UiA University of Agder

Master

Evaluating and Enhancing Custom AI Chat Services

Towards Increasing the Speed, Accuracy and Cost Efficiency of Retrieval Augmented Generation Pipelines

ERIK DALE

SUPERVISORS
Morten Goodwin and Per-Arne Andersen

University of Agder, 2024
Faculty of Engineering and Science
Department of Information and Communication Technology

Obligatorisk gruppeerklæring

Den enkelte student er selv ansvarlig for å sette seg inn i hva som er lovlige hjelpemidler, retningslinjer for bruk av disse og regler om kildebruk. Erklæringen skal bevisstgjøre studentene på deres ansvar og hvilke konsekvenser fusk kan medføre. Manglende erklæring fritar ikke studentene fra sitt ansvar.

1.	Vi erklærer herved at vår besvarelse er vårt eget arbeid, og at vi ikke har brukt andre kilder eller har mottatt annen hjelp enn det som er nevnt i besvarelsen.	Ja
2.	**Vi erklærer videre at denne besvarelsen:** • Ikke har vært brukt til annen eksamen ved annen avdeling/universitet/høgskole innenlands eller utenlands. • Ikke refererer til andres arbeid uten at det er oppgitt. • Ikke refererer til eget tidligere arbeid uten at det er oppgitt. • Har alle referansene oppgitt i litteraturlisten. • Ikke er en kopi, duplikat eller avskrift av andres arbeid eller besvarelse.	Ja
3.	Vi er kjent med at brudd på ovennevnte er å betrakte som fusk og kan medføre annullering av eksamen og utestengelse fra universiteter og høgskoler i Norge, jf. Universitets- og høgskoleloven §§4-7 og 4-8 og Forskrift om eksamen §§ 31.	Ja
4.	Vi er kjent med at alle innleverte oppgaver kan bli plagiatkontrollert.	Ja
5.	Vi er kjent med at Universitetet i Agder vil behandle alle saker hvor det forligger mistanke om fusk etter høgskolens retningslinjer for behandling av saker om fusk.	Ja
6.	Vi har satt oss inn i regler og retningslinjer i bruk av kilder og referanser på biblioteket sine nettsider.	Ja
7.	Vi har i flertall blitt enige om at innsatsen innad i gruppen er merkbart forskjellig og ønsker dermed å vurderes individuelt. Ordinært vurderes alle deltakere i prosjektet samlet.	Nei

Publiseringsavtale

Fullmakt til elektronisk publisering av oppgaven Forfatter(ne) har opphavsrett til oppgaven. Det betyr blant annet enerett til å gjøre verket tilgjengelig for allmennheten (Åndsverkloven. §2). Oppgaver som er unntatt offentlighet eller taushetsbelagt/konfidensiell vil ikke bli publisert.

Vi gir herved Universitetet i Agder en vederlagsfri rett til å gjøre oppgaven tilgjengelig for elektronisk publisering:	Ja
Er oppgaven båndlagt (konfidensiell)?	Nei
Er oppgaven unntatt offentlighet?	Nei

Acknowledgements

This master's thesis was conducted at the University of Agder (UiA) in Grimstad, Norway, by Erik Dale, completing the master's program at the Department of Information and Communication Technology (ICT). I want to first of all give a big thank you to Egde and especially their CTO Øyvind Brekkhus Sandåker, for letting me write this thesis in cooperation with them. EgdeAI was a project that a group of students and I started working on as summer interns at Egde in 2023, and I have loved working on it every step of the way. I of course also want to give a big thank you to my supervisors at the University of Agder: Morten Goodwin and Per-Arne Andersen. You guys are great. Lastly, I would like to thank family and friends, and of course you Mariana: gracias amor.

During the writing process of this thesis, I have used a few AI writing tools. OpenAI's ChatGPT and Grammarly are great tools that I have used for inspiration and language cleaning. That being said, all the text in the thesis is my own (except for some of the text in Appendices B, C, D, E and F due to the nature of this thesis).

Grimstad,
May 24, 2024
Erik Dale

Abstract

Custom AI chat services that utilize Retrieval Augmented Generation are becoming more and more common within different sectors and businesses. When these types of services are used in professional settings like a workplace, they have to be fast, cost-effective, and reliable. The Large Language Models that these services use are often provided as a service with limited access to the software itself, making fine-tuning downstream tasks a challenge. This thesis introduces a Retrieval Augmented Generation pipeline designed to enhance the time and cost efficiency, as well as the reliability, of AI chat services tailored to specific user needs. These advancements have been realized without modifying the underlying architecture of Large Language Models. Instead, they leverage prompt engineering strategies, including prompt compression and prompt classification, to elevate performance and efficiency. Given the critical role of embeddings in these services, this thesis has conducted an exploration of embedding models to find the best one for maximum enhancement. This thesis shows that the enhanced Retrieval Augmented Generation pipeline can achieve a cost reduction of almost 69% compared to a standard Retrieval Augmented Generation pipeline without using any of the proposed enhancements. It has also been shown that utilizing an open-source embedding model can increase efficiency by as much as 41% compared to similar OpenAI models. This advancement highlights the potential of this method to enhance custom AI chat services.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large Language Models (LLMs) like ChatGPT [23] are often provided as a service, with limited or no access to model parameters. Fine-tuning them for downstream tasks can therefore pose a challenge [32]. Retrieval Augmented Retrieval (RAG) has become a popular method for fine-tuning LLMs and custom AI chat services, which allows for creating comprehensive, external knowledge bases for LLMs through resources such as vector databases [25, 7]. Companies are now trying to utilize RAG pipelines with LLMs to streamline their own productivity [8]. While this approach creates ample opportunities, it also presents considerable challenges in terms of cost and time efficiency, accuracy, and explainability [11]. Custom AI chat services used by large companies in professional settings need to be effective both in terms of cost and speed. Such services also have to be accurate and explainable, as more and more people in professional settings put their trust in them to provide truthful information. Achieving this in RAG pipelines is non-trivial, as it involves complex interactions between the underlying model and the external knowledge bases [15, 5]. It involves navigating through an abundance of choices and trade-offs each with its own implications for performance, efficiency, and cost.

We present a Retrieval Augmented Generation pipeline to enhance the efficiency, cost-effectiveness, and reliability of AI chat services made for specific user requirements. These improvements have been achieved without altering the structure of Large Language Models. Instead, they leverage prompt engineering strategies, such as prompt compression and prompt classification, to boost both performance and efficiency. Recognizing the important role of embeddings in these services, we also explore various embedding models to identify the most effective one for the best enhancement. The results, as described in Section 4.6.1, demonstrate that the enhanced Retrieval Augmented Generation pipeline reduce costs up to 69% compared to a traditional Retrieval Augmented Generation pipeline without the suggested improvements. Furthermore, we

show that using an open-source embedding model can increase efficiency by as much as 41% when compared to similar models from OpenAI. This progress underscores the potential of this approach to enhance custom AI chat services, as further detailed in Section 4.6.

1.1 Motivation

In the current landscape of LLMs there is a need for more advanced customization and fine-tuning targeted for more specific, professional use cases. In settings like these, reliability is key, and ensuring it is important. Reliability when it comes to cost is also crucial for ensuring a good AI chat service, as a service like this can get quite pricey in a big company. That is why this thesis explores techniques like prompt classification and prompt compression to reduce cost and increase efficiency. There is also a need for a better understanding of LLMs' reasoning, and why they answer the way they do. These models are known to lie and fabricate non-existent facts or inappropriate information also known as hallucinations [35]. This becomes a problem when more and more people put their trust in these models at work and in their daily lives. It is important that they can be relied upon as sources of information. This thesis looks into enabling explainability with things like sources and reasoning to solve this issue. It enables the end users to fact-check and verify the answer.

1.2 Field of Research

This thesis focuses on custom LLMs made for professional use cases. One of the many tasks in this domain is to increase the reliability and explainability of such models. Due to commercial considerations and risks of misuse, these models are mostly provided as a service, with no access to model parameters. This makes it challenging to fine-tune LLMs for specific use cases and achieve explainability. However, with the recent and rapid advances in mechanisms to fine-tune LLM's, we now have new avenues that needs further study. This thesis will look into combining some of them like prompt engineering, CoT prompting, and prompt classification to achieve an increase in cost-effectiveness, reliability, and explainability of custom LLMs.

1.3 Thesis Definition

The main focus and goal of this thesis is to enhance custom AI chat services that utilize RAG pipelines. To accomplish this, the project is broken down into the following research questions and hypotheses.

1.3.1 Research Questions and Hypotheses

RQ1: To what extent can the reliability and cost-effectiveness of state-of-the-art AI driven chat services be increased?

Different methodologies are used in order to answer this research question. Prompt engineering and prompt compression are utilized to reduce cost and increase reliability of answers. An exploration of prompt classification is conducted to test if it is possible to use different LLMs for different prompts. A comparison of embedding models will also be done to find the most accurate and cost-effective one. The following hypotheses (H1 - H4) are attached to **RQ1**:

- **H1:** It is possible to utilize prompt engineering to make custom AI chat services faster, more cost-effective, and more accurate than without prompt engineering.
- **H2:** Prompt classification can be used to reduce cost and completion time in a RAG pipeline.
- **H3:** A reduction in cost and completion time is possible to achieve in a RAG pipeline by using prompt compression.
- **H4:** Open-source embedding models are as good as OpenAI's alternative, and will therefore be possible to use to reduce cost and completion time.

RQ2: To what extent is it possible to achieve explainability for custom AI chat services?

Prompt engineering will here be used to add reasoning, Chain of Thoughts (CoT) and sources to the LLM's answer. The following hypothesis is attached to **RQ2**:

- **H5:** Prompt engineering, Chain-of-Thoughts, and sources can be used to make LLMs like the GPT models more explainable.

1.4 Contributions

The work in this thesis proposes an advancement in reliability and explainability when it comes to custom chatbots built on LLMs such as the GPT models. The main contribution of the work is an enhanced RAG pipeline that starts with the user sending a prompt and then getting the answer back from an LLM. In this RAG pipeline, different techniques like prompt engineering, prompt classification, and embedding enhancement have been used. Different

methodologies and frameworks for evaluating how these techniques increases reliability, explainability, efficiency, and cost-effectiveness have also been introduced and conducted.

1.5 Pre-project

This master thesis is a continuation of work done in a previous project titled *Optimizing and Evaluating EgdeAI - A Custom Chatbot for Employees of Egde* by Dale [8]. Work done here can be read more about in the Related Work section (2.2.1) below.

1.6 Thesis Outline

The contents of this thesis report are divided into the five following chapters. These should be read sequentially from chapter to chapter. The following is a short description of the content of each chapter.

Chapter 1 - Introduction provides an overview of the thesis task, research area, and motivation for commencing with the project work.

Chapter 2 - Background goes through the key terms and work done before related to LLMs, RAG pipelines, and methods used to enhance them.

Chapter 3 - Methods describes in detail the RAG pipeline itself, as well as the different methodologies used to enhance it.

Chapter 4 - Results and discussion includes the results derived from the enhanced RAG pipeline in the *Methods* chapter, as well as a discussion of them.

Chapter 5 - Conclusion is where the work is concluded and limitations are specified. Potential future work is also included here.

Chapter 2

Background

This chapter is a walk-through of key terms and work done before related to LLMs, RAG pipelines, and methods used to enhance them. The theory and key terms can be found in Section 2.1, while Section 2.2 contains related work.

2.1 Theory

2.1.1 Large Language Models

Huge datasets are needed to train LLMs, often internet-scale ones. Recognition, summarizing, translation, classification, and generating content are some of the tasks they are capable of. LLMs are made up of something called transformer models. Introduced in 2017 by Google, transformer models utilize something called positional encodings and self-attention, which makes them perfect for LLMs. Positional encodings take into consideration the order in which the input occurs in a given sequence, which is why they are important when embedding text. The self-attention on the other hand gives weight to each word in the input, signifying the importance of each word in relation to the other ones. Together with massive datasets, these two techniques have made it possible for LLMs to generate human-like content [22].

2.1.2 Similarity Measures

When it comes to creating custom knowledge bases for LLMs, vector databases are a solid and popular option [25] [7]. It is possible to query a vector database with natural language, by doing something called a similarity search. Euclidean Distance, Manhattan Distance, Jaccard Similarity, Minkowski Distance, and Cosine Similarity are some of the most popular similarity search algorithms. The last one of these, Cosine Similarity Search, is a metric of how similar two vectors are irrespective of their size, as it measures the angle between them. The dot product of the vectors is first calculated, before being divided by the product of their magnitudes. Equation 2.1 below shows the operation of finding

the cosine similarity between a and b [10].

$$cossim(a, b) = \frac{a \cdot b}{||a|| \cdot ||b||} \tag{2.1}$$

2.1.3 Microsoft Prompt Flow

Microsoft Prompt Flow (MSPF) is designed to make the development cycle of LLMs a lot easier. The concept of MSPF is to take a prompt as input, before leading it through various stages to create meaningful output at the end of the flow. These different stages are called nodes and can be LLMs, prompts, vector databases, python scripts, and various other tools. Figure 2.1 below shows four nodes connected in a flow. MSPF also makes debugging, sharing, iterating, and deploying AI applications easier.

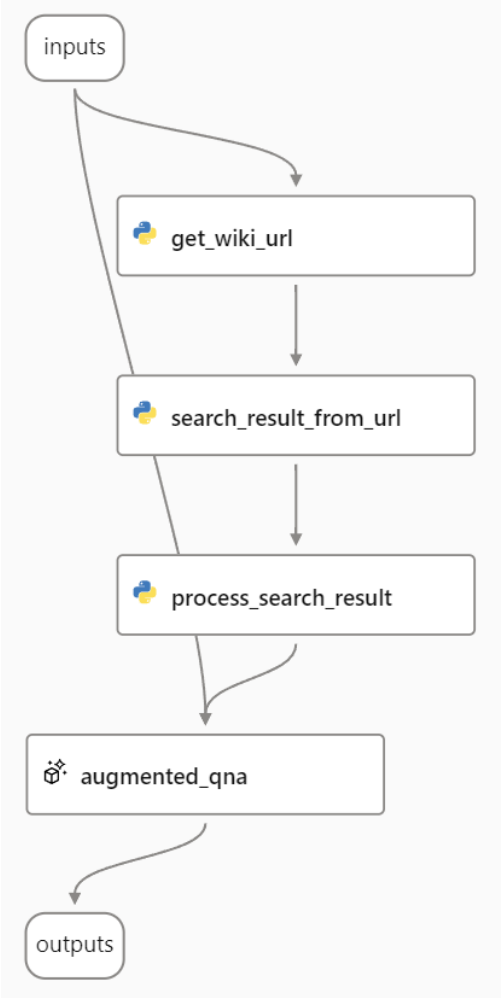

Figure 2.1: This flow provides an LLM with contextual information obtained from Wikipedia, in response to the prompts submitted. Each box represents a node.

2.1.4 Prompt Engineering

As LLMs are becoming increasingly popular in an increasing number of fields and domains, it is crucial to have a way of customizing or programming them to answer the best way possible. This is where prompt engineering comes in. Through so-called prompt templates, prompt engineering can be used to customize, enhance, and refine the capabilities of an LLM. Prompt engineering is in other words the means by which LLMs are told how to behave in a variety of different situations. A common way to use prompt engineering is to create a persona for the LLM, and instruct it what its main objectives are [33]. This might look something like this:

> Your task as an AI assistant is to help computer engineering students understand docker. I would like you to ask them questions about their repositories. When you have enough information to create a Dockerfile, do that for them, and then show them how to deploy it [8].

Other common techniques include giving the LLM some example questions and answers so that it knows what the general structure of its answer should be. There might be times when an AI chat service is not what is needed, but rather a classifier. The classifying task can then be stated in the prompt template, and the LLM will output a classification result rather than an answer [33].

