.5.7 Prompt engineering

Prompt engineering is the process of creating prompts that let us guide the behavior of the model and achieve desired results without altering the model parameters. Most models have a property the user can set called system instructions or something similar that lets the user guide the model with natural language to tailor the results for many different use cases. Prompts should be precise and unambiguous as most LLMs are trained on large amount of data, so ambiguous questions can lead to lot of similar context. Common prompt engineering techniques [12]:

- Role prompting is one of the most used methods. It involves instructing the LLM to act as a specific persona when generating an answer, this could be an expert in the field, an assistant in the field, a teacher, etc. It is used to format the way output is generated in a way that the chosen persona would do it.
- One-shot or few-shot prompting is method where user gives the model example to learn from for what should output look like for some input, guiding the model on how to structure and generate output. If no examples are given, it is called zero-shot.

[6] LearnDataSci, Cosine Similarity. Available at: https://www.learndatasci.com/glossary/cosine-similarity/ [Accessed: May 2025]

[7] Google Cloud. „Ranking“. https://cloud.google.com/generative-ai-app-builder/docs/ranking [Accessed: May 2025].

- Structuring the Prompt. Usually instructions are placed at the start of the prompt or in system instructions parameter if model allows that. Often markdown or XML elements such as `###`, `##` or `<document>,<question>` are used to distinguish different parts of the prompt, like context, question, instructions, etc. Also, delimiters like äre common.

2.6 Evaluation of RAG systems

The evaluation of the RAG system can be very challenging as it has many parts to evaluate, not just the final output. Because of this, special frameworks had to be made for RAG systems, that take into consideration all of its parts. One of them is RAGAs introduced in RAGAs: Automated Evaluation of Retrieval Augmented Generation [13]. RAGAs provides python library which can be used to evaluate large amount of metrics, some of which even use LLMs for evaluation. These utilize calls to LLMs to get the score. These metrics can be nondeterministic, as LLMs, even with temperature set to 0 can sometimes return different results for some inputs. Second category RAGAs provides are Non-LLM-based metrics. Those are deterministic for the same inputs. However experiments in RAGAs: Automated Evaluation of Retrieval Augmented Generation [13] have shown LLM-based metrics are closer aligned with human judgments.

2.6.1 RAGAs metrics explanation

The metrics[8] described here are the ones used for evaluation in later chapter. The ones important for question answering part are Faithfulness, Answer Relevancy, Semantic similarity, and Rubrics based criteria scoring. The ones important for retrieval evaluation are Context Precision, Context Recall.

Faithfulness

For this metric the LLM identifies number of claims in the response supported by the retrieved context and divides that value with total number of claims in the response. It ranges from 0 to 1, with higher scores indicating better consistency.

$$\text{Faithfulness Score} = \frac{\text{Number of claims in the response supported by the retrieved context}}{\text{Total number of claims in the response}}$$

Answer Relevancy

For this metric N amount of questions are generated for the answer and then cosine similarity is computed with embeddings of the new generated questions and the original one. It ranges from 0 to 1, with higher scores indicating more relevant answers.

$$\text{answer relevancy} = \frac{1}{N}\sum_{i=1}^{N}\text{cosine similarity}(E_{gi}, E_o)$$

Here N number of questions are generated for the answer

- E_{gi} is the embedding (vector representation) of the generated question i.

[8] List of available metrics, Ragas. Available at: https://docs.ragas.io/en/latest/concepts/metrics/available_metrics/ [Accessed: May 2025].

- E_o is the embedding (vector representation) of the original question.
- N is the number of generated questions, which is set to 3 by default.

Rubrics based criteria scoring

This metric calculates the score depending on user defined rubrics. Typically numeric scores are used with description belonging to them. The LLM compares the generated answer with retrieved context and reference. Then it scores them based on user defined scale.

Semantic similarity

This metric is calculated by making vectors from the answer and reference using the same embedding model and computing cosine similarity between the two vectors, ranging from 0 to 1, with higher scores meaning more similarity.

Context Precision

K here is total number of retrieved chunks, the @K represents precision with K number of retrieved chunks. Context Precision@K is calculated as the mean of the *precision@k* for each chunk in the context. It ranges from 0 to 1, with higher scores indicating a more precise context.

$$\text{Context Precision@K} = \frac{\sum_{k=1}^{K}(\text{Precision@k} \cdot v_k)}{\text{Total number of relevant items in the top K results}}$$

Precision@k is number of relevant chunks at rank k divided by amount of total number of chunks at rank k.

$$\text{Precision@k} = \frac{\text{true positives@k}}{(\text{true positives@k} + \text{false positives@k})}$$

Context Recall

For this metric LLM identifies number of claims in the reference supported by the retrieved context and divides that value with total number of claims in the reference. It ranges from 0 to 1, with higher scores indicating that better context was retrieved.

$$\text{Context Recall} = \frac{\text{Number of claims in the reference supported by the retrieved context}}{\text{Total number of claims in the reference}}$$

2.7 Ragas Dashboard

Ragas also has online dashboard that stores the results of experiments seen on Figure 2.11. When the user chooses an experiment, he sees a table presenting all the samples in the experiment with all the metric scores. The user can annotate the results one by one. There is also an option to see what the input, retrieved contexts, and answer were for the sample. This is shown on Figure 2.12. For LLM-based metrics, it can be seen what queries the RAGAs internally used to evaluate them. For the Non-LLM-based metrics only the output score of the metric and the parts the calculation of score depended on.

