Bachelorarbeit

AI – Using a Large Language Model (LLM) together with Retrieval-Augmented Generation (RAG) for answering questions on custom data

Verfasser

Alexander Pintsuk

angestrebter akademischer Grad

Bachelor of Science (BSc)

Wien, 2025

Studienkennzahl lt. Studienblatt:	A11724556
Fachrichtung:	Informatik: Spezialisierung Data Science
Betreuerin / Betreuer:	Dipl.-Ing. Dr.techn. Marian Lux

Abstract

This bachelor thesis presents the development of a fully local application designed to enhance the factual accuracy of language model responses through retrieval augmented generation. The project focuses on creating an intelligent agent capable of basing its responses on external knowledge sources, specifically uploaded documents and web information. The application integrates a user-friendly interface, a system for storing and retrieving document information, and an intelligent agent framework equipped with specialised tools. These tools enable document and web searches, perform mathematical calculations, and include mechanisms for language translation and profanity checks. The design prioritizes transparency and performance, enabling users to examine the agents' reasoning process and managing the knowledge base. By implementing a retrieval-augmented generation approach, this application aims to improve the reliability and factual grounding of language model outputs, offering a solution for creating more accurate AI agents.

Contents

1 Motivation

With the appearance of GPT-3, a so-called Large Language Model (LLM), the industry and private users quickly used it to their advantage. Huge LLMs, meaning those with more than 100 billion parameters, are able to respond to many questions in a way that humans would respond. The LLM's knowledge seems infinite at first, but after an in-depth conversation, one may find out that its knowledge is limited. This is partly because LLMs by default have to resort to the knowledge they have 'learned' during their training and cannot access other knowledge without extra mechanisms behind. Even worse, the models might start to hallucinate: making up false facts that are not based on actual ground truth from the training data. This becomes especially relevant, since humans can trust AI-generated content as much as that produced by humans. [18]

Even more importantly, large language models of the LLaMa, BLOOM and GPT families tend to demonstrate ultracrepidarianism as their size increases (effectively higher FLOPs), meaning they offer opinions beyond their own knowledge. It was observed in early models that often user questions were avoided, but scaled-up, improved models tend to provide seemingly sensible yet wrong answers much more often, including errors in difficult questions that human supervisors often overlook.

This becomes even more of a problem when the model is able to actively persuade people of their content. It is especially dangerous when considering that generative AI, combined with personality inference from consumed text, could potentially create a highly scalable "manipulation machine" that targets individuals based on their unique vulnerabilities – without requiring human input. [34]

Additionally, large language models trained via RLHF (Reinforcement Learning via Human Feedback) can implicitly learn to mislead humans, while following their objective of reaching minimal loss. [44]

However, since these huge LLMs require a lot of computing power and memory storage, running one on a private device is not possible. In order to achieve the best results, users may opt to use the LLM as a service.

This comes at the cost of privacy, as the entire conversation of the chat leaves one's immediate boundary and is sent to a company's infrastructure, where it is processed by the large LLM.

This is why IT-savvy people or companies that value privacy, started using smaller LLMs that can be run even on less powerful devices, such as a laptop, instead of a datacenter – though this comes at the cost of lower-quality answers. [37]

Shortly after the first big success of LLMs, a technique called RAG (Retrieval Augmented Generation) became well-known. The idea is to augment the LLM with knowledge beyond its realm. In practice, this means supplying the LLM with additional information from other sources, such as files or the web, to improve the answer quality or provide relevant contextual information.

Additionally, these language models can also be further fine-tuned, if the necessary compute is available. This allows companies or individuals to better align the AI's responses with their own vision or provide an LLM with more information on relevant topics, without relying solely on RAG. It is important to note that fine-tuning a language model's weights is not the only available approach to improve performance. One can also modify the vector embeddings itself.

The motivation of this thesis is to use a language model large enough to fit on a modern desktop computer and answer questions based on data from uploaded documents and websites using information retrieval.

2 Background

To better understand how the whole system actually works, one must first understand the underlying concepts.

2.1 Language Models

One of the first 'intelligent' models was a rule-based model called "Eliza", created by Joseph Weizenbaum in 1966. This program used pattern recognition to simulate conversation. [43]

With the appearance of hidden Markov models, described by Leonard E. Baum and other authors [6] , there was a significant leap toward the system we know today. Instead of predefined rules, these models use probabilities to predict the next word based on previous ones.

Another significant discovery was the development of N-gram models. [7] They estimated the likelihood of a word appearing by looking at the words that came before it in a sequence. N-gram models emphasised the importance of context in language which was the foundation of future techniques.

With the introduction of Convolutional Neural Networks (CNN), such as the Neocognitron by Dr. Kunihiko Fukushima [13], the so-called "Deep Learning Models" were introduced. The Neocognitron presented the two basic types of layers in CNNs:

A convolutional layer consists of units with filters covering patches of the previous layer. Each unit's weight vector, known as a filter, is shared among the units. Downsampling layers contain units that cover patches of previous convolutional layers, computing the average activations within their patches. This downsampling aids accurate object classification in visual scenes, even when objects are shifted.

So-called Recurrent Neural Networks (RNNs) enabled capturing sequential dependencies in language but struggled with long-range dependencies. [31]

The so-called Long Short-Term Memory Models (LTSTM), introduced in the paper [15], also played a critical role. LSTMs allowed for the creation of deeper, more complex neural networks capable of handling vast amounts of data.

In 2015, Bahdanau et al. introduced Sequence-to-Sequence models (Seq2Seq) [36]. A Seq2Seq model architecture is composed of two key components: an encoder and a decoder. The encoder is responsible for processing the input sequence, generating a fixed-length context vector that encapsulates the input sequence's meaning. This context vector is subsequently utilised by the decoder to produce the output sequence, step by step.

In detail, the encoder typically employs a Recurrent Neural Network (RNN) to process the input sequence element by element, creating a fixed-length hidden state vector at each step. The last hidden state vector functions as the context vector and is passed on to the decoder.

The decoder, also usually implemented as an RNN, takes the context vector and generates the output sequence sequentially. It achieves this by generating a probability distribution for the potential output elements at each step and then selecting the next element of the output sequence by sampling from this distribution.

The final major architectural change, the transformer architecture, which was introduced just in 2017, addressed many of the shortcomings of Seq2Seq models, leading to significant improvements in results [39]. One of the major benefits of this architecture was the ability to capture extensive dependencies across large sequences. The key innovation lay in their use of self-attention mechanisms, enabling seamless integration of contextual information from all positions within a sequence, removing the need for recurrence and convolutions, resulting in superior quality models that were more parallelisable and faster to train.

2.2 Model tuning

Traditional finetuning requires updating and storing all parameters, which is computationally very expensive. Parameter-Efficient Fine-Tuning (PEFT) methods enable efficient adaptation of large pretrained models to various downstream applications by only finetuning a small number of (extra) model parameters instead of all the model's parameters. This significantly decreases the computational and storage costs. Recent state-of-the-art PEFT techniques achieve performance comparable to fully fine-tuned models. One such technique that freezes most of the pre-trained parameters and updates only a tiny module, called an adapter layer. [16]