2.1.5 Explainability in Large Language Models

The ability to explain the decision-making of AI in a human-understandable manner is referred to as explainability. When it comes to LLMs there are two main reasons why explainability is crucial. For the end user, explainability builds trust by providing reasoning in an understandable manner, without the need to have technical expertise. It makes it easier for the user to understand the capabilities, limitations, and potential flaws of the LLM. Explainability does not only benefit the end user, however, but also researchers and developers. It provides them with insight into LLMs, which allows them to identify biases, risks, and areas for performance improvement. The scale of LLMs in terms of parameters and training data makes explainability both challenging and exciting. Types of explainability include attribution-based explanation, attention-based explanation, example-based explanation, and natural language explanation. Attribution-based can, for example, measure the relevance of each word, phrase, or text span. Attention looks more into meaningful correlations between input, like words or phrases. Natural language explanation on the other hand makes the LLM automatically generate its own explanations as to why it answered the way it did. Chain of thought (CoT) explanation is a way

for the LLMs to create explanations themselves using natural language explanation. This is done by steering its generation in a particular direction, by giving it a sequence of prompts, and having it explain its reasoning [36].

2.1.6 Langchain

Langchain is an open-source software library to use when working with LLM applications. It, among other things, offers a way to connect LLMs to sources of context like vector databases. Other than that, Langchain also offers functionality for creating prompts, memory, chains, and agents. Prompts are, as mentioned in Section 2.2.3, a way to customize, enhance, and refine an LLM's capabilities. Langchain offers functionality for giving an LLM memory, which is neat when wanting to have a longer conversation with an LLM. Maybe the most important and unique feature of them all is chains. A chain allows for chaining together prompts and LLMs, as well as chaining together chains with each other. These are called sequential chains where the output of one chain becomes the input of another, as illustrated below in Figure 2.2 [28].

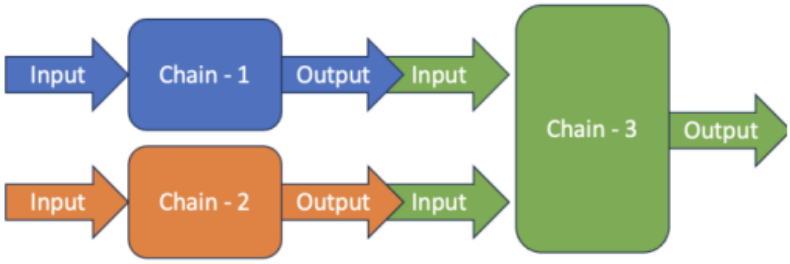

Figure 2.2: Example of a Sequential Chain that gets two inputs and outputs one result [28].

Langchain also offers something called agents, which can be seen as flexible chains. Agents have the ability to utilize tools that give the chain's LLM more functionality. A tool can for example be an API that lets the LLM perform internet searches or a calculator that gives the LLM the ability to perform better and more precise calculations [28].

2.1.7 Hugging Face

Hugging Face is a data science platform whose main goal is to enable the sharing of knowledge and resources to accelerate the development of AI. As a platform, it provides different tools for building, training, and deploying machine learning models based on open-source technologies and code. They believe no one will be able to 'solve AI' on their own, so they provide a community *Hub* where the goal is to democratize AI for all. On this *Hub* everyone can share and explore AI models as well as datasets. Through the use of different libraries, mostly in Python, users can download and use the pre-trained models and datasets from

the *Hub* themselves. These are divided into the main categories of Computer Vision, Natural Language Processing (NLP), Audio, and Multimodal, which are again divided into sub-categories. There is a collection of text sentence-transformer models used to create contextualized word embeddings in a sub-category called *Sentence Similarity* under the category of NLP [19].

2.1.8 AI and Sustainability

Critics of AI often point out that the usage and training of AI, requires a substantial amount of energy to work as intended. This could be in direct conflict with the United Nation's sustainability goal 7: access of clean energy to everyone. Stanford scientist Shana Lynch estimated that the total amount of CO_2 released both directly and indirectly when training the *gpt-3* model was 502 ton. Bloom, a similar models to those of the OpenAI models, is said to use 0.003 KWh per user prompt. ChatGPT had almost 600 million users just a month after release, and wit the same numbers as the Bloom model, it can be estimated that the CO_2 emissions amounted to 21 tons. AI can luckily, in many cases, also be used to reduce emissions. Google DeepMind managed to achieve great results when using AI to reduce the energy consumption of Google data centers. They did this by using historical data collected from thousands of sensors. Using this, they trained neural networks to recognize and understand various patterns, which allowed them to optimize the energy usage effectively. Methods like these can be used in various other industries and industrial systems to increase efficiency [12].

2.2 Related Work

The following sections include related work in the field of AI chat services utilizing RAG and the methodology that can be used to enhance them. These are customized to be used in professional settings across various companies, as stated in the introduction. The chapter starts in Section 2.2.1 with some examples of such services and what has been done to enhance them. Then, in Section 2.2.2, it takes a deep dive into the current state-of-the-art of Large Language Models. The following sections, go more in-depth into previous enhancement methodologies used when it comes to AI chatbot services. Firstly, it takes a closer look at *Prompt Engineering* in Section 2.2.3, which is a very prominent enhancement methodology. Section 2.2.4 includes literature about various ways to achieve explainability for Large Language models. Finally, Section 2.2.5 contains research and comparisons of embedding models, which are vital when doing Retrieval-Augmented Generation (RAG).

2.2.1 Enhancing Custom AI Chat Services

In a paper called *Optimizing and Evaluating EgdeAI - A Custom Chatbot for Employees of Egde* Dale proposes ways of improving and enhancing a custom AI chat service called EgdeAI. These include utilizing MSPF, a platform whose goal is to make the development of AI applications easier. Given a prompt as input, MSPF takes the input through various stages to create meaningful input at the end of the flow. This allows for prompt classification and prompt engineering, as well as testing and evaluating the flows. Through the use of prompt engineering and prompt classification, cost is reduced and efficiency is improved. To reduce cost Dale also takes a look at utilizing open-source sentence embedding models. This proves to be a much faster and cheaper way of embedding documents and prompts. Dale concludes by saying that future work could be to try out different classification models for prompt classification. Future work could also be to reduce the size of the prompt template or remove it entirely, by exploring fine-tuning[1] of OpenAI's GPT models. Fine-tuning is a way of pre-training a model to behave a certain way, similar to using a prompt template. The last thing Dale proposes as future work is to remove the need to send context with each prompt, by using OpenAI's *Assistants API*[2]. With this, it is possible to pre-train a model on up to 20 files of max size 512MB, which removes context input tokens altogether and greatly reduces cost [8].

The paper *PipeRAG: Fast Retrieval-Augmented Generation via Algorithm-System Co-design* from Jiang et al discovered that retrievals from large databases could take up a big portion of the generation time, especially when retrievals are carried out at regular intervals to ensure the content retrieved is in sync with the most recent states of generation. Therefore, they introduce PipeRAG, a model designed to enhance the efficiency of RAG systems. This is done by integrating pipeline parallelism to enable concurrent retrieval and generation processes, flexible retrieval intervals to maximize the efficiency of pipeline parallelism, and dynamic adjustment of retrieval quality through performance modeling. As database they use a RETRO database, which is a large database consisting of chunks of documents that are vectorized. RETRO conducts retrievals periodically while generating a sequence of t tokens $X = (x_1, ..., x_t)$. It divides X into l segments $(C_1, ..., C_l)$, with each segment containing m tokens. Combining this with the three aforementioned methods, their evaluation shows that PipeRAG achieves up to 2.6x faster end-to-end generation while improving generation quality. These encouraging results demonstrate the success of co-designing algorithms and underlying systems, setting the stage for the inte-

[1] https://platform.openai.com/docs/guides/fine-tuning
[2] https://platform.openai.com/docs/assistants/overview

gration of PipeRAG in upcoming RAG systems [15].

FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance is a paper authored by Chen et al. They are motivated by the fact that using LLMs on a large collection of queries and text can be expensive, leading them to discuss how to reduce cost and increase accuracy when using LLMs. To achieve this they propose three approaches: *prompt adaptation*, *LLM approximation* and *LLM cascade*. *Prompt adaptation* is a process where the size of the prompt is reduced, thereby also reducing the cost. They use *prompt selection*, which is a type of *prompt adaption*, to limit the amount of examples in a prompt, making it smaller and cheaper. Another type of *prompt adaptation* called *query concatenation* involves sending multiple queries to an LLM API at a time. *LLM approximation* involves, among other things, storing LLM responses in a cache to limit LLM API calls. Finally, *LLM cascade* is the process of sending different queries to different LLMs, as they have their own strengths and weaknesses for different queries. This can reduce cost and increase performance. Their proposed solution, called *FrugalGPT* can match the best individual LLM (e.g. *gpt-4*) with up to 98% reduction in cost or achieve 4% higher accuracy than the *gpt-4* model at the same cost. Their concepts and findings establish a basis for utilizing LLMs in a sustainable and efficient manner [5].

2.2.2 Large Language Models

Since the release of *ChatGPT* in November 2022 [23], a lot of competitors have emerged. In July 2023, Meta released a free-to-use, semi-open-source LLM called *Llama 2*. Unlike the GPT models, the *Llama 2* model is semi-open-source, allowing for transparency and customization. Users can fine-tune it for specific tasks, so as to achieve better results in niche areas [29]. Later that year in December, Google released their new family of LLMs called *Gemini* [26]. One of the impressive features of these is that the model *gemini-1.5-pro* has a token input limit of up to 1 million tokens. This equates to about 700,000 words and means that the model can understand entire books or podcast series [6]. Quite recently in March 2024, Anthropic introduced *Claude 3*, a family of large multimodal models [2]. The best of these models, *Claude 3 Opus*, outperforms the *gpt-4* model on various popular datasets like the *HellaSwag* and *GSM8K*. The actual numbers can be seen below in Table 2.1. Mistral AI is another company that has released open-source models, meaning they are free to use and can be fine-tuned like the *Llama 2* model. These models were released in September 2023, but their newest models, *Mistral Small, Medium, and Large*

were released quite recently in February 2024 [1].

Various datasets have been made to evaluate how such LLMs perform on various tasks in various domains. Some datasets, like the *HellaSwag*, assess an LLM's ability to do common sense reasoning. Others, like the *GSM8K*, are designed to evaluate a model's ability to do multi-step mathematical reasoning [20]. Other methods include using other LLMs for evaluation. A company called *Vectara* has created a quite interesting open-source hallucination evaluation model called HHEM [31]. The model that scores the best on the *HellaSwag* and *GSM8K* is the *Claude 3 Opus* model, with scores of 95.4% and 95% respectively. This HellaSwag result is quite impressive as it requires a deep understanding of the world and human behavior. In comparison, the average human score on the dataset is 95.6%, only 0.2% above that of the *Opus* model [13]. The model that best deals with hallucinations is the *gpt-4* model, with a score of 97%, indicating that the evaluated text is 97% factually consistent with the source information [20]. These results, along with the rest of the LLM's performances can be seen in Table 2.1 below.

Model	HellaSwag	GSM8K	HHEM	Released
GPT-4	95.3	87.1	**97**	March 2023
Llama 2 70B	87.33	56.8	92.9	July 2023
Gemini Ultra	87.8	94.4	95.2	December 2023
Mistral Large	89.2	91.21	-	February 2024
Claude 3 Opus	**95.4**	**95**	92.6	March 2024

Table 2.1: How the various LLMs mentioned have performed in various benchmarks. The HHME column contains the HHEM factual consistency score. A higher value is better for all the columns. Sources: [20, 13]

2.2.3 Prompt Engineering

A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT is a paper by White et al. where they propose a plethora of prompt patterns that offer reusable solutions to specific problems. As stated in Section 2.2.3, the role of these prompt patterns is to enhance, refine, and customize the capabilities of an LLM. The patterns are placed into different categories after what main functionality they offer. Patterns whose main functionality is to constrain what type, format, and structure the output has, are put into the *Output Customization* category. The *Prompt Improvement* patterns on the other hand are used to improve the quality of the input and the output. Within the *Output Customization* category there are multiple patterns like the *Persona* and *Template* pattern. The *Persona* pattern can be used if the output is wanted from

a certain point of view, like for example a security expert. An example of the *Persona* pattern can be seen in Section 2.2.3 A precise output structure can be achieved by using the *Template* pattern [33].

There are also quite a lot of interesting patterns whose role is to give better answers from the LLM and give reasoning behind the answers. The *Question Refinement Pattern* makes the LLM suggest potentially better or more refined questions to the user, in order to arrive at a more accurate answer. To achieve increased explainability for the LLM, the *Reflection Pattern* can be used. Its goal is to enable the model to automatically explain its reasoning and rationale behind the answers it provides to the user. The *Fact Check List Pattern* can be used to create a list of facts that the LLM bases its answer on. This can be useful if the user would want to perform due diligence on these facts to validate their correctness. Another interesting pattern is the *Refusal Breaker Pattern* whose goal it is to help the end user rephrase a question when the LLM is reluctant to give an answer. Caution is needed when using this pattern, however, as it might lead the LLM to generate answers that violate its policy filters [33]. The rest of the patterns and their corresponding categories can be seen below in Figure 2.3.

Pattern Category	**Prompt Pattern**
Input Semantics	*Meta Language Creation*
Output Customization	*Output Automater* *Persona* *Visualization Generator* *Recipe* *Template*
Error Identification	*Fact Check List* *Reflection*
Prompt Improvement	*Question Refinement* *Alternative Approaches* *Cognitive Verifier* *Refusal Breaker*
Interaction	*Flipped Interaction* *Game Play* *Infinite Generation*
Context Control	*Context Manager*

Figure 2.3: Prompt patterns and their categories [33].

Jiang et al. noticed that prompts fed to LLMs were becoming increasingly

lengthy, due to advancements in technologies like CoT and in-context learning (ICL). In a paper called *LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models* they therefore propose a solution to these lengthy prompts. They present *LLMLingua*, a prompt compression method that keeps the semantic integrity of prompts even under high compression levels. An illustration of their approach can be seen below in Figure 2.4. The motivation behind it is to reduce cost, increase context length and chat history, as well as increase efficiency. In their results, they show that a prompt with a lot of examples, consisting of 2366 tokens, was reduced to a prompt with one example and 446 tokens. As an evaluation, they utilize Exact Match (EM), and the compressed prompt actually gets a higher EM score than the bigger one. This shows that even though the prompt is about 1/5 the size, it still keeps its semantic integrity and the LLM is able to understand it [14].

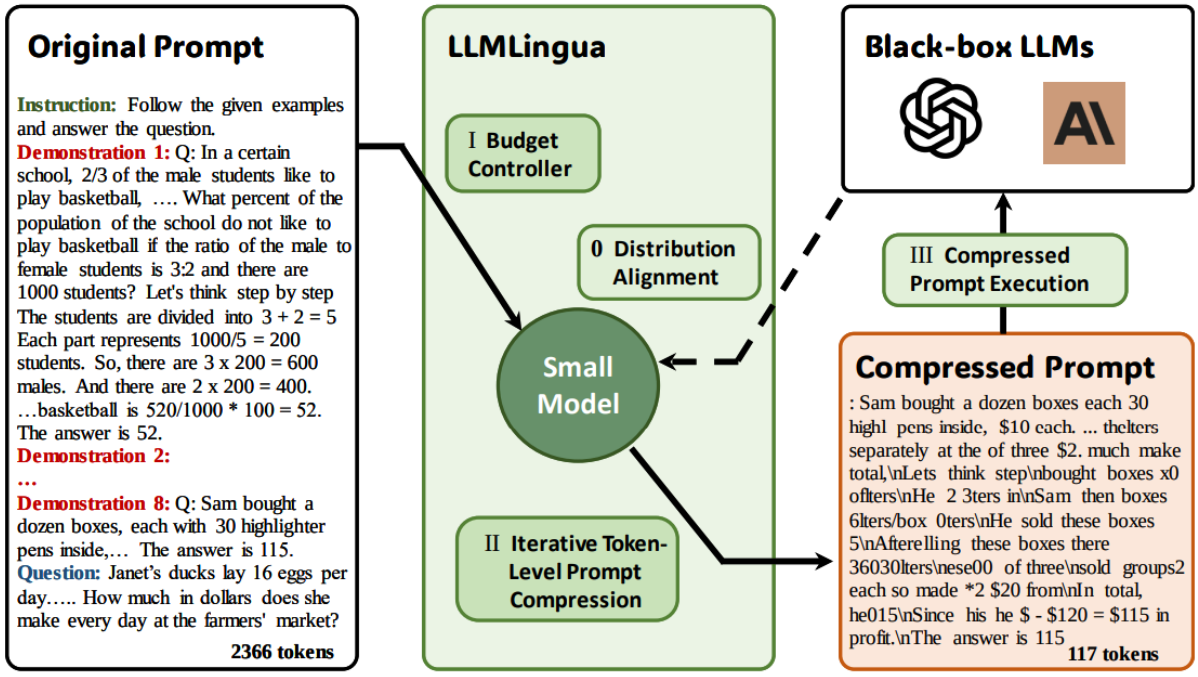

Figure 2.4: Framework of Jiang et al's proposed approach *LLMLingua* [14]

2.2.4 Explainability for Large Language Models

Therefore, understanding and explaining these models is crucial for elucidating their behaviors, limitations, and social impacts [36] is stated in a paper called *Explainability for Large Language Models: A Survey* by Zhao et al. This paper gathers and describes state-of-the-art techniques for implementing explainability when it comes it LLMs. They categorize the training of LLMs into two paradigms: a *traditional fine-tuning paradigm* and a *prompting paradigm*. The prompting paradigm is further divided into *base models* like GPT-3 and more

fine-tuned models with human-level abilities like *GPT-4*. To achieve explainability for these they propose using natural language explanation, which makes the LLM automatically generate its own explanations as to why it answered the way it did. An extension of this is the Chain of Thought (CoT) explanations, which includes giving it a sequence of prompts and having it explain its reasoning [36].