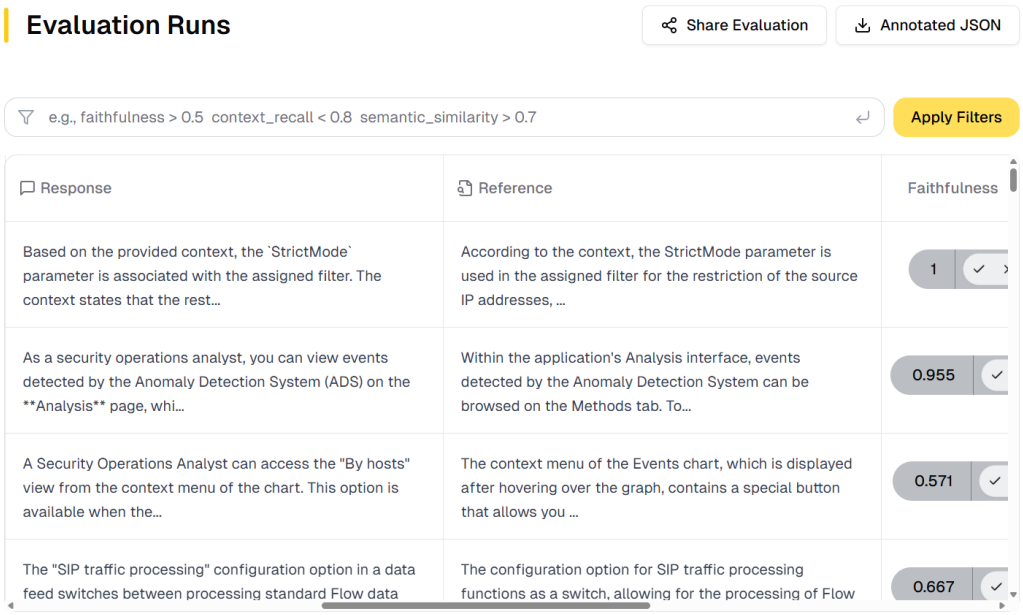


Figure 2.11: RAGAs Dashboard — all results

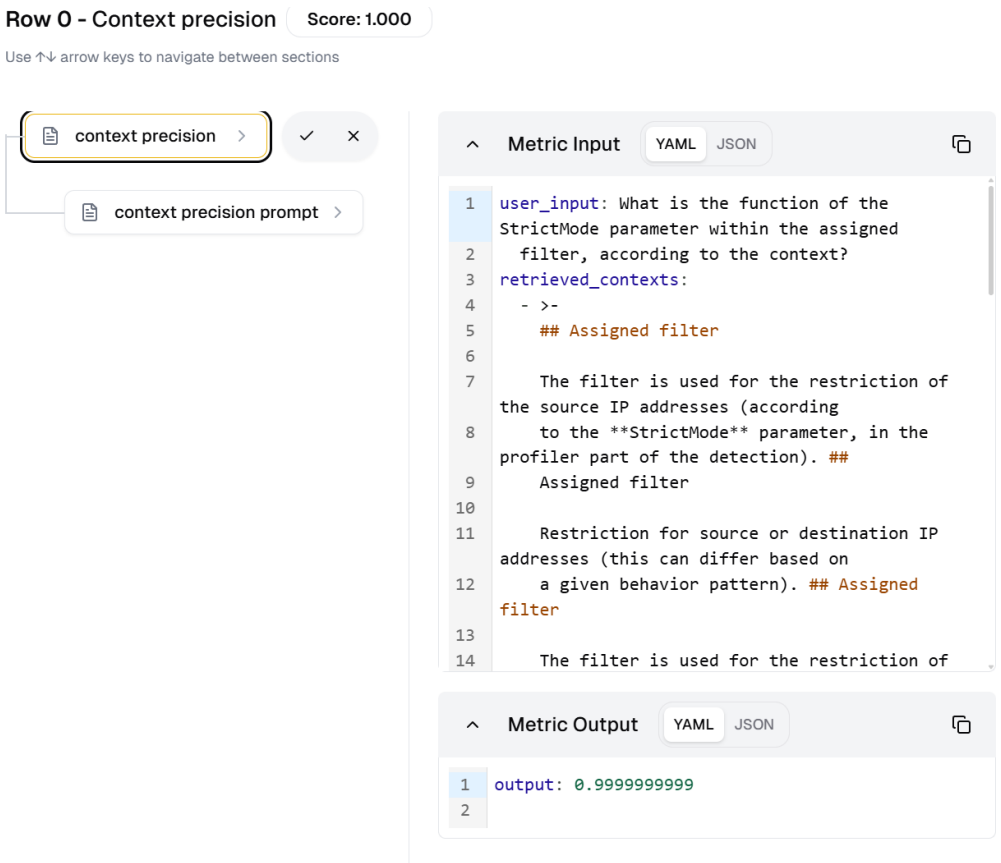


Figure 2.12: RAGAs Dashboard — LLM outputs

2.8 Summary

Chapters 2 covers essential concepts from basic NLP and neural networks, to the most important details of transformer architecture. This chapter explains RAG, its parts, the evaluation framework for RAG systems, and its metrics in detail. The following chapters will build on these concepts and implement the application and its evaluation using the methods and frameworks mentioned here.

Chapter 3

Format of input documents

The structure and format of input documents is one of the most important parts of RAG. With structured files, most commonly used methods are the ones that leverage their structure. If they are not structured, we need to use more generic methods like fixed-size, recursive, or semantic chunking.

3.1 Markdown files

The dataset used for this thesis was provided by Flowmon Networks. It is in the form of Markdown files, which contain technical documentation. These documents provide information about Flowmon ADS, a modern system for the detection of anomalies and patterns of undesirable network behavior. The documents describe the introduction to the system, the installation and configuration, the user interface, and the methods to detect various potentially undesirable activities on the network and accumulate appropriate information. These markdown files contain markdown elements such as headings 3.2, lists, tables 3.3, JSON blocks 3.1, etc. This is the main dataset the application focuses on during implementation and evaluation in the following chapters. It consists of 107 files, sized 382 KB.

Body of the email per event format:

```
<ID>: (unique event identifier);
<Category>: (code of the detection method);
<Type>: (name of the detection method);
<Perspective>: (name of the perspective assigned to the report);
<Severity>: (priority of the event);
<Time>: (start time in UTC or in Flowmon appliance local time, based on the report settings);
<Protocol>: (protocol related to the event or empty value);
<Source>: (source IP address);
<Target IPs>: (first 10 target IP addresses);
<Ports involved>: (port numbers related to the event or empty value);
```

Body of the RT email, the event details were shortened:

```
IP: 192.168.1.1 // event source
Type: DNS traffic anomaly // detection method description
Severity: Critical
Time: 2015-11-05 17:00:13 GMT+0100 // leader event detection time
Incident details: // event id, detection time, detail, targets
5102353 2015-11-05 17:00:13 GMT+0100 Use of ... (connections: 20). 8.8.8.8
5102382 2015-11-05 17:00:13 GMT+0100 Use of ... (connections: 20). 8.8.8.8
```

Figure 3.1: Showcase of different types of formatting in markdowns — JSON blocks

Syslog

The application also supports event export in the Common Event Format (**CEF**). It is possible to set multiple targets for the syslog messages in the **Settings** → **Processing** → **Syslog message** section. Syslog messages are assigned to the *local6* facility. It is possible to configure the following parameters:

- **Name**: Name of the configuration.
- **Perspective**: The perspective that serves as a base for the syslog messages. The priority is translated to the *CEF severity* and, with accordance to the configuration, also to the *syslog severity* (see the table below).
- **Active**: Activation of sending the syslog messages.
- **Priority as severity**: Activate the translation of the Flowmon ADS priority to the *syslog severity*. See the table below.
- **Target**: Choose whether the global Flowmon OS configuration should be applied (from the *Flowmon Configuration Center*) or whether to send syslog messages directly to the specified target (using the *ADS syslog sender*). Note that if the Flowmon appliance operates in a multi-tenant mode, it is recommended to set this option to **ADS syslog sender** to ensure that events from the current tenant will be sent to the correct Syslog server. The global Flowmon OS configuration is system-wide. Tenants that set their Syslog to respect the global configuration will send the detected events to all syslog servers defined in the Base tenant (**Flowmon Configuration Center** → **System** → **System Settings** → **Syslog Event Logging**). For further details about multi-tenancy, refer to the following page: Tenants.
- **Protocol**: The transport protocol to transmit syslog messages (TCP or UDP).
- **Remote IP**: IP address of the target syslog server.
- **Port**: Port number of the target syslog server.
- **Use event ID**: Adding the event ID to the syslog messages (can be used to find the event in the user interface).
- **Divide by event targets**: Decide whether to send event targets in one syslog message or in a single syslog message for each event target.
- **Maximal number of messages per event**: Maximal number of syslog messages (if syslog messages are divided according to the single event target assignment). The last message contains all the remaining event targets.
- **Machine-readable detail**: Activation of the event detail in the format suitable for the following machine processing. The variables of the event details are written in the form *parameter:value*.

Figure 3.2: Showcase of different types of formatting in markdowns — paragraphs and lists

Priority of the perspective	Syslog severity	CEF severity
CRITICAL	Alert	10
HIGH	Critical	8
MEDIUM	Error	6
LOW	Warning	4
INFORMATION	Notice	2

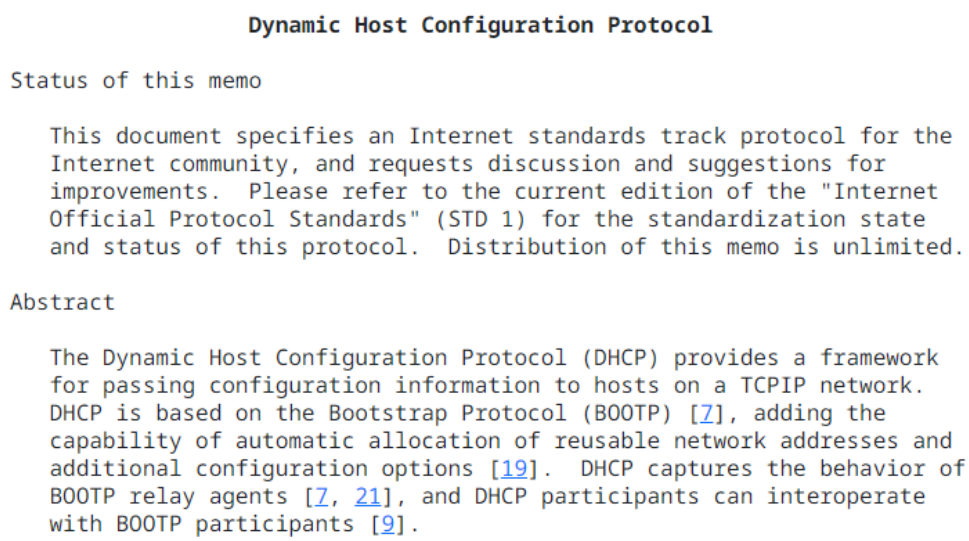
Figure 3.3: Showcase of different types of formatting in markdowns — tables

