Real-Time Chat Bot for Live-stream Content

Bachelor Thesis

Submitted to
IMC University of Applied Sciences Krems

Bachelor Programme Informatics

by

Patrik Palenčár

for the award of academic degree
Bachelor of Science in Engineering (BSc)

under the supervision of

Dr. Rubén Ruiz Torrubiano,
MSc, MBA Leto Baxevanaki

Submitted on 29.04.2025

Declaration of honour

Patrik Palenčár
29.04.2025

ABSTRACT

This thesis aims to find a way to create a chat bot for live-stream video content by comparing retrieval augmented generation and prompt engineering approaches combined with Large Language Model (LLM). The core objective was to design an effective method to provide relevant context from live-stream to a LLM in real time, using transcribed video captions as the primary input source.

During the study, various chunking and retrieval strategies were implemented and evaluated. Multiple configurations were tested across several LLMs, with a particular focus on smaller, cost-efficient models that are better suited for real-time performance.

Keywords: Retrieval Augmented Generation, Prompt engineering, Large Language Models, Chatbot, Live-RAG

ACKNOWLEDGEMENTS

This is an **optional** page. Use your choice of paragraph style for text on this page. Usually, this space is for thanking your supporters and getting emotional about how grateful you are to everyone.

Contents

List of Tables

List of Figures

Chapter 1
Introduction

Recent advancements in the field of artificial intelligence have unlocked many topics for research. The development of large language models (LLMs), like Generative Pre-trained Transformers (GPT) [4], has made AI the most talked about area in modern information technology. This thesis explores the application of LLMs in scenarios that need real-time integration of external knowledge, specifically by the use of Retrieval-Augmented Generation (RAG) and prompt engineering. The study will specifically focus on how can we use RAG and prompt engineering with LLMs to create a chat bot capable of communicating about sports video live stream. The key objective is to implement real-time context retrieval to create a chat bot, which will be researched with the aim of developing a state-of-the-art solution. There are several limitations that I might be facing during this research such as the need to provide context in real time without access to future data or any problems that could affect the final chat bot.

1.1 Motivation

The motivation behind this thesis is based in improving the viewer experience during live sports broadcasts by integrating artificial intelligence technologies such as Large Language Models (LLMs).

Live sports content is fast paced, and viewers often face moments where they are forced to miss short segments of the game due to many different interruptions, such as stepping away from the screen or answering a call. In these situations, being able to ask an intelligent assistant capable of providing truthful and relevant information about the match in real time can significantly improve the overall watching experience. It would make the viewer feel like watching with a friend.

Live audience often seek additional information about players, teams, statistics, or specific events occurring during the game. Rather than manually searching through various sources, viewers would benefit from a conversational interface (chat bot) that allows them to ask questions about the match directly while watching. This kind of interaction not only makes it easier for people to get the information they want, but also connects understanding of additional knowledge and watching in a smooth way.

This research goes over the practical use case as it contributes to the field of real-time natural language processing and conversational AI. It looks at the integration of Retrieval-Augmented Generation and prompt engineering in a time constrained streaming context, creating something new in comparison to many static LLM use cases. By building and evaluating a chat bot capable of operating on live streamed content, this study aims to explore how modern AI models can be tweaked for time sensitive, dynamic data scenarios.

Lastly, the goal is to build a tool that is both technically useful and easy to use chat bot. Which gives viewers a smarter, and more fun way to follow live stream content.

1.2 Research Questions

- How can context from live-stream content be provided to a Large Language Model (LLM) in order to build an accurate chat bot?
  - What are the most effective hyperparameters for Retrieval-Augmented Generation and prompt engineering in the context of a live-streaming chat bot?
- How does Retrieval-Augmented Generation compare with prompt engineering in optimizing the chat bot's performance for live video stream applications?
  - What are effective methods to evaluate and compare the performance of RAG and prompt engineering in this use case?

1.3 Research Method

My research aims to explore techniques for providing context to LLM for live video streaming content to create a chat bot. For this kind of topic quantitative research method will be used. With this approach, I can conclude that findings of the study are based on objective and measurable outcomes.

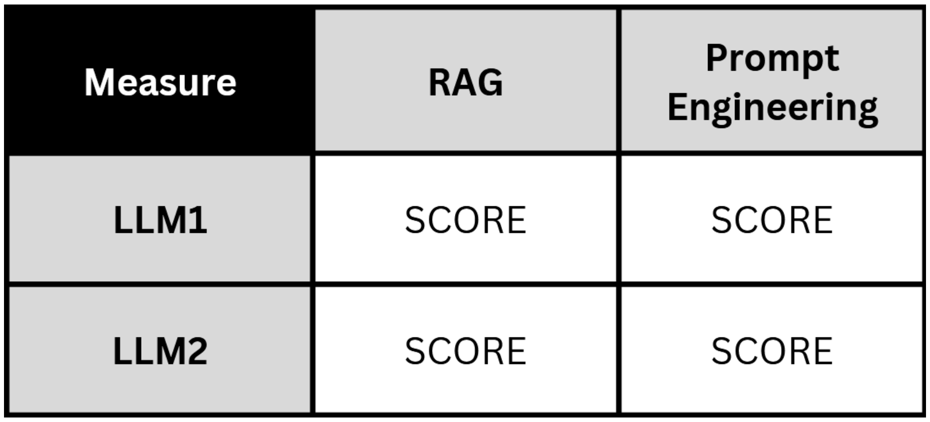
The data used for the research, consists of commentary captions from replays of live sports games, simulating real data flow. Technique to get these data is a speech to text technology that converts the live stream commentary to text in real time.

Next, to answer the research questions, the study will focus on how to evaluate the chat bot as accurately as possible. For this, it will examine different objective evaluation methods, such as statistical comparisons of system performance or ground truth analysis

[5]. It will create a solid ground for comprehensive analysis between RAG and prompt engineering approach. And it will also help determine which approach is most suitable for the use case.

After the best-performing parameters for the chat bot are researched with LLM GPT-4o-mini [6]. Its performance will be further evaluated using an additional LLM. These LLMs will be chosen based on current LLM benchmarks for the informational chat bot LLMs. Important measures for choosing a LLM for this use case will be further researched in upcoming chapters. The responses generated by both LLMs and the two approaches will then be analyzed and compared in contingency tables as shown in Figure 1.1.

Lastly, the thesis will address the best approach to implementing live-rag as a service that can be used by other developers. To achieve this, a server with different APIs will need to be built.



Measure	RAG	Prompt Engineering
LLM1	SCORE	SCORE
LLM2	SCORE	SCORE

Figure 1.1: Comparison of the two LLMs with RAG vs Prompt Engineering approaches.

1.4 Thesis Structure

This thesis is divided into several parts to guide the reader from background knowledge to the final evaluation of the chat bot.

The structure of this thesis begins with an explanation of needed technologies such as Large Languafe Models (LLM's), Retrieval-Augmented Generation (RAG), and prompt engineering.

Then, the thesis takes a closer look at the parts that were needed for working with live-stream content. This includes how live video captions are handled, how text is prepared and divided, and how the chat bot finds the right information from the context.

After the background and technical explanation, the thesis continues with the step-by-step process of building the chat bot. This includes details about the tools and methods used, and how they were put together to create a working system. It shows how the live data is received, processed, and turned into answers by the chat bot.

In the last part, the chat bot is evaluated and its performance is measured. The results from using RAG and prompt engineering are compared, and the best options for the live-stream chat bot are discussed. This will help answer the research questions. The thesis ends with discussion and by talking about what could be improved and what future work can be done.

Chapter 2
Background

The field of AI is currently one of the most active in research advancements, and new scientific papers are being released daily. This chapter introduces the technologies essential for the research and explains why they are important for this study.

2.1 Related Technologies

The technologies used in the research will include the Python programming language, specifically the open-source library LangChain, which is designed for working with Large Language Models (LLMs). Furthermore, two different LLMs will be used and compared in the research. To implement the result of the research as a service, the study will use another Python library, Flask, to create the server. For evaluation purposes, this thesis will leverage the Ragas Python library, which is specifically designed for evaluating Retrieval-Augmented Generation (RAG) pipelines. Lastly, for data visualization will be used Plotly visualization library with Seaborn for modern looking visualizations.

2.2 Transformers

Understanding the transformer architecture is needed in the context of this research, as the study relies on models built with this technique.

Transformers are the revolutionary technology that enabled transformer based LLMs to gain bigger visibility in community of researchers and developers. Introduced in the paper "Attention is All You Need"(2017) [1], transformers presented a paradigm shift in how parts of data are processed in natural language processing (NLP) tasks. Differently from previous architectures such as recurrent neural networks (RNNs), which processed data sequentially and were limited by vanishing gradient issues, transformers allowed for parallel processing of sequence data. This significantly improved efficiency and scalability.

The main part of the transformer architecture is a technique called self attention. This enables the model to look at all the words in a sentence and decide which ones are most important for understanding each word. Because of this, the model can understand connections between words even if they are far apart in the sentence, which helps it better understand the overall meaning compared to older models.

The architecture of transformers is build of an encoder and decoder structure, however many modern applications, such as BERT [7] and GPT [8], use only the encoder (BERT) or decoder (GPT) for specific tasks. Each encoder or decoder layer has several parts: A multi-head self-attention mechanism that helps the model focus on the most relevant words in the sentence. A small neural network (feed-forward network) that processes each word individually after the attention step and additional techniques such as layer normalization and residual connections, which help the model learn better and train more efficiently.

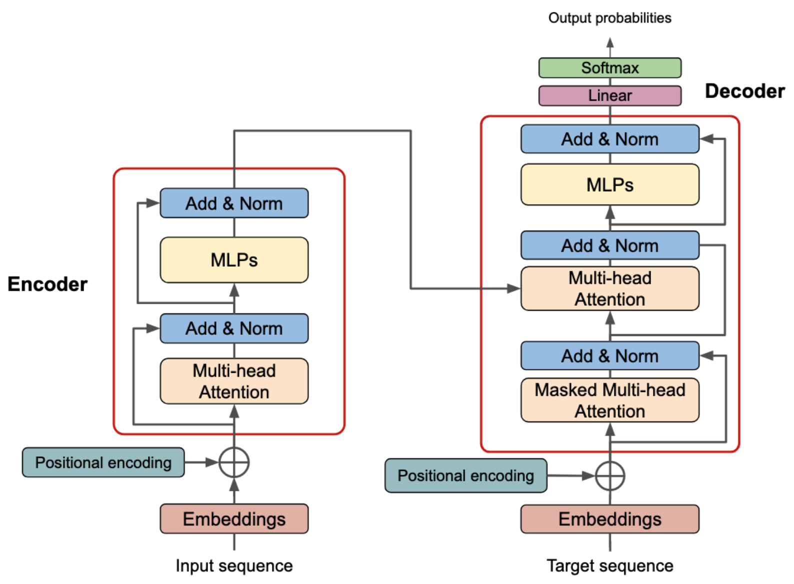


Figure 2.1: Transformer model architecture. Adapted from [1]

One problem with self attention is that it does not care about the order of the words in the sentence. Therefore a key innovation of transformers is the use of *positional encoding*, which provides information about the position of words in the sequence. This allows the model to maintain an understanding of word order.

Transformers have become the foundation for many state of the art models in NLP, such as image processing, speech recognition, and protein folding. Their modularity, scalability, and ability to generalize across tasks have made them a base of modern AI systems.

2.2.1 Large Language Models

The invention of transformers enabled the creation of transformer based LLMs which are also used in my thesis. These models are trained on a huge amount of text data, from which LLMs learn to understand and generate human language. Thanks to the transformer architecture, LLMs are faster and more accurate than older models. Thanks to this, many new use cases started to appear, including the topic of this study.

LLMs are trained by showing them lots of examples of human text. Out of this text the model learns how words and sentences are usually built. The results of this training are saved in the form of model parameters, which are just numbers that help the model decide what word to generate next. The more parameters a model has, the more information it can store and use and it also needs more computing power to run.

Model size: LLMs can have from millions to even hundreds of billions of parameters. For example, GPT-2 has 1.5 billion parameters, while GPT-4 is believed to 1.7 trilion parameters. A bigger model usually gives better results, but it is also more expensive and slower to use.

Model specialization: The running cost of LLM gets higher as the vague LLMs get more parameters. There are therefore many LLMs that are trained for specific tasks like solving math problems, writing code, or working with images. Since this research is focused on text, the model used must be good at language understanding and generation. So the model must be specialized for text-based tasks like chatbots, summarization, or question answering. Additionally, the LLM used for this study should be smaller and efficient, even if it is not as smart as a big, slow one.

Tokenization: LLMs do not read text word by word like humans. Instead, they split the text into smaller parts called tokens. A token can be a word, part of the wort or just a letter. The model understands and processes everything using these tokens. Understanding tokenization is useful when you want to know how much text can fit into the context window or how much the model will cost to run.

Context window: One important parameter of LLMs is the size of their context window. This shows how much text the model can process at once. The more context is provided to the LLM the better responses it tends to give. But this also makes the LLM slower to process the context. A large context window is important for the prompt engineering approach of this research as we need the LLM to process massive prompt. Good thing is that the most of the newest LLMs have big context windows of 100 000 tokens and more.

Temperature: The next parameter of the LLM is temperature. This setting controls how random and creative the model's answers are. A lower temperature (like 0.1 or 0.2) makes the model give more safe and focused answers. A higher temperature (like 0.8 or 1.0) makes the model more creative, but it can also start making things up and hallucinate. In this research, the chat bot needs to stay close to facts, so lower temperature settings are more useful.

LLM usage and token pricing: Most public LLMs (like GPT models from OpenAI) are not free. They charge users based on the number of tokens used. For example, asking a very long question or giving a lot of tokens context costs more than asking something short. Also, different models have different token prices depending on their size and quality. This is important when deciding which model to use in production or for experiments.

2.3 Chat Bots

Chat bots are computer programs designed to simulate conversation with human users. They are used in many areas today, from customer support to personal assistants or helping users interact with websites or apps. A chat bot usually understands what the user writes and tries to respond in a helpful way.

Traditionally, many chat bots are so called straightforward chat bots. This means the LLM used for the chat bot receives just the user's message (query) together with a small text prompt called a context prompt. A context prompt includes written instructions or rules that help guide the LLM on how to answer. These prompts can include things like tone, personality, or specific information about the topic. They are often improved using a technique called prompt engineering [5]. Which means carefully designing the prompt to get better answers from the LLM.

However, there are some limits to this straightforward approach. These chat bots can only work well if all the information they need is already in the model or the prompt. If they need to answer based on data that changes often or is not known in advance (like live sports commentary), this approach becomes difficult.

This thesis focuses on developing a chat bot that goes beyond the basic "straight forward" method. To handle more complex or dynamic information, one more step of retrieval-augmented generation (RAG) or advanced prompt engineering techniques is added. With RAG, the chat bot can first look up information from external sources, then use the information (called context) to answer the user's question. This makes the chat bot more flexible and useful for live-stream use cases. More about how RAG work is described in the next section.

2.4 Retrieval-Augmented Generation

The Retrieval-Augmented Generation (RAG) introduced in [2], is a process that enhances responses of LLMs by providing custom context. Through RAG, LLMs can effectively use specific data that may not be publicly accessible or that the model was not initially trained on. Common data for RAG use cases are for example confidential data in a company or dynamically updated data — such as the data used in this thesis. This allows LLMs to generate responses with contextually relevant information to specific use cases. By augmenting the input with relevant external information, RAG strengthens the LLM's understanding and reduces the inaccuracies or irrelevant outputs, commonly known as “hallucinations”. These might otherwise occur without contextual guidance.

RAG is useful in situations where the base model is not allowed or able to learn all the information in advance. This can include live updates, private data, or large collections of documents. Instead of fine-tuning the LLM with this data (which can be expensive or slow), the data is stored separately and only retrieved when needed.

In simple terms, the process of RAG has two main parts:

Retrieval – finding the most relevant documents or pieces of information from a database.

Generation – giving this information to the LLM along with the user's query so that it can generate a more accurate answer.

This creates a dynamic system that instead of using only the model's pre-trained knowledge becomes more like a smart assistant that looks something up before answering. Because of this, RAG is very useful in real-time applications like the live sports chat bot researched in this thesis.

To make this work, the external information (such as transcripts or documents) needs to be prepared in a special way. This process includes three key steps:

Chunking – splitting long texts into smaller parts that are easier to search through.

Embedding – converting each chunk into a vector (a list of numbers) so that the system can search them and understand their meaning.

Retrieval – finding the most similar chunks based on the user's question and generating the answer based on it.