In a paper called *Plan-and-Solve Prompting: Improving Zero-shot Chain-of-Thought Reasoning by Large Language Models* Wang et al. propose Plan-and-Solve (PS) Prompting, which is an improvement of Zero-shot-CoT prompting. Zero-shot CoT is, in turn, an improvement to few-shot Cot because it eliminates the need for manually crafted step-by-step reasoning demonstrations that the latter requires. PS prompting consists of two components. The first component includes creating a plan for subdividing the bigger tasks into smaller ones, while the second one involves carrying out those subtasks. They also propose something called PS+ prompting, which is an extension of PS prompting with more detailed instructions. To evaluate their proposed prompting method they test it on ten datasets across three different reasoning problems. Across all datasets, their proposed prompting strategy outperforms Zero-shot-CoT. It is comparable to or better than Zero-shot-Program-of-Thought Prompting, and has comparable performance with 8-shot CoT prompting on the math reasoning problem [32].

Confidence elicitation is referred to as the task of enabling LLMs to accurately articulate their confidence in the answers they give. This is a topic heavily explored in a paper called *Can LLMs Express Their Uncertainty? An Empirical Evaluation of Confidence Elicitation in LLMs* by Xiong et al. They explore confidence elicitation and how it can ensure an increase in an LLM's reliability and trustworthiness. This need arises as some of the best and most popular LLMs are closed-source and *non-logit-based* approaches are needed. In the paper, they introduce three categories of methods: verbalize-based, consistency-based, and hybrid methods that utilize both. They found out that LLMs often show overconfidence when verbalizing their confidence, but that it can be calibrated using CoT, Top-K, and Muti-step confidence. Consistency-based methods outperform the verbalized ones, while the hybrid approach delivers the best performance [34].

2.2.5 Embedding Models

Muennighoff et al. have written a paper called *MTEB: Massive Text Embedding Benchmark* covering the Massive Text Embedding Benchmark (MTEB). They benchmark 33 different embedding models mostly found on Hugging Face[3], but also other models like OpenAI's *text-embedding-ada-002* model. MTEB covers eight embedding tasks like classification and semantic textual similarity (STS) spanning 58 datasets and 112 languages. Their results are presented in a big leaderboard. A thing that can be derived from their results, is that a model's vector dimension size has little-to-no correlation with its performance. There are some benefits of having fewer dimensions, however, as it requires a smaller amount of computation. This makes for faster queries and less Random Access Memory (RAM) usage [21].

RAGAS: Automated Evaluation of Retrieval Augmented Generation, a paper written by Es et al., introduces a framework for the evaluation of RAG pipelines. The implementation of RAG requires a great deal of tuning, as its performance is dependent on a lot of different components, like the retrieval model, the context itself, the LLM, prompt template, among others. A method to evaluate the RAG pipeline when working with this tuning is therefore paramount. RAG systems are often evaluated by their ability to tackle the LLM task itself by measuring perplexity on some context. They consider a standard RAG setting, where a question is used to fetch some external context before being used to generate an answer. The evaluation focuses on three different quality aspects. *Faithfulness* is a metric of how well the answer is grounded in the provided context. Secondly, *Answer Relevance* refers to the fact that the generated answer should address the actual question that was given. Finally, *Context Relevance* focuses on how good the RAG pipeline is at retrieving only relevant context, containing as little irrelevant information as possible [9].

[3] https://huggingface.co/

Chapter 3

Methods

This chapter contains a walk-through of approaches used to create and test the enhanced RAG pipeline that can be used in custom AI chat services. Three smaller pipelines are introduced that can all be put together into a bigger one, or used separately. The chapter is divided further into several sections each describing their own methodology. Section 3.1, labeled *The Retrieval Augmented Generation Pipeline*, illustrates and explains the three different pipelines separately, but also how they work together. In Section 3.3, labeled *Classification of Prompts*, the prompt classification process is clarified, and why it is a solution to the problem that not all prompts should cost the same. The following section, labeled *Exploration of Embedding Models* (Section 3.4), compares and evaluates OpenAI and open-source embedding models using various metrics. After that, Section 3.5 labeled *Explainability*, goes through various ways of achieving explainability in custom AI chat services using prompt engineering, CoT, and sources. Finally, the section labeled *Reducing Cost by Utilizing Prompt Engineering* (3.6), goes through the different prompt compression techniques used and how they reduce cost.

3.1 The Retrieval Augmented Generation Pipeline

This thesis proposes a pipeline that starts with the user sending a prompt and ends with an LLM creating an answer. In the following figures, a rectangle represents a process, a parallelogram represents data, and the cylinder shapes represent databases. Starting off, the prompt is sent for prompt classification to decide which LLM model to use, as seen below in Figure 3.1.

Figure 3.1: How the prompt is classified to choose LLM model.

While that is underway, the prompt undergoes embedding before being used to query a vector database for context retrieval. This process can be seen below in Figure 3.2.

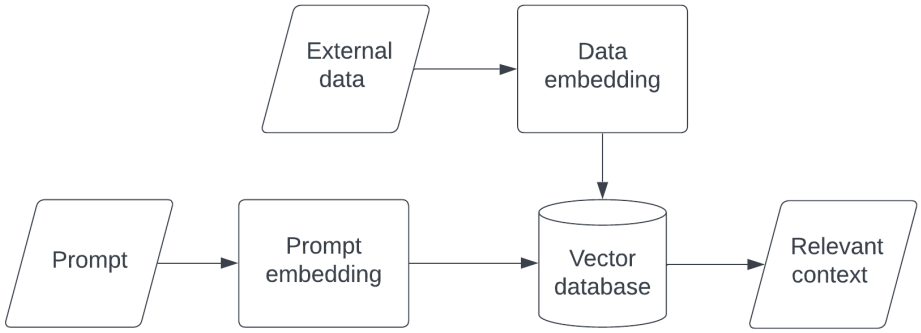

Figure 3.2: How the prompt is used to retrieve relevant context.

The context retrieved is compressed using contextual compression. Simultaneously, explainability instructions are added to the prompt template, and the prompt template itself is compressed and joined with the compressed context. When these two are joined they form the complete prompt that is to be sent to the LLM. Together, however, they can be compressed even further by doing prompt compression. This compressed version of the complete prompt is then finally sent to the LLM model to be answered. The compression pipeline can be seen below in Figure 3.3, and the pipeline as a whole can be seen in Appendix A. The following sections will go through each of these processes as they appear in the pipeline as a whole.

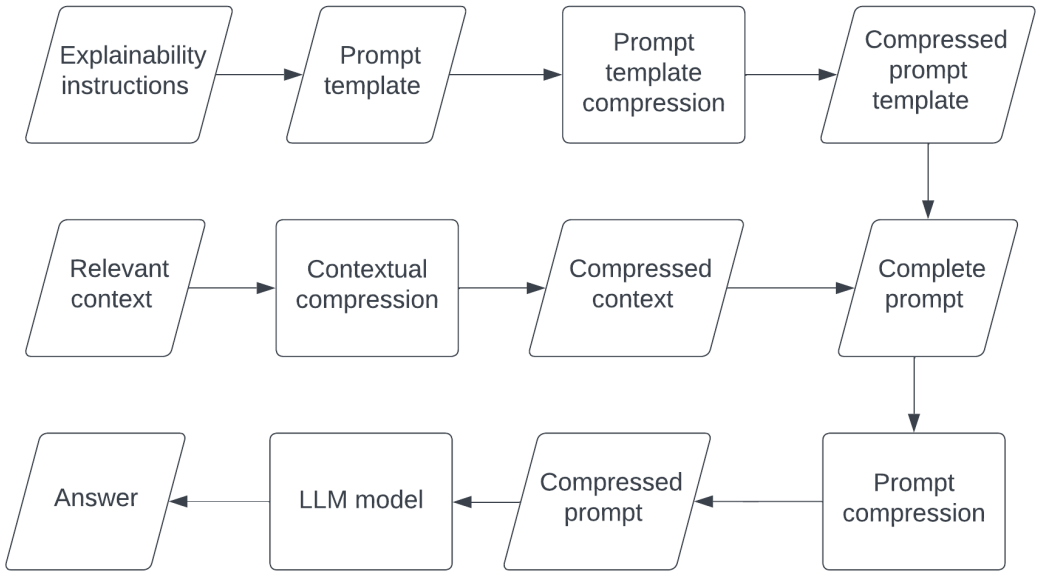

Figure 3.3: How explainability and prompt compression are achieved in the RAG pipeline.

3.2 Evaluating Pipelines

To be able to evaluate the different pipelines and methodologies introduced throughout this chapter, a way to calculate cost and time efficiency is needed. OpenAI calculates the cost of using their models by using something called tokens, which can either be one word or pieces of words. How many characters each token consists of can differ from language to language. In English, one token consists of 4 characters on average, and they can contain trailing spaces and sub-words [27]. An illustration of how a paragraph is tokenized can be seen below in Figure 3.4. MSPF can give you the total number of tokens used in a flow, and also by each LLM node. By looking at the OpenAI model pricing[1], it is possible to calculate the cost of each flow depending on which models are used. MSPF also times each flow, which makes it easier to not only compare the cost of them but also their completion times.

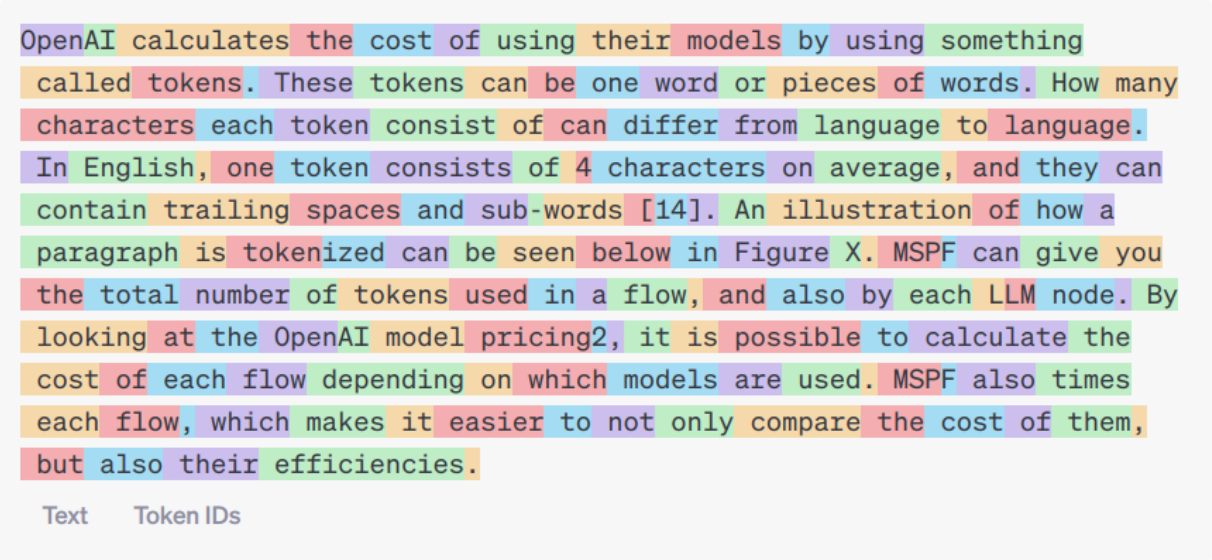

Figure 3.4: How OpenAI tokenizes the paragraph above using their open-source tokenizer called tiktoken.

3.3 Classification of Prompts

Prompt classification is here done to see if it can reduce cost and completion time of a RAG pipeline. There are various ways of classifying words, text, or sentences. A new and popular method is that of using LLMs as classification models. They have been shown to outperform conventional machine learning approaches, especially when it comes to zero-shot and few-shot learning [4]. This project proposes a way of reducing the cost of custom AI chat services

[1] https://openai.com/pricing

by using MSPF and prompt classification. To enable an LLM to do prompt classification, a certain type of prompt engineering has to be utilized. An example of a prompt template that enables this can be seen below in Code Listing 3.1. To do this in MSPF an LLM and Python node have to be created. The Python node is used to provide the LLM node with classification examples that relate to the task at hand, something known as few-shot learning. The prompt template below first specifies the LLM's task, then gives the LLM the different class labels, and finally gives it the examples.

```
system:
Your task is to classify a given prompt into one of the following categories:
gpt-35 or gpt-4 based on the prompt text content.
The prompts classified as gpt-35 will be easier prompts and not very complex,
while the prompts classified as gpt-4 will be more complex.

user:
The selection range of the value of "category" must be within "gpt-35" or ...
    "gpt-4".
Here are a few examples:
{% for ex in examples %}
Prompt content: {{ex.prompt}}
OUTPUT:
{{ex.category}}

{% endfor %}

Prompt content: {{prompt_content}}.
OUTPUT:
```

Listing 3.1: Example of a prompt template that can classify text using few-shot learning.

The example above can be used to reduce the cost of custom AI chat services, as differently priced models could be used to answer prompts that differ in difficulty. Some of the OpenAI models, for example, differ quite a lot in pricing [2], and correctly classifying prompts could potentially save both money and time. Prompt difficulty classification is not the only thing this approach can solve. Often when it comes to custom AI chat services, there are multiple knowledge bases that the end-user wants to prompt. By classifying the user's prompt, it is possible to detect what knowledge base the user wants to prompt. This will remove the need of having to manually select a knowledge base and could save the user both effort and time.

3.3.1 Finding the Best Classification Model

It is important to find the correct classification model as a bad model can potentially add unnecessary cost and completion time. An important thing to note is that the use of an LLM for classification increases the total token count of the flow. It is therefore important to find out which LLM model is

[2] https://openai.com/pricing

sufficient for classification. If their most cost-effective model, *gpt-3.5-turbo-1106* [3] is sufficient, it is most likely worth it because of the very reasonable pricing. If not, utilizing prompt classification might not be more cost-effective. Another factor to take into consideration is the percentage of prompts that are simple and difficult. Two flows that make sense to compare here, are one that uses the most expensive model (*gpt-4*) and one that sends simple prompts to *gpt-3.5-turbo-1106* and difficult prompts to *gpt-4*. The two flows mentioned are illustrated below in Figure 3.5. Assume that 95 percent of prompts that users send are classified as difficult. Then the second flow will save money on only 5 percent of the prompts, while the 95 percent other will have increased cost because of the additional classification cost. In a case like this, it might be better to just use the first flow without classification.