3.2 Text files

To test how the application behaves on plain text files, RFC documents describing official internet protocol standards[1] will be used. To match the size of the first dataset, 37 text files sized 408 KB from the 123 files available in this RFC section will be used for the experiments in the evaluation chapter. They contain technical specifications and organizational notes for the Internet, and they are available in multiple formats like text, PDF, or HTML. They have lot of different ways of formatting inside of them like plain text paragraphs shown in Figure 3.4, tables shown in Figure 3.6, chapters, sections, lists shown in Figure 3.5, ASCII

[1] *Official Internet Protocol Standards*. Available at: https://www.rfc-editor.org/standards. [Accessed: May 2025].

diagrams, etc. In these, evaluation can be made showing how the implementation deals with different types of formatting.



Dynamic Host Configuration Protocol

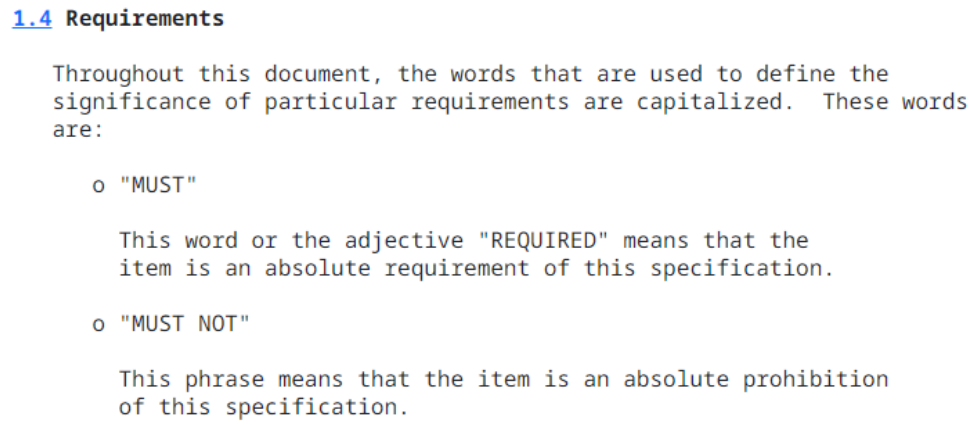
Status of this memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Abstract

The Dynamic Host Configuration Protocol (DHCP) provides a framework for passing configuration information to hosts on a TCPIP network. DHCP is based on the Bootstrap Protocol (BOOTP) [7], adding the capability of automatic allocation of reusable network addresses and additional configuration options [19]. DHCP captures the behavior of BOOTP relay agents [7, 21], and DHCP participants can interoperate with BOOTP participants [9].

Figure 3.4: Showcase of different types of formatting in RFCs — paragraphs



1.4 Requirements

Throughout this document, the words that are used to define the significance of particular requirements are capitalized. These words are:

- o "MUST"

  This word or the adjective "REQUIRED" means that the item is an absolute requirement of this specification.

- o "MUST NOT"

  This phrase means that the item is an absolute prohibition of this specification.

Figure 3.5: Showcase of different types of formatting in RFCs — lists

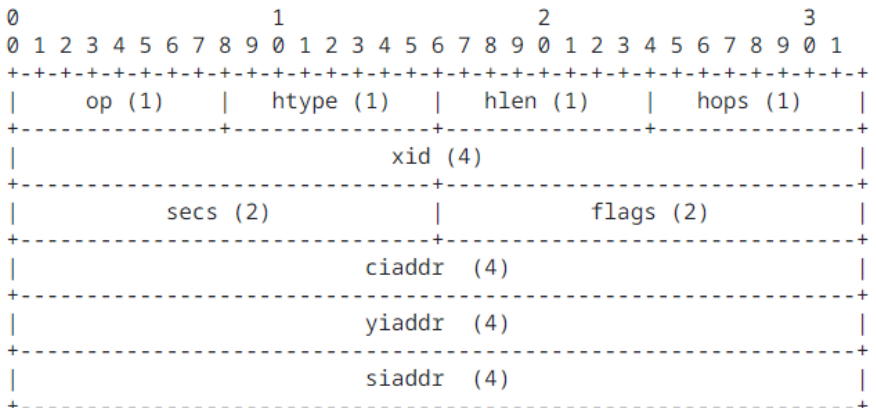


Figure 3.6: Showcase of different types of formatting in RFCs — tables

3.3 Summary

This chapter describes the structural characteristics, formatting elements, and size of both datasets. Recognizing these patterns, such as headings, lists, JSON blocks, and tables in Markdown and RFC files, will provide a way to propose a processing of input documents in the next chapter.

Chapter 4

Design and Implementation of the application for document analysis

This chapter will describe how the QA application's frontend and backend are implemented, what tools have been used, and how. It also shows how input documents are being processed and ingested into the vector database. It also shows diagrams that describe the architecture of this application and show how every part of the architecture is implemented. Implementation is split into two chapters. This chapter is about the web-based application that users can use to make queries about technical documentation related to the project. The next one is about evaluation scripts that were used to choose the models, their parameters, and RAG settings to achieve the best usability of the web-based application.

4.1 User interface implementation

The user interface was designed with user-friendliness as a primary goal, making the process of uploading and querying technical documentation simple for users. The front-end was developed with HTML5, providing the structural foundation, and Bootstrap and CSS were used for the layout and styling. Vanilla JavaScript was used for client-side interaction. It allows users to select the model used to provide answers for their prompts and to select the files that users wanted to use for querying. For the backend, Python was chosen as it is widely used for development involving Artificial Intelligence. It has many libraries for working with neural networks and especially transformers, and makes working with them simple. Python is used with Python web framework Flask, to handle server-side operations and application logic. Flask makes HTTP request processing simple with Flask routes, which map incoming requests to Python functions. Templates are files that contain static data and placeholders for dynamic data. They are used to render HTML with specific data in a web browser, so Flask templates were used for dynamic content rendering.

4.2 Graphical user interface design

These are the main parts of graphical user interface shown in Figure 4.1.

- **Model:** The header section contains the dropdown that lets users switch between different LLMs.
- **Uploaded documents:** Panel with checkboxes that lets the user select which documents he wants to query or summarize.

- **Upload File:** User can upload text documents or markdown files in this section. These documents will be processed and stored in a vector database to be later used for context retrieval.
- **Question Field:** Once the document is uploaded, the user can ask specific questions related to the content of the document.
- **Answer Section:** After submitting question, answer will be shown here.
- **Uploaded Documents:** In this area user can select which documents in the vector database he would like to query.
- **Clear buttons:** Buttons to delete the chat history or clear database.