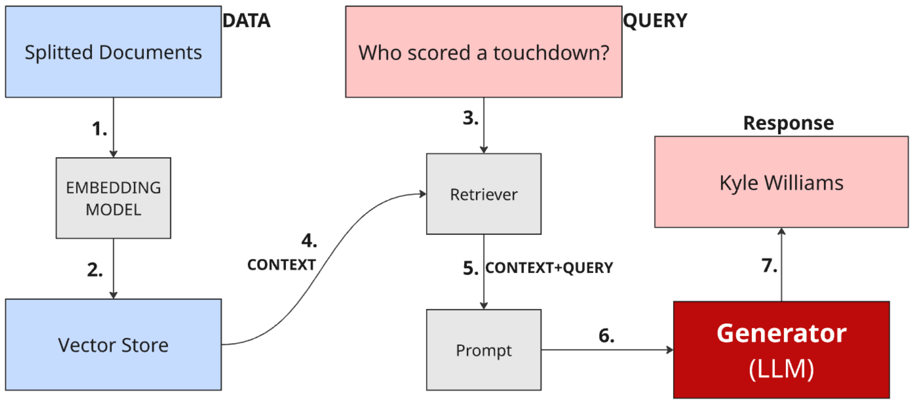


Figure 2.2: RAG Architecture of this Study Adapted from [2]

The following sections will explain each of these steps in more detail.

2.4.1 Chunking

Chunking methods discussed in [9] serve as rules to split the text into parts that can later be searched and retrieved. Chunking is an important step in RAG systems because Large Language Models (LLMs) have a limited context window, meaning they cannot process very long texts at once. And if the provided context is long the LLM looses its efficiency and has higher cost. By dividing documents into smaller parts, the system can later choose only the most relevant ones to include in the final prompt.

In this thesis, the use case deals with shorter but information dense textual data, such as live sports transcripts. This is different from most RAG pipelines which usually work with long articles or documents. Because of that, chunking parameters need to be carefully selected and tested.

Important parameters include chunk size (how many tokens or sentences are in one chunk) and chunk overlap (how much text is repeated between neighboring chunks). These affect both the retrieval quality and performance of the chatbot. Finding the right balance is important — too short chunks may miss context, while too long ones may be not effective or reduce precision.

2.4.2 Embedding

Embedding is the process of turning chunks of text into numbers (vectors) so that a retrieval method can understand and compare them. Each chunk is converted into a vector where similar texts get vectors that are close together in the vector space. This is very useful for searching. For example, when a user asks something, the retrieval method can also turn the question into a vector and then find the most similar chunks based on distance between the vectors.

In RAG, embeddings are used for both storing the chunks and for comparing them to the query. A good embedding model makes sure that chunks with similar meaning are placed close to each other, even if the wording is different.

There are different models available for generating embeddings. These models are usually trained on large amounts of text to understand the meaning behind words and sentences.

Choosing the right model depends on the use case. Some models focus on speed, others on precision. For this thesis, where the texts are short and fact-rich (like sports commentary), the embedding model must be able to capture fine details and return highly relevant chunks.

In short, embedding is a key step that connects the question from the user to the correct pieces of information in the data.

2.4.3 Retrieval

Retrieval methods are described in [9]. Retrieval is a part of RAG that searches for text chunks that have a similar meaning to the user query. Chunks of text are retrieved by using a retrieval method and then provided to the LLM together with the question and prompt as context. This step is important because the LLM does not have access to all data and relies on these selected chunks to generate a good answer.

There are different retrieval methods to chose from. The selection of the right retrieval method depends on the data and the use case.

In this thesis, similarity search is used for the retrieval part of RAG. This method compares the meaning of the user query with the meaning of the stored text chunks. It finds the chunks that are most similar to the question and sends them to the LLM as context for answering.

Similarity search works by comparing vector embeddings. Both the query and the text chunks are turned into vectors using an embedding model. These vectors are then compared using a similarity metric, such as cosine similarity. The more similar the vectors, the more relevant the chunk is to the query.

This method is especially useful when the question and the text do not use the exact same words but still talk about the same topic. This is common in natural language and makes similarity search better than simple keyword matching.

Also, retrieval is usually the step where ranking happens. The selected chunks are ordered based on how relevant they are to the question. This ranking plays a big role in the quality of the final answer, as the LLM receives only a limited number of top-ranked chunks due to the context window size.

2.5 Prompt Engineering

As different LLM-powered agents and chat bots began to appear, the need to navigate and customize their responses was increasingly needed. LLMs, particularly earlier versions, often produced outputs that were vague, generic, or factually inaccurate [10]. To minimize these issues and improve the quality of generated responses, researchers introduced techniques now known as prompt engineering techniques.

Prompt engineering is the process of designing and optimizing the input prompts given to an LLM, in order to have its output in a desired way. This includes not only the writing of the query itself but also additional instructions or contextual information that can influence how the model interprets the task.

As discussed in [5], there are numerous prompt engineering strategies. These techniques sound simple and straightforward, but they are effective. Some of these are: rephrasing the prompt for clarity, explicitly stating the desired format of the response, assigning the LLM a specific role (e.g., "Act as a sports analyst"), or using techniques like resampling to generate multiple outputs and choose the best one. More advanced approaches may also involve few-shot prompting, chain-of-thought prompting, or the use of external tools for dynamic prompt generation.

In this thesis, prompt engineering is an essential part of guiding the chatbot's behavior, especially in combination with retrieved context from the RAG process. Since the effectiveness of an LLM can be heavily influenced by the way it is prompted. A careful prompt design has a crucial role in securing that the chat bot's responses are relevant and accurate.

2.6 Evaluation

To assure the quality and reliability of the chat bot developed in this thesis, a powerful and systematic evaluation framework is needed. Without clearly defined evaluation metrics, it would be difficult to objectively assess whether the system meets its intended goals, such as providing accurate, contextually relevant, and trustworthy answers. Evaluation enables both the identification of system strengths and the exposure of weaknesses.

There are several approaches to evaluating a chat bot, from subjective manual human annotation to automated objective metrics based on language similarity or information retrieval performance. Since this thesis focuses on a chat bot built using Retrieval-Augmented Generation (RAG), the evaluation is structured around both the generation and retrieval components. The selected evaluation metrics align with the key goals of such a system: retrieving meaningful context, generating faithful and relevant answers, and minimizing hallucinations.

To be able to create results with all these metrics there is need for some data to compare the RAG outputs. For this it is possible to create so called "Ground Truth Question-Answer Pairs". These represent the truthful standard against which the RAG system's outputs can be compared. Ground truth QA pairs serve two primary purposes: they provide a reliable benchmark for evaluating the result metrics of the chat bot's responses, and they help identify weaknesses in both retrieval and generation.

In the context of this thesis, ground truth QA pairs are especially important for assessing the semantic similarity and answer relevancy of the system-generated responses. Without them, it would be difficult to perform objective, repeatable, and meaningful evaluations.

There is discussed in paper [11] that for precise and reliable evaluation RAG system needs at least 50 ground truth questions and answer pairs. The number can be much bigger depending on the use case and how big the document data set is.

But how do we define and select these ground truth QA pairs? To answer this, there are several strategies for generating ground truth pairs:

- **Manual Annotation:** Knowledgeable users read through the documents and create questions based on specific factual information. They then write precise answers that are considered 100 percent accurate.

- **Document-Derived Questions:** Questions can be semi-automatically generated from the documents used in the retrieval pipeline. Techniques such as answer aware question generation can be used to create realistic pairs from existing knowledge. After questions generation domain experts create factually correct answers to these questions.

There are also other techniques but understanding of these two is the most important for this study.

Lastly, for the evaluation of different metrics, naturally a LLMs is used. There can be multiple LLM calls for each question and answer pair per each metric what can make the evaluation really expensive.

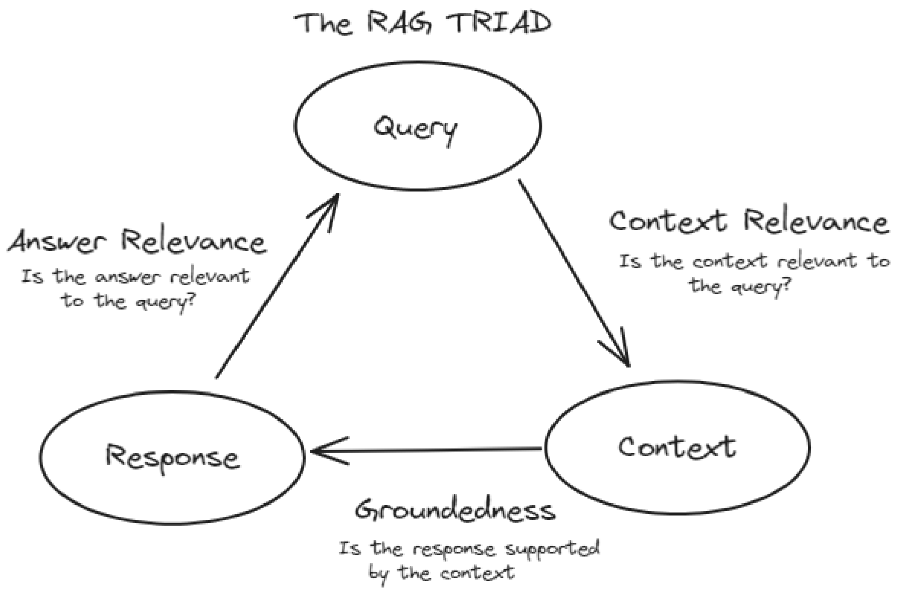

The RAG TRIAD
Query
Context Relevance
Is the context relevant to the query?
Answer Relevance
Is the answer relevant to the query?
Response
Context
Groundedness
Is the response supported by the context

Figure 2.3: Evaluation diagram from blog about RAG evaluation [3]

The following subsections describe the evaluation metrics used in this research:

2.6.1 Context Precision

This metric evaluates how accurately the retrieved documents align with the information required to answer the query. High context precision indicates that the retriever component successfully identifies the most relevant chunks from the knowledge base, reducing noise and increasing the LLM's ability to generate accurate responses.

2.6.2 Faithfulness

Faithfulness means how closely the answer follows the information from the retrieved documents. In simple words, it checks if the response is making things up or if it is sticking to the facts that were actually found. If the answer is based only on the provided context, it is considered faithful.

2.6.3 Answer Relevancy

This criterion checks how well the answer responds to the user's question. Even if the answer is based on the context, it should still be clearly connected to the original question. We can check this by comparing it to correct answers (ground truth questions and answers).

2.6.4 Context Retrieval

Context retrieval evaluation looks specifically at how effective the retriever is in selecting the most appropriate documents. This can be measured using numbers (like recall or precision), or by looking at what was retrieved and comparing it to the ideal context that was prepared in advance.

2.6.5 Semantic Similarity

Semantic similarity measures how close the meaning of the generated answer is to a ground truth answer. This is often computed using embedding based similarity scores (cosine similarity using sentence embeddings). It is not always perfect, but it provides an automated way to approximate answer quality.

Chapter 3
Related Work

This chapter reviews the existing literature and examines the current state of use of RAG and prompt engineering with LLMs in both research and practice. It aims to provide a comprehensive understanding of the advancements and challenges in technologies related to this study. It provides an important foundation for addressing the research questions in this thesis. There are three fundamental concepts critical to understanding upcoming research content: Large Language Models (LLMs), Retrieval augmented generation (RAG), and prompt engineering. This chapter will review the existing literature on each concept in detail.

3.1 Academic Research

Since the topic of the thesis consists of multiple connected parts, it's helpful to explore key research papers that contribute to the understanding of each part.

3.1.1 Chat bots

The idea of chat bots and artificial intelligence was first mentioned by Alan Turing in 1950. He asked a question if a computer program can 'think' and artificially reply in a human like way. This was called the Turing test [12].

The first form of a chat bot, called ELIZA appeared in the years 1964 and 1966 [13]. It was designed to act as a psychotherapist. Since the hardware back then was very limited the technology behind this chat bot was simple. ELIZA worked on identifying keywords in user input, matching them to pre-programmed patterns, and generating responses by rephrasing the input. This gave an illusion of understanding, even though it had no real idea what the user was writing [14].

Later in 1988, another chat bot called Jabberwacky was developed. It was one of the first bots to use some form of artificial intelligence. It worked by remembering past conversations and trying to respond with context based on them. Still, it was far from perfect and often gave slow or incorrect answers [15].

These chat bots improved as the years went by, and they started to be used as a real world products by companies. . Around 2001, chat bots were built into messaging platforms and used for things like customer support, where they helped users by pulling answers from a database. Then, in the early 2010s, big tech companies introduced voice assistants like Siri, Google Assistant, and Alexa. These voice-based chat bots used speech recognition and web services to answer basic questions [16]. Over time, these assistants became smarter thanks to natural language processing, artificial intelligence, and even on-device learning [17].

However, these earlier voice assistants still lacked deeper understanding. In the context of live video streaming, creating such a chat bot wouldn't be possible with these technologies, because the content of a livestream is unpredictable. This all started to change with the introduction of transformers, a new model architecture introduced in the paper Attention is All You Need [1]. Transformers made a huge difference in how machines understand and generate language. They became the building blocks for today's Large Language Models (LLMs), which are the core of modern chat bot systems like ChatGPT.

These improvements opened the door to more intelligent and flexible chat bots,especially those that can work with external knowledge using techniques like Retrieval-Augmented Generation (RAG). That shift from static, rule-based bots to context aware, real-time LLMs is what made projects like this thesis possible.

3.1.2 Attention is All You Need

A major turning point in natural language processing came with the paper Attention is All You Need by Vaswani et al. in 2017 [1]. In this paper, the authors introduced the transformer architecture, which replaced older methods like RNNs and LSTMs. The big idea was that instead of processing words in sequence, transformers look at all words at once using something called self-attention. This allows them to better understand the full meaning of a sentence or paragraph and handle longer-range dependencies.

Thanks to this new architecture, models could be trained much faster and could handle much more data. This also set the foundation for the large language models we use today. Without transformers, modern systems like GPT or Claude wouldn't exist.

3.1.3 GPT – Generative Pre-trained Transformer

Based on the transformer architecture, OpenAI introduced GPT (Generative Pre-trained Transformer) [4] 2018, starting with GPT-1 in 2018. Over time, each version grew bigger and more capable. The latest, GPT-4 [6], can understand and generate high-quality language in many domains. GPT models are trained in two main steps: first on huge amounts of text (pre-training), and then fine-tuned on more specific tasks (like answering questions or summarizing text).

GPT models are also autoregressive, which means they generate one word at a time by predicting the next most likely word, making them great at continuing conversations or writing responses. Because of this structure, GPT models are widely used in chatbots and other LLM applications today.

3.1.4 Retrieval Augmented Generation

Even though GPT and similar models are strong, they still have a problem: they don't know about new information after their training cutoff and can "hallucinate" facts. To solve this, the paper Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks by Lewis et al. (2020) proposed a method called RAG [2].

RAG combines the power of LLMs with external knowledge. It works by first retrieving documents from a database that are relevant to a user's question and then generating a response based on those documents. This makes the answers more accurate, grounded in real data, and up-to-date. The original paper used dense retrievers and BART-style generators, but modern versions use even stronger models such as GPT-4 or Claude [18].

The study [10] describes a step by step guide for the development of enterprise RAG pipelines. This practical approach outlines this study's solid path to follow. That is interesting for this thesis as the final product is supposed to be production ready.

The paper [5] provides a detailed analysis of prompt engineering techniques, which are critical for customizing the responses of LLMs. It explores different strategies for constructing prompts, optimizing their structure, and adapting them to improve LLM performance. This is relevant to this thesis, as effective prompt engineering is needed for creating an intelligent live RAG chatbot. The insights from this paper can help customize the structure of prompts to deliver accurate responses.

3.1.5 Prompt Engineering

As LLMs got better, people realized that how you give them the information, how you "prompt" them, makes a huge difference. This led to a new field called prompt engineering. The paper [5] provides a detailed analysis of prompt engineering techniques, which are critical for customizing the responses of LLMs. It explores different strategies for constructing prompts, optimizing their structure, and adapting them to improve LLM performance. This is relevant to this thesis, as effective prompt engineering is needed for creating an intelligent live RAG chat bot. The insights from this paper can help customize the structure of prompts to deliver accurate responses.

3.1.6 RAG Evaluation

Finally, since RAG systems combine both search and generation, evaluating them is more complicated. In the paper RAGAs: Evaluating Open-Domain Question Answering with Faithfulness and Factuality by Shuster [19], the authors propose a framework called RAGAs that focuses on different parts of the pipeline, like whether the retrieval step gets useful documents, whether the response is faithful to that context, and whether it actually answers the question.