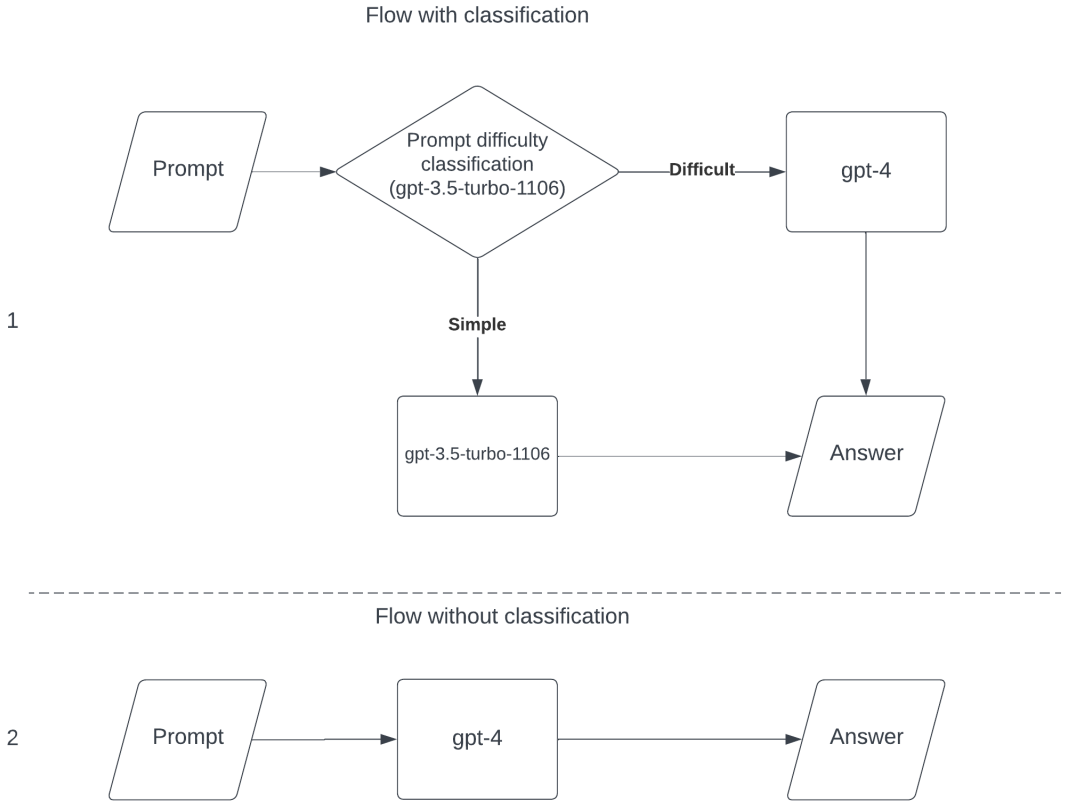

Figure 3.5: The two flows that are being compared in the paragraph above.

3.4 Exploration of Embedding Models

In a RAG pipeline embeddings play a significant and crucial role. The choice of embedding model can affect various factors like cost, accuracy and time efficiency. It was mentioned in Section 2.1.7 that Hugging Face offers an abundance

[3] See Footnote 2

of open-source models. This collection of models also includes embedding models, that can all be utilized using the *sentence_ transformers* library in Python. The open-source embedding models that were chosen were six of the most popular and best-performing sentence similarity models on Hugging Face[4]. The ones chosen, together with the latest OpenAI embedding models, can be seen below in Table 3.1.

Model	Vector Dimension	Parameters	Usage Price
text-embedding-3-large (OpenAI)	3072	unknown	\$0.00013/1K tokens
text-embedding-3-small (OpenAI)	1536	unknown	\$0.00002/1K tokens
text-embedding-ada-002 (OpenAI)	1536	350 million	\$0.00010/1K tokens
e5-large-v2	1024	335 million	free
gte-large	1024	335 million	free
ember-v1	1024	335 million	free
all-mpnet-base-v2	768	133 million	free
all-MiniLM-L6-v2	384	22.7 million	free
e5-small-v2	384	33.4 million	free

Table 3.1: Open-source Hugging Face and OpenAI models used [8]. The prices are as of Feb. 12th 2024.

In the case of custom AI chat services, embedding models like these are used to create custom knowledge bases, often stored in a vector database. To fetch data from a vector database something called a similarity measure has to be performed. When comparing embedding models for custom AI chat services, it then makes sense to compare them by performing similarity measures on their embeddings.

A testing dataset was created in the form of a document as context, as well as relevant example questions. It was then possible to use these example questions to see which embedding model best retrieves relevant context. Another evaluation methodology that was used is the one mentioned in Section 2.2.5 from a paper called *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. This evaluation method requires a dataset similar to the one already mentioned, in addition to answers given by the RAG pipeline and their respective ground truths. The RAGAS metrics that were used include *context precision*, *context recall*, and *context relevancy*. Context precision evaluates whether or not the most relevant text chunks are ranked higher or not when fetching the top four most relevant text chunks. The more relevant text chunks should be ranked at the top and given first to the LLM. Context recall is a measure of how well the retrieved context aligns with the ground truth answer. Finally, context relevancy is a measure of how relevant the retrieved context is

[4] https://huggingface.co/models?pipeline_tag=sentence-similarity&sort=downloads

in accordance with the question. If this number is low, a lot of irrelevant context has been retrieved, whilst a higher number means more relevant context. A more in-depth description of experiments and results can be found below in Section 4.3.

3.5 Explainability

To increase the accuracy of LLMs and RAG pipelines, explainability is essential. It also serves as a powerful tool to combat LLM hallucinations, making it easier for the users to verify and fact check the answers given. The following sections look into how to implement explainability using prompt engineering, CoT and sources respectively.

3.5.1 Explainability Using Prompt Engineering

Getting an LLM to reflect, provide verifiable facts and give its confidence are all ways to achieve explainability. As mentioned in Section 2.1.5, one way to achieve explainability for LLMs is to utilize natural language explanation. One way to do this is through the use of prompt engineering. What comes to mind then is using some of the relevant prompt patterns introduced in Section 2.2.3 [33]. The pattern that is most relevant here is the *Reflection Pattern*, as its job is to provide reasoning with every answer. Adding this to one of the prompt templates used in custom AI chat services would look something like this:

> You are a custom AI chat service, an advanced chatbot developed by <company> for <company> employees. Your job is to... When you provide an answer, please explain the reasoning and assumptions. If possible, use specific examples or evidence associated with your answer. Moreover, please address any potential ambiguities or limitations in your answer, in order to provide a more complete and accurate response [33].

The *Persona Pattern* is here first used to establish the LLM's role and its main job. This is followed by the *Reflection Pattern* that tells the LLM to explain its reasoning and possibly show evidence of its answer. It is also important for the LLM to specify the limitations of its answer if that is necessary. Another pattern that can be useful to achieve explainability is the *Fact Check List Pattern*, which lists out all the facts used in the LLM's answer. This pattern can be used by adding something like the following to the prompt template:

> ...when you generate an answer, create a set of facts that the answer depends on that should be fact-checked and list this set of facts at the end of your output [33].

In addition to this, explainability can be achieved by having the LLM assess its confidence with the answer it has given. This approach was explained in Section 2.2.4, and can be achieved by using the following prompt template:

> Please answer this question and provide your confidence level. Note that the confidence level indicates the degree of certainty you have about your answer and is represented as a percentage. Answer and Confidence (0-100): [34].

3.5.2 Explainability Using Chain of Thoughts

Enabling an LLM to share its thought process can greatly increase its explainability. This is the primary function of CoT prompting. CoT can be achieved using the Python and JavaScript framework Langchain mentioned in Section 2.1.6. Through its Software Development Kit (SDK), Langchain offers agents whose main purpose is to give LLMs more functionality. This functionality can be things like enabling it to perform web searches or giving it access to external information sources like vector databases [16]. An LLM is used as a voice of reason to determine which actions to take in what situation and in which order. Langchain currently supports a Plan-and-execute agent based on the *Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models* [32] paper mentioned in Section 2.2.4. This agent is perfect to use for achieving CoT because it first creates a plan to answer the query with clear steps, before executing the plan using an action agent. To achieve this it first uses an LLM as a planner with the following prompt template:

> Let's first understand the problem and devise a plan to solve the problem. Please output the plan starting with the header 'Plan:' and then followed by a numbered list of steps. Please make the plan the minimum number of steps required to accurately complete the task. If the task is a question, the final step should almost always be 'Given the above steps taken, please respond to the users original question'. At the end of your plan, say '<END_OF_PLAN>' [32].

Then an LLM is used as executor to start working on the plan created by the planner. If the executor had tools called *Calculator* and *Search* its prompt template would look as follows:

> Respond to the human as helpfully and accurately as possible. You have access to the following tools:
>
> Search: Useful for answering questions about current events, args: {{'tool_input': {{'type': 'string'}}}}

> Calculator: Useful for when you need to do math, args: {{'tool_input': {{'type': 'string'}}}}
>
> Use a json blob to specify a tool by providing an action key (tool name) and an action_input key (tool input).
>
> Valid "action" values: "Final Answer" or Search, Calculator
>
> Provide only ONE action per $JSON_BLOB, as shown:
>
> {{ "action": $TOOL_NAME, "action_input": $INPUT }}
>
> Follow this format:
> Question: input question to answer Thought: consider previous and subsequent steps Action:
>
> $JSON_BLOB
>
> Observation: action result ... (repeat Thought/Action/Observation N times) Thought: I know what to respond Action:
>
> {{ "action": "Final Answer", "action_input": "Final response to human" }}
>
> Begin! Reminder to ALWAYS respond with a valid json blob of a single action. Use tools if necessary. Respond directly if appropriate. Format is Action: $JSON_BLOB then Observation:.
> Thought: [32].

3.5.3 Explainability Using Sources

A powerful method to achieve explainability for LLMs is by making the information it provides verifiable. A simple yet effective way of doing this is with the use of sources. Natural language in the form of prompts can actually be used to incentivize the LLM to provide sources with its answers. This can be done by using a prompt as such:

> You are a custom AI chat service, an advanced chatbot developed by <company> for <company> employees. Your job is to... Please provide a source in markdown format if you feel that is needed with "source:" prepended.

This will work for questions where the LLM uses its own knowledge base as a source of information. For instances where it uses an external knowledge base, metadata will have to be added to the knowledge base to enable the LLM to cite its sources. This is yet another problem that can be solved using Langchain. Langchain can be used to split up documents, embed them, and then upload them to a vector database. In this process, Langchain enables the functionality of adding metadata to documents that are uploaded, which can be seen below in Algorithm 1.

Algorithm 1 Process of Adding Sources to PDFs

```
Initialize langchain pdf-loader
Load pages from pdf-loader
Initialize pinecone index with index name and namespace
for each page in pages do
    Add file name and page number as metadata
    Initialize text-splitter with chunk-size 1000 and chunk-overlap 200
    Split page into texts using text-splitter
    Initialize embeddings
    for each text in texts do
        Push embedded text to pinecone with metadata and embeddings
    end for
end for
```

3.6 Reducing Cost by Utilizing Prompt Engineering

When it comes to the cost of using LLMs, it all comes down to the model used and the number of input and output tokens. Therefore it is important to reduce these as much as possible without affecting the quality of answers negatively. Options for reducing output tokens are somewhat limited, as reducing this too much would result in a decrease in answer quality. Prompt engineering can, however, be used to affect both of these numbers. It can of course be used to alter the number of input tokens by changing the prompt template, but it can also affect the number of output tokens.

3.6.1 Prompt Template Compression

A prompt template is a static, textual template that is typically included with each user prompt. It is therefore important that the template is as short as possible without losing its meaning. To reduce the size of the prompt template, this project proposes a quite simple solution. The proposed method is to first write out the prompt with one's own natural language, before giving it to an LLM model like *gpt-4*, asking it to reduce the number of tokens in it. Doing this achieves two things: it reduces the token count while still making sure the

model understands the prompt, as it was made by the model itself.

The prompt template can also be used to reduce the number of output tokens used by a model. This prompt template would be inspired by the *Template* and *Persona* patterns mentioned in Section 2.2.3. A fitting template could then look something like the following:

> You are EgdeAI, an advanced chatbot developed by Egde for Egde employees. Your job is to... Please keep your answer short and concise with clear and simple language.

These two approaches can both be used to reduce the static prompt template and the LLM's answer, but will not reduce the input of the user or the context provided.

3.6.2 Contextual Compression

One of the major challenges when it comes to context retrieval is that a lot of unnecessary data may be retrieved. Not only might this confuse the LLM, but it will also unnecessarily increase the token count. Contextual compression is a tool that can be used to solve this issue [17]. When a PDF document is used as context to an LLM, it is usually split up into text chunks, and the text chunks with the highest textual similarity to the user prompt are given as context. There are two main problems that can arise here. First of all, there might be very similar text chunks with basically the same information that gets sent as context. Secondly, these text chunks often contain information not relevant to the prompt. The first one can be solved by doing similarity measures between the text chunks themselves. If two or more of the text chunks have a similarity score higher than a certain threshold, all but one will be removed, being the one with the highest similarity score with the prompt. The latter issue can be solved by further splitting up the remaining text chunks and checking what parts share the most textual similarity with the prompt. Doing this will filter out the irrelevant parts of the text chunks, making the context only consist of the most relevant information. To implement contextual compression, Langchain[5] can be used, as seen below in Algorithm 2.

[5] https://python.langchain.com/docs/modules/data_connection/retrievers/contextual_compression

Algorithm 2 Document Compression and Retrieval with Pinecone and Langchain

1: Initialize pinecone with api key
2: Fetch pinecone index with the index name
3: Fetch the correct pinecone namespace within the index
4: Initialize embedding model
5: Initialize text-splitter with chunk-size and chunk-overlap
6: Initialize redundant filter with embedding model and similarity threshold
7: Initialize relevant filter with embedding model and similarity threshold
8: Create document-compressor-pipeline with text-splitter, redundant-filter and relevant-filter
9: Create contextual-compression-retriever
10: Retrieve relevant documents by using the contextual-compression-retriever and the prompt

On lines 1-3 above in Algorithm 2, a Pinecone retriever is created, which is used as a context provider. Then, from lines 5-7, the text-splitter and filters are initialized. The *text-splitter* is used to further split the chunks into smaller ones, while the filters remove unnecessary context. The *redundant_filter* is there to remove very textually similar text chunks, meaning chunks that have a higher similarity score than the threshold value. The *relevant_filter* goes through the now smaller text chunks and removes any that share a similarity score with the prompt that is smaller than the given threshold value. These tools are then put in a *document-compressor-pipeline* that is again used in a *contextual-compression-retrieve*.

3.6.3 Prompt Compression

The total prompt, including prompt template, context and the user prompt itself, is often quite large. Compressing this even further can therefore reduce cost quite a lot. In sections 3.6.1 and 3.6.2 methods to compress the prompt template and the context were explained. In this section, a methodology for compressing all the text being sent to an LLM will be gone through. This will be the final compression step after the two other methods before the text is sent to an LLM. To achieve this, LLMLingua will be used. It was also mentioned in Section 2.2.3, and can be used for not only reducing the prompt template size but also the user's input and provided context. All this while still preserving the essential information of the whole input. This is especially powerful in the case of long user prompts and contexts, where the algorithm has been shown to reduce the whole prompt down to as little as 1/20 of the original prompt. This is with a minuscule amount of performance loss [14].

Chapter 4

Results and Discussion

This chapter contains the experiments that test all the hypotheses stated in Section 1.3.1. The experiments were conducted using one of the two systems specified in Section 4.1 below. If relevant, the system used is specified beneath the corresponding experiment.

4.1 Hardware

- **System 1:** Machine with Ubuntu 22.04.3 LTS with a *NVIDIA A100 80GB* GPU. The machine has an *Intel(R) Xeon(R) CPU E5-2660 v4 2.00GHz* CPU with 2 sockets, 14 cores per socket, and 2 threads per core.

- **System 2:** Machine with Microsoft Windows 11 Home. The machine has an *AMD Ryzen 7 3700U with Radeon Vega Mobile Gfx, 2300 Mhz, 4 Core(s), 8 Logical Processor(s).*

4.2 Prompt Classification

4.2.1 Finding the Best Classification Model

The following experiment is done to test **H2**, as described in Section 1.3.1. To find out which OpenAI model suffices for prompt classification a prompt flow in MSPF was created. The flow was a simple classification flow with one Python node for preparing a few examples and one LLM node for doing the actual classification. Three of OpenAI's LLM models, *gpt-35-turbo-1106*, *gpt-4-1106* and *gpt-4* were used for testing. Using Azure AI Machine Learning Studio[1], it was possible to perform batch runs with multiple inputs so as to obtain accuracies for the different models. To test each model, 40 test prompts were created with their respective ground truths. The test prompts can be seen below in Table C.1 in Appendix C. Batch runs for each of the models were then

[1] https://ml.azure.com/home

run to find their accuracies. The results from the comparison can be seen below in Table 4.1.

Model	Accuracy	Price (input/output)
gpt-35-turbo-1106	0.95	**$0.0010/$0.0020**
gpt-4-1106	0.97	$0.01/$0.03
gpt-4	**1**	$0.03/$0.06

Table 4.1: Accuracies and prices of the different OpenAI models tested. These are the prices as of 30.01.2024.