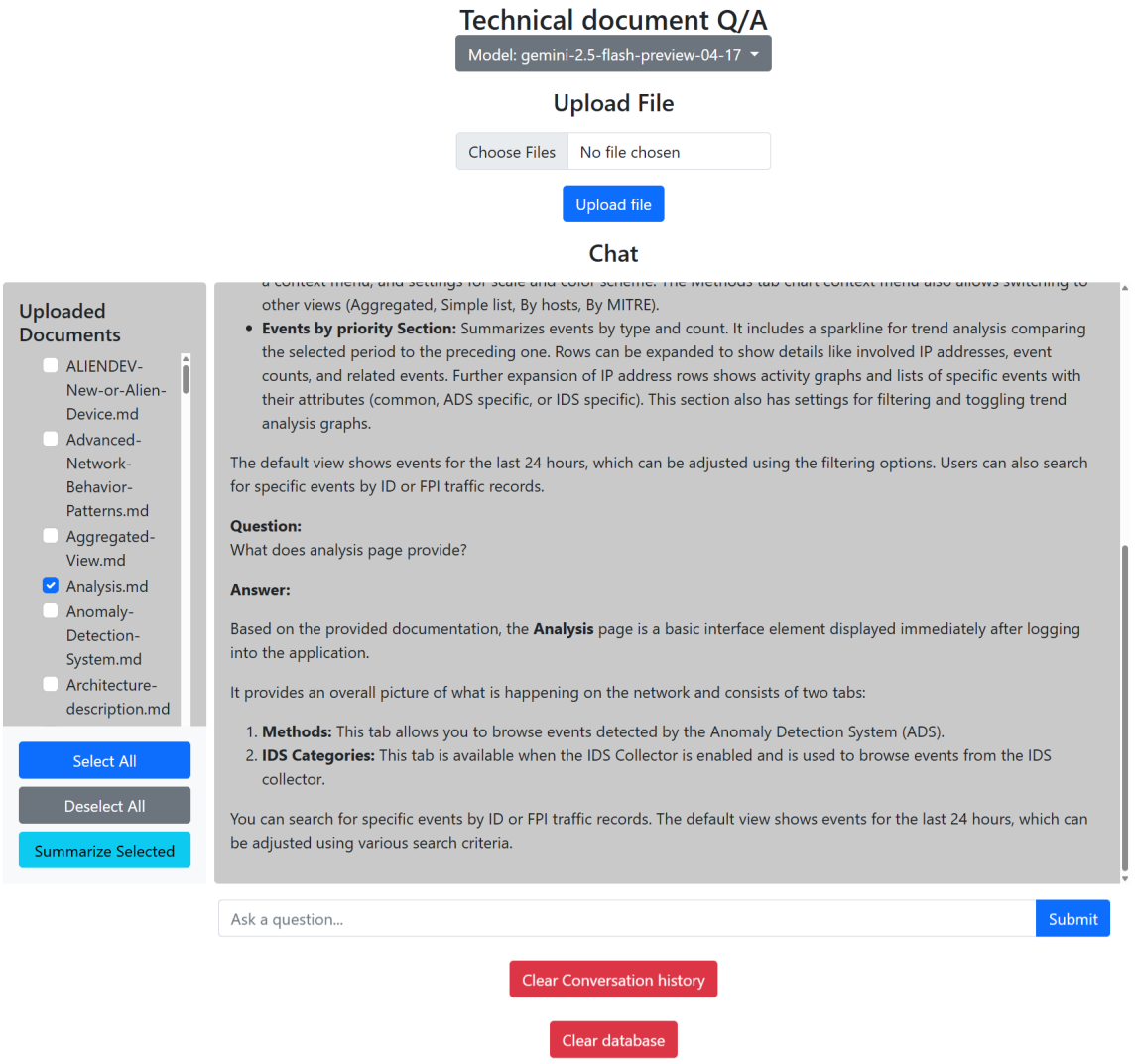


Figure 4.1: Graphical user interface

4.3 Preprocessing input documents

Documentation preprocessing for this tool consists of 4 steps shown in the Figure 4.2.

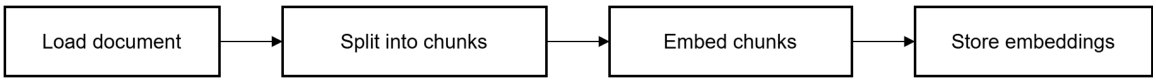


Figure 4.2: Steps for preprocessing files[1]

4.3.1 Load

The loading is done by user inserting files into HTML form with the attribute which allows file uploads `enctype="multipart/form-data"`. Supported formats are markdown files and plain text files. When the file is submitted, POST request is made and it is handled by Flask `/upload` route. In there `upload_file` function is called. There every file is processed, first calling `add_file_name_to_db` function which adds file name to `uploaded_documents` table in Postgres. This table is used to display name of all uploaded documents in the left panel used for filtering in the web interface. After this `process_and_store_file` function is called and takes file content, filename and database connection as parameters. In there `decode('utf-8')` is used to convert file content into a string. The file content is then split into chunks.

4.3.2 Split into chunks

For splitting into smaller chunks, I use different approaches, which are set with input parameters for the application, based on the input file format. All character splitters I am using are from langchain[2].
First approach is using `MarkdownHeaderTextSplitter`.
In this approach `MarkdownHeaderTextSplitter` class from `langchain_text_splitters` is used create instance of `MarkdownHeaderTextSplitter`, here is what parameters and methods I am using.

Parameters

- `headers_to_split_on (List[Tuple[str, str]])`: Headers we want to track.
- `strip_headers (bool)`: Strip split headers from the content of the chunk.

Methods

- `__init__(headers_to_split_on[, ...])`:Create a new `MarkdownHeaderTextSplitter`.
- `split_text(text)`: Split markdown file into multiple components.

The headers used are `("##", "Header 2")` and `("###", "Header 3")`. `strip_headers` is set to `False`. `split_text(text)` method is called on the instance with file content as a parameter, which splits it into smaller chunks.

[1] Langchain Documentation, RAG. Available at: https://python.langchain.com/docs/tutorials/rag/ [Accessed: May 2025]
[2] Langchain Text splitters. Available at: https://api.python.langchain.com/en/latest/text_splitters/index.html [Accessed: May 2025]

Second approach is using `RecursiveCharacterTextSplitter`.
In this approach `RecursiveCharacterTextSplitter` class from `langchain_text_splitters` is used create instance of `RecursiveCharacterTextSplitter`, here is what parameters and methods I am using.

Parameters

- `chunk_size (int)` – Maximum size of chunks to return
- `chunk_overlap (int)` – Overlap in characters between chunks
- `length_function (Callable[[str], int])` – Function that measures the length of given chunks

Methods

- `__init__([chunk_size, ...])`: Create a new `RecursiveCharacterTextSplitter`.
- `split_text(text)`: Split text into multiple components.

I am setting `chunk_size` to `1000`, `chunk_overlap (int)` to `200` and `length_function` to `len`. `split_text(text)` method is called on the instance to split it same way as with `MarkdownHeaderTextSplitter`.
Third approach is using `SemanticChunker`.
In this approach `SemanticChunker` class from `langchain_experimental.text_splitter` is used create instance of `SemanticChunker`, here is what parameters and methods I am using.

Parameters

- `embeddings` – Interface for embedding models

Methods

- `__init__(embeddings[, buffer_size, ...])`: Create a new `SemanticChunker`.
- `split_text(text)`: Split text into multiple components.

I am setting `embeddings` to instance of `GoogleGenerativeAIEmbeddings` class, which is imported from `langchain_google_genai` library. It is used to connect to Google's generative AI embeddings service. As `model` I set `models/text-embedding-004`. `split_text(text)` method is called on the instance to split it same way as with `MarkdownHeaderTextSplitter`.

4.3.3 Embed chunks

For embedding the chunks into vector representation, I use different models, based on input parameters for application.
First approach is using google API to make embeddings with `models/text-embedding-004`[3].
Python library genai, which is made by Google to use generative artificial intelligence with APIs, is used to make embeddings out of the document chunks. The model used is `models/text-embedding-004` and function used is `embed_content` as shown in Listing 1. This maps sentences to a 768 dimensional dense vector space that can be used for sentence or paragraph similarity search.

[3]Gemini API — Embeddings Available at: https://ai.google.dev/gemini-api/docs/embeddings [Accessed: May 2025].

```python
client = genai.Client(api_key=os.environ.get('GEMINI_API_KEY'))
def generate_embeddings_genai(text):
    result = client.models.embed_content(
        model="models/text-embedding-004",
        contents=text,
    )
    return result.embeddings[0].values
```

Listing 1: This code snippet demonstrates how embeddings are made with genai library.