One key challenge in this area is having good ground truth Q and A pairs and reference contexts to compare the model's output against. Without these, it's hard to say if the system is doing a good job or just sounding smart. The paper also talks about automating these evaluations to scale them across datasets.

3.2 State of Practice

There is currently no chat bot for live-stream video content in practice. Although chat bots created by combining LLM with RAG or prompt engineering approach are currently widely applied across various domains.

Firstly, chat bots are used in customer support, where they improve efficiency and reduce the need for human employees [20]. For example, in eToro,[1] uses customer support chat bot that provides guidance on functionalities of the platform and financial regulations for different countries.

Secondly, chat bots are used in education. They provide personalized learning by retrieving data consisting of course materials from different study fields. Applications can interactively explain topics or give exercises for the student with these data. An example is the Duolingo Lily chat bot [2], which engages users in conversations to improve their language skills. Lily's conversational approach makes learning fun and interactive while helping users practice and retain knowledge.

Furthermore, they are applied in healthcare to help patients by accessing medical databases and retrieving accurate and relevant information. For example, Babylon Health[3] provides a chat bot that offers personalized medical consultations based on patient input. The chat bot evaluates symptoms and offer possible diagnoses, it helping patients understand their condition and determine when to seek professional medical advice.

[1]eToro is an investing broker platform where users can invest in stocks, commodities, and more. Further details at: https://www.etoro.com

[2]Duolingo is a language learning platform with many languages.
https://blog.duolingo.com/duolingo-chatbots/ for more information.

[3]Babylon Health a digital health service that uses artificial intelligence to provide medical consultations https://www.babylonhealth.com/uk for more details.

Lastly, the thesis topic has potential for real-world application, as it is being researched in collaboration with the video stream solutions company Bitmovin.

3.3 State of the Art

Currently, there is no state of the art solution specifically for live-RAG chat bots. However, advanced document-based chat bots have been already studied. For example, dynamic RAG has been explored in different applications, such as the approach described in [21]. It focuses on adapting retrieval and generation processes dynamically based on the needs of large language models.

Chapter 4
METHODOLOGY

This chapter presents the practical implementation of the proposed research. It writes about a detailed description of the technical choices, system design, development process and evaluation. It begins with the selection of right software tools and programming languages used for the projects's development. This is followed by a explanation of the system architecture, which illustrates how the components interact with one another through data flow mechanisms and API endpoints.

Next, it will be described how are the data handled, collected, preprocessed, and managed to support efficient processing. Backend development is discussed with a focus on the programming language and frameworks in the language used for implementation. Next, the chunking strategy is explained, detailing how documents are divided into smaller units to support retrieval-augmented generation. The embedding step covers how these chunks are transformed into vector representations using a suitable embedding model and different models are introduced as well.

Continuing with explanation of response generation where is shown how did I implemented the system so the relevant chunks are used to generate meaningful and context aware responses. Finally, the chapter concludes with a discussion of the evaluation process and visual representations of experimental results to understand the best parameters for the chat bot.

4.1 System Architecture and Implementation

The project consists of three main parts. First is the video live stream which is the main data source. The second is the frontend, which holds the web browser user interface. It is responsible for collecting data from the video stream and communicating with the third component, the backend server. The backend holds most of the project logic. From embedding of the data till generating the respose and evaluation. The following sections describe each component in detail and explain how they are integrated to form a connected system.

4.1.1 Software and technologies used

The decision of which software to use seemed relatively straightforward. Since the project is focused on practical implementation, I prioritized tools that I know already and I am comfortable with. Whole project was developed in the Windows Subsystem for Linux 2 (WSL2), which provides a full Linux environment running directly on Windows. This allowed me to use Linux-based tools and packages.

For code development, I chose Visual Studio Code as the integrated development environment (IDE). It offers good support for Python, Git integration and has a WSL extension that allows editing and running code directly in the WSL environment.

Technologies used for this project are:

1. **Backend**
   (a) **Python** – Programming language used for backend development, evaluation, and result visualization.
   (b) **Flask** – Python framework used for API development.
   (c) **Git** – Version control system used to track project progress in the cloud.
2. **Retrieval-Augmented Generation and Prompt Engineering**
   (a) **LangChain** – Open-source framework that provides tools for working with LLMs and AI technologies.
   (b) **ChromaDB** – Vector database used for locally storing embedded documents.
   (c) **RAGAS** – Framework for evaluating LLM applications such as RAG.
3. **Frontend**
   (a) **TypeScript** – Programming language used for frontend development.
   (b) **BitDash** – Bitmovin's media player framework for adaptive streaming (DASH, HLS), with support for DRM and low-latency playback.
   (c) **Jest** – TypeScript testing framework used for unit testing.
   (d) **HTML and CSS** – Markup and styling languages used to create the user interface.
4. **Result Interpretation**

(a) **Matplotlib** – A library used for creating static and interactive visualizations in Python. It was used to generate radar charts and grouped bar plots that clearly communicate performance metrics across different RAG and prompt engineering configurations.

(b) **Seaborn** – A Python data visualization library built on top of Matplotlib, used for generating aesthetically appealing and informative statistical graphics.

(c) **Pandas** – A data analysis and manipulation library that provided efficient data structures (DataFrames) for cleaning, grouping, and analyzing large evaluation result datasets.

(d) **NumPy** – Numerical computing library in Python, used for array operations and intermediate calculations during visualization preparation.

4.1.2 System architecture

This section provides high level overview of the system architecture.

The overall purpose of the project is to enable real-time interaction between users and a domain-specific large language model system based on video stream content. As shown in the diagram, the user interacts with the system by sending a question through the BitDash Player interface. This frontend component gets the query and sends it to the server, where is the project logic based on RAG. This setup allows users to receive accurate answers based of the video content, transforming the passive video consumption into an interactive experience.

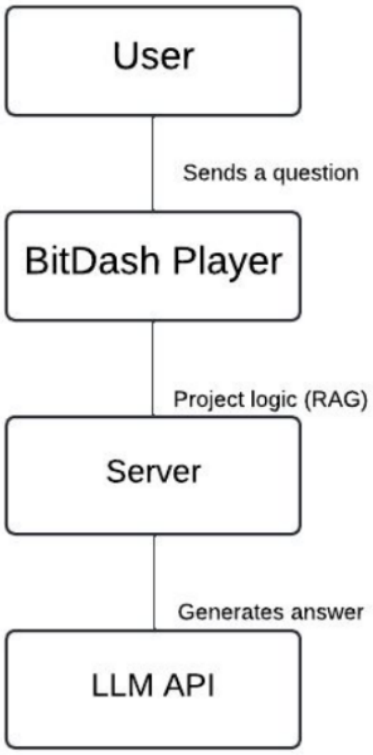


Figure 4.1: Overall Project Flow

The first component is the video stream, which is a data source for the research. The data are then processed in the frontend part of the project. Once the data is cleaned and prepared, it is sent to the backend via two API endpoints:

1. **POST /Embed** - The purpose of this endpoint is to send the data that are ready to be embedded to the backend. This API endpoint has two parameters. First is `"text"`: since the chunking logic is implemented in the frontend part of the project, these chunks are sent independently to the backend, where the embedding logic is executed. The second parameter of this endpoint is `"sourceID"`. In the research I thought that there might be multiple video stream data sources in the embedding database. So each chunk that is sent to be embedded has its source identification tag based on the video stream source. This means that all embedded chunks from one source end up in the same vector database.

2. **POST /ask** - The role of this endpoint is to handle user queries. It accepts two parameters that are sent to the backend. First `"query"`: this contains the user's input, which is sent in JSON format to the backend for processing. Second `"character"`, which defines the chat bot assistent's persona in the final prompt. This parameter is intended to be configured on the frontend by a system administrator. This parameter value can for example be, "American Football Expert", "Tennis Commentator", "Bas-

ketball Enthusiast" etc. This flexibility enables the assistant to act in the conversation, in communication style based on the video stream sport type.

Last part of the architecture is the backend. It handles the core logic of embedding, querying, response generation and evaluation.It retrieves relevant contexts from embedded video data and formulates a prompt for the LLM API. The LLM API then generates an answer based on this prompt and returns it to the backend that passes the response to frontend. All the main components are also shown in the figure 4.2

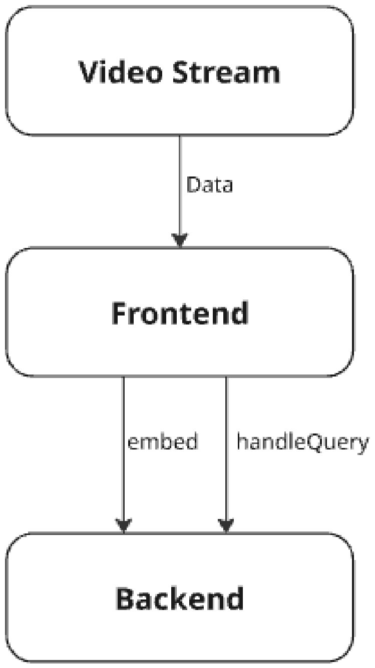


Figure 4.2: Overall Data Flow

4.1.3 Data Handling

The data handling is done mostly in the frontend part. The data with which this project works are commentary captions from sports live stream. These captions are passed by the stream in an object that can be accessed. This object with captions is send by the HTTP Live Streaming (HLS) video stream every time when there are any words said by commentators. The HLS is a video streaming protocol, works by breaking down video files in to smaller downloadable HTTP files and delivers them using the HTTP protocol [22].

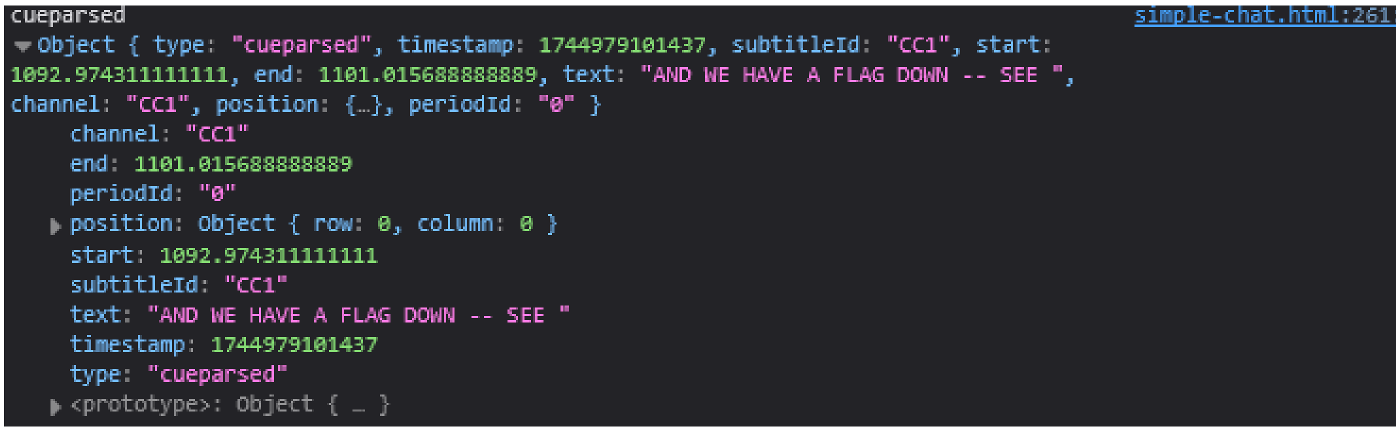
In the figure 4.3 is shown the one object passed to the frontend. There are many fields in the object and each field in the object has a specific role. I will explain the most important fields that are used in the project:

timestamp: A numeric value representing the exact time the caption was generated or received. With this field we know when the text was spoken and we can ensure that text stays in the order.

subtitleId and channel: These both show the subtitle track from which the cue originated. This is important for keeping namespaces for each embedding source as discussed in System Architecture section in the API endpoints.

start and end: Floating point values that mark the start and end times (in seconds) of the subtitle cue in the video timeline. These fields may be useful for providing metadata for the chat bot.

text: Contains the actual spoken content extracted from the video stream. It is the most important field because it is the core data used for chunking, embedding and retrieval.



```
cueparsed                                                                     simple-chat.html:261
▼Object { type: "cueparsed", timestamp: 1744979101437, subtitleId: "CC1", start:
1092.974311111111, end: 1101.015688888889, text: "AND WE HAVE A FLAG DOWN -- SEE ",
channel: "CC1", position: {…}, periodId: "0" }
    channel: "CC1"
    end: 1101.015688888889
    periodId: "0"
  ▶ position: Object { row: 0, column: 0 }
    start: 1092.974311111111
    subtitleId: "CC1"
    text: "AND WE HAVE A FLAG DOWN -- SEE "
    timestamp: 1744979101437
    type: "cueparsed"
  ▶ <prototype>: Object { … }
```

Figure 4.3: Data Object passed from video stream

This structured data object is the foundation for further processing in the frontend, where it is chunked, tagged with metadata (such as source ID), and sent to the backend for embedding into the vector database.

4.1.4 Backend Development

When thinking about the backend implementation the first thing I thought about was the choice of programming language and frameworks. Since this project has to do a lot with AI functionality and most of the frameworks supporting large language models (LLMs) are naturally built for Python, Python seemed like the logical choice. However, the company, which I work on this research for, technical stack is fully build in TypeScript. This led me to explore the possibility of building the backend in TypeScript with Node.js for the API endpoint logic and using the Llama Index framework for the LLM and AI logic.

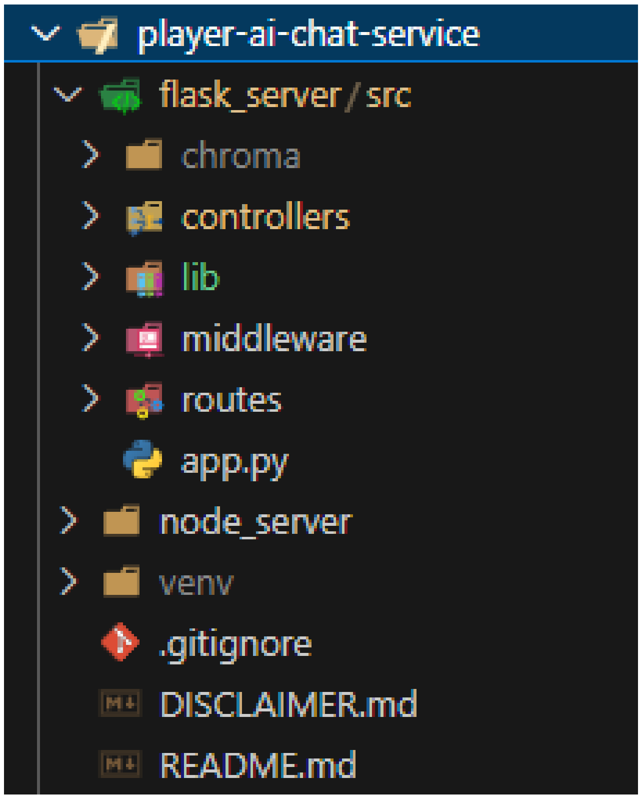
The first version of backend server was therefore built in TypeScript using technologies described above. This version worked and communicated well with the frontend, serving as a Minimal Violable Product. However, as this was only the initial version, I managed to implement only basic logic for the RAG system. As I began exploring more advanced methods, it was challenging due to insufficient documentation. The more advanced the methods I wanted to use, the less resources were available. This made development in TypeScript increasingly time consuming. As a result, I decided to rewrite the entire backend server in Python, which provided a more suitable environment for implementing advanced RAG functionalities.

I recreated the whole project into Python language, using Flask for API endpoint handling and LangChain for LLM and RAG logic, as there is much more documentation and active development with this framework.

Figure 4.4 shows the backend folder structure of the project. The main directory is organized as follows:

- `flask_server/src/`: This is the main source directory for the Flask backend server.
  - `chroma/`: Contains ChromaDB collections of embeddings and it is the main data storage of the project.
  - `controllers/`: Holds the controller files, which are responsible for handling incoming requests and dealing with responses.
  - `lib/`: Contains the core logic and functions used across the backend. All main logic implementations are placed here.
  - `middleware/`: Includes middleware components that process requests and responses, such as authentication or logging.
  - `routes/`: Defines the API routes and endpoints. It connects URLs to specific controller actions.
  - `app.py`: The entry point of the Flask application, where the server is initialized and configured.
- `node_server/`: Contains the Node.js server. The first version of the project.
- `venv/`: The Python virtual environment. It manages package dependencies for the backend.
- `.gitignore`, `DISCLAIMER.md`, `README.md`: Standard project files for version control, legal information, and project documentation.