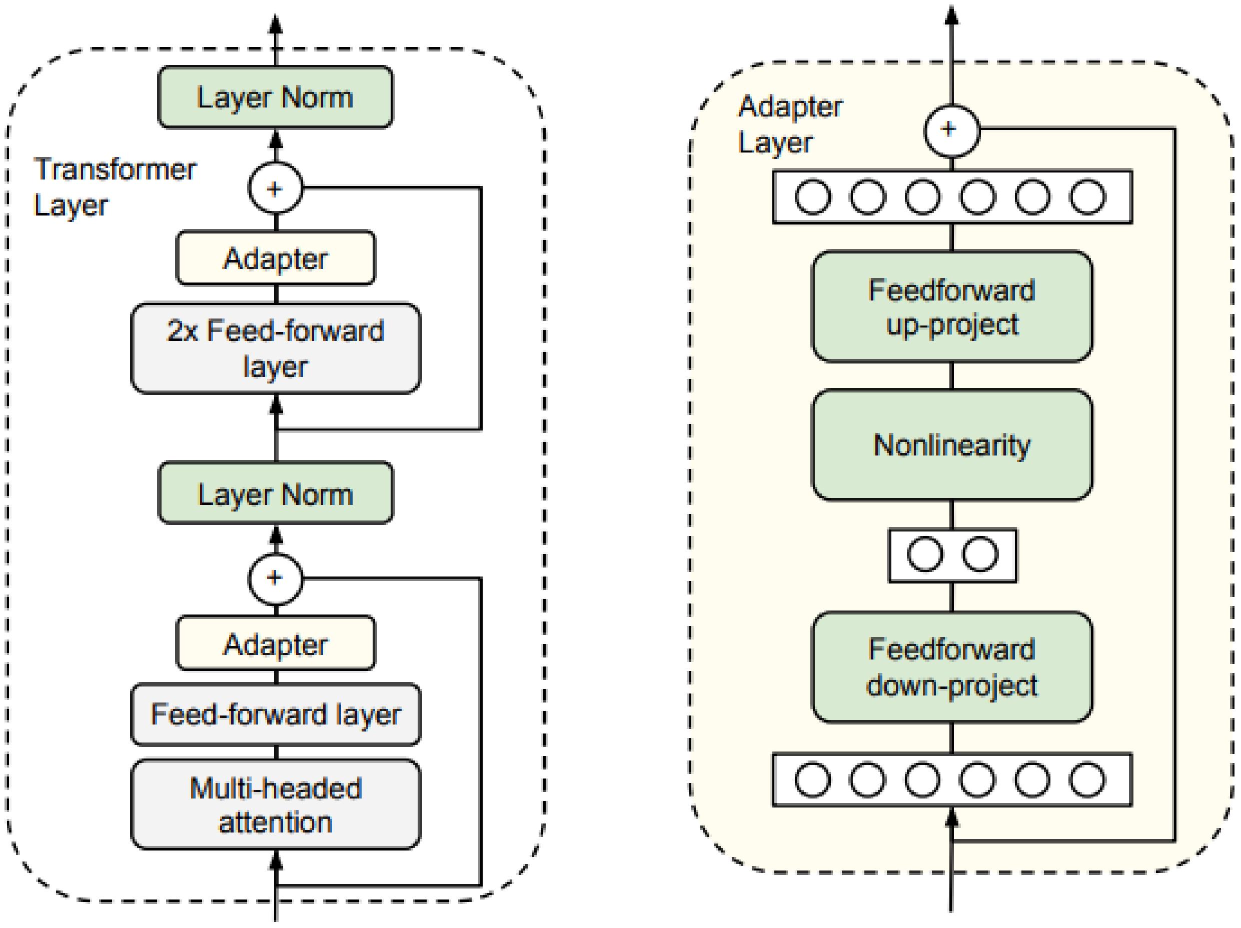

Figure 1: Architecture of the adapter module and its integration with the Transformer.
© Parameter-Efficient Transfer Learning for NLP, Neil Houlsby and Andrei Giurgiu and Stanislaw Jastrzebski and Bruna Morrone and Quentin de Laroussilhe and Andrea Gesmundo and Mona Attariyan and Sylvain Gelly

A groundbreaking training method, which allowed to finetune huge language models with less resources, was the Low-Rank Adaptation approach from the paper [17]. Instead of fine-tuning the entire model, it introduces small, low rank matrices (A and B) to the existing model's weights. This is because LORA auhors made the discovery that the effective "rank" of the necessary adjustments to the pre-trained model is often much lower than the rank of the original weight matrices. This approach offers two distinct advantages: firstly, the pretrained model's weights remain unchanged, reducing the risk of catastrophic forgetting. Secondly, LoRA's rank-decomposition matrices have significantly fewer parameters compared to the original model, making the trained LoRA weights easily transferable. This for example allows to switch the finetuning behaviour of an LLM at runtime, nearly instantaneously.

Shortly afterwards a technique called QLora (Quantized Low-Rank Adaptation) introduced novel techniques hat not only reduce memory requirements but also enhance the efficiency of the fine-tuning process. At its core, QLoRA introduced several key innovations, including 4-bit NormalFloat (NF4), Double Quantisation, and Paged Optimisers, which collectively allow for the fine-tuning of massive models while maintaining performance. [11] 4-bit NormalFloat (NF4) is a quantisation method that represents real numbers with just 4 bits, significantly reducing memory usage. NF4 uses more bits to represent values near zero and gradually fewer bits for larger or smaller values. This is based on the fact that many activations and weights in neural networks have a high density around zero. Its goal is not to represent every real number perfectly, but instead, it focuses on approximating the most frequent and important values with sufficient accuracy. This allows for significant compression, meaning a lowered memory footprint, while maintaining a reasonable performance. Double Quantisation further compresses the model by quantizing the quantisation constants themselves. Paged Optimisers optimise memory usage during training by only loading and updating a subset of the model's parameters at a time.

One brand new transformer architecture, swaps the given precision with ternary operators -1, 0, 1, achieving similar results, while reducing latency, memory and energy consumption hugely. [25]. An Official inference framework for 1-bit LLMs is BitNet from Microsoft.

A new approach, to further improve Lora is the Weight-Decomposed Low Rank Adaptation (DoRA) [23]. DoRA decomposes the pre-trained weight into two components, magnitude and direction, for fine-tuning, specifically employing LoRA for directional updates to efficiently minimise the number of trainable parameters. By employing DoRA, the learning capacity and training stability of LoRA is enhanced, while avoiding any additional inference overhead.

Another approach from the paper [10] is to use deep reinforcement learning from human preferences (RLHF- Reinforcement Learning Human Feedback).It has been adopted to scale instruction tuning wherein the model is aligned to better fit human preferences. This recent development powers models like ChatGPT.

There is also a different idea called Knowledge Distillation (KD). KD is a method where a smaller model, called the student model, learns to perform tasks by copying the behavior of a larger, pre-trained model, known as the teacher model. This technique helps the student model become good at tasks while being much smaller and easier to run. KD is especially useful when it's hard to use a large model because of limited computer power or memory. By learning from the teacher model, the student model can do well on tasks without needing as much processing power, making it more practical for devices that need to work in real-time or with limited resources. The student model learns not just the correct answers but also the soft predictions from the teacher, which helps it understand the task better.[45].

This was demonstrated in the so-called MiniLLM paper [14]. Extensive experiments in the instruction-following setting showed that the distilled LLM generated more precise responses with higher overall quality, lower exposure bias, better calibration, and higher long-text generation performance than the baselines. The method is scalable for different model families with 120M to 13B parameters, but the general idea was applied across models of all sizes.

2.3 Embeddings

A major component of a retrieval system are word embeddings. Encoding words as dense vectors in a continuous space, significantly improved language understanding and the modelling of semantic relationships between words. A popular example is the Word2Vec architecture. [27]

In order to represent any kind of data, such as text, image, audio, or video, into meaningful clusters, embeddings are used. An embedding is a continuous vector space where the locations of those points in space are semantically encoded. There exist a few approaches:

Word Embeddings In word embeddings each individual word is presented as a vector. Different methods are used to create these word embeddings. One popular method is called Word2Vec, which focuses on how words appear together in text. Two main techniques within Word2Vec are Skip-gram and Continuous Bag-of-Words (CBOW). FastText is a similar method that goes beyond whole words. It also considers smaller parts of words, like groups of characters, to improve how well it understands the meaning of words. Another method called GloVe focuses on how often words appear together in text. It creates word embeddings by looking at the relationships between the probabilities of different word combinations. By representing words as these numerical codes, computers can better understand and process human language. [21]

Sentence embeddings In sentence embeddings, the complete sentence is represented as a vector. Techniques like the Universal Sentence Encoder (USE) use deep learning models to create these sentence representations. USE embeddings are designed so that sentences with similar meanings are closer together. Another method called SkipThought is particularly good at understanding the flow of ideas within a text. It considers the sentences that come before and after a given sentence, capturing the overall context. [9]

Document embeddings In document embeddings entire documents are represented as vectors. One example is Doc2Vec. It learns a unique vector representation for each document by considering not only the words within the document but also the document itself as a context. Essentially, it tries to predict words within a document given the other words and the document's overall theme. This process implicitly captures the semantic meaning and style of the document, allowing us to compare documents based on their overall similarity, not just the presence of individual words. [35]

Other embeddings Other types of embeddings include image embeddings, user embeddings, product embeddings and multi modal embeddings. [20]

2.4 Tokenisation

Tokenisation is a process where raw text is broken down into smaller units called tokens. These tokens can be words, subwords, or even characters, which serve as the basic building blocks for machines to understand and process human language. Traditionally, NLP models treated words as discrete units, but this approach posed challenges, particularly with rare words or words with multiple forms (like "play," "playing," "played"). Treating a word like "running" as individual characters ("r," "u," "n," "n," "i," "n," "g") also loses semantic meaning. Tokenisation helps manage vocabulary size, as frequent token combinations are prioritised in the vocabulary. It also improves the handling of rare words, as well generalised subword units can provide better understanding and representation of the text.

Subword-based approaches like Byte-Pair Encoding (BPE) have become widely used because they address these limitations. BPE is a technique that learns a vocabulary of subword units from training data. It starts with individual characters and iteratively merges the most frequent character pairs. For example, in the sentence "playing the piano," the algorithm might initially merge "pl" and "ay" due to their frequency, eventually forming subword units like "play" and "ing" until a desired vocabulary size is reached.

BPE offers several advantages since it significantly reduces the vocabulary size compared to word-level tokenisation, leading to more efficient model training and faster inference. It also handles rare and out-of-vocabulary words by representing them as combinations of learned subword units. For example, the word "unprecedented" can be represented as "un" + "prec" + "ed" + "ent" + "ed," even if it was unseen during training. This flexibility enhances the model's ability to generalise to new data and words.

Although subword-based approaches like BPE have revolutionised many NLP tasks, there is no one-size-fits-all solution. The optimal tokenisation strategy depends on the application and the context it is in. There is still ongoing research for exploring the best strategies for tokenisation. [26]

2.5 Chunking

Chunking strategies are used to break down large amounts of text into smaller pieces so that language models can better understand and process them. There are two main types: rule-based and semantic-based chunking. Rule-based methods rely on straightforward rules like using spaces or predefined character lengths to split text. Examples include libraries like NLTK, which uses these simple approaches. On the other hand, semantic chunking uses machine learning and context-based algorithms to group text by meaning rather than by fixed sizes or simple separators.

