Technical University of Košice
Faculty of Electrical Engineering and informatics

Automatic Generation and Optimization of Large Language Model Prompts II

Master thesis

2025 **Bc. Ihor Kasatkin**

Technical University of Košice
Faculty of Electrical Engineering and informatics

Automatic Generation and Optimization of Large Language Model Prompts II

Master thesis

Study Programme:	Informatika
Field of Study:	9.2.1. Informatika
Department:	Department of Computers and Informatics (DCI)
Supervisor:	Ing. Eugen Šlapak, PhD.
Consultant:	Bc. Vladyslav Moskalenko

Košice 2025 **Bc. Ihor Kasatkin**

Abstract in English

Every year, the demand for LLMs is growing significantly and is being actively integrated into every type of activity. The efficiency of these models largely depends on the quality and accuracy of input queries that direct the model to the desired result. Creating effective queries requires an investment of time and expertise in Prompt Engineering. Existing methods are limited to manual approaches and cannot automatically adapt to various tasks and contexts. Currently, no trivial and versatile solutions can fully automate and optimize context-specific queries for LLMs. This thesis proposes and implements an approach for automatic query generation and optimization for LLMs based on different optimization techniques and the availability of other models. Experimental results have shown that the developed methodology improves the quality and accuracy of the answers.

Keywords in English

prompt optimization, large language models, artificial intelligence, prompt engineering, attention mechanism

Abstract in Slovak

Dopyt po LLM každoročne výrazne rastie a aktívne sa začleňuje do všetkých druhov činností. Účinnosť týchto modelov do veľkej miery závisí od kvality a presnosti vstupných dopytov, ktoré model nasmerujú k požadovanému výsledku. Vytvorenie efektívnych dotazov si vyžaduje investíciu času a odborných znalostí v oblasti Prompt Engineering. Existujúce metódy sú obmedzené na manuálne prístupy a nedokážu sa automaticky prispôsobiť rôznym úlohám a kontextom. V súčasnosti neexistujú triviálne a univerzálne riešenia, ktoré by dokázali plne automatizovať a optimalizovať kontextovo špecifické dopyty pre LLM. V tejto práci sa navrhuje a implementuje prístup na automatické generovanie a optimalizáciu dotazov pre LLM na základe rôznych optimalizačných techník a dostupnosti iných modelov. Experimentálne výsledky ukázali, že vyvinutá metodika zlepšuje kvalitu a presnosť odpovedí.

Keywords in Slovak

optimalizácia dopytov, veľké jazykové modely, umelá inteligencia, návrh dopytov, mechanizmus pozornosti

Bibliographic Citation

KASATKIN, Ihor. *Automatic Generation and Optimization of Large Language Model Prompts II*. Košice: Technical University of Košice, Faculty of Electrical Engineering and informatics, 2025. 78s. Supervisor: Ing. Eugen Šlapak, PhD.

76159

TECHNICAL UNIVERSITY OF KOŠICE

FACULTY OF ELECTRICAL ENGINEERING AND INFORMATICS

Department of Computers and Informatics

D I P L O M A T H E S I S A S S I G N M E N T

Field of study: **Computer Science**

Study programme: **Informatics**

Thesis title:

Automatic Generation and Optimization of Large Language Model Prompts II

Automatické generovanie a optimalizácia dopytov pre veľké jazykové modely II

Student: **Bc. Ihor Kasatkin**

Supervisor: **Ing. Eugen Šlapak, PhD.**

Supervising department: **Department of Computers and Informatics**

Consultant: **Bc. Vladyslav Moskalenko**

Consultant`s affiliation: **Deutsche Telekom IT Solutions Slovakia**

Thesis preparation instructions:

1. Prepare an overview of current knowledge in the field of large language models (LLMs).
2. Analyze existing methods for automatic generation and optimization of prompts for LLMs and their effectiveness.
3. Implement a system capable of automatic generation and optimization of prompts for LLMs.
4. Quantitatively and qualitatively evaluate the effectiveness and accuracy of the generated and optimized prompts using appropriate metrics.
5. Develop system and user documentation containing a detailed description of the implemented solution and its usage.

Language of the thesis: English

Thesis submission deadline: 17.04.2025

Assigned on: 31.10.2024

.....

prof. Ing. Liberios Vokorokos, PhD.

Dean of the Faculty

Declaration

I hereby declare that this thesis is my own work and effort. Where other sources of information have been used, they have been acknowledged.

Košice, 22.5.2025 . . . . . .

Signature

Acknowledgement

I am very grateful to my parents for their constant support. I am also grateful to my supervisor for his invaluable help and constant guidance during the work on the thesis.