This modular structure has clear separation of roles, making the backend codebase maintainable and scalable.



```
v player-ai-chat-service
  v flask_server / src
    > chroma
    > controllers
    > lib
    > middleware
    > routes
      app.py
  > node_server
  > venv
    .gitignore
    DISCLAIMER.md
    README.md
```

Figure 4.4: Folder structure of the project

API Endpoint Handling The backend uses Flask as the web framework to manage API endpoints. The main endpoints are the same as mentioned in frontend description:

POST /**ask**: Handles user queries by retrieving relevant context from the vector database and generating a response using a LLM.

POST /**embed**: Accepts text data from the client, generates embeddings using a pre-trained model, and stores them in ChromaDB.

As mentioned above, the core logic for the RAG system is implemented in the lib directory. It includes these parts of the projects:

Embedding Generation: In the lib directory are three files with different embedding logic. One for the usage of OpenAI embedding models, second one for usage of Gemini models, and last one for prompt engineering approach. These similar files were made for flawless experimenting. These embeddings are always stored in ChromaDB, which serves as the vector database for efficient similarity search. More about embedding in section 4.1.6

ChromaDB Integration: ChromaDB is used as the primary vector database for storing and retrieving embeddings. The chroma/ directory contains the collections of embeddings, organized by different configurations such as chunk size and overlap. The backend interacts with ChromaDB through the LangChain library.

Query Handling: The Python files for query handling have multiple responsibilities. It retrieves contexts, creates responses, creates prompt and holds chat history. For easier experimentation there are also multiple files for it. These files also holds ground truth question and answer pair and create evaluation metrics. More in section 4.1.7

Evaluation: Lastly evaluation logic is the backend /lib directory. There are generated responses that are filled into evaluation datasets. These are then evaluated on different metrics and saved in to json files. More about this in seciton 4.1.9

Lastly, I want to describe how the frontend and backend parts of the system communicate together. The connection is done through API endpoints. When the frontend needs to send data to the backend, it sends it in JSON format. On the backend, there is a middleware that checks if the data is correct before using it. It makes sure that the data from the frontend has the correct structure, types, and values. Thanks to this, the backend does not crash because of bad or unexpected data. This middleware can be scaled in the future as it is crucial for data valuadtion and security, so the chat bot cannot be tricked by user prompts into doing something unintended and wrong.

4.1.5 Chunking

To make efficient retrieval of relevant context from the vector database possible, it is needed to chunk the textual data. Without this chunking process, the RAG system would not be able to connect relevant context with user queries, making it unusable.

Chunking is commonly done on static documents where the complete content is available in advance. For these scenarios there exist several chunking methods:

- **Fixed-size chunking:** The text is split into chunks of a predefined size, usually measured in characters or tokens. This method is simple but may break sentences unnaturally.

- **Separator based chunking:** The text is split at predefined separators such as "." or "newline character".

- **Recursive or semantic chunking:** More advanced models use natural language processing to segment text at semantically appropriate points such as, paragraphs or topics.

However, in this thesis, the data consists of live textual captions arriving in real time from a video stream. Since the text is dynamic and comes in small and unpredictable bits, traditional chunking strategies that assume access to full documents are not applicable. Therefore, I had to design a custom chunking algorithm for real-time processing.

A critical decision in designing this system was determining where to implement the chunking logic. On the frontend or the backend. Both approaches have their advantages and disadvantages:

- **Frontend**:
  - **Pros:** Instant access to incoming text data; less load on backend; responsive real-time chunking.
  - **Cons:** More responsibility on the client-side; requires synchronization with backend logic.
- **Backend:**
  - **Pros:** Centralized logic; easier to update and maintain chunking methods.
  - **Cons:** Requires sending raw text data to the server constantly, what means higher network latency and server load.

Because of the the real-time data and the need for responsiveness, I decided to implement the chunking logic directly in the frontend.

As already mentioned above, existing models could not be used. Therefore I developed my own chunking method using a separator based approach. Using models such as sentence based or semantic chunking was not possible because they rely on having access to complete documents, which is not possible with live data.

I implemented the chunking method for dynamic data by creating these functions. Whole data flow is illustrated in figure 4.5:

1. `accumulate(chunk_size)`: This function accumulates incoming text in a buffer. When the total length of the accumulated text reaches the defined `chunk_size` threshold measured in characters, it calls the next function.

2. `trimToLastSeparator(text)`: When the threshold is reached, this function trims the accumulated text up to the last occurrence of a defined separator. Separators help make sure, that chunks end at natural points in the text, such as sentence endings. In my implementation, common separators are`"."` (end of a sentence) or`"»"` (start of commentator speaking indicators). Any remaining text after the last separator stored and passed back to the accumulate function for next cycle.

3. `getLastXSentences(text)` (optional): If overlap is configured, this function extracts the last `x` sentences from the previous chunk to include in the current chunk. This overlap helps maintain contextual continuity between chunks.

4. `generate(text)`: This function sends the finalized chunk to the backend for embedding into the vector database and waits for a response.

These functions work in a loop and continue as long as new text data arrives from the video stream.

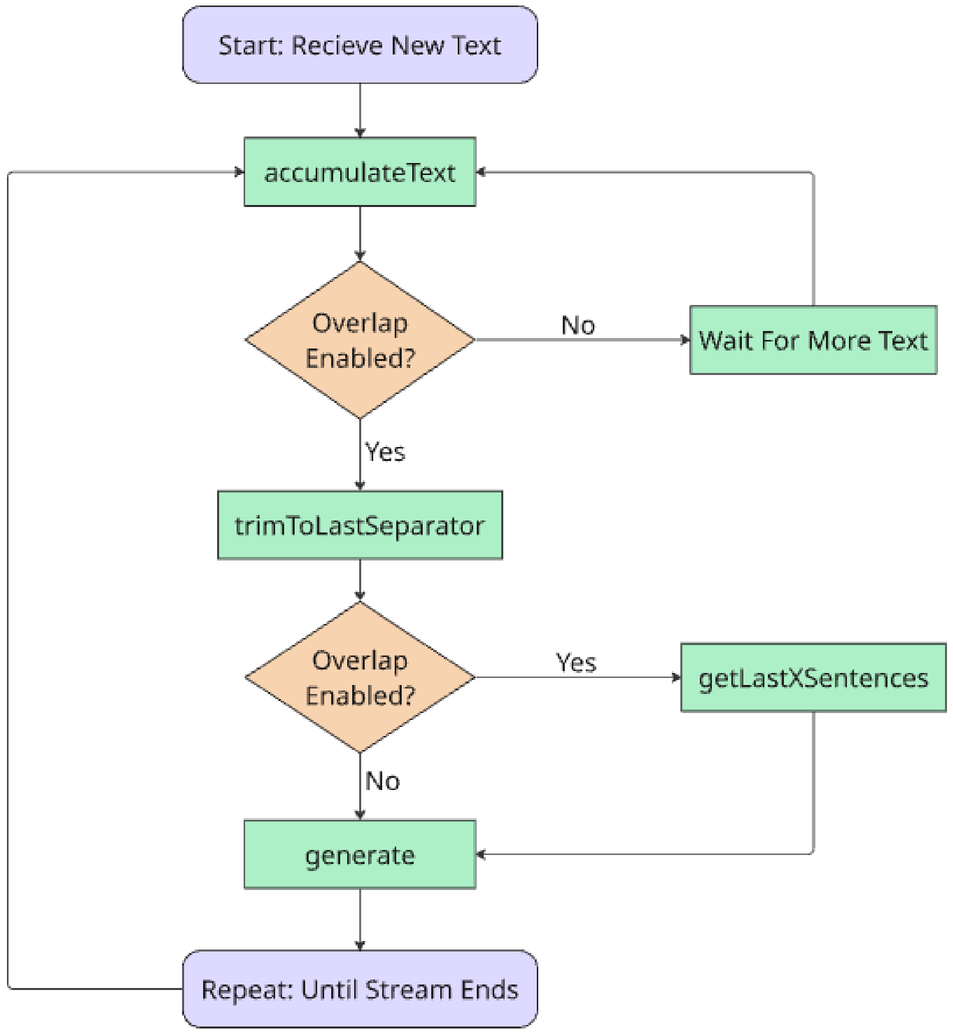


Figure 4.5: Flowchart illustrating the dynamic chunking loop used for real-time data streaming.

Two key research challenges came up during this process:

- **Optimal chunk size:** I need to find balance between chunk size, processing delay and token cost. For example, waiting until 10 000 characters accumulate might take 10 to 15 minutes, resulting in the chat bot without having understanding of recent context. On the other hand, chunks that are too small may contain insufficient information for accurate retrieval. Also bigger chunks contain more tokens to retrieve what makes the response generation more expensive. Therefore data were created with multiple chunking sizes for the evaluation of the best chunking size.

- **Choice of separator and overlap:** The effectiveness of chunking heavily depends on choosing appropriate separators. Bad choice can lead to unnatural splits in the text. In this project, the end of sentence `"."` and the start of commentator speaking indicators `"»"` were found to be the most reliable for ending of the chunks. Additionally, chunks with overlap were added to the datasets for evaluation. Overlap takes x characters from the previous chunk to keep contextual continuity between chunks. This should make the chat bot more effective because it should have bigger awareness of the context.

Overall, this custom real-time chunking approach is important component of the system. It makes possible continuous embedding and context retrieval to maintain a responsive chat bot.

4.1.6 Embedding

The embedding is important part of this project. It converts text chunks into vector representations that can be stored and later retrieved from the vector database, ChromaDB. With good embeddings, it is easier and more accurate to search for right context based on the query from user. As the retriever performs searches in vector space rather than with traditional keyword matching.

In a Retrieval-Augmented Generation system, the quality of the retrieved context has a direct impact on the quality of the chat bot's final response. As retrieval is done only based on similarity in the vector space, good embeddings are essential for retrieving the most relevant chunks. If the embeddings represent the meaning of the text poorly, the retriever might return irrelevant content, which can confuse the model and result in wrong answers. High quality embeddings make sure that semantically close texts are grouped together, what makes the model to give better responses.

There are different embedding models available from different companies. These models can vary in many aspects such as vector dimensionality (for example: 512, 1536, 3072), training data and performance on different types of content. These differences affect how accurate the embeddings are in specific use cases like retrieval.

For this project, I experimented with three models. Two of them are from OpenAI: `text-embedding-ada-002` and `text-embedding-3-large`, which are widely used and are reliable base models. They are known for good performance and easy integration. The third one is `gemini-embedding-exp-03-07` from Google, which I decided to try because it showed very strong results in recent MTEB benchmark tests [23] and seemed to perform especially well in terms of semantic understanding.

Since I was using models from different companies, I needed to structure my code in a flexible way. I created separate implementation files for each provider (OpenAI and Google) because each has its own framework and rules for implementation. This divided approach made it easier to switch between models and test them one by one.

Embedding for prompt engineering approach is also different as for the response generation part where I had to handle prompt engineering in separate file, embeddings are more straightforward. There's no need for context retrieval So I didn't needed to create one more file for prompt engineering approach.

To summarize, embedding determines how well the chat bot understands and connects the live input with stored information. I tested three embedding models which will be evaluated later in evaluation chapter.

4.1.7 Response Generation

This section outlines the core components of the response generation pipeline, which retrieves relevant information from a vector database, composes a prompt for a language model, and generates a character-specific answer grounded in retrieved context.

The first step in the response generation process is retrieving relevant chunks of information from the vector database. For this I needed to take to account two components:

- **Retrieval Method:** When a user submits a query, the retriever performs a cosine similarity search against the stored embeddings. This means that the user query is embedded to vectors with a embedding model and compared to the stored vector space. The `topK` parameter determines how many chunks are retrieved ($k = 2$). The best top K number was researched, because lower `k` can lead to more focused context, while a higher `k` may introduce noise but offer broader coverage. One additional aspect to take to account is that, the higher K number the more tokens does one query cost and the longer response time is.
- **Context Construction:** Retrieved chunks are grouped into a single context string, which forms the basis for prompt construction and response generation.

When the most relevant chunks are retrieved, they are added into a prompt for the language model. The prompt is carefully designed to guide the model towards generating responses that are accurate.

- **System Prompt:** A static system prompt defines the overall behavior and tone of the LLM. It includes guidelines on how to understand the retrieved context and how to simulate the voice of a specific character.
- **Dynamic Formatting:** The prompt dynamically inserts the retrieved context, chat history, user query, and character identity to provide the LLM with all necessary information for generating correct response.

Example System Prompt:

```
Context information is below.
-----------
{{context}}
-----------
Given the context information and not prior knowledge, answer the query.
Answer the query as if {{character}} is speaking. I want you to respond and answer like
{{character}} using the tone, manner, and vocabulary {{character}} would use.
Use the context as {{character}} would use it. Provide information that you know from
context; if you don’t know, don’t guess—just write "I don’t know that information."
```

Although the experimental version of the system handles only individual queries, it is designed with support for back and forth conversations with understanding chat window history.

- **Chat History Placeholder:** The system prompt includes a placeholder for past messages. This allows the LLM understand context of the chat window and which can be used for the response creation.

- **Query Reformulation:** If chat history is available. One more step is added. The user’s query is reformulated into a better question, that should give more accurate responses.

The final step is invoking a LLM to generate a response based on the built prompt.

- **LLM Invocation:** The system supports models such as OpenAI’s `gpt-4o-mini` and Google’s `gemini-2.0-flash-001`. The formatted prompt is passed to the selected LLM, which generates a character specific response based on the prompt. Based on different LLM there can be different latency for the response because LLMs have different token processing speeds.

- **Error Handling:** If the LLM fails to produce a response, the system returns a fallback message to the user.

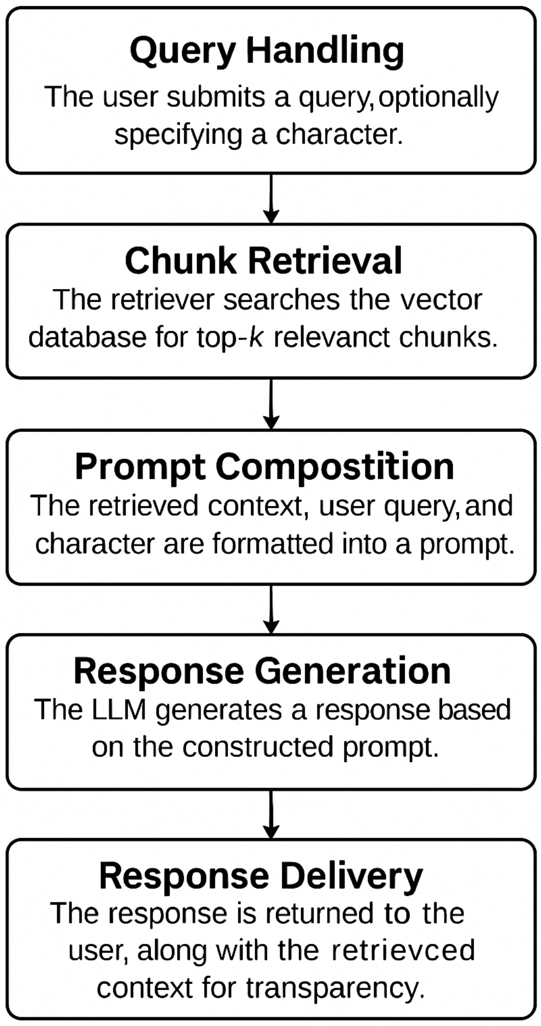

Query Handling
The user submits a query,optionally specifying a character.
Chunk Retrieval
The retriever searches the vector database for top-k relevanct chunks.
Prompt Compostition
The retrieved context, user query,and character are formatted into a prompt.
Response Generation
The LLM generates a response based on the constructed prompt.
Response Delivery
The response is returned to the user, along with the retrievced context for transparency.

Figure 4.6: Flowchart illustrating the pipeline of response generation in the project.

This pipeline makes sure the answers are based on the right information, match what the user asked, and sound like the character they chose.

4.1.8 Prompt Engineering

As previously discussed, this project also includes an implementation of a chatbot based on a prompt engineering approach. The logic of this approach differs from RAG by loading all of the text into the prompt as context instead of just retrieved chunks.

This approach simplifies the retrieval logic but introduces new technical challenges, primarily related to the model's context window limitations.

One of the primary constraints encountered during implementation was the size of the context window supported by different Large Language Models (LLMs). The full text of the video stream used in this project is approximately 35,000 tokens long. Therefore, to use this approach effectively, only LLMs with a context window exceeding 35,000 tokens can be used. For example, newest models such as OpenAI's `gpt-4-128k` or Google's `Gemini Flash-2` with extended context capacities are okay to use.

Attempting to use models with smaller context windows resulted in error in the response.