One common rule-based method is Fixed Size Chunking, where text is divided into chunks of equal size based on a predetermined number of characters or words. This method can also allow overlap between chunks to maintain context between segments. It is simple, computationally efficient, and ideal for structured data where chunk size matters more than context, such as standardised surveys or genetic data analysis. Another rule-based approach, Recursive Chunking, splits the text until an optimal size and structure are achieved, prioritizing paragraph and line breaks to keep semantic meaning. It is slower but better for general, unstructured text, useful in applications like summarizing long documents or processing large collections of customer support records.

Sentence-Aware Chunking uses natural language processing tools to split text by sentence boundaries. This method ensures chunks contain full sentences, preserving grammar and meaning, making it useful for analyzing educational content, medical guidelines, or legal documents where sentence integrity is important. Lastly, Semantic Chunking uses embeddings created by language models to measure the contextual distance between text groups. This technique is highly effective for complex data such as legal texts or information retrieval systems, as it keeps related content together and helps reduce misinterpretation.

The effectiveness of these strategies varies depending on the use case. Fixed-size chunking works best for well-structured data, offering a simple and fast approach. However fixed-size chunking often disregards the underlying textual structure, leading to incoherent segments and potentially irrelevant information. Recursive chunking provides better context for large volumes of general text but requires more resources. Sentence-aware chunking helps maintain sentence coherence, ideal for fields where clarity and grammatical structure are critical. Semantic chunking excels in applications requiring a deep understanding of context and meaning, like search engines and legal analysis, but may still struggle with complex or non-standard text formats. [19]

2.6 Prompting

Every language model has in common that the require a prompt as input. This prompt has special tokens like in the case of Meta LLama3. This is equivalent to the Beginning of sequence (BOS) token:

```
<|begin_of_text|>
```

```
<|end_of_text|>
```

The next is equivalent to the EOS token. For multiturn-conversations it's usually unused. Instead, every message is terminated with `<|eot_id|>` instead:

`<|eot_id|>`

The next token indicates the end of the message in a turn or to put it another way, it indicates the end of a single message by a system, user or assistant role as shown below:

`<|start_header_id|>{role}<|end_header_id|>`

The tokens below each enclose the role for a particular message. The possible roles can be: system, user, assistant. This is best demonstrated in a multiturn-conversation with Meta Llama 3 prompt template:

```
<|begin_of_text|><|start_header_id|>system<|end_header_id|>

{{ system_prompt }}<|eot_id|><|start_header_id|>user<|end_header_id|>

{{ user_message_1 }}<|eot_id|><|start_header_id|>assistant<|end_header_id|>

{{ model_answer_1 }}<|eot_id|><|start_header_id|>user<|end_header_id|>

{{ user_message_2 }}<|eot_id|><|start_header_id|>assistant<|end_header_id|>
```

Chat vs Instruct Additionally a Language Model might be trained to either or both tasks: Chatting or following instructions. While chat models prioritise the flow of the conversation a instruction tuned language model would prioritise fulfilling the task it was assigned to. A simple instruction could be something like "Summarise this text for me", but the user might still want to chat in a normal manner afterwards.

Zero-Shot Prompting This approach is the simplest and easiest to understand approach. Here we just supply our instruction and our question, which the Language model responds to. It is called zero shot because we are giving zero examples of the task execution. An example input could be:

```
Classify the text into neutral, negative or positive.
Text: I think the vacation is okay.
Sentiment:
```

In [41] it was shown that finetuning language models on a collection of tasks described via instructions substantially improved zero-shot performance on unseen tasks. This approach is called instruction tuning.

Few-Shot Prompting While large-language models demonstrate remarkable zero-shot capabilities, they still fall short on more complex tasks when using the zero-shot setting. Few-shot prompting can be used as a technique to enable in-context learning where we provide demonstrations in the prompt to give the model better performance. [8]

```
Classify the text into neutral, negative or positive.
Text: This is awesome! Sentiment: Positive
Text:This is bad! Sentiment:Negative
Text:Wow that movie was rad! Sentiment:Positive
Sentiment: What a horrible show!
```

.

This approach works good for easier things, but fails for more complex tasks, which is where the next popular approach comes in.

(Automatic) Chain of though prompting (COT) Introduced in the paper [42], chain-of-thought (CoT) prompting enables complex reasoning capabilities through intermediate reasoning steps. One can combine it with few-shot prompting to get better results on more complex tasks that require reasoning before responding. The basic idea is to spell out the logic of the 'thoughts' before doing the actual task. This 'tricks' the model into more structured outputs, which lead to better results.

It is also worth to mention that Chain-of-Though Reasoning can be handled without explicit prompting, simply altering the decoding process. Instead of greedy decoding, meaning choosing the next token with the highest probability, one can investigate the top-k alternative tokens. [40]

When applying chain-of-thought prompting with demonstrations, the process involves hand-crafting effective and diverse examples. This manual effort could lead to suboptimal solutions. In [48] an approach to eliminate manual efforts by leveraging LLM's with "Let's think step by step" prompt to generate reasoning chains for demonstrations one by one, was proposed. This automatic process can still end up with mistakes in generated chains. To mitigate the effects of the mistakes, the diversity of demonstrations matter. This work proposes Auto-CoT, which samples questions with diversity and generates reasoning chains to construct the demonstrations.

Auto-CoT consists of two main stages:

Stage 1): question clustering: partition questions of a given dataset into a few clusters Stage 2): demonstration sampling: select a representative question from each cluster and generate its reasoning chain using Zero-Shot-CoT with simple heuristics The simple heuristics could be length of questions (e.g., 60 tokens) and number of steps in rationale (e.g., 5 reasoning steps). This encourages the model to use simple and accurate demonstrations.

ReAct - Reasoning and Acting In the paper [47] framework named ReAct where LLMs are used to generate both reasoning traces and task-specific actions in an interleaved manner, was introduced-

Generating reasoning traces allows the model to induce, track, and update action plans, and even handle exceptions. The action step allows to interface with and gather information from external sources such as knowledge bases or environments.

The ReAct framework can allow LLMs to interact with external tools to retrieve additional information that leads to more reliable and factual responses.

Results show that ReAct can outperform several state-of-the-art baselines on language and decision-making tasks. ReAct also leads to improved human interpretability and trustworthiness of LLMs. Overall, the authors found that best approach uses ReAct combined with chain-of-thought that allows use of both internal knowledge and external information obtained during reasoning.

ReAct is inspired by the synergies between "acting" and "reasoning" which allow humans to learn new tasks and make decisions or reasoning.

Chain-of-thought (CoT) prompting has shown the capabilities of LLMs to carry out reasoning traces to generate answers to questions involving arithmetic and commonsense reasoning, among other tasks . But its lack of access to the external world or inability to update its knowledge can lead to issues like fact hallucination and error propagation.

ReAct is a general paradigm that combines reasoning and acting with LLMs. ReAct prompts LLMs to generate verbal reasoning traces and actions for a task. This allows the system to perform dynamic reasoning to create, maintain, and adjust plans for acting while also enabling interaction to external environments (e.g., Wikipedia) to incorporate additional information into the reasoning. The figure below shows an example of ReAct and the different steps involved to perform question answering.

Here is an example of what the ReAct prompt exemplars look like, obtained from the paper:

```
    Question What is the elevation range for the area that the eastern sector of the
Colorado orogeny extends into?
Thought 1 I need to search Colorado orogeny, find the area that the eastern sector
of the Colorado orogeny extends into, then find the elevation range of the
area.
Action 1 Search[Colorado orogeny]
Observation 1 The Colorado orogeny was an episode of mountain building (an orogeny) in
Colorado and surrounding areas.
Thought 2 It does not mention the eastern sector. So I need to look up eastern
sector.
Action 2 Lookup[eastern sector]
Observation 2 (Result 1 / 1) The eastern sector extends into the High Plains and is called
the Central Plains orogeny.
Thought 3 The eastern sector of Colorado orogeny extends into the High Plains. So I
need to search High Plains and find its elevation range.
Action 3 Search[High Plains]
Observation 3 High Plains refers to one of two distinct land regions
Thought 4 I need to instead search High Plains (United States).
Action 4 Search[High Plains (United States)]
Observation 4 The High Plains are a subregion of the Great Plains. From east to west, the
High Plains rise in elevation from around 1,800 to 7,000 ft (550 to 2,130
m).[3]
```

```
Thought 5 High Plains rise in elevation from around 1,800 to 7,000 ft, so the answer
is 1,800 to 7,000 ft.
Action 5 Finish[1,800 to 7,000 ft]
...
```

Reflexion In the paper [33] a framework to reinforce language-based agents through linguistic feedback was introduced. At a high level, Reflexion converts feedback (either free-form language or scalar) from the environment into linguistic feedback, which is referred to as self-reflection, which is provided as context for an LLM agent in the next episode. This helps the agent to rapidly and effectively learn from prior mistakes leading to performance improvements on many advanced tasks.