A thing to keep in mind when looking at these results is that the two classes *simple prompt* and *difficult prompt* have no clear boundaries. How difficult a prompt is can be subjective, so the ground truth is not necessarily the absolute truth. Some prompts are located somewhere in the middle between the two classes. There is, however, room for a few "misclassifications". As long as the vast majority of prompts are classified "correctly", it has the potential to increase cost efficiency. Considering all of this, the model that is natural to use for prompt classification is the *gpt-35-turbo-1106* model. It is by far the cheapest model (as of 30.01.2024) and it has an acceptable accuracy of 0.95. It does have the lowest accuracy of all the models, but as previously discussed, some "misclassifications" are tolerable.

4.2.2 Reduction in Cost by Using Prompt Classification

The experiment done in this section is conducted to test **H2**, as stated in Section 1.3.1. Because an extra LLM node is needed, prompt classification in MSPF leads to an increase in the total token count. A prompt template with a token count of 254 (with few-shot examples), much like the one in Code Listing 3.1 was used for comparison. From the previous section, it was deduced that the best model to use for classification was the *gpt-35-turbo-1106* model. One prompt, which was classified as "simple" (*How many people live in Grimstad*), was used to compare the different flows. Since this is a "simple" prompt, it allows for comparing a flow with two *gpt-35-turbo-1106* LLMs and one with a single *gpt-4* LLM, as can be seen in Figure 3.5.

Looking at the first of these flows, flow one in Figure 3.5, the total number of tokens can be obtained. The total number of input tokens for this flow was 679 (254 + 425), and 25 (5 + 20) output tokens. Using the OpenAI model pricing[2] for the *gpt-3.5-turbo-1106* model, the total flow cost can be calculated as following in Equation 4.1.

[2] https://openai.com/pricing

$$\underbrace{\$0.0010 * \frac{679}{1000} + \$0.0020 * \frac{25}{1000} = \$0.00073}_{\text{cost per 1000 input tokens * input tokens/1000 + cost per 1000 output tokens * output tokens/1000 = \$0.00073}} \tag{4.1}$$

The same prompt can then be used to test the second flow without prompt difficulty classification. This flow only has one LLM node, which is the *gpt-4* model. The answering models in both these flows share the exact same prompt template, as well as a temperature of 0.4. Finding the total token count as well as the correct pricing[3] then allows for calculating the cost as seen below in Equation 4.2.

$$\underbrace{\$0.030 * \frac{425}{1000} + \$0.060 * \frac{20}{1000} = \$0.01395}_{\text{cost per 1000 input tokens * input tokens/1000 + cost per 1000 output tokens * output tokens/1000 = \$0.01395}} \tag{4.2}$$

The pricing of the *gpt-4* model is so much steeper than the pricing of *gpt-3.5-turbo-1106* that even with the extra 254 classification tokens, it is a lot cheaper. In fact, the flow without classification is about 19 times more expensive than the one with classification. Another comparison that has to be made is between the flows if a prompt was classified as "complex". In this case, the cost will actually increase a bit, as there will be additional tokens used for an "unnecessary" prompt classification in flow one. A big question is if this increased cost will be smaller than the decrease in the previous comparison. Assume that the prompt used above is classified as "complex", to keep the token counts identical when calculating the total cost of flow one in Equation 4.3.

$$\underbrace{(\$0.030 * \frac{425}{1000} + \$0.060 * \frac{20}{1000}) + (\$0.0010 * \frac{254}{1000} + \$0.0020 * \frac{5}{1000}) = \$0.014214}_{(\textit{gpt-4} \text{ input and output cost}) + (\textit{gpt-3.5-turbo-1106} \text{ input and output cost}) = \$0.014214} \tag{4.3}$$

From this calculation, it is apparent that the additional cost added by the LLM classification node is minuscule. The additional cost is $\$0.014214 - \$0.01395 = \$0.000264$. Comparatively, the decrease in cost when feeding a "simple" prompt to the flow was $\$0.01395 - \$0.00073 = \$0.01322$, a 94.8% price reduction. As shown, the decrease in the cost of using classification is a lot bigger than the increase in using it. The relation between these is $\$0.01322/\$0.000264 \approx 50$. This means that if the user base of the custom AI chat service sends at least one or more "simple" prompts for every 50 "complex" prompts, the flow with classification is the most cost-effective. A summary of the price calculations

[3] https://openai.com/pricing

between the two flows can be seen below in Table 4.2. These results can be compared to those of Chen et al. in [5]. Their proposed solution, *FrugalGPT* is able to reduce cost in the range of 50% to 98%, depending on dataset, by using *LLM Cascade*. This similar approach uses various LLMs to respond to prompts of differing complexity. A difference here is that they use a dataset with ground-truth values to decide which LLMs to use. This is not possible in the case of a custom AI chat service, as it is impossible to obtain a ground truth dataset for all possible user prompts.

Flow	**Prompt Complexity**	**Price**
1	Simple	**$0.00073**
1	Complex	$0.014214
2	Simple	$0.01395
2	Complex	$0.01395

Table 4.2: Resulting prices obtained from the two different flows when using "simple" and "complex" prompts as input.

Completion times are crucial in a custom AI chat service used in professional settings. If the chat service is too slow in retrieving information, users might as well find the answers on their own. Therefore, another factor to take into consideration is the completion times of the two flows. Will adding a classification LLM node increase the time it takes to finish?

Prompt	Flow	
	1	2
What is the population of Grimstad?	5.62s	4.22s
What is 2+2?	5.78s	3.17s
What is the tallest mountain in Norway?	8.96s	5.48s
What is the biggest country in the world?	7.97s	4.49s

Table 4.3: Completion times of the two different flows. These completion times were measured on System 2 from the Hardware section (4.1).

From Table 4.3 it can be seen that the classification LLM does in fact add a bit of completion time. For some prompts, it adds about a second, but for others it almost doubles the completion time, averaging at an increase of 2.74s. Considering this increase in completion time, it would be worth exploring the relation between "simple" and "complex" prompts, so as to be sure there is a possibility to reduce cost.

4.3 Comparison of Embedding Models

H4 states that: **"Open-source embedding models are as good as OpenAI's alternative, and will therefore be possible to use to reduce cost**

and completion time". This hypothesis is tested in the following section by doing a comparison of accuracy, time efficiency, context precision, context recall and context relevancy on some of the best and most popular open-source embedding models on Hugging Face[4], as well as the three best OpenAI embedding models[5]. To achieve this, the famous paper *Attention is All you Need* [30], was used as a knowledge base. Twenty test prompts, which can all be seen in Appendix D, were made to perform similarity measures against it. Some examples of the prompts are listed below:

1. How do you add positional encodings to the input embeddings?
2. What were their results when it comes to machine translation?
3. What do they say in their conclusion?

To be able to use the paper as a knowledge base, it is first split up into text chunks of 1000 characters each with a chunk overlap of 200 characters each. This overlap is there to prevent sentences from being cut in half and context from being lost. Before doing the embedding, the ground truths are found. These are the text chunks that should be returned when doing a similarity measure with each of the twenty prompts. All of the text chunks and all of the prompts are then embedded using the current embedding model. A prompt's top four most similar text chunks are then found using a cosine similarity search. These top four text chunks are then compared to the ground truths, and if one of the text chunks matches, a score of one will be given. This procedure was repeated for each of the twenty prompts found in Appendix D.

Another methodology used for comparing embedding models was the one mentioned in Section 2.2.5 from a paper called *RAGAS: Automated Evaluation of Retrieval Augmented Generation* [9]. To utilize the RAGAS framework, a dataset similar to the one used above has to be created. Firstly, example questions have to be created, followed by the ground truth answers to these questions. Then the context chunks retrieved by each embedding model for each of the example questions have to be collected. Finally, the answer to each of the questions when using each of the embedding models has to be found. As example questions, the twenty test prompts were used. The ground truth answers were generated by using OpenAI's *gpt-4* model and giving it the correct context chunks manually. The context retrieved from each embedding model was found by using cosine similarity as in the methodology above. Then finally

[4] https://huggingface.co/models?pipeline_tag=sentence-similarity&sort=downloads
[5] https://platform.openai.com/docs/guides/embeddings

the answers derived when using each of the embedding models were found using the same *gpt-4* model, with the same temperature (0.5), given the retrieved context using the respective embedding model.

Accuracy is not the only relevant measure when it comes to embedding models. Efficiency is also key when it comes to custom AI chat services and can vary from model to model depending on its vector dimension and potential number of parameters. The following experiment was done on System 1 specified in the Hardware section (4.1), and was conducted by embedding the whole paper that was used above and putting the embedded text chunks into a vector database. After that, all the twenty test prompts from Appendix D were used to query the database. This procedure was timed and repeated for all the embedding models resulting in the numbers under *Time* in Table 4.4 below. The table also includes experimental results from all the other embedding comparisons.

	Metrics						
Embedding Model	*Correct*	*Accuracy*	*MTEB score (STS) [21]*	*Time*	*Context Precision*	*Context Recall*	*Context Relevancy*
V3 large (OpenAI)	**19/20**	**0.95**	81.73	9.79s	**0.9750**	0.9750	0.0580
e5-small-v2	18/20	0.90	81.05	4.26s	0.8861	0.9300	0.0410
V3 small (OpenAI)	18/20	0.90	81.58	7.25s	0.9694	0.9875	**0.0697**
e5-large-v2	17/20	0.85	82.50	6.30s	0.9667	**0.9900**	0.0531
Ada v2 (OpenAI)	17/20	0.85	80.97	6.64s	0.9333	0.9375	0.0515
gte-large	16/20	0.80	**83.35**	6.37s	0.9097	0.9250	0.0376
all-mpnet-base-v2	15/20	0.75	80.28	4.84s	0.9403	0.8700	0.0340
ember-v1	15/20	0.75	83.34	6.40s	0.9153	0.9400	0.0403
all-MiniLM-L6-v2	14/20	0.70	78.90	**3.89s**	0.8750	0.9325	0.0292

Table 4.4: Results of embedding models on the twenty test prompts [8].

From these results, it is apparent that the OpenAI models fare quite well against the open-source models. The OpenAI models, however, use more time to embed the paper and then query it with all the test prompts. One reason for this is quite clearly vector dimension. The results show that a higher vector dimension leads to a slower embedding process. Another factor that should be

considered is that all the open-source embedding models are run locally, while the OpenAI embeddings are retrieved from their API by sending the text there. One of the fastest models was the *e5-small-v2* and also one of the best in terms of accuracy. *V3 large* from OpenAI obtained the best accuracy, but it was also the slowest.

While the accuracy calculated above gives a good indication of how well the embedding models do both when it comes to accuracy and efficiency for this particular paper, the MTEB STS score and RAGAS metrics give a better indication of how well the models fare in general on all knowledge bases. Looking at the RAGAS metrics, it is apparent that the smaller models in terms of vector dimension and parameters, fare worse than their bigger counterparts. These are the *e5-small-v2*, *all-MiniLM-L6-v2* and *all-mpnet-base-v2* models, which averagely show among the lowest scores when it comes to context precision, recall and relevancy. For the *e5-small-v2* this comes as a bit of a surprise, as it achieves quite good accuracy in the former experiment. It seems this model is quite good at finding the right context, but also finds some irrelevant context and does not rank the context ideally for all the test prompts.

Given these results, what should a company that is looking to create its own custom AI chat service use as an embedding model? The most apparent answer seems to be either the *e5-small-v2* or the *e5-large-v2* model. They are free to use, among the fastest models of their respective sizes and they obtain accuracies just below that of the best model. One thing to keep in mind is that the models are only "free" to use if they are in possession of the right hardware. Consider the *e5-small-v2* model as an example in the following experiment. Given that the model is quite small, hardware as good as the one specified in Section 4.1, is not really needed. However, the system should at least have a CUDA-capable Graphical Processing Unit (GPU) [18].

Assume say for production, a company used an in-house machine like the one specified in Section 4.1 under *Testing embeddings*. If not taking into account the cost of the hardware and maintenance, the only cost remaining is that of electricity. The following experiment was done on System 1 specified in the Hardware section (4.1). Using the *Eco2AI* [3] Python library, it is possible to calculate the power consumption of using these embedding models. This was used to compare the price of embedding one prompt (*What is an attention function?*) and the text chunks from the *Attention is All You Need*

paper 1000 times. 1.144kWh of electricity was used to complete this procedure. According to Statistisk sentralbyrå[6], the average price of electricity in Norway in the third quarter of 2023 was 0.1139 NOK/kWh. Considering this, the whole procedure costs roughly $1.3NOK = \$0.12$ (As of Feb 12th, 2024). Using the cheapest OpenAI model (*V3 small*) to do the same procedure costs $(949,600,000 tokens/1000) * \$0.00002 = \$18.99$. These findings show that utilizing an open-source embedding model is more cost-effective, particularly if the necessary hardware is already readily available.

4.4 Explainability

In this section about explainability, **H5** (as described in Section 1.3.1) is tested. To test the three approaches, reflection, fact checklist, and confidence score, introduced in Section 3.5.1, they were all put into one big prompt template:

> When you provide an answer, please explain the reasoning and assumptions. If possible, use specific examples or evidence associated with your answer. Moreover, please address any potential ambiguities or limitations of your answer. When you generate an answer, create a set of facts that the answer depends on that should be fact-checked, and list this set of facts at the end of your output. Please answer this question and provide your confidence level. Note that the confidence level indicates the degree of certainty you have about your answer and is represented as a percentage. Answer and Confidence (0-100): Please provide a source with "source:url", IMPORTANT: On a new line and in the same language as the question. If there are multiple sources, I want you to write each source in the same format on a new line every time.

To test how well this prompt template works, three test prompts were made and tested with the *gpt-4* model from OpenAI with a temperature of 0.5. The test prompts used were as follows:

- How tall is the Burj Khalifa?
- Who won World War 2?
- How many people live in the municipality of Grimstad?

Using this prompt template when answering the test prompt, resulted in the answers found in Appendix B. From the answers, it can be seen that the LLM

[6] https://www.ssb.no/energi-og-industri/energi/statistikk/elektrisitetspriser

gives a much more thorough answer when using the explainability prompt template compared to not using it. It also provides sources in the form of a Uniform Resource Locator (URL) or multiple URLs. Some of the sources point to URLs that do not exist anymore, as they likely existed when the model was trained, but not as of 08.03.2024. Limitations, ambiguities, and assumptions are also listed if necessary. For the first prompt, for example, it mentions one limitation with the answer being that there are different ways of measuring the height of a building. There are different rules when it comes to things like antennas, spires, and flag poles. For the first two prompts, the LLM answered with a confidence of 100%, as those prompts have a definite answer, making the model quite sure of its ability to answer. The last prompt gets a confidence level of 85% as the model does not have access to real-time data, and therefore can not give an accurate answer regarding population numbers.

To test if CoT prompting adds a layer of explainability the prompt *How many people live in the municipality of Grimstad?* was used. Langchain's *DuckDuckGo Search*[7] was given to the agent as a tool. This allows the LLM to use the DuckDuckGo search engine to retrieve necessary context. The results when running this prompt can also be found below in Appendix B, Section B.2. The results show that the planner first pinpoints the problem at hand before arriving at a plan for how to solve it, which is to use the search tool to find the relevant information. As can be seen from the executor output, it first finds the population as of 2017. It then decides that this information is not current enough, does one more search, and finds a more current population number. To defend its source it states that it seems reliable because it also provides other specific details about the municipality. The source also says that the population has increased by 12.8%, so the executor uses the number from 2017 to compare and conclude by saying that the increase aligns with the current population specified by the user.

Sources were also made possible when prompting an external knowledge-base, through the addition of metadata to the external data. This makes it so that when a user prompts their own documents, the source document and page number are displayed, as seen in Figure 4.1. As seen in the figure, when a user hovers above the information box, the source/sources appear.