The implementation of the prompt engineering approach had several advantages in comparison to:

- **Simplified Logic:** There was no need to implement a separate retrieval mechanism or maintain a vector database.
- **Full Context:** The language model receives the exact same input context for every query, leading to consistent responses when asked the same question. It will also never miss a part of the context (what can happen when wrong context is retrieved in RAG), because it has always full context.

The implementation of prompt engineering approach was easy, however there might be some limitations. All of these concerns will be addressed in the results chapter.

4.1.9 Evaluation

To assess the effectiveness of the implemented RAG chatbot system, I designed a structured evaluation process. The goal of this evaluation was to determine what are the best chat bot parameters to respond most accurately to user queries. This section describes the evaluation pipeline, the ground truth data, selection of parameters that are tested and the selection and interpretation of performance metrics.

The first step in the evaluation process was to get a reliable set of ground truth question answer pairs. These were generated using an LLM with access to the full match transcript. A total of 66 pairs were created, covering a range of fact based questions and answers to them. This dataset formed the basis of all evaluations.

Next, I create testing data set. Each evaluation sample in the dataset consists of the following fields:

- `question`: The user query.
- `answer`: The response generated by the LLM using the RAG pipeline described in section 4.1.7.
- `contexts`: The retrieved context chunks used for response generation.
- `ground_truth`: The reference answer used for metric comparison.

A typical evaluation sample looks like this:

```
{
 "question": "Who are the commentators calling the game?",
 "answer": "The commentators calling the game are Mark Jones, Louis Riddick, and Quint Ke
 "contexts": [..., ..., ...],
 "ground_truth": "Mark Jones, Louis Riddick, and Quint Kessenich."
}
```

These datasets were generated for different system configurations and were stored as a json format to be easily accessable.

To explore the impact of different parameters like retrieval and embedding strategies, multiple configurations of these parameters were tested. All experiments were done using the GPT-4o-mini model as the generation component.

In the first round of the evaluation, I wanted to analyze the impact of chunk overlap on RAG performance. For this, data chunks were generated using three different character lengths: 600, 1500, and 3000. For each chunk size, there were two configuration created:

- **No Overlap:** Documents were divided into chunks without any overlapping content.
- **With Overlap:** Each chunk partially overlapped with the next chunk to keep contextual continuity.

The retrieval component used `top_k = 4`. All evaluations in this round were done using the `text-embedding-ada-002` model for embedding generation.

The goal of this evaluation round was to understand how overlap affect core RAG metrics and understand if overlap matters in the chunking with this use case. The better version embeddings are then used in evaluation with different LLMs in the end.

In the second evaluation round I tested different parameter variations. These were:

- **Embedding Models:**
  - `text-embedding-ada-002`
  - `text-embedding-3-large`
  - `gemini-embedding-exp-03-07`
- **Top-K Retrieved Chunks:** $K = 2$, $K = 4$, $K = 6$
- **Chunk Sizes:** 600, 1500, and 3000 characters

Each combination of embedding model and top-K setting was evaluated using all three chunk sizes, resulting in a total of $3 \times 3 \times 3 = 27$ distinct RAG configurations.

I continued the evaluation by selecting the two best performing combinations of embedding model, top-K value, and chunk size. Then I re-evaluated them using overlapping chunks to compare the resulting metrics overlap versus no overlap.

After selecting the best performing setup based on the GPT-4o-mini results, the same configuration was re-evaluated using Gemini 2.0 Flash LLM to see performance using different LLMs. Since new LLM models were released after evaluating with these two, I did also evaluate the two best configurations with newest cost efficient LLMs from OpenAI gpt-4.1-mini and Google Gemini Flash 2.5. Finally I evaluated the best configurations with one state of the art model that is not cost efficient just to see if there is a difference.

The evaluation was done using the RAGAS framework[1], which is made for evaluating chat bots and RAG systems. These evaluation metrics were computed:

- **Context Precision**: Measures the accuracy of the retrieved chunks.
- **Context Recall**: Measures the completeness of the retrieved chunks.
- **Faithfulness**: Measures if the generated answer is grounded in the retrieved context.
- **Answer Relevancy**: Measures if the answer is relevant to the question.
- **Answer Correctness**: Semantic match between answer and ground truth.
- **Semantic Similarity**: Embedding-based comparison of generated and reference answers.

All of these metrics are described in depth in section 2.6. The Results for each configuration were stored in json file, so it is possible to access them for visual interpretation.

The prompt engineering evaluation was different from the RAG evaluation. Since this approach does not involve chunk retrieval, less evaluation metrics were used. There is no context, and therefore metrics that rely on context relevance such as context precision/recall are not applicable. So I used only output focused metrics such as:

- **Answer Correctness**
- **Answer Relevancy**
- **Semantic Similarity**

Additionally, both approaches have average time metric. This metric times how long does it take the LLM to generate a response. I evaluated the prompt engineering approach with different LLMs and with the best RAG approach on these four metrics. For this I must

[1]`https://github.com/explodinggradients/ragas`

always evaluate the responses with the same LLM because different LLMs behave differently at evaluating and it could skew the end results. So the responses are created with different LLMs but evaluation is **always** done with gpt-4o-mini.

Lastly, it is important to note that this evaluation process has pretty big financial cost. For example, one evaluation run using $K = 4$, chunks size 600 with GPT-4o-mini consumed approximately 820,000 tokens and did 925 LLM requests. This shows the practical challenge of evaluating large scale RAG systems, especially when experimenting with multiple configurations.

4.1.10 Experimental Visualization

This section describes the thinking behind the selection of visualizations, the tools used to implement them, and how the visualizations support the interpretation of the results outlined in Section 4.1.9.

I used several types of visualizations, to effectively interpret the impact of different parameter configurations and model choices on the RAG system's performance. The primary technologies used for visualization were `matplotlib.pyplot` for standard plotting, `seaborn` for better and aesthetically improved statistical plots, `numpy` for numerical computations, and `pandas` for data manipulation and analysis.

The first evaluation goal focused on determining if chunk overlap influences the quality of the RAG responses. For this, I had computed six RAGAS metrics scores with the average response time. To compare the results of using chunks with overlap versus no overlap, radar plots were used (Figure 5.1). Radar plots are good for this purpose as they allow for multi metric comparison in one plot.

However, the differences between overlapping and non-overlapping chunk configurations were small and it is hard to see clearly. I also included tabular representations of the full metric scores for better interpretability (Table 5.1).

As a second evaluation goal I chose to evaluate 27 different RAG configurations of three embedding models, three chunk sizes and three top K retrieved chunks values. Visualizing such a large number of combinations was a challenge. To visualize these configurations, I designed a set of heatmaps. One aggregate heatmap displays the average performance across five RAGAS metrics for each configuration (only five because answer correctness metric was skewing the results), this makes possible easy overview of which combinations perform best. In addition to that, six individual heatmaps were created for each metric. This provides detailed insights into how different configurations influence specific aspects of performance. These visualizations helped see trends between the different metrics.

After identifying the two top performing parameter configurations, I re-evaluated both with chunk overlap. These results were again visualized using radar plots for high-level multi-metric comparison and a summary table showing exact metric values (Table 5.2).

Next, the selected top-performing RAG configurations were evaluated using different LLMs. The results of these evaluations were visualized using grouped bar plots. This made possible direct comparison of performance across LLMs while holding other parameters constant. A similar approach was taken for the prompt engineering method, where different LLMs were evaluated without retrieval, and the results were plotted to compare their effectiveness Figure **??**.

As a last visualization I created comprehensive visualization to compare the performance of the best prompt engineering setups versus the best RAG configurations across different LLMs. This comparison was visualized using combined bar plots showing the core output metrics (Answer Correctness, Relevancy, Semantic Similarity, and Average Time). This visualization provides a clear picture of the trade-offs between prompt engineering and RAG systems in terms of performance and latency.

Finally, these visualizations were made to help better understand what are the best configurations for live stream chat bot. I will look on the final results in the next chapter 5.

Chapter 5
Results

In this chapter, I present and analyze the evaluation results obtained during the experiments. There will be figures and tables that hopefully help interpret the findings and to help identify the optimal parameters and the most suitable LLM. Furthermore, I look at the efficiency of the approaches and their costs.

5.1 Best RAG Parameters

In this section there are be described the results for different parameter configurations in the RAG approach.

First, I evaluated and visualized RAG responses with and without overlap. In Figure 5.1, we can see that the overlap configuration achieves slightly better metric scores. This is particularly noticeable with the smaller chunk size of 600 tokens. Additionally, responses with overlap show shorter response times, indicating better efficiency. Since the differences in the metrics are relatively small, I created Table 5.1 to support better interpretation of the results.

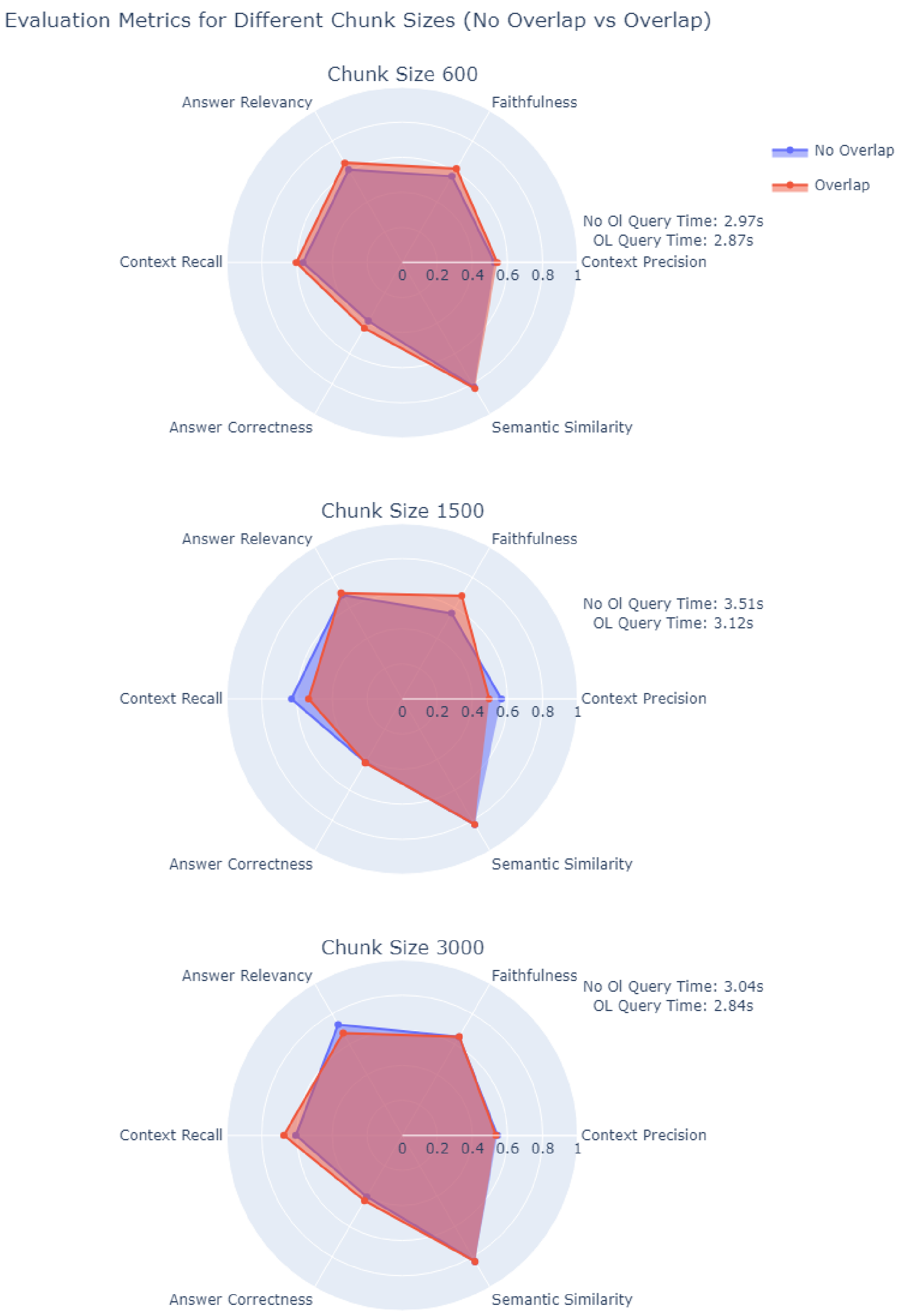


Figure 5.1: Evaluation metrics for different chunk sizes (no overlap vs Overlap)

Also in the table 5.1 is shown that most of the metrics are in bold text for the overlap versions. This suggests that chunk sizes with overlap generate better responses.

Table 5.1 confirms that most of the metrics for the overlap configuration are highlighted in bold, showing better performance. This suggests that chunking with overlap generally has better response quality.

Table 5.1: Evaluation Results by Chunk Size and Overlap Configuration

Chunk Size	**Config**	**Ctx Prec**	**Ctx Recall**	**Faithfulness**	**Ans Rel**	**Ans Corr**	**Sem Sim**	**Avg Time (s)**
600	No Overlap	0.527	0.566	0.567	0.612	0.386	0.821	2.97
	Overlap 150 CH	**0.542**	**0.604**	**0.617**	**0.656**	**0.432**	**0.830**	**2.84**
1500	No Overlap	**0.565**	**0.631**	0.564	0.685	0.418	0.827	3.51
	Overlap 250 CH	0.495	0.533	**0.680**	**0.697**	**0.420**	**0.827**	**3.12**
3000	No Overlap	**0.543**	0.606	0.648	**0.729**	0.404	**0.830**	3.04
	Overlap 400 CH	0.535	**0.674**	**0.650**	0.674	**0.430**	0.830	**2.87**

Figure 5.2 visualizes the average performance of different RAG configurations as a heatmap. Each cell shows the average value of five core evaluation metrics (excluding answer correctness) for a specific combination of embedding model (rows), top-K retrieved chunks, and chunk size (columns). Darker blue colors indicate higher average metric values, while lighter colors indicate lower performance. The highest scores are achieved by the `gemini-embedding-exp-03-07` model with larger chunk sizes and higher top-K values ($K = 6$, chunk size 1500 or 3000), reaching an average metric value of 0.75. In contrast, the `text-embedding-ada-002` model generally generated lower scores, especially for smaller chunk sizes and lower K. This heatmap makes it easy for identification of the most effective parameter combinations. It highlights that both the choice of embedding model and retrieval configuration significantly influence RAG system performance.

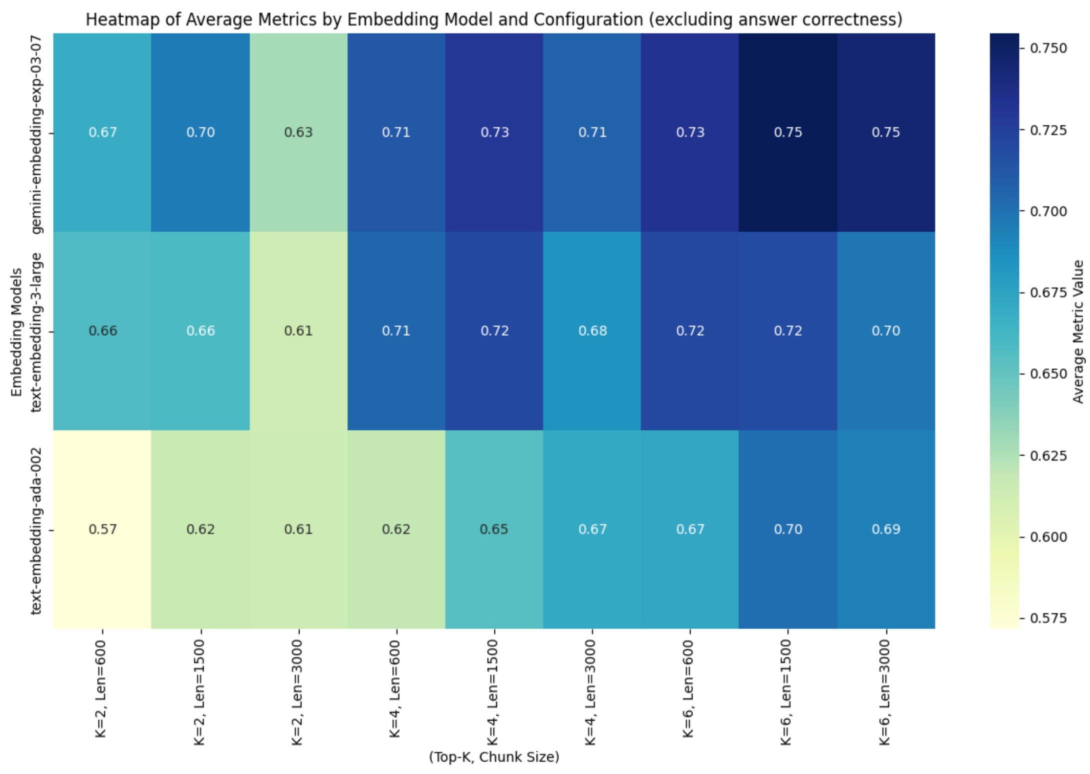


Figure 5.2: Heatmap of average metrics by embedding model, topK and chunk size configurations. (excluding aswer correctness metric)

Since I created the first heat map plot for average of each metric I did not see how the configurations behave on each metric. Now I will look at the heat map for each of the 6 metrics and describe the findings.