Reflexion consists of three distinct models:

An Actor: Generates text and actions based on the state observations. The Actor takes an action in an environment and receives an observation which results in a trajectory. Chain-of-Thought (CoT) and ReAct are used as Actor models. A memory component is also added to provide additional context to the agent. An Evaluator: Scores outputs produced by the Actor. Concretely, it takes as input a generated trajectory (also denoted as short-term memory) and outputs a reward score. Different reward functions are used depending on the task (LLMs and rule-based heuristics are used for decision-making tasks). Self-Reflection: Generates verbal reinforcement cues to assist the Actor in self-improvement. This role is achieved by an LLM and provides valuable feedback for future trials. To generate specific and relevant feedback, which is also stored in memory, the self-reflection model makes use of the reward signal, the current trajectory, and its persistent memory. These experiences (stored in long-term memory) are leveraged by the agent to rapidly improve decision-making. In summary, the key steps of the Reflexion process are a) define a task, b) generate a trajectory, c) evaluate, d) perform reflection, and e) generate the next trajectory. The figure below demonstrates examples of how a Reflexion agent can learn to iteratively optimise its behavior to solve various tasks such as decision-making, programming, and reasoning. Reflexion extends the ReAct framework by introducing self-evaluation, self-reflection and memory components.

Reflexion is well suited if an agent needs to learn from trial and error: Reflexion is designed to help agents improve their performance by reflecting on past mistakes and incorporating that knowledge into future decisions. This makes it well-suited for tasks where the agent needs to learn through trial and error, such as decision making, reasoning, and programming.

Tree of Thoughts (ToT) For complex tasks that require exploration or strategic lookahead, traditional or simple prompting techniques fall short. In [24] and [46] tree of thoughts was introduced. A framework that generalises chain-of-thought prompting and encourages exploration over thoughts that serve as intermediate steps for general problem solving with language models. ToT maintains a tree of thoughts, where thoughts represent coherent language se-

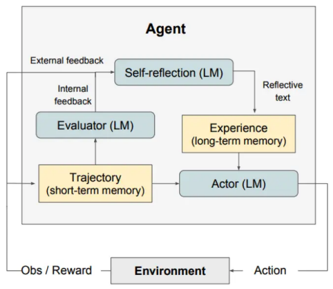

Figure 2: Reflexion architecture [33]

quences that serve as intermediate steps toward solving a problem. This approach enables an LLM to self-evaluate the progress through intermediate thoughts made towards solving a problem through a deliberate reasoning process. The LLM's ability to generate and evaluate thoughts is then combined with search algorithms to enable systematic exploration of thoughts with lookahead and backtracking.

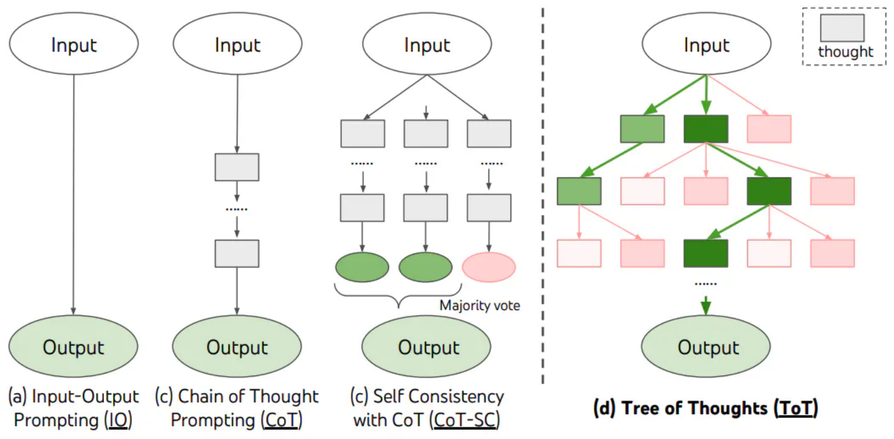
Input
Output
(a) Input-Output Prompting (IO)
(c) Chain of Thought Prompting (CoT)
Majority vote
(c) Self Consistency with CoT (CoT-SC)
thought
(d) Tree of Thoughts (ToT)

Figure 3: Tree of Thoughts [5]

2.7 Retrieval Augmented Generation

Retrieval Augmented Generation (RAG) is a methodology that combines a pre-trained retriever, which includes a Query Encoder and a Document Index, with a pre-trained seq2seq model, referred to as the Generator, and fine-tunes them in an end-to-end manner [22]. In the context of natural language processing, RAG is employed to generate text by leveraging both retrieval and generation mechanisms.

RAG might work the following way: given a query x, the retriever employs Maximum Inner Product Search (MIPS) to identify the top-K documents z_i relevant to the query. Subsequently, the generator utilises these retrieved documents to produce the final prediction y. In summary, RAG facilitates the generation of text by integrating information retrieval techniques with sequence-to-sequence modelling. By leveraging both retrieval and generation components, RAG enhances the relevance and informativeness of the generated text, making it particularly useful in applications requiring comprehensive and contextually rich responses.

There was also an idea to combine RAG with reciprocal rank fusion (RRF) [29]. Reciprocal rank fusion (RRF) is a method for combining multiple result sets with different relevance indicators into a single result set. RRF requires no tuning, and the different relevance indicators do not have to be related to each other to achieve high-quality results.

We can look at methods like semantic similarity and statistical methods like BM25. Semantic similarity and statistical methods have their strengths in retrieving information, depending on the query type. But finding a balance to prioritise which results should be displayed under specific circumstances is not easy.

Directly sorting the search results becomes tricky due to the mismatch in document score ranges. Keyword search scores fall within a positive range, usually between 0 and some maximum value, depending on the relevance of the query. On the other hand, semantic searches with cosine similarity generate document scores between 0 and 1. Reciprocal Rank Fusion (RRF) is a method used to combine results from different retrieval systems by leveraging the positions/rank of the documents, displayed in figure 4.

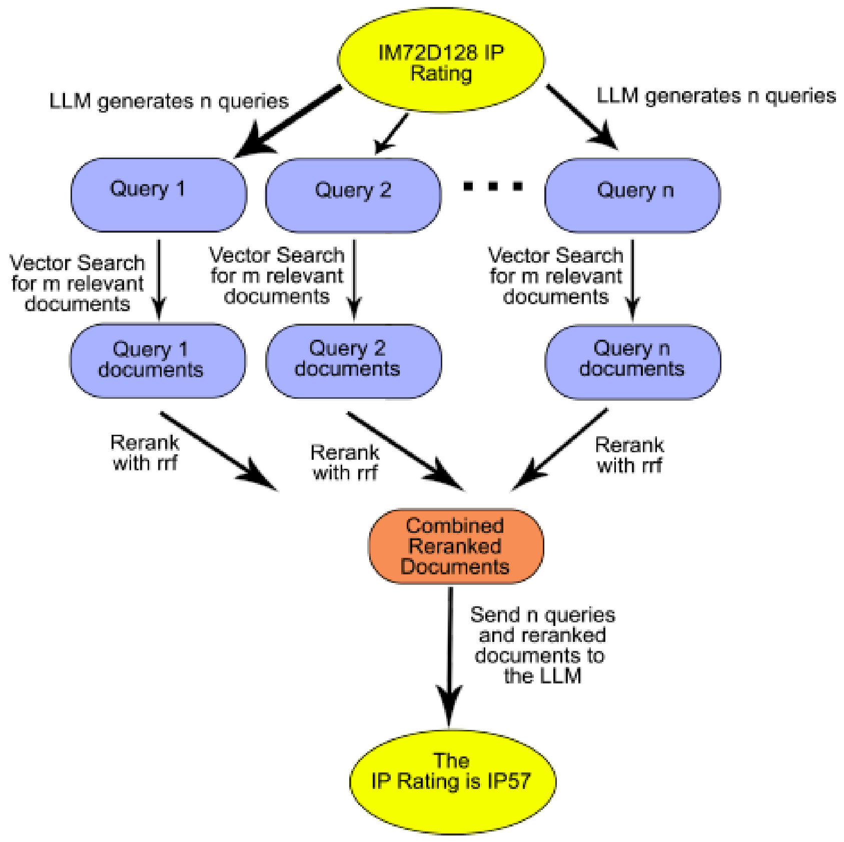

Figure 4: Rag fusion [29]

2.8 Sentence transformers

Sentence transformers are used for a wide range of tasks including Semantic Search, Information retrieval, Text classification and clustering. They have to major components: Bi-encoder and Cross encoders. [38],[30] Bi-encoders generate separate embeddings for each sentence independently. These embeddings can then be compared using a similarity metric (like cosine similarity) Cross-encoders process both sentences simultaneously within a single model. They directly learn a similarity score for the sentence pair, often achieving higher accuracy but at a higher computational cost.

2.9 Reranking

If documents or passages are retrieved, their contents may be too big to fit into the context of a language model, or even if the context window allows it, the

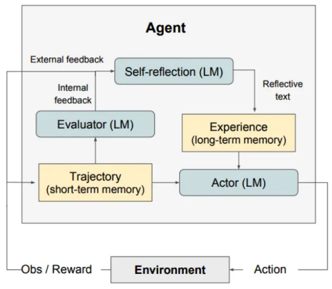

Figure 5: Sentence Transformers [3]

runtime would be too big to give a quick response, because of quadratic runtime complexity. Additionally even the order of the contents can play a major role in terms of answer quality, as research of the context of large language models shows [32].

As a solution, one could order documents or text passages according to their usefulness and provide the top n elements sorted by relevance score descending as context for a language model. This process is called reranking. Reranking can be done through a variety of mechanisms.