Second approach is using Python framework Sentence Transformers to make embeddings with `intfloat/multilingual-e5-large-instruct`. Using Sentence Transformers [14] which is Python library for state-of-the-art sentence, text and image embeddings.[4] Embeddings are made from the document chunks, they are 1024 dimensional. The model used is `intfloat/multilingual-e5-large-instruct` shown in Listing 2.

```python
e5_model = SentenceTransformer('intfloat/multilingual-e5-large-instruct')
def generate_e5_embeddings(text):
    instr_text = f"Instruct: Represent this documentation for retrieval
    ↪  \nQuery: {text}"
    embedding = e5_model.encode(
    instr_text,
    convert_to_tensor=False,
    normalize_embeddings=True
    )
    return embedding.tolist()
```

Listing 2: This code snippet demonstrates how embeddings are made with Sentence Transformers library.

4.3.4 Store

After the embeddings are made `insert_embedding_pgvector` function is called. Here, embeddings are inserted into the database along with the content the embedding was made out of and the file name from which the chunk was taken. As vector database Postgres with pgvector extension is being used. How records with embeddings are stored into the database is shown in Listing 3.

[4] Sentence-Transformers Documentation. Available at: https://sbert.net/docs/sentence_transformer/usage/usage.html [Accessed: May 2025]

```python
def insert_embedding_pgvector(conn, file_name, content, embedding):
    with conn.cursor() as cur:
        cur.execute(
            """
            INSERT INTO markdown_embeddings (file_name, content, embedding)
            VALUES (%s, %s, %s);
            """,
            (file_name, content, embedding)
        )
        conn.commit()
```

Listing 3: This code snippet demonstrates how embeddings are stored in database.

4.4 Question Answering and Document summarization

Question answering for this tool can be described by following diagram in the Figure 4.3.

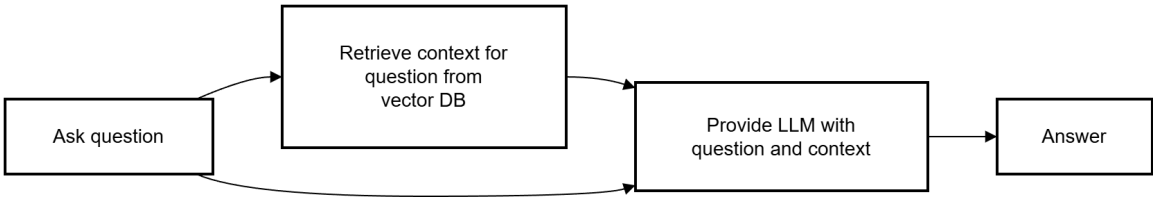


Figure 4.3: Steps for generating answer for question from the given context [5]

4.4.1 Ask question

When a POST request is received at the `/question` endpoint, the Flask routing mechanism directs it to the appropriate handler, which executes the `answer_question` function.

4.4.2 Retrieving context from vector database

First embedding is made out of the question with the same model as embeddings of chunks were made, so either `generate_embeddings_genai` function or `generate_e5_embeddings` function is called. After embedding of is made, most similar vectors in vector database can be searched for. Cosine similarity is being used as similarity metric and k most similar embeddings are being retrieved where `k` is based on input parameters for the application. How query for hybrid search looks like when using pgvector is shown in Listing 4.

[5] Langchain Documentation, RAG. Available at: https://python.langchain.com/docs/tutorials/rag/ [Accessed: May 2025]

```
    """
    SELECT * FROM markdown_embeddings
    WHERE file_name = ANY(%s)
    ORDER BY embedding <=> %s::vector
    LIMIT %s;
    """,
    (selected_files, query_embedding, top_k)
```

Listing 4: Cosine similarity search with SQL query filters.

Then based on input parameters, reranking can be used and can return different number of chunks. For reranking I am using `semantic-ranker-default-004` model from Google. Reranking is done using `discoveryengine` module from `google.cloud` library in a way show in Listing 5.

```
client = discoveryengine.RankServiceClient(credentials=credentials)
ranking_config = client.ranking_config_path(
        project=PROJECT_ID,
        location="global",
        ranking_config="default_ranking_config",
    )
request = discoveryengine.RankRequest(
        ranking_config=ranking_config,
        model="semantic-ranker-default-004",
        top_n=top_n,
        query=query,
        records=records
    )
response = client.rank(request=request)
```

Listing 5: Reranking with semantic-ranker-default-004.

4.4.3 Prompting QA model and getting answer

Model for question answering can be chosen in dropdown in web interface 4.1. Based on which one is selected, one of the functions is called. First option is `answer_genai_api` for model `gemini-2.5-flash-preview-04-17` and the second one `answer_vertex_api` for the model `llama-4-maverick-17b-128e-instruct-maas`. Retrieved context and the question are passed as parameters for these functions. Then prompt for the LLMs is put together. To guide LLMs I used prompt engineering methods mentioned earlier like role prompting and setting the system instructions. The resulting prompt and system instructions are as shown in Listing 6.

```
prompt = f"""
**Chat History:**
{[]}

**Retrieved Context (from vector database based on current question):**
{context}

**Current Question:**
{question}
"""
```

Listing 6: Prompt for large language models

I am using the same instructions for all LLMs. First I used role prompting to set the persona for the LLM.

```
### Role and Goal: ###
    You are an AI assistant specialized in answering questions based
    on provided technical documentation. You are part of a RAG system.
    Your goal is to be helpful, accurate, and context aware, leveraging
    both the retrieved documents and the ongoing conversation history.
```

Then I wanted to set the priority on what should the LLM focus when generating output.

```
### Instructions: ###
    1.  # Analyze the Current Question #: Carefully consider the
    Current Question in the context of the full Chat History.
    2.  # Check for Follow-up #: Determine if the Current Question
    is a direct follow-up, clarification request, or elaboration
    on the preceding answers in the Chat History.
    3.  # Prioritize Previous Answer, if there is one #: If the
    Current Question is a direct follow-up on the previous answer,
    prioritize information from that previous answer first.
    Use the Retrieved Context to supplement or verify information
    for this follow-up question, but ensure your response is focused
    on the previous answer.
    4.  # If the current question is not a follow-up, prioritize the
    Retrieved Context #:  If the current question is not a follow-up,
    base your answer primarily on the Retrieved Context, with history
    potentially supplementing it.
    5.  #Use full provided history#: Provided history consists of last
    5 questions and answers. Understand what the user is trying to achieve
    and use the full context to answer the question.
```

Another goal was to let the LLM know how to synthesize the information, when to use its internal knowledge instead of provided information, and what to do if answer cannot be found in either of those.

```
6.  # Synthesize the retrieved information logically #: In RAG, the
context is sorted by cosine similarity with the question. If relevant
pieces are scattered within the context, resolve potential ambiguities.
6.  # Detail and accuracy #: Provide a detailed, clear, and accurate
answer based on the information sources and their priority.
7.  # Fallback to general knowledge #: If the answer cannot be
```

```
reasonably constructed from the Retrieved Context and relevant parts
of the Chat History, use your general knowledge as a source with the
lowest priority. If you use your general knowledge, let the user know
that the information doesn't come from the provided technical
documentation but from your general knowledge.
8.  # Handling Uncertainty: # If you cannot confidently answer the
question using context and history, let the user know, don't make up
the answer.
```

The last part was there specifically because answers are being rendered on the web interface and they should be formatted in a way the user can understand them with ease.

```
9.  # Make the answer markdown format #: The whole generated text needs
to be formatted according to markdown rules, to make it easier to
display on webpage. Don't use big headings, use only ###.
```

After the prompt is built, the function `answer_genai_api` genai client is used to query the LLM using API as shown in Listing 7.

```
client = genai.Client(api_key=os.environ.get('GEMINI_API_KEY'))
response = client.models.generate_content(model=session['selected_model'],
contents=prompt,config=types.GenerateContentConfig(
    temperature=0.0,
    max_output_tokens=3000,
    system_instruction=[
        types.Part.from_text(text=SYSTEM_INSTRUCTIONS),
    ]
))
return response.text
```

Listing 7: Call to the genai API

Function `answer_vertex_api` calls vertex API the way shown in Listing 8.

```
SERVICE_ACCOUNT_KEY_PATH = "path_to_vertex_key.json"
cred = service_account.Credentials.from_service_account_file(
        SERVICE_ACCOUNT_KEY_PATH
)
vertexai.init(project=PROJECT_ID, location=LLAMA_REGION, credentials=cred)
vertex_model = GenerativeModel(model_name=f"publishers/meta/models/{qa_model}",
system_instruction=SYSTEM_INSTRUCTIONS)
chat = vertex_model.start_chat()
response = client.models.generate_content(model=session['selected_model'],
contents=prompt,config=types.GenerateContentConfig(
    temperature=0.0,
    max_output_tokens=3000,
    system_instruction=[
        types.Part.from_text(text=SYSTEM_INSTRUCTIONS),
    ]
))
return response.text
```

Listing 8: Call to the vertex API

After the answer is received from the API, it is rendered on the web interface for the user. It also gets appended along with the question to the chat history that is saved in the Flask session.