Figure A.1 shows how well different RAG configurations performed in terms of answer relevancy. The best performance was achieved using gemini-embedding-exp-03-07 with K=6 and chunk size 1500 or 3000, reaching scores around 0.84 and 0.81. Text-embedding-3-large performed consistently well across all configurations, though not as high as Gemini. Text-embedding-ada-002 showed the weakest performance, especially for small chunk sizes and low K values. This heatmap corresponds with the average values for all metrics in Figure 5.2.

Next the context precision metric. The metric measures the proportion of retrieved chunks that are actually relevant to the user's query. We can see A.2 that the gemini-embedding-exp-03-07 model consistently outperforms the others, achieving the highest context precision of 0.66 at multiple configurations (K=2 and K=4, chunk size=600). Smaller chunk sizes (600 tokens) generally perform better across all models when K is low (K=2 or 4), likely because shorter chunks reduce irrelevant content per retrieval unit. This metric highlights the trade-off between chunk size and retrieval count (K). Smaller, focused chunks perform better, especially when fewer chunks are retrieved. However, without strong embedding model, even good configurations have bad results.

To continue, I will describe context recall. The heatmap in Figure A.3 reveals that the `gemini-embedding-exp-03-07` model achieves the highest context recall, peaking at 0.82 with K=6 and a chunk size of 3000. The larger chunk sizes (1500 and 3000) and higher k (4 and 6) generally show higher recall values across all models, suggesting that larger chunks capture more of the relevant context, especially when more chunks are retrieved. However, the `text-embedding-ada-002` model consistently underperforms, regardless of the configuration.

The Faithfulness metric, which evaluates the degree to which the generated answers are grounded in the retrieved context. The heatmap in Figure A.4 illustrates that the `gemini-embedding-exp-0` model generally has higher faithfulness scores, reaching a peak of 0.74 at K=6 and a chunk size of 1500. In general, increasing K tends to improve faithfulness. This suggest that a wider context helps reduce hallucination, however if the context is too big the score gets worse as in configuration with K6 and chunk size 3000.

Further, I analyze Semantic Similarity, which reflects how closely the meaning of the generated answer aligns with ground truth. The heatmap in Figure A.5 shows that all configurations achieve relatively high semantic similarity scores. The `gemini-embedding-exp-03-07` model is still the most consistent. While differences between configurations are small, larger K values (4 and 6) with larger chunk sizes (1500 and 3000) tend to get slightly better semantic similarity scores. Overall, semantic similarity is less sensitive to configuration changes compared to other metrics such as context precision or faithfulness.

Lastly, Answer Correctness, which checks the factual accuracy of the generated answers compared to the ground truth. As shown in Figure A.6 this metric has the lowest results out of all metrics. The answer correctness scores were excluded from the average performance heatmap in Figure 5.2 because including them could skew the trends observed in the other metrics down.

To conclude this part, I am choosing the configuration **chunk size 1500, K6 and gemini-embedding-exp-03-07** as the best RAG configuration because it performed the best out of all configurations across most metrics.

There is a configuration with the same embedding model and top K, but with different chunk size that performed equally well. However, I decided not to select this configuration because it uses a chunk size that is twice as large, resulting in twice the token cost per query. Additionally, a chunk size of 3000 characters is less suitable for real-time applications, as it typically takes around 4 to 5 minutes to accumulate enough content. In comparison, a 1500 character chunk can be gathered in approximately half the time, making it more responsive for real-time retrieval scenarios.

Next I tried out to make sure that the overlap results are valid. I compared the two best configurations of heatmap with overalp version of them. And based on results in table below we can see that overlap versions are performing better. It confirmed the findings from the Figure 5.1.

Table 5.2: Evaluation Results for `gemini-embedding-exp-03-07` by Chunk Size and Overlap Configuration

Chunk Size	Config	Ctx Prec	Ctx Recall	Faithfulness	Ans Rel	Ans Corr	Sem Sim	Avg Time (s)
1500	No Overlap	**0.626**	0.735	**0.736**	**0.838**	**0.462**	0.837	3.00
	Overlap 250 CH	0.618	**0.768**	0.727	0.783	0.440	**0.839**	**3.16**
3000	No Overlap	0.570	0.818	0.696	0.806	0.469	0.836	2.99
	Overlap 400 CH	**0.624**	**0.818**	**0.710**	**0.841**	**0.464**	**0.842**	**3.19**

In summary, the experiments show that chunking with overlap slightly improves RAG performance across most metrics. We discovered that the best embedding model for all the metrics is `gemini-embedding-exp-03-07` and the overall best RAG configuration with chunk size of 1500 characters and topK of 6 with overlap.

5.2 Prompt engineering versus RAG Comparison Through Different LLMs

In this section I will compare the prompt engineering approach with the best RAG configuration accross different LLMs.

We already know that the best RAG configuration is embedding model gemini-embedding-exp-03-07, topK 6 and chunk size 1500 with overlap of 250 characters. Therefore all of the RAG evaluation will be conducted with this configuration. I tested both approaches with these LLMs: gpt-4o-mini (initial testing), gemini-2.0-flash-001, gemini-1.5-pro, gemini-2.5-flash-preview-04-17, gemini-2.5-pro-preview-03-25 and gpt-4.1-nano. The visualizations of the most interesting will be now shown and analyzed.

To see how do these two approaches compare metric-wise through different LLMs. I created bar plots showing comparisons of each common metric of both approaches as well as average time comparison bar plots per LLM.

The first figure 5.3 shows the comparison of the approaches having generated responses with gpt-4o-mini. We can see that prompt engineering approach got slightly better results for each metric. However the average time comparison shows that it takes four times longer for the prompt engineering approach to generate a response in average. The metric differences are only slightly better, therefore the RAG approach wins the comparison for this LLM.

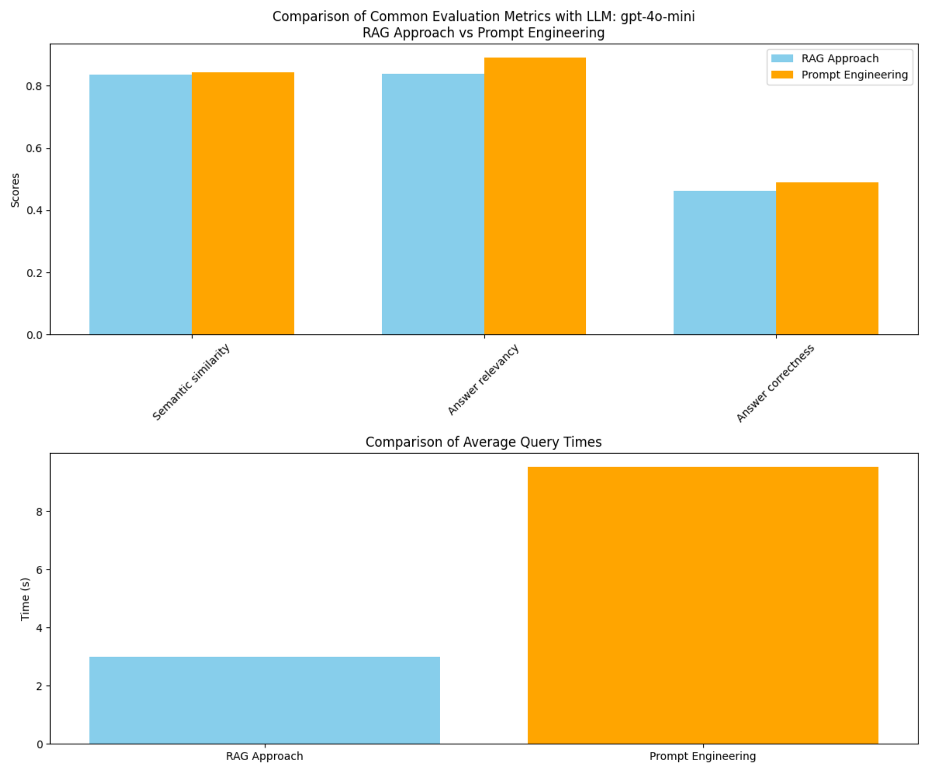


Figure 5.3: Comparison of Common Evaluation Metrics and average response generation times with LLM: gpt-4o-mini between best RAG configuration and Prompt engineering approaches

The second comparison of the approaches is using gpt-4.1-nano. Prompt engineering again slightly outperforms RAG in all evaluation metrics. However, the average query time for prompt engineering is significantly higher — about five times longer than for the RAG approach. Given the minimal differences in scores but the major difference in response time, the RAG approach is the more efficient choice for this LLM. The visualization can be found in the appendix A.7

Next comparison is with the first of Google's models gemini-2.0-flash It most interesting one because we can see prompt engineering approach outperforming the RAG approach on two out of three metrics and in average time as well. It has around staggering 1.3 seconds average response generation time. 5.4 Which is the best of all evaluations through all approach/LLM evaluation combinations.

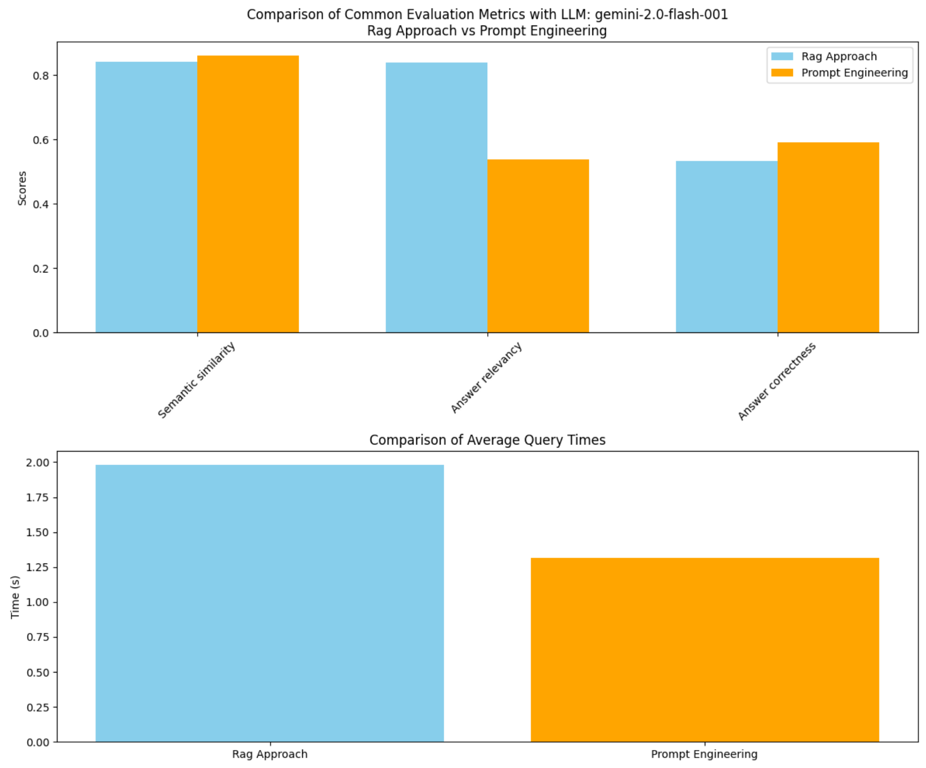


Figure 5.4: Comparison of Common Evaluation Metrics and average response generation times with LLM: gemini-2.0-flash between best RAG configuration and Prompt engineering approaches

The next comparison is with Google's gemini-1.5-pro model. Here, the RAG approach outperforms prompt engineering in answer relevancy, while prompt engineering slightly leads in semantic similarity and answer correctness. Interestingly as this is a pro model it doesn't give better responses and the query times are worse than for smaller compact models we have seen before. Overall, due to better performance in the key metric of answer relevancy and faster responses, the RAG approach is slightly better for this LLM.A.8

To determine which LLM performs best, I compared the average evaluation metrics and normalized query times for each model using the RAG approach, as shown in the bar chart above.

We can see in figure5.5 that all of the four LLMs have similar average performance metrics with slight differences. However the average time to generate answer is in the gemini-2.0-flash-001 and gpt-4.1-nano much faster. If we look at the average metrics values of these two, then we can conclude that the best performing LLM for RAG approach is the newest cost efficient LLM from OpenAI **gpt-4.1-nano**.

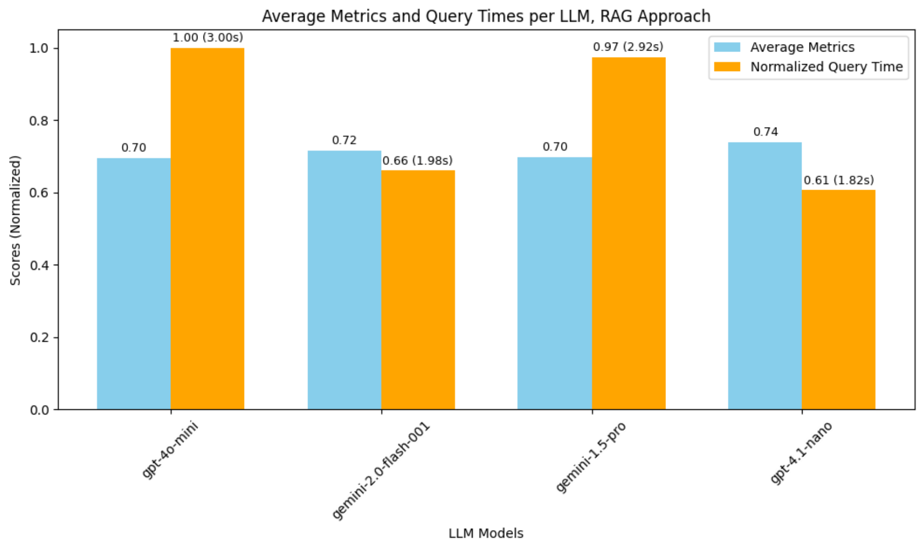


Figure 5.5: Comparison of average metrics for all the LLMs for RAG approach with normalized average times to generate a response

In the next figure5.6, we see the same comparison for the Prompt Engineering approach. We can notice a trend: models with longer average response times tend to generate better results. However, an average response time of 9 seconds is not feasible for our use case. Interestingly the fastest response is achieved by the gemini-2.0-flash model and is only 1.31 second, faster than the best RAG approach. Its average metric scores are the lowest in this comparison but as shown in the previous figure5.4, all prompt engineering approaches still outperform most of the average scores of the RAG approach.

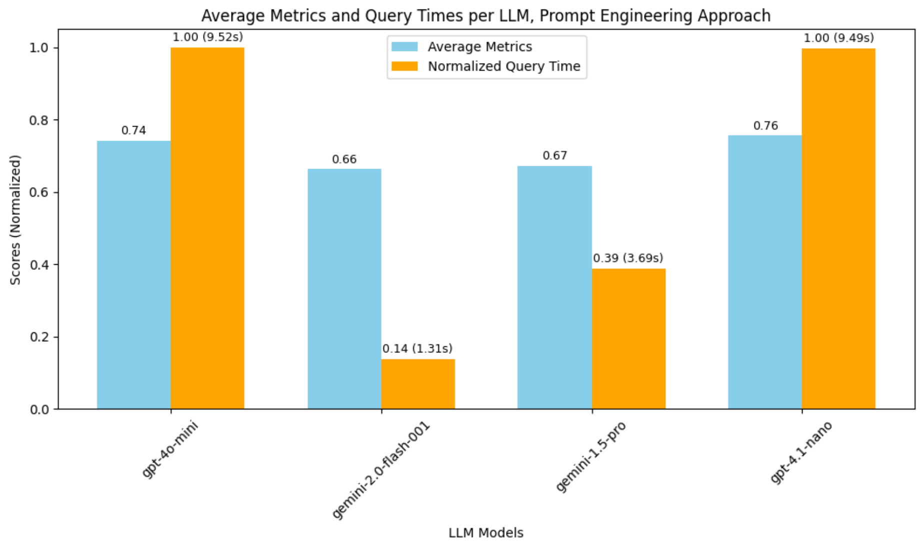


Figure 5.6: Comparison of average metrics for all the LLMs for Prompt Engineering approach with normalized average times to generate a response

This comparison shows the trade-offs between the Prompt Engineering and RAG approaches across various LLMs. While Prompt Engineering often has slightly better metric scores, it mostly results in significantly longer response times, which is not practical for real-time applications. The RAG approach when combined with models such as **gpt-4.1-nano** and **gemini-2.0-flash** offers a favorable balance between performance and speed. Notably, **gemini-2.0-flash** stands out in Prompt Engineering for its exceptional speed and better performance. However, RAG still seems to be the more suitable choice for real time applications because of the efficiency and cost of the LLM calls that will be described in the next section.