In general, effective cross-encoders (coupled with strong retrievers) outperform Large Language Models in terms of runtime by a wide margin, while also outperforming many of them in terms of accuracy. [12]

2.10 Tool/Function Calling

In some specific use cases language models should have the ability to decide to choose from a range of tools. One example of fine tuning a model with a massive dataset of API's was Gorilla [28], a finetuned LLama model. This model not only surpassed GPT-4 performance on writing API calls but also had fewer parameters than GPT-4. One can imagine a user request like 'How is the weather in Florida'. In that case the model can not give a truthful answer, because it has no knowledge of the current weather, since it just predicts the most likely answer to the question. One solution is to also provide a list of available functions and the user question. The model then decides what tool to call by returning the description of the function with the respective parameters, that should be called. If a function is returned in a correct format, the arguments and function name then can be parsed and the actual function can be executed and the result returned.

2.11 Creating Multi-Agents/Graphs

For figure 6 the independent agents are actually just a single LLM call. Specifically, each LLM has a custom prompt template. The main thing controlling the state transitions is the rule-based router. After each LLM call it looks at the output. If a tool is invoked, which is indicated by a JSON output by the LLM, then it calls that tool. If no tool is called and the LLM responds with a "final message" answer then it returns to the user. If no tool is called and the LLM does not respond with the final answer, then it goes to the other LLM.

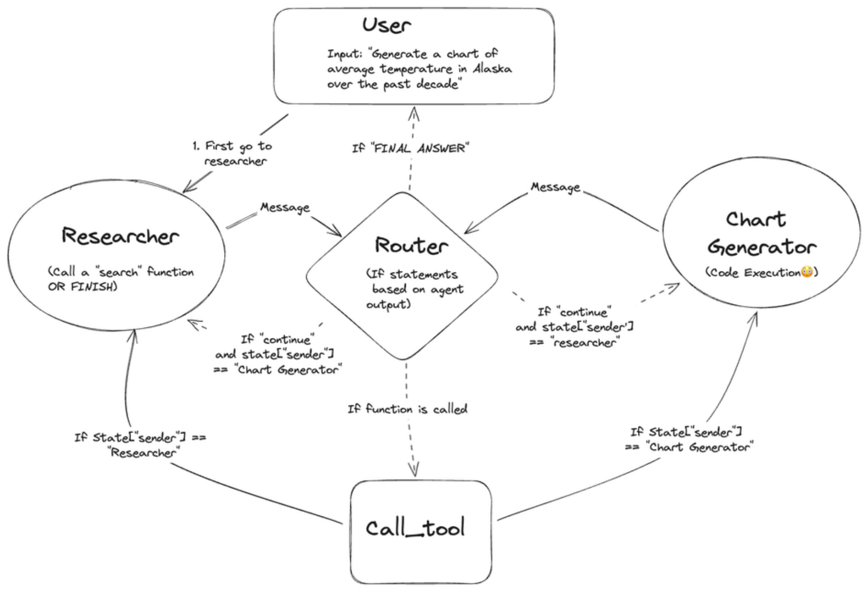
User
Input: "Generate a chart of average temperature in Alaska over the past decade"
1. First go to researcher
If "FINAL ANSWER"
Message
Message
Researcher
(Call a "search" function OR FINISH)
Router
(If statements based on agent output)
Chart Generator
(Code Execution)
If "continue" and state["sender"] == "Chart Generator"
If "continue" and state["sender"] == "researcher"
If function is called
If State["sender"] == "Researcher"
If State["sender"] == "Chart Generator"
Call_tool

Figure 6: Simple Multi Agent Graph [2]

For figure 7. This is similar to the above example, but now the agents in the nodes are fully fledged agents which can be customised very well to the specific use case. The supervisor then orchestrates the agents.

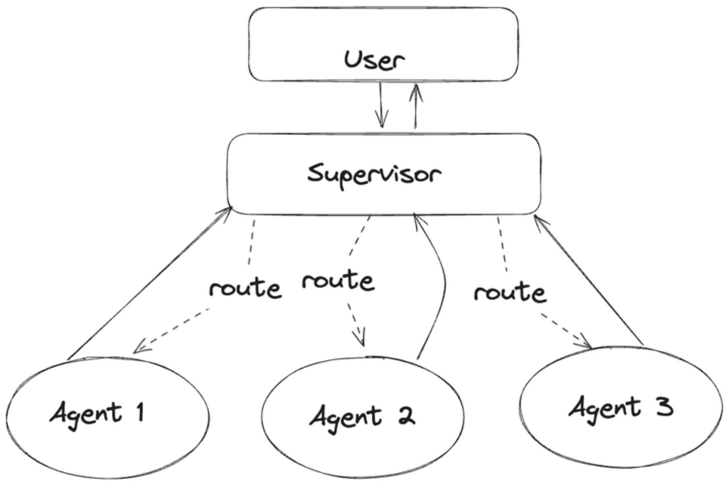

Figure 7: Supervisor Graph [4]

For figure 8 each independent agent has his own individual prompt, LLM, and tools. These agents are connected by the agent supervisor which is responsible for routing to individual agents. The supervisor can also be thought of as an an agent whose tools are other agents.

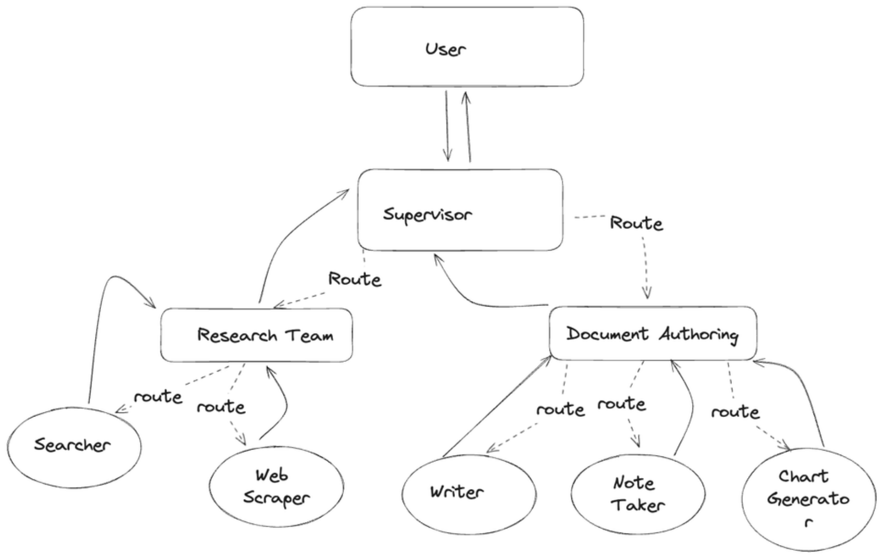

Figure 8: Hierarchical Graph [1]

3 Implementation

3.1 Key Components

3.1.1 Streamlit Application

The streamlitapp_with_langgraph class serves as the primary interface for users to interact with the Langgraph natural language processing model through a Streamlit application. Its purpose is to facilitate user interaction with the Langgraph model by providing a user-friendly interface for uploading documents, querying the model with questions, and displaying responses. Additionally, the class allows users to manage the document vector database used by the Langgraph model. It relies on dependencies such as Streamlit, a Python library for creating web applications, and Langchain, a Python package containing the Langgraph model.

The Streamlit application file operates in the following way: every time the code in the script receives a requested event (for example, a button being pressed), the entire page is reloaded. This means that even large classes are reloaded for every message sent to the underlying Python application.

In practical usage, users can navigate the Streamlit application using the sidebar to access different tabs, such as the default interface for interacting with the Langgraph model and the vector database management interface. In the vector database management interface, all indexed file chunks and their contents are displayed as a table. Each entry (file chunk) also indicates the file from which it originated.

Users can upload documents using the file uploader widget, enter questions in the chat input box, and receive responses from the Langgraph model. Uploaded documents and responses are displayed in their respective sections, providing users with a clear overview of their interactions.

To enhance debugging, a collapsible element is included above the chat. It allows the user to view the last full agent response, including intermediate steps from all nodes. This increases transparency by enabling users to see the outputs of different nodes during their respective steps.

Finally, users have the option to manage the document vector database, including clearing it if necessary.

In conclusion, the streamlitapp_with_langgraph class is the core of the web application user interface. It offers a way to interact with the Langgraph agent. By combining Streamlit with the features of the Langgraph model, users can easily upload documents, ask questions, and receive relevant responses.

Document Loading The first step before using documents for language models is loading the documents into text. For JSON files, the code directly loads the JSON data and creates a Document object, which contains the text. For PDF files, it employs the PyPDFLoader to extract text content. For other file types, UnstructuredLoader is used to extract text.

Text Chunker The text Chunker takes the text of all loaded documents and chunks them using Semantic Chunking. This chunker works by determining when to "break" apart sentences. This is achieved by identifying differences in embeddings between any two sentences. When this difference exceeds a certain threshold, they are split. The Semantic Chunker Model is "BAAI/bge-small-en-v1.5", available on huggingface.

Vector Database The vector database class performs multiple functions. Upon initialisation, it downloads an embedding model. This allows it to embed future documents more quickly, as the embedding model only needs to be loaded once. The embdding model used is also "BAAI/bge-small-en-v1.5".

Furthermore, the vector database class supports two primary functionalities:

Initialisation: During initialisation, all files within a special folder (indexedFiles) are automatically read, split, and embedded for the vector database. These embeddings are then saved in the internal vector database called ChromaDb. A ChromaDb Langchain wrapper is used to simplify database handling.