[7] https://python.langchain.com/docs/integrations/tools/ddg

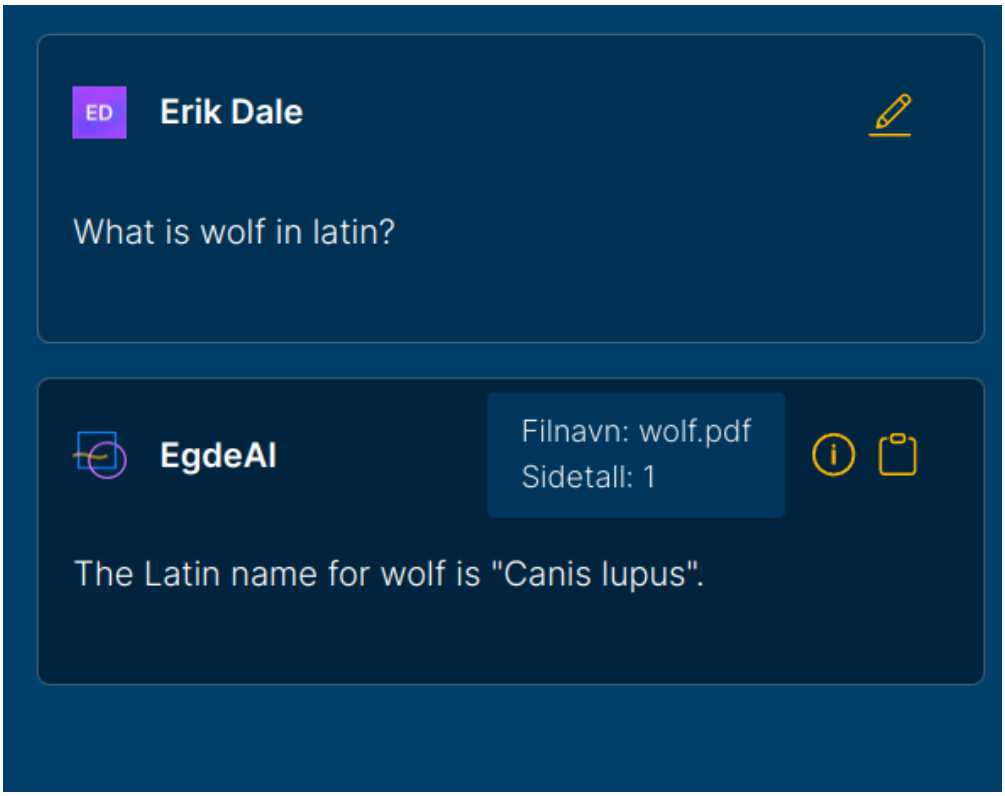

Figure 4.1: How the display of sources looks in a custom AI chat service like EgdeAI.

4.5 Prompt Engineering

4.5.1 Prompt Template Compression

In the following section, **H3**, as described in Section 1.3.1, is tested. This project proposes a quite simple, but effective way of reducing the size of prompt templates. This is done by feeding the prompt template to the LLM model in question and instructing it to *reduce the number of words, while still making sure you understand it.* Doing this procedure on Code Listing 3.1, results in the prompt template seen below in Code Listing 4.1.

```
system:
Classify prompts into 'gpt-35' (simpler) or 'gpt-4' (complex) categories ...
    based on their text content.

user:
Choose 'category' value from 'gpt-35' or 'gpt-4'.
Examples:
{% for ex in examples %}
Content: {{ex.prompt}},
OUTPUT:
{{ex.category}}
{% endfor %}

Content: {{prompt_content}}.
OUTPUT:
```

Listing 4.1: Result of doing prompt template compression on Code Listing 3.1.

This new, compressed prompt template has a token count of 93 compared to 131 of the original one. This leads to a reduction in the cost of about 30% while obtaining the same accuracy as the bigger one. Additionally, it slightly decreases the flow completion time, particularly for longer prompts.

4.5.2 Contextual Compression

One concern when doing prompt compression is that important parts of the context are lost in the process. To test this assumption the five first prompts from the twenty test prompts (Appendix D) were used. A comparison of token count and answer quality was then conducted. The answer was given by OpenAI's *gpt-4* with a temperature of 1, and the *text-embedding-ada-002* was used as the embedding model.

Prompt No.	**Context Token Counts**	**Retrieval Time**
1	805/275	0.48s/3.99s
2	878/365	0.51s/3.17s
3	1069/470	0.33s/2.26s
4	827/176	0.51s/2.15s
5	672/282	0.40s/2.16s

Table 4.5: Result data from the contextual compression comparison. The 'Context Token Counts' and 'Retrieval Time' columns are in the form uncompressed/compressed. The retrieval times were measured on System 2 from the Hardware section (4.1).

From the results in Table 4.5, it is apparent that the token count reduction is quite significant. At most (prompt 4) the token count was reduced by a factor of almost 5. It is therefore apparent that contextual compression is able to reduce the price of giving LLMs context. It does come at a price of a bit longer retrieval time, however, but that is acceptable considering the improved cost-effectiveness. A comparison of the answers to the first two prompts can be found in Appendix E. They show that the answers are pretty much identical with or without contextual compression. This means that even with a quite hefty reduction in prompt tokens, the model is still able to answer the prompts just fine.

4.5.3 Prompt Compression

LLMLingua can be used with MSPF, as seen in Code Listing 4.2, but not on any device, as the models needed to run it are Compute Unified Device Architecture (CUDA) dependable. To test how well LLMLingua works, a similar methodology to the one in the previous section (4.5.2) was used. Two prompts from Appendix D were used to compare token count and answer quality between one flow with and one without LLMLingua. Different compression rates were also

tested, so as to find the best one. Ideally, the compression rate should be as high as possible without losing any important information from the prompt. OpenAI's *gpt-4* with a temperature of 1 was used for the comparison, and the *text-embedding-ada-002* was used as the embedding model.

```
1  from promptflow import tool
2  from llmlingua import PromptCompressor
3  import tiktoken
4
5  @tool
6  def compress_prompt(prompt: str):
7      llm_lingua = PromptCompressor("TheBloke/Llama-2-7b-Chat-GPTQ", ...
       model_config={"revision": "main"})
8      encoding = tiktoken.get_encoding("cl100k_base")
9      compression_rate = 0.8
10
11     prompt_tokens = len(encoding.encode(prompt))
12     compressed_prompt = llm_lingua.compress_prompt(prompt, instruction="", ...
       question="", target_token=int(prompt_tokens*compression_rate))
13
14     return compressed_system_prompt["compressed_prompt"]
```

Listing 4.2: Example of how to use LLMLingua in MSPF.

In Code Listing 4.2, a prompt compressor is initialized on line 7. This uses an open-source LLM based on Meta's *Llama 2 7B Chat*. On line 11 the token count of the prompt is counted using *tiktoken*. The *target_token* is then set to be *prompt_tokens*compression_rate*. A *compression_rate* of 0.8 then equals 20% compression.

One of the big questions to ask when doing prompt compression is what the compression rate should be. The compression rate should be as high as possible without the loss of any important information from the prompt as a whole.

Prompt No.	**Prompt Token Counts**	**Completion Time**	**Compression Rate**	**Cosine Similarity**
1	917/658	23.4s/25.7s	20%	85.3%
3	1182/750	30.1s/25.1s	20%	81.6%
3	1182/634	30.1s/24.5s	30%	70.6%
3	1182/561	30.1s/24.3s	40%	69.2%
3	1182/471	30.1s/22.6s	50%	65.8%

Table 4.6: Result data from the prompt compression comparison. The 'Prompt Token Counts' and 'Completion Time' columns are in the form uncompressed/compressed. The 'Cosine Similarity' number is the textual similarity score between the compressed and uncompressed answers using cosine similarity. The completion times were measured on System 1 from the Hardware section (4.1).

Table 4.6 above shows the difference in prompt token counts and completion times for different flows ran on different prompts with different compression rates. It is shown that an increase in the compression rate leads to a decrease in token counts and completion time. It is also apparent that it leads to a decrease in cosine similarity with the uncompressed prompt answer. Now the uncompressed prompt answer can not be seen as the "ground truth", but it should be the most fleshed-out answer since it is given the longest prompt and context. Using the cosine similarity score only, it can be quite hard to determine what the best compression rate is. The answers given by the *gpt-4* model, which can be seen in Appendix F, are all quite impressive actually. They do become shorter and details are left out the higher the compression rate number gets, but they all manage to answer the question. It can be seen that the answer when doing 50% prompt compression is a little lackluster, and it kind of tries to guess what multi-head attention is. Which compression rate is the best kind of comes down to how long and what level of detail is wanted in the answer.

4.5.4 Overall Compression

It has now been shown how three different compression techniques can reduce token counts and how they affect completion times and LLM answers. An interesting thing to look at now is how they work when put together. How much is it possible to reduce the token count of a prompt and how will it affect completion time and answer? Also, what are the best hyper-parameters to use? It starts off with a prompt template of 181 tokens. As an external knowledge base the *Attention is All You Need* paper is used with the prompt: *What is an attention function?* This prompt together with the prompt template and context adds up to 1005 tokens and a completion time of 21.6 seconds.

Method	**Total Token Reduction**
Prompt Template Compression	$1005 \rightarrow 902$
Contextual Compression	$902 \rightarrow 357$
Prompt Compression	$357 \rightarrow 289$
Overall Compression	$\mathbf{1005 \rightarrow 289}$
Overall Compression Completion Time	**21.6s → 21.7s**

Table 4.7: Compression level achieved by each of the compression methodologies.

Which level of compression each of the compression methodologies achieved can be seen above in Table 4.7. The prompt template is the first to get compressed, by using the method described in Section 3.6.1 (Prompt Template Compression). Using this method, the prompt template token count went from 181 to 78, decreasing the total token count from 1005 to 902. Next in the proposed contextual pipeline is the contextual compression described in Section 3.6.2. By using the exact methodology as described there, the contextual tokens were

reduced from 827 to 272, making the total token count decrease further from 902 to 357. The final step of the compression pipeline is the compression of the whole prompt, as explained in Section 3.6.3. In this case, the compression rate was set to 30%, as this was proved to be the highest compression rate that also kept a lot of context. This method further reduces the token count from 357 to 289, and the whole flow from 1005 to 289. A prompt size that is almost 3.5 times smaller than the original, and that achieves a similar flow completion time of 21.7 seconds. The answers, given by OpenAI's *gpt-4* model, from the two flows, are also quite similar and both answer the question adequately:

- **Uncompressed prompt answer:** An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.
- **Compressed prompt answer:** An attention function is a type of mapping that relates a query and a set of key-value pairs to an output. All elements such as the query, keys, values, and output are vectors. It's a mechanism that enables the modeling of dependencies without considering their distance in the input or output sequences. Therefore, this function plays a crucial role in sequence modeling and transduction models in various tasks.

This shows that a 70% reduction in prompt size does not affect the LLM's ability to understand the textual and contextual meaning of the prompt to answer the question.

4.6 The Whole Pipeline

The following sections test all the hypotheses (**H1-H5**) when they are put together into one pipeline. It has been demonstrated how much costs and completion times are affected by a variety of different methodologies in the RAG pipeline. This section, however, looks at the pipeline as a whole. It looks into how significant the cost reduction is, and if the methodologies affect completion times positively or negatively.

4.6.1 Cost Reduction

To test and calculate the total cost reduction of an enhanced RAG pipeline a realistic scenario has been created. Assume the users of an AI chat service send an average of 10,000 prompts per day, with an average length of

20 tokens each. To retrieve context from an external source, the prompts have to be embedded using an embedding model. The total token count of these prompts amounts to 200,000. Assume also that there are uploaded 1000 documents on average each day, and that each document has an average of 100,000 tokens. This brings the total number of embedding tokens up to $200,000 + 100,000 * 1000 = 120,000,000$. Using the OpenAI *V3 small* embedding model this would cost $(120,000,000 tokens/1000) * \$0.00002 = \$2.4$. In Section 4.3, it is shown that using an open-source model like *e5-small-v2* can reduce the cost by a factor of about 158. This would mean that using the *e5-small-v2* model instead could save $\$2.4 - \$0.015 = \$2.385$.

Of the 10000 prompts, 10% (1000) are classified as "simple" prompts. For simplicity, assume each prompt amounts to an average of 500 input tokens together with their respective prompt template and context. They each use an average of 20 output tokens, and the classification model (*gpt-3.5-turbo-1106)* uses 250 tokens to classify the prompts. The calculations for the "simple" prompts can be seen below in equations 4.4 and 4.5.

$$\underbrace{\$0.0010 * \frac{((500+250)*1000)}{1000} + \$0.0020 * \frac{(20*1000)}{1000} = \$0.79}_{\text{input and output cost for the 1000 "simple" prompts with classification = \$0.79}} \tag{4.4}$$

$$\underbrace{\$0.030 * \frac{(500*1000)}{1000} + \$0.060 * \frac{(20*1000)}{1000} = \$16.2}_{\text{input and output cost for the 1000 "simple" prompts without classification = \$16.2}} \tag{4.5}$$

With the 1000 "simple" prompts it is therefore possible to save $\$16.2 - \$0.79 = \$15.41$. In Section 4.2.2, it is shown that when doing prompt classification and the prompt is classified as "complex", the cost compared to a flow without classification actually increases. This increase can be calculated as follows in equations 4.6 and 4.7.

$$\underbrace{\$0.030 * \frac{(500*9000)}{1000} + \$0.060 * \frac{(20*9000)}{1000} = \$145.8}_{\text{input and output cost for the 9000 "complex" prompts without classification = \$145.8}} \tag{4.6}$$

$$\underbrace{\begin{aligned}&\$0.030 * \frac{(500*9000)}{1000} + \$0.060 * \frac{(20*9000)}{1000} \\ &+\$0.0010 * \frac{(250*9000)}{1000} + \$0.0020 * \frac{(5*9000)}{1000} = \$148.14\end{aligned}}_{\text{input and output cost for the 9000 "complex" prompts with classification = \$148.14}} \tag{4.7}$$

It is here apparent that the increased cost of using classification, $\$148.14 - \$145.8 = \$2.34$, is smaller than the decrease from the "simple" prompts. Total cost saved here is $\$15.41 - \$2.34 = \$13.07$, and total cost this far in the pipeline amounts to $\$0.015 + \$0.79 + \$148.14 = \$148.945$. Without classification and without using an open-source embedding model total cost would have been $\$2.40 + \$16.2 + \$145.8 = \$164.4$. The more prompts are classified as "simple", the more money there is to save by using the prompt classification.

The next step in the RAG pipeline is prompt compression. In Section 4.5.4 (Overall Compression), it is shown that a prompt with 1005 tokens including prompt template and context, can be reduced to 289 tokens. This is a reduction by a factor of 3.5, and using the example stated above it is possible to calculate the cost saving. Assume that the 250 tokens the RAG pipeline uses for classification remain the same, as the tokens here have already been reduced by doing Prompt Template Compression. The input tokens, being 500 per prompt, can then be reduced by a factor of 3.5 amounting to $500/3.5 \approx 143$. This token count can then be used in the consecutive calculations in equations 4.8 and 4.9.

$$\underbrace{\$0.0010 * \frac{((143 + 250) * 1000)}{1000} + \$0.0020 * \frac{(20 * 1000)}{1000} = \$0.433}_{\text{input and output cost for the 1000 "simple" prompts with compression} = \$0.433} \tag{4.8}$$

$$\underbrace{\begin{aligned} &\$0.030 * \frac{(143 * 9000)}{1000} + \$0.060 * \frac{(20 * 9000)}{1000} \\ &+\$0.0010 * \frac{(250 * 9000)}{1000} + \$0.0020 * \frac{(5 * 9000)}{1000} = \$51.75 \end{aligned}}_{\text{input and output cost for the 9000 "complex" prompts with compression} = \$51.75} \tag{4.9}$$

Prompt compression shows here a reduction in cost from \$148.945 to \$51.75, making the total cost reduction $\$164.4 - \$51.75 = \$112.65$. An overview of the total cost reduction derived from the above calculations can be seen below in Table 4.8.

Method	**Total Cost Reduction**
Open-source Embedding Model	\$164.4 → \$162.015
Prompt Classification	\$162.015 → \$148.945
Overall Compression	\$148.945 → \$51.75
Overall Cost Reduction	**\$164.4 → \$51.75**

Table 4.8: Cost reduction achieved by each of the different methodologies.