Contents

List of Figures

List of Tables

Motivation

LLMs capabilities are attracting growing interest from developers, researchers, and everyday people. Neural networks imitate human speech, support dialog, and allow text to be written on a given topic. However, as the range of capabilities and applications of intelligent systems expands, the scope for misuse of the results of their work increases. It becomes more challenging to realize that when a neural network generates an incomplete or inaccurate answer, more information can be made up without relation to reality.

The effectiveness of LLM depends entirely on the correct formulation of the input query. Most users have difficulty matching the proper query, which leads to more query iterations and unexpected results. When dealing with a large amount of data or limited computational power, an incorrectly formulated query will cause an unnecessary waste of computational resources.

Optimizing and generating updated queries will make it much easier for people to use LLM. The model will help with query generation to get an accurate and relevant answer the first time without unnecessary adjustments. Getting correct information from the models will become easy for the average user and reduce the model's response time.

Introduction

Everyone is familiar with the concept of "artificial intelligence" today. However, fewer people understand the structure of large language models as AI systems despite their active use in work and life. Their ability to understand and generate text in response to complex queries opens new perspectives for information retrieval and automated decision support. However, as the size and scope of these models increase, there is a need to generate queries as accurately as possible and optimize them to minimize resource consumption and improve response rates.
Query automation and optimization for LLM are driven by the requirements for accuracy and efficiency in processing large real-time data sets. Automating this process will significantly improve the interaction with the language model in an environment where manual query generation is time-consuming and error-prone. It will help generate customized queries, leading to a better understanding of the user's needs and developing more relevant and accurate results.
One of the main challenges is the lack of effective quality control methods, limited support in complex search tasks, and relevance of generated queries. When creating queries manually, users often face uncertainty in terminology selection and query construction, leading to unreliable results. In addition, LLM requires significant computational resources, and without proper query optimization, these costs would be prohibitive. Another important limitation of traditional methods is their inability to automatically adapt to model specifications, so a more versatile query generation method is needed.
The main hypothesis of this paper is that automated query generation and optimization for large language models will significantly improve their performance, increase the relevance of the answers, and reduce the time and computational cost of data processing. The optimized query generation system will be used to exploit LLM's full potential as a powerful decision support and data analysis tool.

Task formulation

This thesis aims to study and develop practical approaches to automatically generating and optimizing queries when working with LLM. The task is complex because it requires selecting and applying the most appropriate methods and techniques to improve the quality of interaction with the model, reduce the time needed to generate queries and achieve the most accurate and valuable answers.
One important task is to perform an in-depth review of current LLMs and existing techniques for generating and optimizing prompts to identify the key advantages and limitations of each approach.
The next step is to create a system capable of automatically generating and optimizing queries using modern techniques. Given the constraints on the number of accesses and the size of messages, the system should be designed to minimize the number of requests to the external APIs of the language models.
The penultimate step is developing mechanisms for objectively evaluating the quality and efficiency of the received queries. These mechanisms allow for comparing different methods of generation and optimization and monitoring their impact on the final result.
Finally, the last task is to prepare detailed documentation describing the principles of operation, structure, and use of the developed software to ensure ease of its further exploitation.

1 Analytical part

1.1 Large Language Models

At first, neural networks were a source of jokes: They drew people and animals with strange limbs and often "hallucinated" — gave silly answers to simple questions. But artificial intelligence learns quickly, and for several years, it has been used in business, science, production, and, of course, in the home. Neural networks can significantly save us time and effort when solving routine tasks. They do what can take us long minutes and hours in a couple of moments. They are already able to generate a wide variety of texts:

- Answer questions, write letters, and to-do lists.
- Write class and workout schedules, travel itinerary plans, and meal recipes.
- Write technical instructions and code in different programming languages.
- Write scripts, fairy tales, and stories.

A large language model is learned by analyzing the content of books, articles, instructions, dialogues, and web pages. By processing information, the model memorizes how sentences are constructed in language, what words are often used together, and how topics are related. Consequently, it learns how to answer questions and maintain dialogue. It is almost like a human, except that the model does not understand the meaning of words as we do. If you ask it a question, it will answer similarly to the many texts it is familiar with. If the answer is not suitable in content, form, or depth, the model can be trained using additional data and setting parameters for its answer.

1.1.1 Development history of the LLM

In early 1966, the first chatbot, ELIZA, was created by a human, Joseph Weitzen-Baum, at the Massachusetts Institute of Technology. ELIZA was a pioneering experiment that provided human-computer interactions. ELIZA did not follow the

context of a conversation and could not find relevant answers, but could create the illusion of a conversation using pattern matching and substitution methodology. At that time, many variations of chatbots were just being made. For example, DOCTOR was created with the likeness of a therapist who could mimic the responses of a Rogerian psychotherapist. The therapist could review the questions and return them to the patient as clarifications. As seen in Figure 1.1, these chatbots were just a bright beginning in the field of chatbots and natural language processing [1].