File Handling: There is also a function that takes a single input file, splits and embeds it, and adds it to the underlying vector database. Every time an embedded chunk is saved into the internal database, the file from which it originally originated is also saved as metadata associated with the given chunk. This enables instant tracking of the file source for a given word query.

The class handles loading different file formats by utilizing a generic Unstructured File loader and a PDF loader provided by Langchain. This enables the vector database to read multiple file formats. After loading, these documents must be split and embedded. I decided to use the Semantic Chunker provided by Langchain. At a high level, this splits the text into sentences, then groups them into groups of 3 sentences, and finally merges groups that are similar in the embedding space.

3.2 Tools

3.2.1 Document search Tool

The task of the document search is to respond with the best chunks of text from all documents stored within the vector database, using a list of queries. A list of queries was chosen to increase the likelihood of finding a suitable chunk that contains the answer. For each search query, the tool searches the vector database for chunks and returns them all together in a list. Since this yields many results, they must then be reduced. This is achieved by a "Reranker," an encoder-decoder model trained to rank text passages according to their usefulness, given a query. The result with the highest score is then chosen to be provided as context for the program that called the document search tool. The reranker model used is "ms-marco-MiniLM-L-12-v2".

Additionally, another mechanism was added to further enhance the result's usefulness. The top results from the Reranker are further enriched with the context of the document they come frome, further improving the result, since much needed context for the snippet within the document is given.

3.2.2 Websearch tool

The task of the websearch tool is to search on the web for content that the local system does not have a response for. It takes only a stringle string as argument, imitating a regular search query. After that a library called "duckduckgosearch", is called with the query. The number of results is limited to 5 items. It returns a list of text web results, already well formated, so the html is already parsed into the most relevant text. Since these results could again have different quality, they are too reranked with the reranker. The result with the highest score, is then as context for the program that called the tool

3.2.3 Mathtool

The task of this tool is to evaluate a mathematical expression and returns its result as a string. It uses the python library "numexpr" in the backround, which does the parsing and actual calculation.

3.2.4 Tool Node

The actual pice of code responsible for calling the functions is the Tool Node, which is already supplied by Langgraph.

3.3 Nodes

3.3.1 User Message Translation Nodes

Many language models can understand multiple languages. However, if language models are fine-tuned for instructions, their ability to understand other languages may decrease. Therefore, it is reasonable to generate a language model specifically designed for translating into a unified system language on which all subsequent models operate. This is where the translation node comes into play. Within the agent, its sole purpose is to translate the user message from any language into English. The translated message is then used for all subsequent operations within the agent. At the end of the agent graph, there is another node that translates the English response back into the original user language. The language chosen for the task is the "aya 8b" model, a multilingual model that support 23 languages. For detecting the language "papluca/xlm-roberta-base-language-detection" is used. The System prompt to translate into English is:

> You are a professional translator. You must only translate the given human message into English. Even if the user writes a question, you have to translate the question to english and you are NOT allowed to respond to the question. Provide only the translated text without

any additional information, comments, or explanations. Examples: Input: "Bonjour, comment ça va ?" Output: "Hello, how are you?

The System prompt to translate back into the users language is:

You are a professional translator. Your job is to translate the user message into user language. Only respond with the translated sentence in user language.

3.3.2 User profanity Check Node

The purpose of the profanity check node is to detect forbidden user request that inact violence or any kind of harm. The profanity check model uses a method implemented for models that provide native APIs for structuring outputs, like tool/function calling or JSON mode, and makes use of these capabilities under the hood. This is achieved by create to a Pydantic schema, which the language model should adhere to. The structure is as follows:

```
 class SafetyCheck(BaseModel):
"""Checks if a given input contains inhumane, illegal, or unethical content."""

input: str = Field(description="The phrase, question, or demand to be classified.")
is_unethical: bool = Field(
    default=False,
    description="True if the input contains unethical content. Unethical content
    includes, but is not limited to, mentions of violence, self-harm, illegal
    activity, lawbreaking, harm or danger to others, terrorism, or anything
    intended to cause injury or suffering."
)
```

3.3.3 Language detection for conditional routing

To detect the user language, a small transformer model is used, called "xlm-roberta-base-language-detection". This model is a XLM-RoBERTa transformer model with a classification head on top. It was trained to classify 20 languages. It takes the user question and returns a list of scores, associated to the respective classified language. The conditional routing function then sets a value based on if the language is English, or not.

3.4 Agent architecture

The LanggraphLLM is the heart of the actual agent. The figure 9 illustrates this. The top node start indicates the beginning, while the end node at the bottom indicates the end node. One can imagine the graph like a control flow chart. Depending on the choices from the routers, indicated by the naming of the edges, the next node is determined. Since all nodes need acces to data, the Langgraph framework added a State mechanism. Every node and router node has access to a global state and is allowed to modify it. This allows to process things in on node, write the results into the graph state and retrieve them in another node.

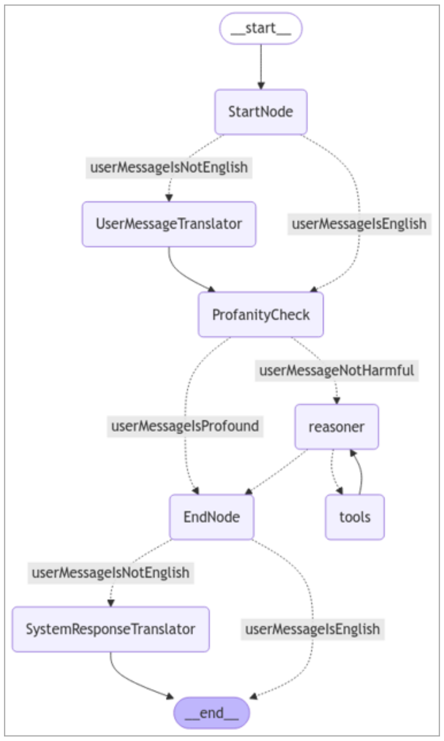

Figure 9: The graph of the smart agent

Start Node The sole function of this Node is to set some initial state when first being called. It sets the user question.

Conditional Routing to Profanity Check After the start Node, a function is called which determines the language of the user question. If the language is not english the User Message Translator will be called, else the Profanity Check is called directly.

Profanity Check This node checks if the user message is profound. If it is, the agent quits reasoning and returns a message that the supplied user message violates the softwares policy.

Intermediate Node The intermediate Node acts just as a placeholder and has no functionality

Reasoner The job of the reasoner is evaluate if to call tools and if it does to generate the correct parameters in the correct format. The underlying language model used for reasoning is ”Llama3.1:8b”.

The system prompt for reasoning looks as such:

> You are a helpful assistant with access to tools. You can search for relevant information using the provided tools and perform arithmetic calculations. For each question, determine if you can answer the question directly based on your general knowledge, or If necessary Use the ‘Search in document‘ tool to find the necessary information within the available documents. If you do not get an answer from the ’Search in document’ tool Message or get an error, use the websearch tool, but the websearch tool should have lower priority.

Tools After the reasoner message is generated it is checked for its content. It examines the current state of the conversation, specifically focusing on the latest message exchanged between the agent and the user, the reasoner message.

The function first extracts the latest message from the state input, which can be a list of messages, a dictionary containing messages, or an instance of a Python data class (BaseModel). It then checks if the message contains any tool calls. If any tool calls are found within the message, the function returns the string ”tools”, indicating that the agent should proceed to a ”ToolNode” in its workflow. This Node will then execute the specified tools and gather the results. If no tool calls are found in the message, the function returns ”EndNode”, signifying that the conversation has reached its conclusion and no further processing is required.

EndNode The job of the EndNode is to unify the paths from the reasoning step or profanity check. It does not modify state.

System Response Translator This node is responsible to translate the last message from the graph back into the original language, detected at the beginning. Again it uses the ”aya” model to translate back

3.5 Using the tool

The user interface is split into three main tabs. The first tab, ”Default,” provides a chat interface where users can chat with the language model. In this tab, users can upload documents, such as text, PDF, JSON, or markdown files, which are processed and indexed into a vector database for document retrieval capabilities. The interface displays a chat history managed by the ‘StreamlitChatMessageHistory‘ object, allowing users to view and interact with the ongoing conversation. Users can also reset the conversation memory to clear previous interactions and refresh the agent's state. The reason for this is the following: Since the whole conversation is kept in the graph, the whole conversation is used for the agents response. This may lead to a lower response quality. A special expander component allows users to inspect the reasoning path or graph state used by the language model to generate responses. This allows to see the state returned by each node. As a result one can track which tools were called in what order and what their returned values were.

The second tab, "vectordb," focuses on managing the vector database that stores document embeddings. Users can empty the entire vector database if needed by pressing the respective button. The interface presents a table view of indexed filenames and corresponding text chunks, giving transparency into the contents of the database. The content of each chunk can be even further inspected for certain words. These chunks are crucial for enabling context-aware responses by the language model when referencing uploaded documents.

The third tab, "settings," allows customisation of model behaviour. Users can select which language model to use for function calling/profanity check, by choosing from available model files. An additional toggle option enables or disables profanity checks, which can increase performance even more if disabled, because the profanity model does not have to be kept in memory and called. Configuration changes trigger callback functions to update the agent's settings dynamically, ensuring that adjustments are applied immediately during the session.

4 Evaluation

To evaluate the application's performance, I used a test document and a set of predefined questions provided by my professor. However, evaluating the system's actual usefulness posed several challenges.