4.6.2 Time Reduction

To test how much the different methodologies affect the completion time of the RAG pipeline, the Attention is All You Need paper was used as an external knowledge base together with the prompt: What is an attention function? The total completion time of the RAG pipeline without using any of the proposed enhancements was 23.4s. This was done using the *text-embedding-ada-002* model from OpenAI. From the results in Section 4.3, it is shown that the *e5-small-v2* model is 2.38s faster than the Ada model. It achieves similar results here being 2.45s faster than the Ada model coming in at 20.95s. Prompt classification, on the other hand, does not decrease completion time. Section 4.2.2 shows that prompt classification in the pipeline adds an average of 2.74 seconds. The completion time is therefore back to more than originally: 23.69s. In Section 4.5.4 (Overall Compression), it was shown that the completion time difference between a RAG pipeline with compression and one without is minuscule. Overall this means that the total completion time of the RAG pipeline is increased by a couple of milliseconds, as can be seen below in Table 4.9. From the table, it is clear that the only thing decreasing completion time is using an open-source embedding model.

Method	**Total Completion Time Change**
Open-source Embedding Model	$23.4s \rightarrow 20.95s$
Prompt Classification	$20.95s \rightarrow 23.69s$
Overall Compression	$23.69s \rightarrow 23.69s$
Overall Completion Time Change	**23.4s → 23.69s**

Table 4.9: Completion time changes achieved by each of the different methodologies.

4.7 Discussion

One question remaining now is what strategies a company looking to acquire its own AI chat service should use. This really comes down to the company's main preferred features of such a service. Do they value, efficiency, cost-effectiveness, or quality of answers the most?

	Most Important Feature		
Method / Model	Efficiency	Cost Effectiveness	Quality of Answers
Prompt classification	No	Yes	Yes
Embedding model	e5-small-v2	e5-small-v2	V3 large / V3 small (OpenAI)
Compression	Optional, but encouraged	Yes	Optional, but encouraged
Explainability	No	No	Yes

Table 4.10: Recommended methods to use according to what the most important feature is. Compression is set as optional for efficiency and quality of answers as it does not really affect those in any major way. It is of course recommended to use compression as it will reduce cost.

Table 4.10 above outlines the recommended methods and models for building an AI chat service, depending on whether the primary focus is efficiency, cost-effectiveness, or answer quality. If efficiency is most important no prompt classification should be done, the fastest embedding model (e5-small-v2) should be used, compression is optional and explainability should not be enabled. Similar recommendations are also given if a high level of cost-effectiveness or quality of answer is wanted.

Section 2.2.2 (Large Language Models) shows that the landscape of LLMs evolves fast, with frequent changes in both pricing and performance. Just between the time that section was written and the finalization of this thesis, OpenAI has released a new production-ready model called *gpt-4-turbo-2024-04-09*. This model has not only a better performance than the *gpt-4* model, but also a better pricing[8]. Embedding models are also frequently being improved and released, which could make some of the recommendations in this discussion section obsolete [24].

[8] https://openai.com/pricing

Chapter 5

Conclusions

This thesis proposes an enhanced RAG pipeline for custom AI chat services, by investigating, implementing, and testing various state-of-the-art methodologies. The proposed pipeline reduces costs and completion times while enhancing the quality of answers and incorporating explainability. To achieve this, prompt engineering, prompt classification, prompt compression, and embedding optimization were used. Explainability was achieved by using natural language in the form of a reflection prompt template, CoT prompting, and sources. Classification of prompts was achieved by using LLMs in MSPF. A compression pipeline was introduced that includes a prompt template compression methodology, contextual compression using Langchain, and a prompt compression using LLMLingua. Finally, embedding models were compared to find the best and most cost-effective model. The project's results show that cost can be decreased quite drastically (69%) by using prompt classification, prompt compression, and open-source embedding models in the RAG pipeline. Prompt compression minimally affects completion times; however, open-source embedding models have been shown to outperform the Embedding-as-a-Service models offered by companies like OpenAI, achieving speeds up to 41% faster.

In the thesis, a lot of LLM answers have been shown as results, and a quite subjective measure of their quality has been given. To acquire a more objective quality measure, a survey or user-test could have been conducted among the users of such AI chat services. Doing this would result in an indication of whether or not explainability increases answer quality or makes it more confusing. It could also help decide if the different compression techniques make the LLM give better or worse answers. When it comes to the prompt classification testing that was done in this project, it would also be possible to test more than only OpenAI models. An open-source model like *bert-base-uncased*[1] could be pre-trained to do prompt difficulty classification. Although the price

[1] https://huggingface.co/google-bert/bert-base-uncased

of classifying with the *gpt-35-turbo-1106* is minuscule, this would eliminate the classification cost altogether. In the thesis, RAGAS metrics were used to compare the embedding model. This could also have been used for comparing things like compression techniques. Since the start of this project a lot of LLM-as-a-service products have been released like Google's Gemini models [26] and Anthropic's Claude 3 models[2]. It would be interesting to test the proposed RAG pipeline with other LLMs, not from OpenAI's catalog of models.

Appendix A

The Proposed Pipeline

See Figure A.1 on next page ↓

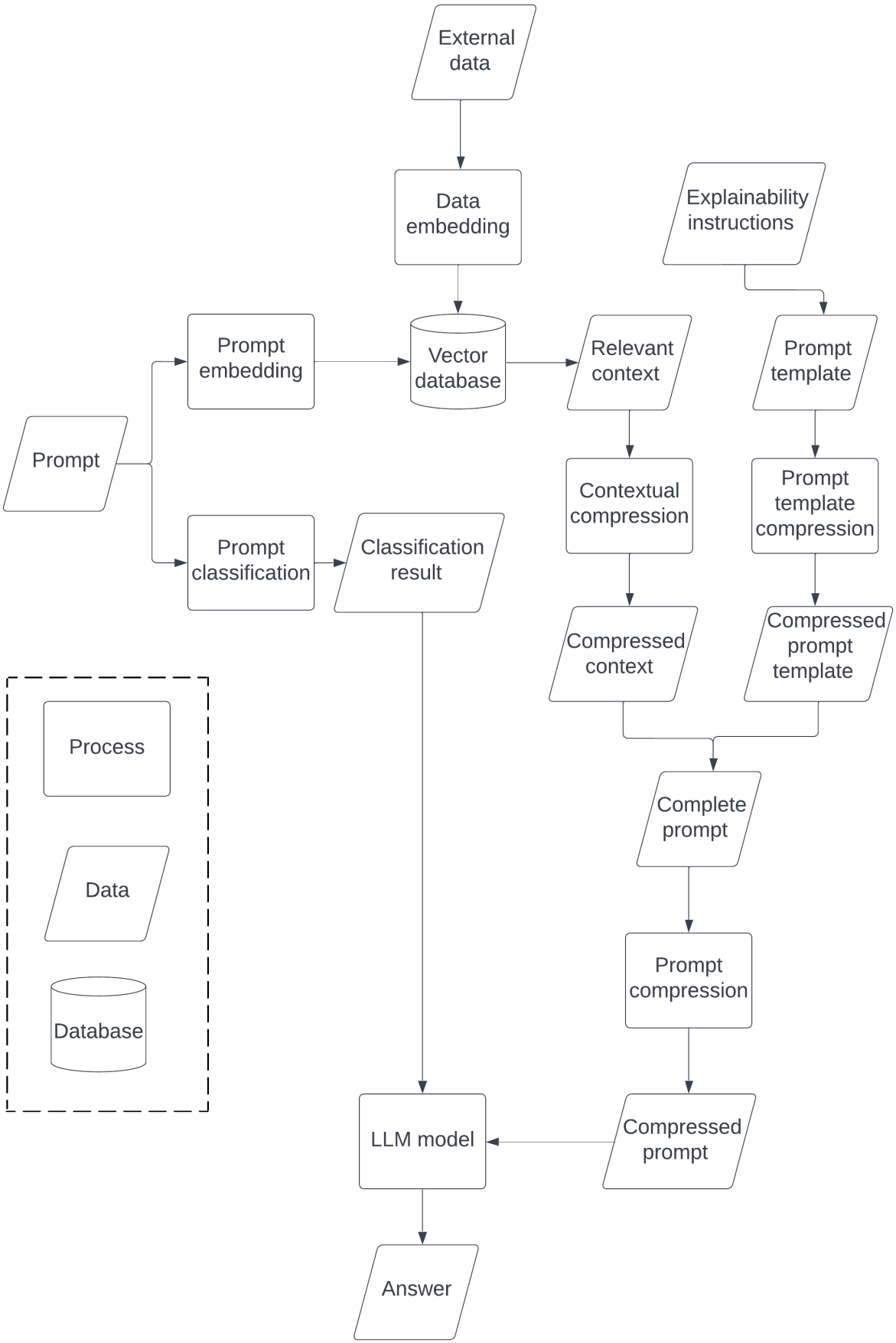

Figure A.1: The whole RAG pipeline proposed in this project.

Appendix B

Explainability Results

B.1 Explainability Using Prompt Engineering

The following prompts were created by using OpenAI's *gpt-4* model[1].

1. **How tall is the Burj Khalifa?**

   - **Non-explainability answer:** The Burj Khalifa is approximately 828 meters tall.
   - **Explainability answer:** The Burj Khalifa, located in Dubai, United Arab Emirates, is approximately 828 meters (2,716.5 feet) tall. This measurement includes all of its architectural elements.

     The Burj Khalifa is currently the tallest structure and building in the world. It was designed by the Chicago-based architectural firm Skidmore, Owings & Merrill and was completed in 2010. The building's height was made possible by a number of technological innovations, including high-strength concrete and a buttressed core structural system.

     However, it's worth noting that buildings and structures are often measured in different ways, and the definition of "tallest" can vary. For example, some measurements include antennas or other non-architectural features, while others do not. In the case of the Burj Khalifa, the stated height includes all architectural elements but does not include any broadcast antennas or other equipment.

     Facts to be fact-checked:
     1. The Burj Khalifa is approximately 828 meters tall.
     2. The Burj Khalifa is the tallest structure and building in the world.
     3. The Burj Khalifa was designed by Skidmore, Owings & Merrill.

[1] https://openai.com/research/gpt-4

4. The Burj Khalifa was completed in 2010.

Confidence Level: 100%

Sources:
1. Burj Khalifa:
https://www.burjkhalifa.ae/en/the-tower/facts-figures/
2. Skidmore, Owings & Merrill:
https://www.som.com/projects/burj_khalifa

2. **Who won World War 2?**

- **Non-explainability answer:** World War II was won by the Allies, which was a group of countries that included the United States, the United Kingdom, the Soviet Union, China, and France, among others.
- **Explainability answer:** World War II was won by the Allied Powers, which included major nations such as the United States, the United Kingdom, and the Soviet Union. These countries, along with others, fought against the Axis Powers, which included Germany, Italy, and Japan. The war officially ended in 1945 with the unconditional surrender of the Axis Powers.

  The victory of the Allies in World War II was not the result of a single battle or event, but rather a series of victories in various theaters of war around the globe. The Allies were successful due to a combination of factors including superior numbers, better resources, and strategic decisions. However, the war resulted in the loss of millions of lives and widespread destruction.

  There are some ambiguities and limitations to this answer. For instance, the specific contribution of each Allied nation to the victory is a topic of ongoing debate among historians. Furthermore, the aftermath of the war led to significant geopolitical changes, including the start of the Cold War between the United States and the Soviet Union, and the decolonization of Africa and Asia.

  Facts to be fact-checked:
  1. World War II ended in 1945 with the unconditional surrender of the Axis Powers.
  2. The Allied Powers included the United States, the United King-

dom, and the Soviet Union.
3. The Axis Powers included Germany, Italy, and Japan.

Confidence Level: 100%

Sources:
1. https://www.history.com/topics/world-war-ii/world-war-ii-history
2. https://www.britannica.com/event/World-War-II
3. https://www.nationalww2museum.org/war/articles/who-were-allies
4. https://www.nationalww2museum.org/war/articles/who-were-axis-powers

3. **How many people live in the municipality of Grimstad?**

- **Non-explainability answer:** As an AI, I don't have real-time data. As of my last update in October 2021, the population of Grimstad, Norway, was approximately 23,000. For the most accurate and current population count, I would recommend checking the latest statistics on the official website of Statistics Norway (Statistisk sentralbyrå).

  Please note that population numbers can fluctuate due to various factors such as birth rates, death rates, and migration. Therefore, the number provided might not be exact at the time of your query.

- **Explainability answer:** As an AI, I don't have real-time data. However, according to the data from Statistics Norway as of January 1, 2021, the population of Grimstad, Norway, was approximately 23,862.

  However, this number could have changed by now due to births, deaths, and migration. For the most accurate and up-to-date population count, you should refer to the latest data from official statistical sources or the municipality itself.

  Facts to be fact-checked:
  1. The population of Grimstad, Norway, was approximately 23,862 as of January 1, 2021.
  2. This data was sourced from Statistics Norway.

  Confidence level: 85% (This is based on the reliability of the source and the fact that population counts can change over time.)

Sources:
Statistics Norway:
https://www.ssb.no/en/befolkning/statistikker/folkemengde/aar-per-1-januar

B.2 Explainability Using CoT

The following prompts were created by using OpenAI's *gpt-4* model[2].

Thought: I need to find the current population of the municipality of Grimstad. I will use the search tool to find this information.

Action:
"action": "Search", "action_input": "Current population of the municipality of Grimstad"

Observation: Arendal municipality in Agder, Norway Demographics Population by years (2016 - 2020) By 1st January 2021, Norway's population stood at 5 391 369. Population growth for 2020 came in at 23 800, making 2020 the year with the slowest population growth since 2001. Article / 12 May 2020. Here you will find weekly figures on the number of dead. Drammen is today the fifth largest city in Norway, with a population of 111,000. It is also one of the fasted growing urban regions in the country. ... Larvik municipality has a population of around 48,000 and like the other communities we have passed through, is today known as a popular summer spot for Norwegians. ... Grimstad municipality is ... Law enforcement and church services are provided at a national level in Norway. Municipalities are undergoing continuous consolidation. In 1930, there were 747 municipalities in Norway. ... Population (2017) (2021 for Oslo, Viken, Rogaland and Vestland) Area (km 2) ... Grimstad: Grimstad: Agder: 22,692 303.58 Bokmål: Beate Skretting: H: 4203 ... Almost 13 percent of the population live in Oslo, the biggest city of Norway and the country's capital. The most populated county is Viken which surrounds the capital Oslo. Thought:The search results provided the population of Grimstad municipality as of 2017, which was 22,692. However, the current population data is not available. I will provide this information to the user.

Action:

[2]https://openai.com/research/gpt-4

"action": "Final Answer", "action_input": "The population of Grimstad municipality was 22,692 as of 2017. Unfortunately, I was unable to find the current population data."

Action:

"action": "Search", "action_input": "Current population of Grimstad municipality"

Observation: Map of the United States with North Dakota highlighted. North Dakota is a state located in the Midwestern United States.All incorporated communities in North Dakota are considered cities, regardless of population; there are no towns, villages, or hamlets in the state. There are 356 municipalities. Cities Larvik municipality has a population of around 48,000 and like the other communities we have passed through, is today known as a popular summer spot for Norwegians. ... Grimstad has been known for a long time, but it was recognised as a harbour town in 1622. By 1747, Grimstad had a sizeable sailing community and was a known place for smugglers. ... Arendal municipality in Agder, Norway Demographics Population by years (2016 - 2020) Map of municipalities (2020) Norway is divided into 15 administrative regions, called counties (fylker in Norwegian, singular: fylke), and 357 municipalities (kommuner/-ar, singular: kommune - cf. communes).The capital city Oslo is considered both a county and a municipality.. Municipalities are the atomic unit of local government in Norway and are responsible for primary education (until ... The 304-square-kilometre (117 sq mi) municipality is the 264th largest by area out of the 356 municipalities in Norway. Grimstad is the 51st most populous municipality in Norway with a population of 24,017. The municipality's population density is 88.2 inhabitants per square kilometre (228/sq mi) and its population has increased by 12.8% over ... Thought:The search results indicate that the population of Grimstad municipality is 24,017. This information seems reliable as it provides specific details about the municipality, including its size and population density. It also mentions that the population has increased by 12.8% over an unspecified period, which aligns with the previous information that the population was 22,692 in 2017. Therefore, I can conclude that the current population of Grimstad municipality is 24,017.

Action:

"action": "Final Answer", "action_input": "The current population of Grimstad municipality is 24,017."

Thought: The user's original question was about the current population of the municipality of Grimstad. The most recent and reliable information found indicates that the population is 24,017.

Action:

"action": "Final Answer", "action_input": "The current population of the municipality of Grimstad is 24,017."