5.3 Efficiency and cost

In the previous sections, we discovered the most efficient configurations based on the trade-off between response quality and latency. The best performing setup in terms of average evaluation metrics and response time was the Prompt Engineering approach using the `gemini-2.0-flash` model. The second best result was achieved by the RAG approach with the configuration: chunk size of 1500 characters, topK set to 6, embedding model gemini-embedding-exp-03-07, and an overlap of 250 characters, evaluated with the LLM `gpt-4.1-nano`.

As explained in Section 2.2.1, LLM query cost is typically measured in tokens. This aspect is critical when considering the scalability and efficiency of deploying such a system in a production environment.

In the RAG configuration, the LLM receives a context composed of the top 6 most relevant chunks, each containing approximately 1500 characters. This results in a total of roughly 9,000 characters (or about 2250 tokens) passed to the LLM per query. In comparison, the Prompt Engineering approach sends the full concatenated text of the source document approximately 108,000 characters (or about 27,000 tokens) as input context. This results into a multiple times larger token count per query, resulting in higher computational costs.

The Prompt Engineering approach introduces several challenges when deployed at scale:

- **Scalability Issues:** As the size of source data increase (long sports match broadcasts), even models with extended context windows may be unable to keep all relevant information. This affects their effectiveness.

- **Reduced Relevance:** Including all content together can introduce noise. It cause the LLM to care about less relevant parts of the input therefore we can see low answer relevancy for the best resulting prompt engineering approach 5.4.

- **Higher Cost per Query:** Because to the increased token usage, each Prompt Engineering query is multiple times more expensive in terms of API cost. This makes it less suitable for large scale deployment.

In contrast, the RAG approach provides a more cost efficient and scalable alternative by dynamically selecting only the most relevant content. With minimizing token usage while preserving high response quality.

Chapter 6
SUMMARY

The summary will consist of answering research questions based on evaluating different methods used in the study and sharing my own opinion about it. My thoughts, limitations and possible future improvements on this work will be discussed in the discussion section. And lastly I will propose what future improvements can be made to this study. This will hopefully help us understand the final result of the thesis and answer the research questions.

6.1 Conclusion

This thesis explored the problem of delivering accurate and timely responses in a chat bot built for sports live-streams using LLMs. The main goal was to investigate how relevant context from a live-stream can be provided to an LLM to maximize performance. Two strategies were evaluated: prompt engineering (providing full raw content directly) and Retrieval-Augmented Generation, which uses a vector store to retrieve and send as context selected chunks of information.

The first research question focused on *how context from live-stream content can be provided to a Large Language Model (LLM) in order to build an accurate chat bot*. Two main approaches were explored to answer this question: (1) prompt engineering, where the entire available context is appended directly to the user query, and (2) RAG, where only the most relevant content chunks are retrieved and passed to the model. The experiments showed that both approaches can lead to high performing results under specific conditions. However, RAG offered better scalability and adaptability to increasing data size. Prompt engineering performed strongly only with specific LLM (gemini-flash-2.0) but it was cost unefficient.

To answer the sub-question on *what are the most effective hyperparameters for RAG and prompt engineering*, an extensive evaluation was performed on 27 RAG configurations involving different chunk sizes (600, 1500, 300), top-K retrieval values (2, 4, 6), and three embedding models (text-embedding-ada-002, text-embedding-3-large, gemini-embedding-exp-03-07). By visualizing these configurations we found out that the best-performing RAG setup used a **chunk size of 1500 character with overlap of 250 characters, topK=6, and the gemini-embedding-exp-03-07 embedding model**. For the prompt engineering approach there were no hyperparameters to be researched.

Regarding *how Retrieval-Augmented Generation compares with prompt engineering*, results showed that prompt engineering with the LLM gemini-flash-2.0 generated the highest average performance across evaluation metrics. However, it required passing the full context (equivalent to roughly 27,000 tokens or more) for each query, resulting in high cost. On the other hand, the best RAG configuration achieved slightly lower but still strong performance. However, it was passing only approximately 2.250 tokens per query, leading to a much more cost efficient solution. Therefore, for real-time application like live-streaming chat bot, RAG is a more sustainable approach than prompt engineering. Especially under high user load and longer source content.

Finally, to answer *what are effective methods to evaluate and compare the performance of RAG and prompt engineering*, the thesis used RAGAS metrics: Answer Correctness, Answer Relevancy, Semantic Similarity, Context Precision, Context Recall, and Faithfulness for RAG approach. For prompt engineering were used only Answer Correctness, Answer Relevancy and Semantic Similarity metrics because there is no context retrieval part to be evaluated. Additionally , these metrics were supported by average response time measurements. The combination of these metrics gave a detailed way to compare different LLM-based chatbot methods for live-streaming environments. The visualizations used — radar plots, heatmaps, and grouped bar charts. These enabled to interpret the comparisons across configurations.

In conclusion, both RAG and prompt engineering are good techniques for building LLM-based chat bots for live-stream content. However, RAG with optimized hyperparameters shows to be more scalable and cost effective solution for real-time responsiveness. Especially in scenarios with large or continuously growing content.

6.2 Discussion

Now when I concluded the research question it is time to discuss the ideas, limitations and future improvements of this topic. The main objective of this thesis was to design a chatbot that could effectively interact in real-time by using the context of a live video stream. To achieve this, I tried out many RAG configurations and LLMs.

The main focus was on maximizing the chatbot's effectiveness by experimenting with various setups and comparing their performance. LLMs were selected based on research and the constraints of real-time application. Smaller LLMs were suitable more for this kind of approach because they are faster and cheaper. Although some larger, more advanced models (such as gemini-1.5-pro) were tested, the amount of evaluation with these was limited by financial constraints. The entire evaluation process was conducted with a total cost of approximately 25 EUR.

6.2.1 Limitations

Despite the promising findings, there were some limitations hindering this research:

- **Lack of human evaluation:** The chatbot's performance was evaluated entirely through automated metrics. While these metrics provide objective comparability, they do not capture subjective qualities like user satisfaction or engagement. Human evaluation, either through surveys or direct user feedback, was outside the scope of this thesis due to time and resource limitations.
- **Lack of metadata and multi-source context:** The current system relied solely on textual transcripts for context. Incorporating metadata, such as player statistics, real-time game state, or historical match information, could have significantly enhanced chat bot responses.

6.2.2 Future Improvements

By stating the limitations there opened new parts of the project for exploring:

- **Integration of real-time context and metadata:** Incorporating additional structured data sources—such as player statistics, game metadata, or audio-visual event triggers—could significantly improve the retrieval quality and contextual accuracy of the chatbot's responses.
- **Dynamic retrieval and summarization:** A dynamic system that continuously updates its retrieval window and summarizes older content could help manage long-form context effectively and avoid exceeding the LLM's token limit.
- **Human evaluation:** Future studies should involve real users to validate the qualitative aspects of chatbot responses. A/B testing different configurations or collecting real-time user ratings could offer valuable insights for tuning model behavior.
- **Hybrid strategy adaptation:** Depending on the situation (e.g., low activity periods vs. high-intensity moments in a stream), the system could dynamically switch between prompt engineering and RAG strategies to optimize for cost, speed, and performance.

In summary, while this thesis achieved promising results with limited resources, there is still significant room for optimization and improvement.

Bibliography

[1] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Kaiser, and I. Polosukhin, “Attention is all you need,” in *Advances in neural information processing systems*, 2017.

[2] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W.-t. Yih, T. Rocktäschel, S. Riedel, and D. Kiela, “Retrieval-augmented generation for knowledge-intensive NLP tasks,” *Advances in Neural Information Processing Systems*, vol. 33, 2020.

[3] MyScale, “The ultimate guide to evaluate rag system components,” 2024.

[4] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving language understanding by generative pre-training,” *OpenAI*, 2018.

[5] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, “Unleashing the potential of prompt engineering in large language models: a comprehensive review,” *Guangdong Provincial Key Laboratory of Interdisciplinary Research and Application for Data Science*, 2023. BNU-HKBU United International College, Beijing Normal University.

[6] OpenAI, “Gpt-4 technical report,” *arXiv preprint arXiv:2303.08774*, 2023.

[7] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2018.

[8] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, “Language models are few-shot learners,” *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[9] P. Finardi, L. Avila, R. Castaldoni, P. Gengo, C. Larcher, M. Piau, P. Costa, and V. Caridá, “The chronicles of rag: The retriever, the chunk and the generator.” arXiv preprint arXiv:2401.07883, 2024. Preprint.

[10] R. Akkiraju, A. Xu, D. Bora, T. Yu, L. An, V. Seth, A. Shukla, P. Gundecha, H. Mehta, A. Jha, P. Raj, A. Balasubramanian, M. Maram, G. Muthusamy, S. R. Annepally, S. Knowles, M. Du, N. Burnett, S. Javiya, A. Marannan, M. Kumari, S. Jha,

E. Dereszenski, A. Chakraborty, S. Ranjan, A. Terfai, A. Surya, T. Mercer, V. K. Thanigachalam, T. Bar, S. Krishnan, J. Jaksic, N. Algarici, J. Liberman, J. Conway, S. Nayyar, and J. Boitano, "Facts about building retrieval augmented generation-based chatbots," 2024.

[11] A. Madan, P. Rai, A. Paranjape, P. Bhargava, A. Zareian, M. Kale, and A. Dubey, "Ragas: An evaluation framework for retrieval-augmented generation," *arXiv preprint arXiv:2309.15217*, 2023.

[12] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, 1950.

[13] M. T. ZEMČÍK, "A brief history of chatbots," 2024.

[14] D. M. Berry, "The limits of computation: Joseph weizenbaum and the eliza chatbot," 2024.

[15] E. Adamopoulou and L. Moussiade, "Chatbots: History, technology, and applications," *ScienceDirect*, 2020.

[16] J. R. Bellegarda, "Large–scale personal assistant technology deployment: the siri experience," in *Interspeech 2013*, 2013.

[17] A. Inc., "Siri - apple," 2023.

[18] Anthropic, "Introducing the next generation of claude," March 2024.

[19] K. Shuster, D. Vyas, P. Mazaré, D. Kiela, and S. Roller, "Ragas: An evaluation framework for retrieval augmented generation," 2023.

[20] D. Reddy, "Increasing customer service efficiency through artificial intelligence chatbot," *Revista de Gestão*, vol. 29, no. 1, pp. 19–33, 2021.

[21] W. Su, Y. Tang, Q. Ai, Z. Wu, and Y. Liu, "Dragin: Dynamic retrieval augmented generation based on the information needs of large language models," 2023.

[22] Cloudflare, "What is http live streaming?," 2025.

[23] N. Muennighoff, L. Tunstall, L. Magne, T. Dettmers, T. Li, L. F. Berrios, L. Rimell, A. Williams, T. Schick, S. Ruder, N. Goyal, D. Dey, and T. L. Scao, "Massive text embedding benchmark (mteb) leaderboard." Hugging Face Spaces, 2023. https://huggingface.co/spaces/mteb/leaderboard.

Appendix A

Additional Comparison Visualizations

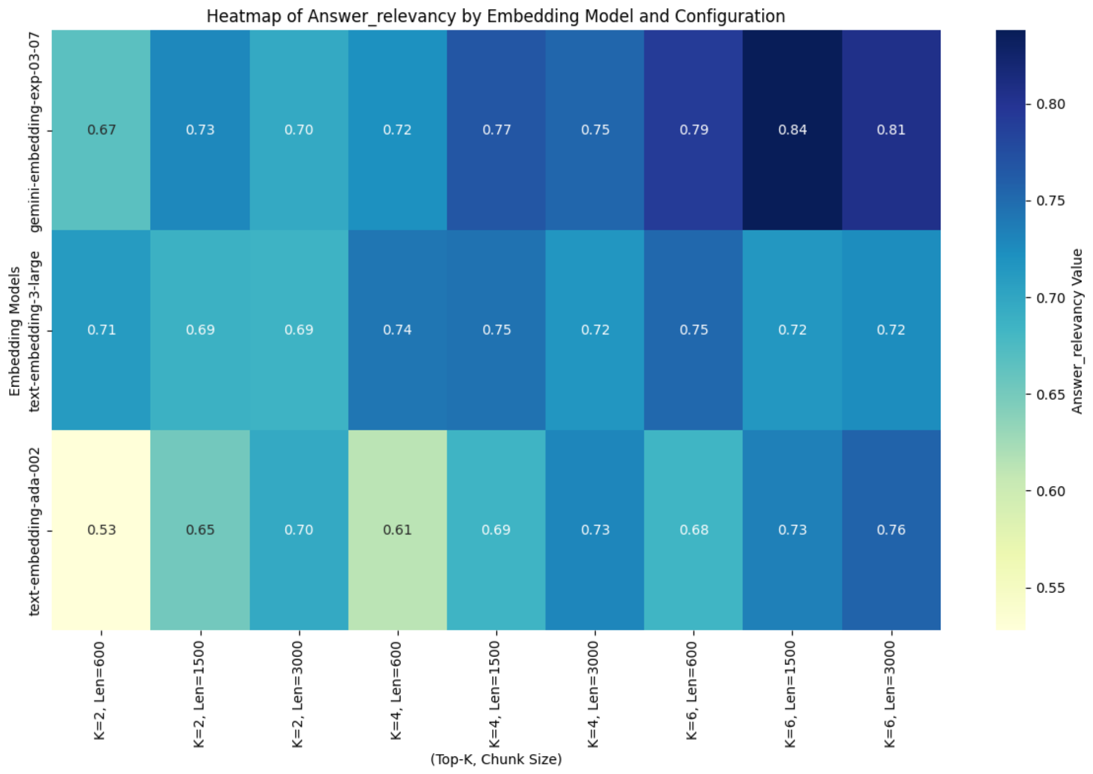


Figure A.1: Heatmap of answer relevancy by embedding model, topK and chunk size configurations.

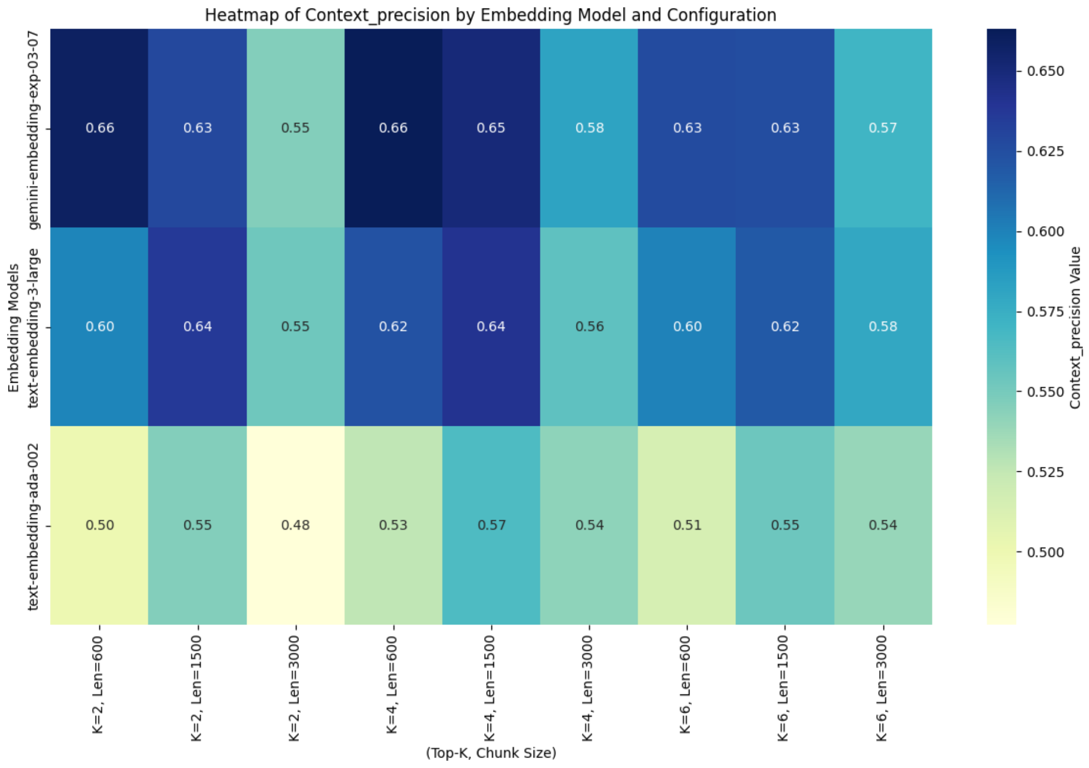


Figure A.2: Heatmap of context precision by embedding model, topK and chunk size configurations.