4.4.4 Document summarization

When a POST request is received at the `/summarize` endpoint, the Flask routing mechanism directs it to the appropriate handler, which executes the `summarize_file` function. Summarizing works same as asking question regarding prompting the LLM, only difference is that the context is not retrieved based on similarity search, but all chunks made from selected file are retrieved and the question part in prompt is `Summarize this document`.

4.5 Summary

This chapter detailed the complete architecture and implementation of the technical document analysis application. Key components, including the user-friendly interface, the document preprocessing pipeline involving loading, chunking, embedding, and storage, and the RAG-based question-answering system utilizing selected LLMs and context retrieval, have been described. The result is an application designed to meet the thesis goal, which involves performing Q&A tasks and summarizing technical documentation. After implementing the application, the next step is to evaluate its performance, which is part of the implementation that the next chapter is focused on.

Chapter 5

Evaluation

To assess the performance of my RAG system, I am using RAGAs framework in the evaluation program. The program is designed to easily evaluate the RAG system by allowing simple adjustments to input parameters like chunking methods, embedding and generation LLMs, and the amount of retrieved context. Evaluation with RAGAs requires questions and ground truths that are related to documents ingested into the database for context retrieval. LLMs are prompted with these questions same way, described in previous chapter then dictionary containing questions, ground truths, chunks of context and answers is made. Using `datasets`[1] python library, dataset type is made out of the dictionary. Then RAGAS `evaluate` function is used to evaluate the dataset, and get scores for the set metrics. How it is used in evaluation program is shown in the Listing 9.

```
eval_data = {
    "question": questions_rfc,  # List of strings
    "ground_truth": ground_truths_rfc, # List of strings
    "contexts": contexts,  # List of lists of strings
    "answer": answers  # List of strings
}
eval_dataset = Dataset.from_dict(eval_data)
result = evaluate(
            dataset=eval_dataset, # Dataset object
            metrics=metrics, # List of metrics
            llm=gemini_llm,  # LLM instance
            embeddings=gemini_embeddings, # Embedding model instance
            raise_exceptions=True,
            run_config=rate_limit_config,
            experiment_name=experiment_name
        )
```

Listing 9: Evaluating datasets

RAGAs has a `TestsetGenerator` that lets a user generate testset from their documentation using LLMs [2]. It generates the scenarios and personas, and then generates sample questions with corresponding ground truths based on the dataset provided. Us-

[1] Hugging Face Datasets Documentation. https://huggingface.co/docs/datasets/index [Accessed: May 2025]

[2] Ragas , Generate Synthetic Testset for RAG https://docs.ragas.io/en/stable/getstarted/rag_testset_generation/ [Accessed: May 2025]

ing `TestsetGenerator` I have generated a testset that includes 70 questions and ground truth related to the question for this Flowmon ADS dataset, which contains 107 markdown files, with a size of 382 KB. For rubrics, I am using the ones mentioned in RAGAs documentation[3].

```
"score1_description": "The response is entirely incorrect and fails to address
↪  any aspect of the reference.",
"score2_description": "The response contains partial accuracy but includes
↪  major errors or significant omissions that affect its relevance to the
↪  reference.",
"score3_description": "The response is mostly accurate but lacks clarity,
↪  thoroughness, or minor details needed to fully address the reference.",
"score4_description": "The response is accurate and clear, with only minor
↪  omissions or slight inaccuracies in addressing the reference.",
"score5_description": "The response is completely accurate, clear, and
↪  thoroughly addresses the reference without any errors or omissions.",
```

5.1 Comparison of used model specifications and prices

Pricing for text inputs and outputs per million tokens is shown in Table 5.1.[4]

Model	**Parameters**	**Pricing ($/million tokens)**	
		Input	**Output**
Gemini 2.5 Flash	Not specified	$ 0.15	$0.60
Mistral Large (24.11)	123 B	$2.00	$6.00
llama-4-maverick	17 B active 400 B total	$0.35	$1.15

Table 5.1: Model Pricing and Parameters Comparison

5.2 Experiments

This section details the experiments conducted to evaluate the performance of the RAG system. Evaluation consists of testing various configurations of chunking methods, LLMs, number of retrieved context chunks and reranking of retrieved context. It contains an evaluation of all of these parameters, with multiple LLMs calculating the score for metrics, and explores the effectiveness of different parameters on different formats of input files. RAGAs scores every question in the data set separately, so all the results mentioned in the tables are mean of all the question scores.

5.2.1 Experiment number 1: Number of retrieved context chunks

For my first experiment, I decided to do the following settings:

[3]Code adapted from: Ragas. Rubrics-based criteria scoring | Ragas Documentation. Available at: https://docs.ragas.io/en/stable/concepts/metrics/available_metrics/general_purpose/#rubrics-based-criteria-scoring

[4]Pricing data from Google Cloud Vertex AI documentation Available at: https://cloud.google.com/vertex-ai/generative-ai/pricing#cost-of-building-and-deploying-ai-models-in-vertex-ai [Accessed: May 2025]

- Chunking Method: Markdown chunking, as based on figure 2.7 it should yield the best results.
- Data: All 107 Flowmon markdown files ingested in database, sized 382 KB
- Embedding model: Google text-embedding-004
- LLMs: gemini-2.5-flash-preview-04-17, llama-4-maverick-17b-128e-instruct-maas, Mistral Large (24.11)
- Number of retrieved context chunks: variable
- LLM used to calculate score for metrics: gemini-2.5-flash-preview-04-17

As RAGAs needs LLM to generate the score for calculating the metrics I first decided to use gemini-2.0-flash-001. I ran evaluation using gemini-2.5-flash-preview-04-17 to generate responses to the prompts, with 20 retrieved chunks. Then I reran the evaluation again and got different results for both. This can happen for multiple reasons, most common ones being that models parameters are continuously being adjusted, the other one floating-point arithmetic precisions. Because operations calculating the probability of the next token are parallelized on multiple resources, the exact order of operations can vary slightly between runs. With this happening, some token that wouldn't otherwise have higher probability of being next token can get picked.

After this I tried using gemini-2.5-flash-preview-04-17 and the results were the same across multiple runs with the same parameters, so I decided to keep using it as the LLM used to calculate score for metrics. *The results of experiment* number 1 are shown in Table 5.2 for llama4, Table 5.3 for gemini-2.5, and Table 5.4 for Mistral Large.