Manually checking the responses for every test question was not practical. Small changes in the code could significantly alter the quality of the answers, making it necessary to create automated tests. To do this, I first defined the correct answers for each question. Then, I used another language model to assess whether the agent's responses accurately answered the questions, using the predefined correct answers as a reference.

The evaluation was divided into two main parts:

Normal question and answering, testing the agent's ability to answer questions in English based on the provided document and Non-English question and answering, evaluating the agent's performance with questions in languages other than English.

All evaluation results were logged in a file, allowing for detailed analysis at key development milestones. At these milestones, I also manually checked responses to ensure the automated tests were accurately reflecting the system's performance.

This approach allowed for a comprehensive evaluation of the application's functionality and performance across different scenarios. The automated tests provided a consistent and efficient way to measure the system's accuracy, while manual checks at milestones ensured the reliability of the overall evaluation process.

5 Challenges

During the development of the project I had to overcome numerous challenges.

The streamlit framework The first challenge was bringing Streamlit to work. Although Streamlit offers prefabricated components to simplify development, it requires mastering state management first. This is because every time an event occurs, the streamlit application script is re-run entirely. This can be problematic if the connected classes have a long startup time, such as the class loading the vector database. In this case, all files need to be read and indexed every time the class is created. State management in Streamlit was not well documented, making it even more difficult. Additionally, I had to be inventive to figure out basic things like passing the current selected value of a button/field to my agent when the value is changed. Another challenge arose when I tried to imitate the citing of sources in the chat. There was no built-in support for adding a hoverable character to display the source text used. Overall, I found Streamlit to be a good framework for prototyping, but I would prefer a different framework for more complex requirements.

Langchain framework The Langchain framework itself was another hurdle. While it offers great value if you understand how to use it, it has a steep learning curve. Many concepts need to be grasped, although they are inherently simple. However, the framework can make them unnecessarily complex. Additionally, since the goal of the thesis was to develop a local chatbot using popular APIs with the most support (OpenAI, Mistral, Gemini), I was unable to leverage these. As a result, I decided to use Ollama, an application that allows you to run any compiled C++ model on any platform where it can be compiled to. Unfortunately, the integration of Ollama into Langchain is not very extensive. There may be a reason for this: function calling, a major component of Langchain agents, was not available to local SLMs until recently due to advances in more efficient models. Consequently, the Ollama library does not provide all the support it could.

Ollama One major challenge with Ollama, compared to other APIs like OpenAI, is that anyone can upload a model. This allows for great flexibility but also increases complexity. Different models may respond completely differently to inputs and require different prompt formats and training. For instance, I encountered a situation where a model did not support the traditional "agentMessage" supported by OpenAI. Accounting for this in the application architecture is challenging because users might want to switch models at runtime, but this could lead to undefined outcomes. Moreover, the endpoints offered by Ollama did not satisfy all my needs, as other language model libraries provide a wider range of functionality, from grammar for tool calling to a generic message endpoint.

Ollama model prompting When creating the agent, the idea was to also change the model used for each tasking and see where they performed best. However, it turned out that every language model has its own Modelfile. This Modelfile specifies the exact format in which the query is supplied to the model. Normally, this is not a problem since Ollama is built to handle this. However, some models are not fine-tuned for certain role messages, such as system, user, or assistant. If one of those roles is missing and the whole system is built with the assumption that it exists, problems may occur. This was the case for the Aya model, because it does not support the "assistant" role.

Loading time For a local chatbot to leverage the benefits of RAG, a vector database needs to be spun up, and documents need to be chunked, embedded, and indexed. This loading time might discourage users from using the application because spinning up the already created vector database can take 5 to 10 seconds. Embedding an uploaded document could take even more time, depending on the document size. My approach was to focus every instantiation of large objects into the constructor, increasing startup time but reducing the time to answer each query. I also made great use of cache directories for all transformer models used for embedding, reranking, and general question answering.

Susceptibility to minor prompt perturbations One of the biggest challenges was actually getting the desired output out of a given language model. Even if the model is instructed on a given task clearly and the prompt is formatted correctly, the model might still fail to give the desired answer. For example, it took me quite a bit of effort to implement a profoundness check. The idea was for the model to respond only with 'true' or 'false,' indicating if the user message is profound. This proved to be more difficult than expected because the model did not respond exactly as anticipated. In my experience, making a model decide function calls is also a very big challenge. If the model is not fine-tuned, the output is not very consistent, making it harder to achieve consistently good results. Additionally, different models might be fine-tuned differently. For example, one model might be fine-tuned to always return tools and, in case a tool is not found, return a "tool not found" tool, while a Mistral model might be fine-tuned to respond normally in case no tool is found.

Model size Another challenge is the model size. If we compare models that we can run locally (about 8 billion parameters), their response quality is measurably lower than that of their higher-parameter models. A smaller model might make more mistakes adhering to a tool call schema and choosing the right tool to call with the correct parameters. However, it is very computationally and memory-intensive to run one of these models, even if they are quantized. This is why I gave the user the ability to dynamically choose between language models in the settings.

Choosing an Embedding model The backbone of the vector database is the underlying embedding model, which is responsible for converting the inputs, in this case, sentences, into a vector representation. I observed that changes in the embedding model significantly affected the chunks retrieved for a query. This meant in practice that I could quickly worsen the quality of the vector database's responses.

Embedding different Document types Another real challenge is the embedding of documents. Before embedding, one has to chunk the text and then encode it. These different encoding techniques can vary significantly for different file formats. A .txt file will be different to process than a .json file because the .json file contains not only text but also the text encoded within the JSON structure. Ideally, different parsers would be created for different file formats to achieve a more meaningful separation of chunks.

Tool calling The first challenge in tool calling is to actually get the language model to send the text that indicates a tool call. One approach is to fine-tune the model to adhere to the OpenAPI .json for function calling. However, the model does not always adhere to the format and may make mistakes, leading to unexpected results. In practice, this meant that I would receive a visible Tool call text with a few wrongly formatted parameters as a user. One elegant way to provide better tool calls is to use "structured generation," which forces the language model to adhere to a given format. This means that in the background, the choice of next available tokens for each token generation becomes restricted. This is a very useful feature, but Ollama does not support enforcing a .json format. Other Python libraries even support providing a grammar, which can be very useful.

Choosing a chunking strategy Choosing a chunking strategy is also a significant decision. There are several major approaches, but they can be refined with arbitrary weights, and no single chunking strategy is optimal for all documents. For example, depending on the use case and underlying documents, it might be useful to chunk by a character limit, with a limit of 250 tokens and an overlap of 10 percent. In other cases, it might be beneficial to increase the token window and reduce overlap. In general, it is easier to use a single chunking strategy, but to achieve optimal results, multiple chunking strategies might need to be used depending on the input and its context.

Dependency management Even though the scale of the software is small, it quickly required many dependencies, making project maintenance more difficult. As a result, I tried to keep only what I considered necessary and valuable. The numerous dependencies also meant that the underlying libraries might change over time during development. This is exactly what happened with the Langchain framework, LangGraph framework, and Ollama software.

Limitations of Langchain Integrating everything with Langchain into a single large agent was also challenging because Langchain, as the name suggests, was designed to handle chains, which are fixed paths that lead to a deterministic result. However, sometimes it is difficult to debug these chains and agents. As a result, I decided to heavily utilize Langgraph instead of Langchain. Learning this framework introduced additional workload, but in the long run, it significantly improved the understandability of the outputs and led to more consistent results.

One surprising challenge was the sheer amount of available language models. Many people use language models provided by popular companies and claim to have fine-tuned versions for specific tasks. This can be very helpful if it fits the use case exactly, but it can also be time-consuming to find the right model.

6 Lessons Learned

In terms of agent architecture, I learned that clearly defining the problem and designing the solution architecture before starting the project is crucial. While there are many ways to solve a problem, and the agent architecture can be fine-tuned extensively, this often comes at the cost of generalisation ability.

Expertise before undertaking the actual project would have been extremely valuable. With numerous moving parts involved (file chunking strategies, embedding models, agent architectures, graph routes, and language models), it is essential to always work towards a defined goal. Otherwise, one might lose sight of the objective and get bogged down in details, which ultimately wastes time.

My belief in the crucial nature of automated tests was further solidified. This was evident when changes to one component significantly impacted the entire system's response. Although the components were loosely connected, they still interacted closely. To quickly verify the continued proper functioning of my agent, running automated tests proved invaluable. In hindsight, I should have also tested more frequently on different platforms. There were instances where something worked on my Windows OS but not on a Linux OS.

I also learned that obtaining useful results from a small language model can be very challenging and requires significant manual effort. Moreover, minor changes, such as altering the embedding model of the vector database, can render this effort useless. Having an 'optimally' finetuned version of a given language model can further improve response quality, but actually finding it is another tough task.

Interestingly, I observed that "bigger is not always better." Increasing the parameters of a reranker or embedding model sometimes led to worse results. It's crucial to carefully weigh the benefits and drawbacks of each unique model.

7 Improvements

One of my first improvements would be to invest heavily in document preprocessing, like done in the lama index framework. Another idea would be to to potentially use a large language model to transform document chunks into more effective tokens. This could be achieved in various ways, but caution is necessary. Any improvement in performance on local files might decrease the agent's ability to generalize to unforeseen scenarios.