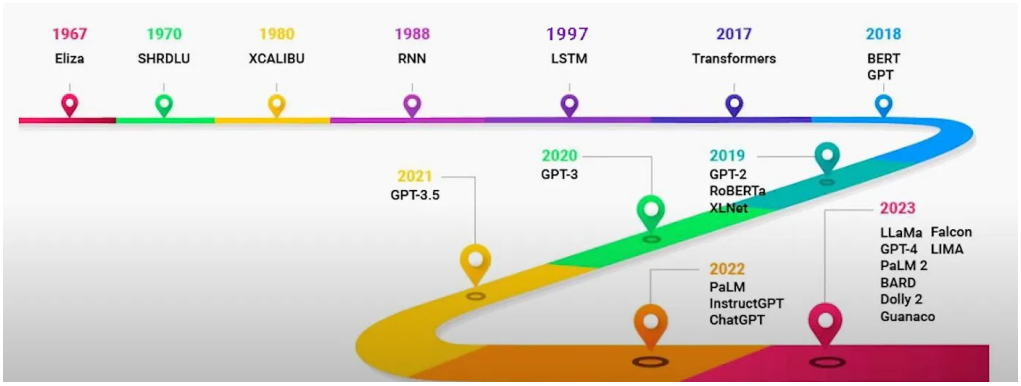

Figure 1.1: LLM timeline [1]

The heyday and growth of recurrent neural networks (1980-2014)

At the end of the 20th century, the first neural networks that were inspired by the human brain and its interconnected neurons emerged. Recurrent neural networks appeared in 1986 and became one of the most popular networks worldwide. The RNN had a unidirectional information flow; it stored past inputs in its internal memory and could answer questions based on context. The network was trained to process and convert the received input data into specific sequential output data that had feedback, but despite this, it had limitations with memory, for example, the perception of long sentences.

In 1997, Long-Short-Term Memory was introduced. Its main advantage was storing information in long sequences, which covered the limitations of RNN short-term memory. LSTM's unique architecture had input, output, and forget valves. These valves determined how much information would be stored, discarded, or produced at each step. This ability to selectively remember and forget helped the LSTM retain the necessary information in memory and effectively capture long-term dependencies in sentences.

Recurrent networks with gates appeared in 2014. They were designed to solve the same problems as LSTMs but with a more straightforward and optimized

structure. GRUs are designed to solve vanishing gradient problems and preserve long-term expression dependencies. GRUs simplified migration using an update gateway and a reset gateway. The update gateway was responsible for maintaining the right amount of past information and considering new information. The reset gateway determined how much of the previous information was worth forgetting. These gateways made GRUs more computationally efficient.

Attention Enhancement Mechanism (2014)

RNN and RNN-based variants of LSTM and GRU partially maintained context even when it was removed. As seen in Figure 1.2, this problem was solved using the concept of attention. The introduction of this mechanism led to a major paradigm shift in sequence modeling and opened a new perspective compared to previous architectures. Since RNNs treat sentences as fixed-size context vectors, they attempt to fit all the original information into a single fixed-length vector despite the length of the entire sentence, causing performance to degrade as sentence length increases. In contrast, attention allows the model to dynamically look through the whole source sequence and select different fragments based on relevance at each inference stage. This will ensure that no critical information is lost or replaced.