Metric	20	25	30
faithfulness	0.808269	0.805194	0.872692
answer_relevancy	0.887196	0.878201	0.900209
context_precision	0.999999	0.999999	0.999999
context_recall	0.957721	0.960578	0.969761
semantic_similarity	0.905009	0.905333	0.902820
domain_specific_rubrics	3.985714	4.128571	4.228571

Table 5.2: llama-4-maverick with different amounts of retrieved chunks

Metric	20	25	30
faithfulness	0.959336	0.956145	0.948480
answer_relevancy	0.903508	0.899954	0.881290
context_precision	0.999999	0.999999	0.999999
context_recall	0.957721	0.965340	0.969761
semantic_similarity	0.920235	0.919857	0.919034
domain_specific_rubrics	4.585714	4.671428	4.614285

Table 5.3: gemini-2.5-flash-preview-04-17 with different amounts of retrieved chunks

Metric	20 mean	25 mean	30 mean
faithfulness	0.870780	0.889669	0.881136
answer_relevancy	0.916198	0.909478	0.911987
context_precision	0.999999	0.999999	0.999999
context_recall	0.957721	0.965340	0.969761
semantic_similarity	0.904830	0.901914	0.902961
domain_specific_rubrics	4.500000	4.471428	4.457142

Table 5.4: mistral-large-2411 with different amounts of retrieved chunks

Experiment conclusion

In this experiment, `gemini-2.5-flash-preview-04-17` consistently showed strong performance across the different chunk settings. With 25 chunks, it achieved the highest domain_specific_rubrics score (4.67) and high faithfulness (0.956) and semantic similarity (0.920). Model `llama-4-maverick` generally scored lower, particularly in faithfulness (0.80-0.87) and domain_specific_rubrics (3.98-4.23) compared to the other two models. Model `mistral-large-2411` performed well, often close to Gemini, especially in answer relevancy (around 0.91) and domain_specific_rubrics (around 4.45-4.50). Model `gemini-2.5-flash-preview-04-17 had` overall, considering its consistently high scores across metrics, particularly the domain-specific rubrics and faithfulness. Considering that increasing prompt context size raises costs, and given that gemini-2.5-flash-preview-04-17 achieved the highest domain_specific_rubrics score along with comparably high scores on other metrics with 25 retrieved chunks, I will use 25 as the number of retrieved context chunks for my next experiments.

5.2.2 Experiment number 2: Changing LLM used for evaluation

Settings for second experiment:

- Chunking Method: Markdown chunking
- Embedding model: Google text-embedding-004
- Data: All 107 Flowmon markdown files ingested in database, sized 382 KB
- LLMs: gemini-2.5-flash-preview-04-17, llama-4-maverick-17b-128e-instruct-maas, Mistral Large (24.11)
- Number of retrieved context chunks: variable
- LLM used to calculate score for metrics: gpt-4o-mini

For my second experiment, I will be using `gpt-4o-mini` as the LLM used to evaluate results in RAGAs, to see if Gemini did not perform better in the first experiment only because it was also evaluating how good the responses are. I will be setting the number of retrieved context chunks to 25 as it got the best result in domain specific rubrics. *The results of experiment* number 2 are shown in Table 5.5.

Metric	25 lama-4-maverick	25 Gemini 2.5 Flash	25 Mistral Large
faithfulness	0.865412	0.954167	0.936639
answer_relevancy	0.868643	0.881560	0.906636
context_precision	0.999999	0.999999	0.999999
context_recall	0.955714	0.948888	0.962381
semantic_similarity	0.893595	0.919857	0.901915
domain_specific_rubrics	4.828571	4.914285	4.885714

Table 5.5: All models with 25 retrieved chunks

Experiment conclusion

In this experiment, Gemini and Mistral outperformed Llama in faithfulness, answer relevancy, semantic similarity and domain specific rubrics, while they were very comparable in all of the metrics. However, since the price per token of Gemini is much lower, as can be seen in Table 5.1, I will choose Gemini to carry out the next experiments. There are small differences between runs in this experiment, which can be seen in context recall, as even with temperature set to 0 on gpt-4o-mini, results changed a small amount between runs, which did not happen with Gemini 2.5 Flash as the LLM used to calculate score for metrics. This inconsistency also happened before with Gemini 2.0 Flash, so as I mentioned, I will be using Gemini 2.5 Flash as LLM used to calculate the score for metrics for all the other experiments.

5.2.3 Experiment number 3: Chunking methods on Markdown Files

For my third experiment, different chunking methods will be used for the model gemini-2.5-flash-preview-04-17 with the setting that performed the best in the experiment before, to find the best chunking method to use for the main Flowmon dataset. *The results of experiment* number 3 are shown in Table 5.6. Settings for the third experiment:

- Chunking Method: Markdown chunking, Semantic Chunking, Recursive Chunking
- Embedding model: Google text-embedding-004
- Data: All 107 Flowmon markdown files ingested in database, sized 382 KB
- LLMs: gemini-2.5-flash-preview-04-17
- Number of retrieved context chunks: 25
- LLM used to calculate score for metrics: gemini-2.5-flash-preview-04-17

Metric	Markdown chunking	Semantic Chunking
faithfulness	0.956145	0.948465
answer_relevancy	0.899954	0.902465
context_precision	0.999999	0.999999
context_recall	0.965340	0.945912
semantic_similarity	0.919857	0.913380
domain_specific_rubrics	4.671428	4.442857

Table 5.6: All models with 25 retrieved chunks

Experiment conclusion

Markdown chunking outperformed Semantic chunking across most evaluated metrics. Specifically, it achieved higher scores in faithfulness (0.956 vs 0.948), context recall (0.965 vs 0.946), semantic similarity (0.920 vs 0.913), and notably higher in domain_specific_rubrics (4.67 vs 4.44). Semantic chunking only slightly edged out Markdown chunking in answer relevancy (0.902 vs 0.900). For the structured Markdown dataset, leveraging the document structure via Markdown chunking proved more effective than the meaning-based Semantic chunking approach.

5.2.4 Experiment number 4: Chunking methods on Text Files

With RAGAs `TestsetGenerator` I have generated testset that includes 150 question and question-related ground truth for the RFC dataset, which contains 37 text files, with size of 408 KB. There are more RFC files describing official internet protocol standards, but I wanted the datasets to be of a similar size, so I picked only 37 files that had around the same disk size as the Flowmon ADS dataset. To ensure comparable embedding lengths, I found that the average embedding length made from semantic chunking is 1610 characters, and set `chunk_size` for `RecursiveCharacterTextSplitter` to be 1610 with 20% overlap. In this experiment, I will be testing how the application performs for a similar dataset to the Flowmon ADS dataset but in a different format, as well as what are the best chunking options for the text format. *The results of experiment* number 4 are shown in Table 5.7. Settings for the fourth experiment:

- Chunking Method: Semantic Chunking, Recursive Chunking
- Embedding model: Google text-embedding-004
- Documents: RFC Text documentation, consisting from 37 text files, sized 408 KB.
- LLMs: gemini-2.5-flash-preview-04-17
- Number of retrieved context chunks: 25
- LLM used to calculate score for metrics: gemini-2.5-flash-preview-04-17

Metric	Semantic	Recursive
faithfulness	0.986529	0.981751
answer_relevancy	0.896361	0.900307
context_precision	0.953333	0.959999
context_recall	0.951777	0.945912
semantic_similarity	0.911525	0.914743
domain_specific_rubrics	4.846666	4.800000

Table 5.7: All models with 25 retrieved chunks

Experiment conclusion

The results show that the application performs very well when handling plain text documents. Both Semantic and Recursive chunking methods yielded high scores across all

metrics. Faithfulness was very high (above 0.98) for both methods, showing the model reliably based its answers on the retrieved text content. Answer Relevancy was high (around 0.90), confirming the answers effectively addressed the questions. Context Precision and Recall were also high (around 0.95), indicating the retrieval process successfully identified relevant document sections. Semantic Similarity scores (above 0.91) suggest the generated answers closely matched the meaning of the ground truths. The Domain-Specific Rubrics scores were excellent (above 4.80 out of 5), demonstrating high overall quality of the responses according to the defined criteria.