Furthermore, I would consider adding more structure to the reasoning tool. After some tool calls, we can eliminate others. For example, if a document search fails, a web search could be automatically triggered instead of querying the model again. This would significantly reduce the time to receive an answer.

Given sufficient time and resources, I would likely discontinue the use of Ollama. It has inherent limitations that cannot be easily overcome. Instead, I would opt for a more low-level library that allows me to define my own grammar and provides greater control during debugging.

Additionally, the current file loading process is not optimal. While the Unstructured library is powerful, it is overly bulky and complex. I would prefer to use specialized readers for each document type to enhance stability during file reading.

Moreover, I would favor implementing a chunking agent that aims to generate high-quality chunks. Assigning an agent the task of "intelligently" chunking a document could prove highly valuable for niche or particularly unstructured data.

8 Conclusion

This thesis resulted in a working local application that uses RAG to improve the accuracy of language model answers. The main goal was to create a system that lets users interact with a language model that is based on their own documents and information from the web.

The application has an easy to use interface, a system to store and find document information, and an agent that uses special tools. These tools help with searching documents and the web, doing math, translating languages, and checking for bad content. The design lets users see how the agent thinks and manage the information it uses. By using RAG, the application makes language model answers more reliable and accurate, which helps create better AI agents.

The core of the application is the 'streamlitapp with langgraph' class, which makes it easy for users to upload documents, ask questions, and get relevant answers. The Streamlit framework was used to build a simple and clear user interface. This interface includes features like vector database content inspection, and a debugging window. This choice of framework made it possible to quickly build a working prototype and create an easy-to-use application.

The application was built with a modular design. Components are used for loading documents, chunking them, and managing a vector database. Using Langchain's Semantic Chunker and ChromaDB for storing vectors was effective for handling different document types and making sure information was found quickly. The integration of tools for searching documents, searching the web, doing math, translating languages, and checking for profanity showed how was crucial in making the agent more useful.

The agent's architecture, based on the Langgraph framework, showed that it is possible to create an agent that can answer questions on custom data by using tools. Using nodes for language detection, translation, and safety checks, it ensured that the application could handle different user inputs and keep a safe environment.

During development, I faced challenges like making information retrieval more accurate and ensuring the application ran fast. I used a reranker for document search, which greatly improved the quality of the information found. The ability to see the agent's reasoning process through the debug view in the Streamlit interface made it easier to understand and fix any problems.

The user interface, allows managing the application with ease.

References

[1] Hierarchical architecture. https://blog.langchain.dev/content/images/size/w1000/2024/01/hierarchical-diagram.png. Accessed: 2025-01-18.

[2] Multi agent architecture. https://blog.langchain.dev/content/images/size/w1000/2024/01/simple_multi_agent_diagram--1-.png. Accessed: 2025-01-18.

[3] Sentence transformers. https://www.sbert.net/examples/applications/cross-encoder/README.html. Accessed: 2025-01-18.

[4] Supervisor. https://blog.langchain.dev/content/images/size/w1000/2024/01/supervisor-diagram.png. Accessed: 2025-01-18.

[5] Tree of thoughts. https://www.promptingguide.ai/_next/image?url=%2F_next%2Fstatic%2Fmedia%2FTOT.3b13bc5e.png&w=1200&q=75. Accessed: 2025-01-18.

[6] Baum, L. E., and Petrie, T. Statistical Inference for Probabilistic Functions of Finite State Markov Chains. *The Annals of Mathematical Statistics 37*, 6 (1966), 1554 – 1563.

[7] Broder, A. Z., Glassman, S. C., Manasse, M. S., and Zweig, G. Syntactic clustering of the web. *Computer Networks and ISDN Systems 29*, 8 (1997), 1157–1166. Papers from the Sixth International World Wide Web Conference.

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

[9] Cer, D., Yang, Y., Kong, S., Hua, N., Limtiaco, N., John, R. S., Constant, N., Guajardo-Cespedes, M., Yuan, S., Tar, C., Sung, Y., Strope, B., and Kurzweil, R. Universal sentence encoder. *CoRR abs/1803.11175* (2018).

[10] Christiano, P., Leike, J., Brown, T. B., Martic, M., Legg, S., and Amodei, D. Deep reinforcement learning from human preferences, 2023.

[11] Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. Qlora: Efficient finetuning of quantized llms, 2023.

[12] Déjean, H., Clinchant, S., and Formal, T. A thorough comparison of cross-encoders and llms for reranking splade, 2024.

[13] Fukushima, K. Neocognitron. *Scholarpedia 2*, 1 (2007), 1717. revision #91558.

[14] Gu, Y., Dong, L., Wei, F., and Huang, M. Minillm: Knowledge distillation of large language models, 2024.

[15] Hochreiter, S., and Schmidhuber, J. Long Short-Term Memory. *Neural Computation 9*, 8 (11 1997), 1735–1780.

[16] Houlsby, N., Giurgiu, A., Jastrzebski, S., Morrone, B., de Laroussilhe, Q., Gesmundo, A., Attariyan, M., and Gelly, S. Parameter-efficient transfer learning for nlp, 2019.

[17] Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models, 2021.

[18] Huschens, M., Briesch, M., Sobania, D., and Rothlauf, F. Do you trust chatgpt? – perceived credibility of human and ai-generated content, 2023.

[19] Kshirsagar, A. Enhancing rag performance through chunking and text splitting techniques. *International Journal of Scientific Research in Computer Science, Engineering and Information Technology 10*, 5 (2024), 151–158.

[20] Kukreja, S., Kumar, T., Bharate, V., Purohit, A., Dasgupta, A., and Guha, D. Vector databases and vector embeddings-review. In *2023 International Workshop on Artificial Intelligence and Image Processing (IWAIIP)* (2023), pp. 231–236.

[21] Kusner, M., Sun, Y., Kolkin, N., and Weinberger, K. From word embeddings to document distances. In *Proceedings of the 32nd International Conference on Machine Learning* (Lille, France, 07–09 Jul 2015), F. Bach and D. Blei, Eds., vol. 37 of *Proceedings of Machine Learning Research*, PMLR, pp. 957–966.

[22] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., and Kiela, D. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Advances in Neural Information Processing Systems* (2020), vol. 33, pp. 9459–9474.

[23] Liu, S.-Y., Wang, C.-Y., Yin, H., Molchanov, P., Wang, Y.-C. F., Cheng, K.-T., and Chen, M.-H. Dora: Weight-decomposed low-rank adaptation, 2024.

[24] Long, J. Large language model guided tree-of-thought, 2023.

[25] Ma, S., Wang, H., Ma, L., Wang, L., Wang, W., Huang, S., Dong, L., Wang, R., Xue, J., and Wei, F. The era of 1-bit llms: All large language models are in 1.58 bits, 2024.

[26] Mielke, S. J., Alyafeai, Z., Salesky, E., Raffel, C., Dey, M., Gallé, M., Raja, A., Si, C., Lee, W. Y., Sagot, B., and Tan, S. Between words and characters: A brief history of open-vocabulary modeling and tokenization in nlp, 2021.

[27] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space, 2013.

[28] Patil, S. G., Zhang, T., Wang, X., and Gonzalez, J. E. Gorilla: Large language model connected with massive apis, 2023.

[29] Rackauckas, Z. Rag-fusion: A new take on retrieval augmented generation. *International Journal on Natural Language Computing 13*, 1 (Feb. 2024), 37–47.

[30] Reimers, N., and Gurevych, I. Sentence-bert: Sentence embeddings using siamese bert-networks, 2019.

[31] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning internal representations by error propagation.

[32] Saito, K., Sohn, K., Lee, C.-Y., and Ushiku, Y. Where is the answer? investigating positional bias in language model knowledge extraction, 2024.

[33] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. Reflexion: Language agents with verbal reinforcement learning, 2023.

[34] Simchon, A., Edwards, M., and Lewandowsky, S. The persuasive effects of political microtargeting in the age of generative artificial intelligence. *PNAS Nexus 3*, 2 (01 2024), pgae035.

[35] Sinoara, R. A., Camacho-Collados, J., Rossi, R. G., Navigli, R., and Rezende, S. O. Knowledge-enhanced document embeddings for text classification. *Knowledge-Based Systems 163* (2019), 955–971.

[36] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks, 2014.

[37] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023.

[38] Tran, T. Q., Kang, M., and Kim, D. Rerankmatch: Semi-supervised learning with semantics-oriented similarity representation, 2021.

[39] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023.

[40] Wang, X., and Zhou, D. Chain-of-thought reasoning without prompting, 2024.

[41] Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. Finetuned language models are zero-shot learners, 2022.

[42] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. Chain-of-thought prompting elicits reasoning in large language models, 2023.

[43] Weizenbaum, J. Eliza—a computer program for the study of natural language communication between man and machine. *Commun. ACM 9*, 1 (jan 1966), 36–45.

[44] Wen, J., Zhong, R., Khan, A., Perez, E., Steinhardt, J., Huang, M., Bowman, S. R., He, H., and Feng, S. Language models learn to mislead humans via rlhf, 2024.

[45] Xu, X., Li, M., Tao, C., Shen, T., Cheng, R., Li, J., Xu, C., Tao, D., and Zhou, T. A survey on knowledge distillation of large language models, 2024.

[46] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023.

[47] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models, 2023.

[48] Zhang, Z., Zhang, A., Li, M., and Smola, A. Automatic chain of thought prompting in large language models, 2022.