Appendix C

Prompts Used for Prompt Classification Testing

Prompt	Category
Write a joke about the Swede, the Dane, and the Norwegian	gpt-35
What is 21 + 19?	gpt-35
Who plays Jack Sparrow?	gpt-35
How to make a hamburger?	gpt-35
How old is Joe Biden?	gpt-35
Who is the president of Russia?	gpt-35
How many people live in India?	gpt-35
Where is Grimstad located?	gpt-35
What color do you get when you mix red and blue?	gpt-35
What is communism?	gpt-35
Hello	gpt-35
Who wrote 'To Kill a Mockingbird'?	gpt-35
What is Pi to 3 decimal places?	gpt-35
What is the capital of Italy?	gpt-35
Who won the World Series in 2020?	gpt-35
How to make a simple pasta dish?	gpt-35
Translate 'Hello, how are you?' to Spanish	gpt-35
What is the boiling point of water in Fahrenheit?	gpt-35
Who is the current Prime Minister of the United Kingdom?	gpt-35
How many metres in a kilometre?	gpt-35
Create a docker compose yaml file for me based on my repository	gpt-4
How can one center a div?	gpt-4
How do you write a tax return?	gpt-4

What is the difference between react and angular?	gpt-4
How to setup and use pnpm?	gpt-4
Explain streaming in nextjs	gpt-4
How do you sort this list in Python and then find the 4 highest numbers?	gpt-4
How to write an introduction to a technical report?	gpt-4
How to connect two docker containers?	gpt-4
Explain the concept of Higgs Boson particle in easy terms	gpt-4
How to navigate by the stars?	gpt-4
Describe the process of photosynthesis in detail	gpt-4
What does Nietzsche mean by 'God is dead'?	gpt-4
Explain the implications of the theory of relativity	gpt-4
Analyze the main themes of 'To Kill a Mockingbird'	gpt-4
What were the geopolitical factors that led to World War I?	gpt-4
Break down the process of DNA replication in cells	gpt-4
How to apply the scientific method in a real-world scenario?	gpt-4
Discuss the effects of climate change on the global economy	gpt-4
Explain the concept and implications of quantum entanglement in quantum physics	gpt-4

Table C.1: 40 test prompts and their "ground-truths" used to test the prompt classification capabilities of the OpenAI LLMs.

Appendix D

Prompts Used for Embedding Comparison

- What is an attention function?
- What is Scaled Dot-Product Attention?
- What is Multi-Head Attention?
- How do you add positional encodings to the input embeddings?
- What is the difference between self-attention layers and recurrent and convolutional layers?
- What dataset did they train on?
- What hardware did they use?
- What kind of optimizer did they use?
- What were their results when it comes to machine translation?
- What do they say in their conclusion?
- What are dominant sequence transduction models?
- Summarize the abstract.
- What are state of the art approaches in sequence modeling?
- Explain end-to-end memory networks.
- What is their model architecture?
- What is the encoder composed of?
- What about their decoder?
- What are their applications of attention in their Model?

- Do they use position-wise feed-forward networks?
- How do they use embeddings and softmax?

Appendix E

Contextual Compression Results

The following prompts were created by using OpenAI's *gpt-4* model[1].

1. **Prompt 1:**

   - **Uncompressed context answer:** An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. It computes the compatibility function using methods like a feed-forward network with a single hidden layer, as in the case of additive attention, or a dot-product for multiplicative attention.
   - **Compressed context answer:** An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

2. **Prompt 2:**

   - **Uncompressed context answer:** Scaled Dot-Product Attention is a form of attention mechanism where the input consists of queries and keys of dimension dk, and values of dimension dv. The function computes the dot products of the query with all keys, each divided by the square root of dk, and applies a softmax function to obtain weights on the values. This mechanism computes the attention function on a set of queries simultaneously, which are packed together into a matrix Q. The keys and values are also packed together into matrices K and V. The output matrix is computed as per a given formula. The mechanism gets its name "Scaled Dot-Product Attention" because it scales the dot products by 1 over the square root of dk, mainly to prevent

[1] https://openai.com/research/gpt-4

the dot products from growing large in magnitude, which would result in very small gradients in the softmax function.

- **Compressed context answer:** Scaled Dot-Product Attention is a type of attention mechanism wherein inputs consist of queries and keys of dimension d_k, and values of dimension d_v. The dot products of the query with all keys are computed, divided by the square root of d_k, and then a softmax function is applied to obtain the weights on the values. In practice, the attention function is computed on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed into matrices K and V. To counteract the effect of large values of d_k, the dot products are scaled by 1 divided by the square root of d_k. This mechanism is faster and more space-efficient in practice, as it can be implemented using optimized matrix multiplication code.

Appendix F

Prompt Compression Results

The following prompts were created by using OpenAI's *gpt-4* model[1].

1. **Prompt 1:**

   - **Uncompressed prompt answer:** An attention function is described as a method that maps a query and a set of key-value pairs to an output, where the query, keys, values, and the output are all vectors. The output is computed as a weighted sum of the values, the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.
   - **Compressed prompt answer:** An attention function can be described as mapping query and key to an output, where the query, keys, and values are vectors. The output is computed as a sum of the values where the weight is computed by a function of the query with the keys.

2. **Prompt 3:**

   - **Uncompressed prompt answer:** Multi-Head Attention is a method employed in the Transformer model which allows the model to jointly attend to information from different representation subspaces at different positions. It consists of several attention layers running in parallel. In this mechanism, the projections are parameter matrices WQi, WKi, WVi and WO. The algorithm first linearly projects the queries, keys, and values h times with different learned linear projections to dk, dk, and dv dimensions, respectively. On each of these projected versions of queries, keys, and values it then performs the attention function in parallel, yielding dv-dimensional output values. These are then concatenated and once again projected, resulting in the final values. The model uses multi-head attention in three different ways: In ëncoder-decoder attentionl̈ayers, the queries come from the previous decoder

[1] https://openai.com/research/gpt-4

layer, and the memory keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence. The computational cost is similar to that of single-head attention with full dimensionality.

- **20% compression rate prompt answer:** Multi-Head Attention is a method in which the model linearly projects the queries, keys, and values h times with different, learned linear projections to dk, dk, and dv dimensions. It employs parallel attention layers, or heads. Instead of performing a single attention function with d model-dimensional keys, values, and queries, this process finds it beneficial to linearly project the queries, keys, and values h times with different, learned linear projections. On each of these projected versions of values, attention function is then performed in parallel, yielding d dimensional output. These outputs are concatenated and once again projected, resulting in final values. This allows the model to focus on different regions of the input sequence, thereby improving its representational power.
- **30% compression rate prompt answer:** Multi-Head Attention is a technique used in the attention mechanism framework. It essentially allows the model to focus on different positions in the input simultaneously, which is different from the traditional one-head attention mechanism. In the given context, it is implemented with 8 layers of heads. This multidimensional model creates several learned linear transformations of the input, with different projections for each attention head. These linear transformations are computed using parameter matrices. The function given is MultiHead(Q, K, V), where Q is the query, K is the key, and V is the value. The dot product between Q and K establishes a sort of match, and this is then applied to V to get the output.
- **40% compression rate prompt answer:** According to the provided context, Multi-Head Attention is a function in the model in which the queries and the keyed values are parameter matrices. This function is designed to apply attention mechanisms multiple times with different parameter matrices. The attention function operates on a set of queries, packed together as a matrix. The weighted values result in another matrix of outputs. This concept is employed using 8 layers or 'heads', and the reduced size of each head helps manage the computational cost.
- **50% compression rate prompt answer:** The context doesn't provide a complete and clear definition of Multi-Head Attention. How-

ever, it is apparent from the text that it is a type of attention mechanism used in certain types of models. Multi-Head Attention employs several attention layers and is used in scenarios with multiple queries. It has some relation to dot products, queries, keys, and values, but the exact interaction is not explained in the provided context. The number of layers or 'heads' can vary, as mentioned with h=8 and h=4 in two different instances.

Bibliography

[1] Mistral AI. *Model Selection*. URL: https://docs.mistral.ai/guides/model-selection/. (accessed: 27.03.2024).

[2] Anthropic. *The Claude 3 Model Family: Opus, Sonnet, Haiku*. URL: https://www-cdn.anthropic.com/de8ba9b01c9ab7cbabf5c33b80b7bbc618857627/Model_Card_Claude_3.pdf. (accessed: 27.03.2024).

[3] Semen Andreevich Budennyy et al. "Eco2ai: carbon emissions tracking of machine learning models as the first step towards sustainable ai." In: *Doklady Mathematics*. Vol. 106. Suppl 1. Springer. 2022, S118–S128. (accessed: 12.02.2024).

[4] Youngjin Chae and Thomas Davidson. *Large Language Models for Text Classification: From Zero-Shot Learning to Fine-Tuning*. en. Aug. 2023. URL: https://files.osf.io/v1/resources/sthwk/providers/osfstorage/64e61d8f9dbc3f068f681622?format=pdf&action=download&direct&version=2. (accessed: 29.01.2024).

[5] Lingjiao Chen, Matei Zaharia, and James Zou. *FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance*. en. arXiv:2305.05176 [cs]. May 2023. URL: http://arxiv.org/abs/2305.05176. (accessed: 05.02.2024).

[6] Gemini Community. *Gemini Pro 1.5*. URL: https://www.gemini-ai.org/gemini-pro-1-5/. (accessed: 24.04.2024).

[7] Jiaxi Cui et al. *ChatLaw: Open-Source Legal Large Language Model with Integrated External Knowledge Bases*. en. arXiv:2306.16092 [cs]. June 2023. URL: http://arxiv.org/abs/2306.16092. (accessed: 05.01.2024).

[8] Erik Dale. "Optimizing and Evaluating EgdeAI - A Custom Chatbot for Employees of Egde." en. In: (Nov. 2023). URL: https://drive.google.com/file/d/13oYGKUYBnYwbK9n7kLjurYsNe2oqJIE1/view?usp=sharing. (accessed: 05.01.2024).

[9] Shahul Es et al. *RAGAS: Automated Evaluation of Retrieval Augmented Generation*. en. arXiv:2309.15217 [cs]. Sept. 2023. URL: http://arxiv.org/abs/2309.15217 (visited on 02/26/2024). (accessed: 26.02.2024).

[10] Sahel Eskandar. "Exploring Common Distance Measures for Machine Learning and Data Science: A Comparative Analysis." In: (2023). https://medium.com/@eskandar.sahel/exploring-common-distance-measures-for-machine-learning-and-data-science-a-comparative-analysis-ea0216c93ba3. (accessed: 20.11.2023).

[11] Yunfan Gao et al. "Retrieval-augmented generation for large language models: A survey." In: *arXiv preprint arXiv:2312.10997* (2023). (accessed: 27.02.2024).

[12] Morten Goodwin. "Smarte algoritmer for en grønnere planet." no. In: (Nov. 2023). URL: https://ny.ntva.no/innhold/artikler/kapittel-9-smarte-algoritmer-for-en-gronnere-planet. (accessed: 29.04.2024).

[13] Stephen M. Walker II. "HellaSwag: Can a Machine Really Finish Your Sentence?" en. In: (2024). URL: https://klu.ai/glossary/hellaswag-eval. (accessed: 10.05.2024).

[14] Huiqiang Jiang et al. *LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models*. en. arXiv:2310.05736 [cs]. Dec. 2023. URL: http://arxiv.org/abs/2310.05736. (accessed: 22.01.2024).

[15] Wenqi Jiang et al. *PipeRAG: Fast Retrieval-Augmented Generation via Algorithm-System Co-design*. en. arXiv:2403.05676 [cs]. Mar. 2024. URL: http://arxiv.org/abs/2403.05676. (accessed: 04.02.2024).

[16] Langchain. *Agents*. https://js.langchain.com/docs/modules/agents/. 2024. (accessed: 24.09.2023).

[17] Langchain. "Contextual compression." In: (2024). https://python.langchain.com/docs/modules/data_connection/retrievers/contextual_compression. (accessed: 08.02.2024).

[18] Kelvin Lu. "Hosting A Text Embedding Model That is Better, Cheaper, and Faster Than OpenAI's Solution." In: (2023). https://medium.com/@kelvin.lu.au/hosting-a-text-embedding-model-that-is-better-cheaper-and-faster-than-openais-solution-7675d8e7cab2. (accessed: 12.02.2024).

[19] Omer Mahmood. "What's Hugging Face?" In: (2022). https://towardsdatascience.com/whats-hugging-face-122f4e7eb11a. (accessed: 08.01.2024).

[20] Shervin Minaee et al. *Large Language Models: A Survey*. en. arXiv:2402.06196 [cs]. Feb. 2024. URL: http://arxiv.org/abs/2402.06196 (visited on 03/28/2024). (accessed: 28.03.2024).

[21] Niklas Muennighoff et al. "MTEB: Massive Text Embedding Benchmark." en. In: *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*. Dubrovnik, Croatia: Association for Computational Linguistics, 2023, pp. 2014–2037. DOI: 10.18653/v1/2023.eacl-main.148. URL: https://aclanthology.org/2023.eacl-main.148. (accessed: 10.01.2024).

[22] Nvidia. "Large Language Models Explained." In: (2023). https://www.nvidia.com/en-us/glossary/data-science/large-language-models/. (accessed: 22.09.2023).

[23] OpenAI. *Introducing ChatGPT*. URL: https://openai.com/blog/chatgpt#OpenAI. (accessed: 05.01.2024).

[24] OpenAI. *New embedding models and API updates*. URL: https://openai.com/blog/new-embedding-models-and-api-updates. (accessed: 29.04.2024).

[25] Ruoling Peng et al. *Embedding-based Retrieval with LLM for Effective Agriculture Information Extracting from Unstructured Data*. en. arXiv:2308.03107 [cs]. Aug. 2023. URL: http://arxiv.org/abs/2308.03107. (accessed: 05.01.2024).

[26] Sundar Pichai and Demis Hassabis. *Introducing Gemini: our largest and most capable AI model*. URL: https://blog.google/technology/ai/google-gemini-ai/#sundar-note. (accessed: 27.03.2024).

[27] Raf. "What are tokens and how to count them?" In: (2023). https://help.openai.com/en/articles/4936856-what-are-tokens-and-how-to-count-them. (accessed: 29.01.2024).

[28] Oguzhan Topsakal and Tahir Cetin Akinci. "Creating Large Language Model Applications Utilizing LangChain: A Primer on Developing LLM Apps Fast." en. In: *International Conference on Applied Engineering and Natural Sciences* 1.1 (July 2023), pp. 1050–1056. ISSN: 2980-3209. DOI: 10.59287/icaens.1127. URL: https://as-proceeding.com/index.php/icaens/article/view/1127. (accessed: 16.01.2024).

[29] Hugo Touvron et al. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. arXiv:2307.09288 [cs]. July 2023. URL: http://arxiv.org/abs/2307.09288. (accessed: 05.01.2024).

[30] Ashish Vaswani et al. "Attention is all you need." In: *Advances in neural information processing systems* 30 (2017). (accessed: 30.01.2024).

[31] Vectara. *vectara/hallucination$_e$valuation$_m$odel*. URL: https://huggingface.co/vectara/hallucination_evaluation_model. (accessed: 28.03.2024).

[32] Lei Wang et al. *Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models*. en. arXiv:2305.04091 [cs]. May 2023. URL: http://arxiv.org/abs/2305.04091. (accessed: 22.01.2024).

[33] Jules White et al. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. en. arXiv:2302.11382 [cs]. Feb. 2023. URL: http://arxiv.org/abs/2302.11382. (accessed: 10.01.2024).

[34] Miao Xiong et al. "Can LLMs Express Their Uncertainty? An Empirical Evaluation of Confidence Elicitation in LLMs." In: *arXiv preprint arXiv:2306.13063* (2023). (accessed: 08.03.2024).

[35] Jia-Yu Yao et al. *LLM Lies: Hallucinations are not Bugs, but Features as Adversarial Examples*. en. arXiv:2310.01469 [cs]. Oct. 2023. URL: http://arxiv.org/abs/2310.01469. (accessed: 24.01.2024).

[36] Haiyan Zhao et al. "Explainability for Large Language Models: A Survey." en. In: *ACM Transactions on Intelligent Systems and Technology* (Jan. 2024), p. 3639372. ISSN: 2157-6904, 2157-6912. DOI: 10.1145/3639372. URL: https://dl.acm.org/doi/10.1145/3639372. (accessed: 15.01.2024).