5.2.5 Experiment number 5: Using reranking

For my fifth experiment, I used `semantic-ranker-default-004` from Google to rerank the 25 retrieved context chunks based on their relevancy and put the most relevant chunks into the context first. *The results of experiment* number 5 are shown in Table 5.8. Settings for the third experiment:

- Chunking Method: Markdown chunking
- Embedding model: Google text-embedding-004
- Data: All 107 Flowmon markdown files ingested in database, sized 382 KB
- LLMs: gemini-2.5-flash-preview-04-17
- Number of retrieved context chunks: 25
- LLM used to calculate score for metrics: gemini-2.5-flash-preview-04-17

Metric	Without reranking	With reranking
faithfulness	0.956145	0.993673
answer_relevancy	0.899954	0.904894
context_precision	0.999999	0.999999
context_recall	0.965340	0.980952
semantic_similarity	0.919857	0.916961
domain_specific_rubrics	4.671428	4.857142

Table 5.8: All models with 25 retrieved chunks

Experiment conclusion

The introduction of a reranking step using semantic-ranker-default-004 yielded significant improvements in the overall quality and reliability of the RAG system's outputs. Every metric improved, except context_precision which stayed the same and semantic_similarity with very minor decrease, which is likely negligible given the substantial gains in faithfulness and overall quality.

5.3 Application Deployment

The application is hosted on the Google Cloud Platform, providing easy access and scalability of all its parts. The application is hosted on the Cloud Run service, which is a

serverless platform that provides automatic scaling. It is deployed with a simple shell script that builds container from specified folder containing python script and text file containing python packages requirements. It also specifies the service name, serverless platform, region, authentication, and connects to the database instance.

```
gcloud run deploy ${SERVICE_NAME} \
    --source ./app \
    --platform managed \
    --region ${REGION} \
    --allow-unauthenticated \
    --add-cloudsql-instances=${INSTANCE_CONN_NAME}
```

The database is also hosted on the Google Cloud Platform, using the Cloud SQL service. It is connected to the same virtual network, so it can communicate with the application.

5.4 Evaluation program Deployment

I am running the evaluation on both my machine and on the Google Cloud Platform. On the cloud the evaluation is run using Batch service, which is a service to schedule, queue, and execute VM scripts and containerized batch jobs. First, a Docker image is made according to a Dockerfile and pushed to Google Artifact Registry.

```
docker build -t $IMAGE_URI .
docker push $IMAGE_URI
```

```
FROM python:3.13-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY main.py .
COPY rag_functions.py .
CMD ["python", "main.py"]
```

Then the batch job gets submitted and executes on the Google Cloud Platform.

```
gcloud batch jobs submit \
  --location=europe-west4 \
  --config=job_definition.json
```

It also requires a `job_definition.json` file which specifies what image on Artifact Registry to build the container from, what local variables to set for the container, the needed compute and storage resources, and how to use them.

5.5 Summary

This chapter detailed the development of an evaluation tool and evaluation steps to assess the performance of the developed RAG system for technical documentation analysis. Using the RAGAs framework and synthetically generated test sets for both Flowmon ADS markdown dataset and RFC plain text dataset describing official internet protocol standards, the goal of evaluation was to find optimal setting for both. Outcomes of the evaluation are described in the conclusion of this thesis. This chapter also covered deployment of the application and evaluation scripts on Google Cloud Platform.

Chapter 6

Conclusion

This thesis explored the application of Transformer neural networks, specifically through the LLMs and RAG techniques, for analyzing technical documentation. The primary goal was to develop and evaluate a system capable of efficiently answering user questions and summarizing content from technical documents, making it easier to navigate the large amount of documentation. The work covered the state-of-the-art in relevant NLP techniques, investigated the formats of provided technical documents, and proposed effective pre-processing steps. A web-based application was designed and implemented, with functionalities such as choosing different LLMs, uploading and processing documents on which user can then perform Q&A and text summarization tasks. A significant portion of this work was dedicated to evaluation using the RAGAs framework. Experiments systematically compared different LLMs, chunking methods, the number of retrieved context chunks, effects of reranking and performance on different input document formats. Key findings from the evaluation indicated that for the structured Markdown dataset, the Markdown chunking method combined with the gemini-2.5-flash-preview-04-17 model and retrieving 25 context chunks provided a strong balance of performance on the chosen metrics and cost-efficiency. The system also demonstrated very good performance on plain text RFC documents using both Semantic and Recursive chunking. Based on these results, the deployed web application was configured to use the `MarkdownHeaderTextSplitter` for ingested Markdown files and Semantic Chunking for plain text files, combined with the gemini-2.5-flash-preview-04-17 model, retrieving 25 context chunks and context reranking for both the formats.

The main contribution of this thesis lies in the practical implementation and thorough evaluation of a RAG system specifically tailored for technical documentation. It provides empirical evidence on the effectiveness of different configurations for this specific domain and document types. The developed application showcases significant potential for real-world deployment to help developers or users quickly understand parts various technical projects. It can help quickly find specific information within extensive documentation or generate concise summaries of lengthy documents for quick reviews.

However, future possible improvements could be made. Future work could involve testing and optimizing the system's performance with much larger documentation, as the current main dataset consists of 107, sized 382 KB. Another area that could be expanded is formats of input documents as this thesis focuses only on two formats, markdown and plain text. Also, all evaluation is done by one framework, so future improvement could be human evaluation or use of more evaluation tools.

Bibliography

[1] Khurana, D.; Koli, A.; Khatter, K. and Singh, S. Natural language processing: state of the art, current trends and challenges. *Multimedia Tools and Applications*. Springer Science and Business Media LLC, july 2022, vol. 82, no. 3, p. 3713–3744. ISSN 1573-7721.

[2] Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L. et al. Attention is all you need. In: *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010. NIPS'17. ISBN 9781510860964.

[3] Joshi, C. Transformers are Graph Neural Networks. *The Gradient* online. September 2020. Available at: https://thegradient.pub/transformers-are-graph-neural-networks/. Accessed: 2025-04-24.

[4] Pennington, J.; Socher, R. and Manning, C. D. GloVe: Global Vectors for Word Representation. In: *Empirical Methods in Natural Language Processing (EMNLP)*. 2014, p. 1532–1543.

[5] Brown, T. B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. et al. Language Models are Few-Shot Learners. In: Larochelle, H.; Ranzato, M.; Hadsell, R.; Balcan, M. and Lin, H., ed. *Advances in Neural Information Processing Systems 33 (NeurIPS 2020)*. Curran Associates, Inc., 2020, p. 1877–1901.

[6] Treviso, M.; Lee, J.-U.; Ji, T.; Aken, B. van; Cao, Q. et al. Efficient Methods for Natural Language Processing: A Survey. *Transactions of the Association for Computational Linguistics*. Cambridge, MA: MIT Press, 2023, vol. 11, p. 826–860.

[7] Kshirsagar, A. Enhancing RAG Performance Through Chunking and Text Splitting Techniques. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology*, september 2024, vol. 10, p. 151–158.

[8] Tarakad, N.; Yu, P.; Samdani, R. and Merrick, L. Long-Context Isn't All You Need: Impact of Retrieval and Chunking on Finance RAG. *Snowflake* online. March 2025. Available at: https://www.snowflake.com/en/engineering-blog/impact-retrieval-chunking-finance-rag/. Accessed: 2025-04-24.

[9] Han, Y.; Liu, C. and Wang, P. A Comprehensive Survey on Vector Database: Storage and Retrieval Technique, Challenge. *CoRR*, 2023, abs/2310.11703.

[10] Karabiber, F. Cosine Similarity. *Learndatasci*. Available at: https://www.learndatasci.com/glossary/cosine-similarity/. Accessed: 2025-04-24.

[11] Mortaheb, M.; Khojastepour, M. A. A.; Chakradhar, S. T. and Ulukus, S. *Re-ranking the Context for Multimodal Retrieval Augmented Generation*. January 2025. Available at: https://arxiv.org/abs/2501.04695.

[12] Chen, B.; Zhang, Z.; Langrené, N. and Zhu, S. Unleashing the potential of prompt engineering for large language models. *Patterns*, 2025, p. 101260. ISSN 2666-3899.

[13] Es, S.; James, J.; Espinosa Anke, L. and Schockaert, S. RAGAs: Automated Evaluation of Retrieval Augmented Generation. In: Aletras, N. and De Clercq, O., ed. *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*. St. Julians, Malta: Association for Computational Linguistics, March 2024, p. 150–158.

[14] Reimers, N. and Gurevych, I. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, November 2019.