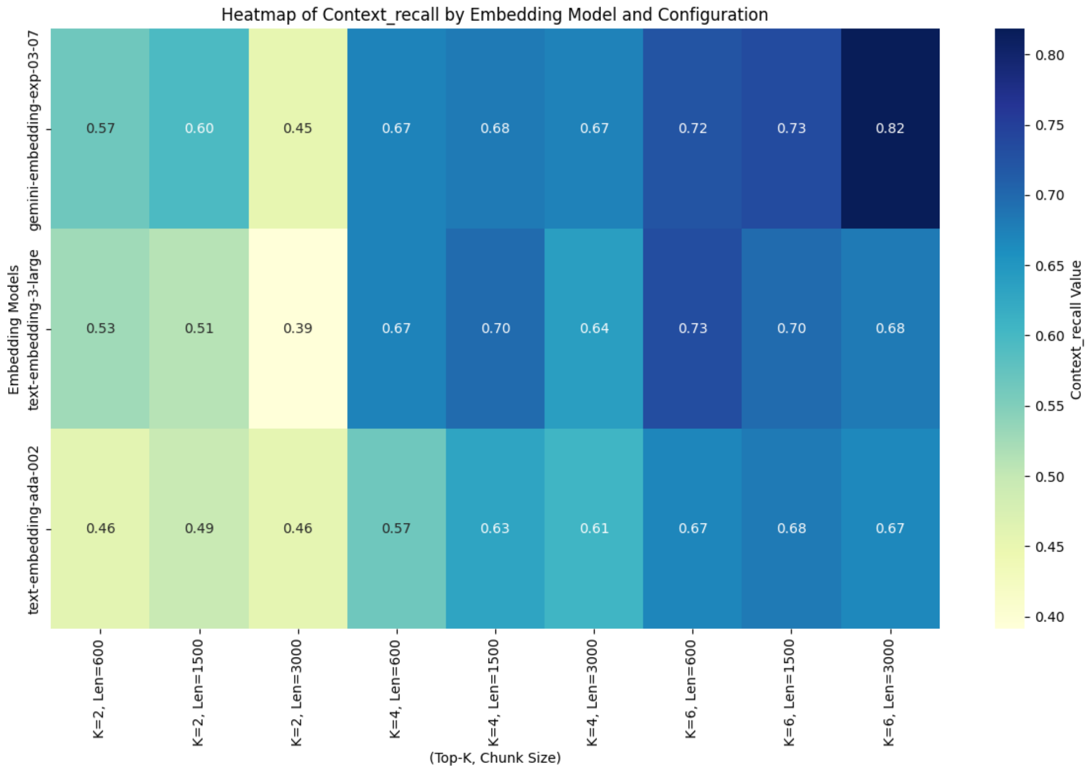


Figure A.3: Heatmap of context recall by embedding model, topK and chunk size configurations.

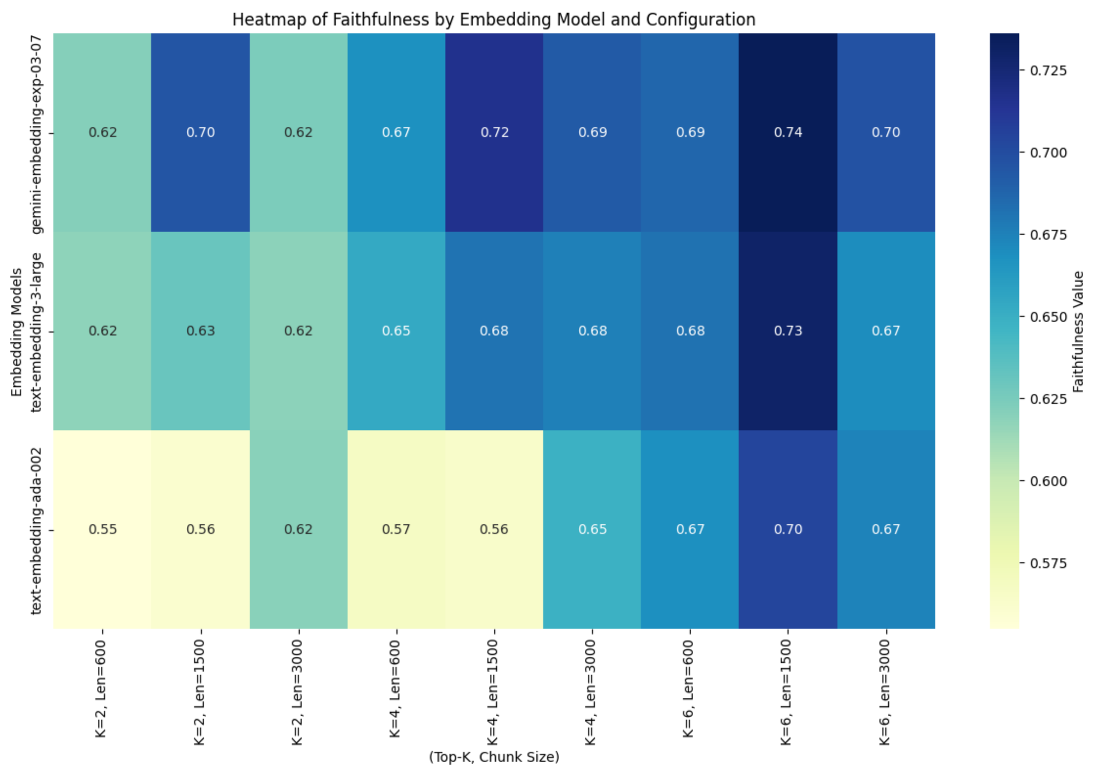


Figure A.4: Heatmap of faithfulness by embedding model, topK and chunk size configurations.

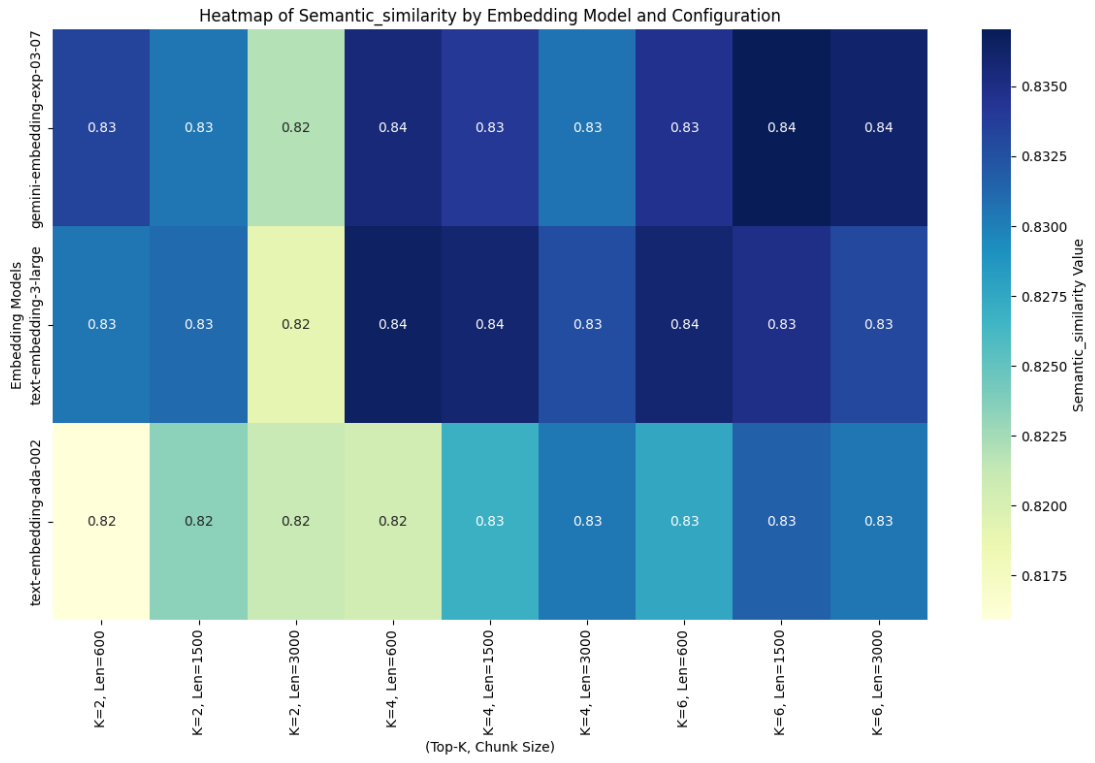


Figure A.5: Heatmap of semantic similarity by embedding model, topK and chunk size configurations.

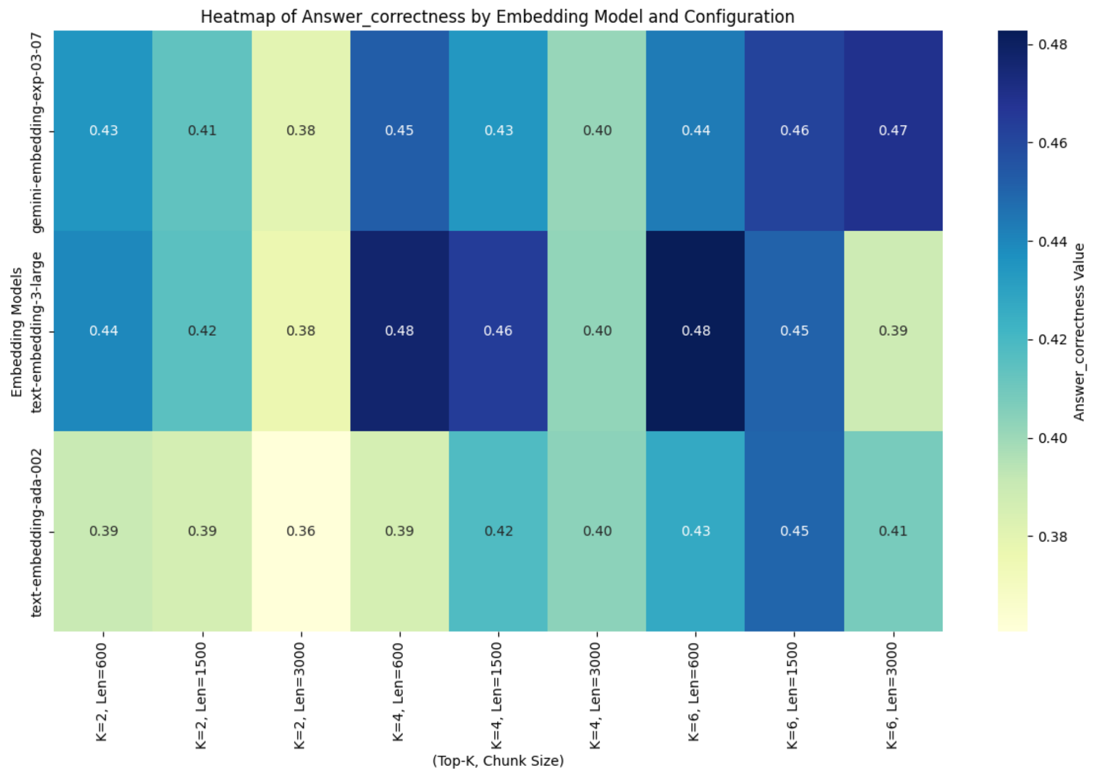


Figure A.6: Heatmap of answer correlation by embedding model, topK and chunk size configurations.

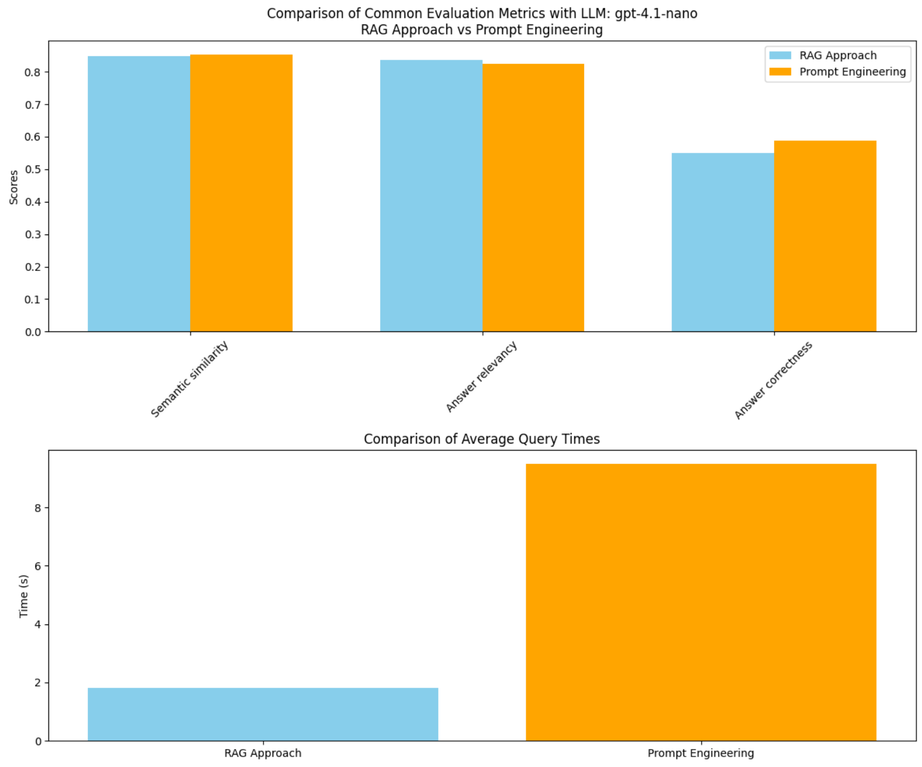


Figure A.7: Comparison of Common Evaluation Metrics and average response generation times with LLM: gpt-4.1-nano between best RAG configuration and Prompt engineering approaches

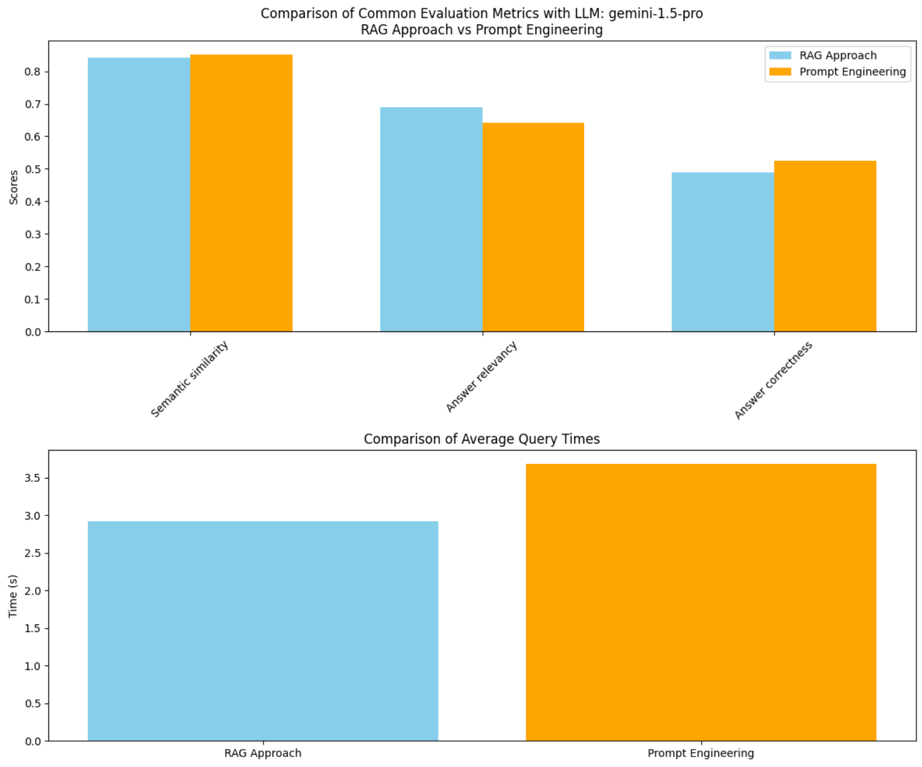


Figure A.8: Comparison of Common Evaluation Metrics and average response generation times with LLM: gemini-1.5-pro between best RAG configuration and Prompt engineering approaches