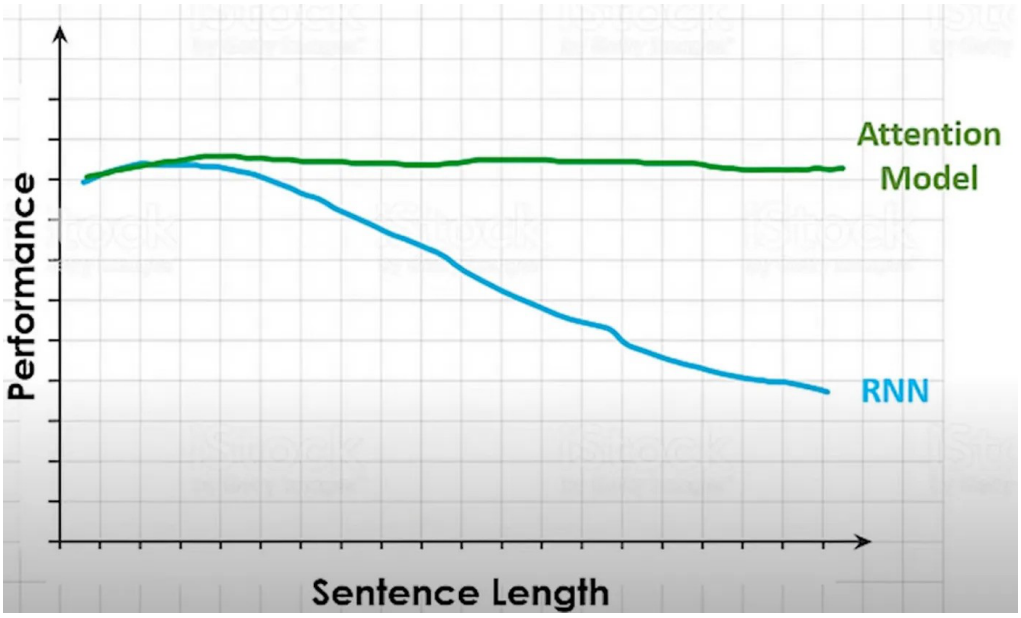

Figure 1.2: Performance comparison [1]

In 2017, the Transformers architecture emerged thanks to the report "Attention is All You Need" by Vaswani and his colleagues on the Google team [2]. This new type of architecture relied on attention mechanisms to process sequences. It

consisted of encoders and decoders, each containing multiple layers of intrinsic attention and feed-forward neural networks.

"Multi-headed" attention was the main distinguishing feature, which made it possible to simultaneously focus on different parts of the input sentence and capture nuances from the context. Another advantage was processing sequences in parallel rather than sequentially. With these advantages, transformers laid the foundation for subsequent models such as BERT, and GPT and ushered in a new era of LLM.

After the success of Transformers, the next logical step was scaling up. Google's BERT model was the pioneering model that was released in 2018. Other models processed text from left to right or vice versa, so BERT was designed to account for both directions simultaneously, hence the name BERT. This model was trained on huge amounts of text. Thus, BERT was the first proper basic language model that could be fine-tuned for specific tasks. After large companies started to release their models, such as GPT-3 [3], GPT-4 [4], BERT [5], PaLM [6], and LLaMA [7]. As can be seen in Figure 1.3, LLMs began to perform a huge number of tasks, marking a paradigm shift in AI capabilities.

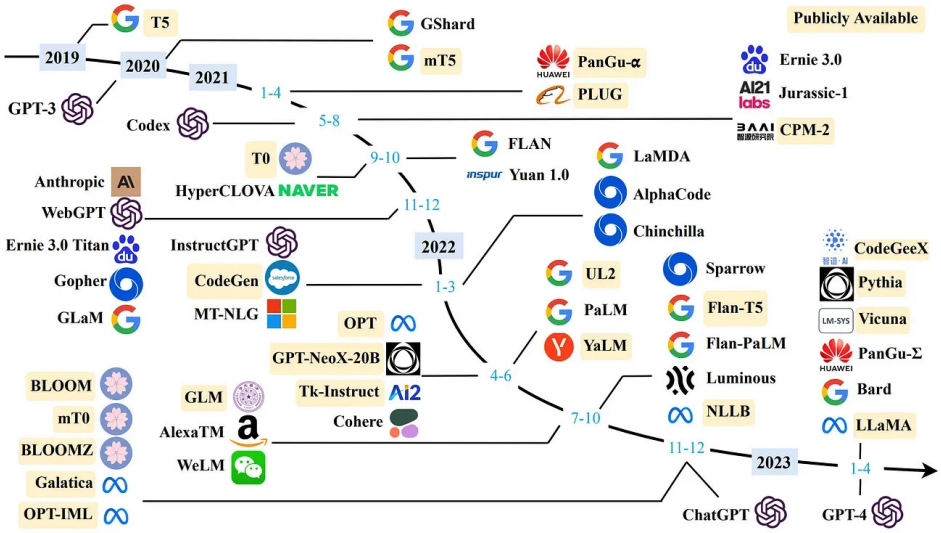

Figure 1.3: Timeline of nowadays LLM [1]

1.1.2 Architecture of LLMs

The basis for most modern language models is the Transformer architecture. This architecture was proposed in 2017 in the article "Attention is All You Need" and has since become the standard for creating models that work with text.

The Transformer architecture consists of two key components, as illustrated in Figure 1.4.

- Encoder: converts the input text into a hidden representation.
- Decoder: generates output text based on the hidden representation.

A key element of Transformers is the attention mechanism, which allows the model to focus on different parts of the input sequence while processing each aspect. Unlike previous approaches, such as recurrent neural networks, which processed sequences step by step, transformers can process all sequence elements simultaneously.

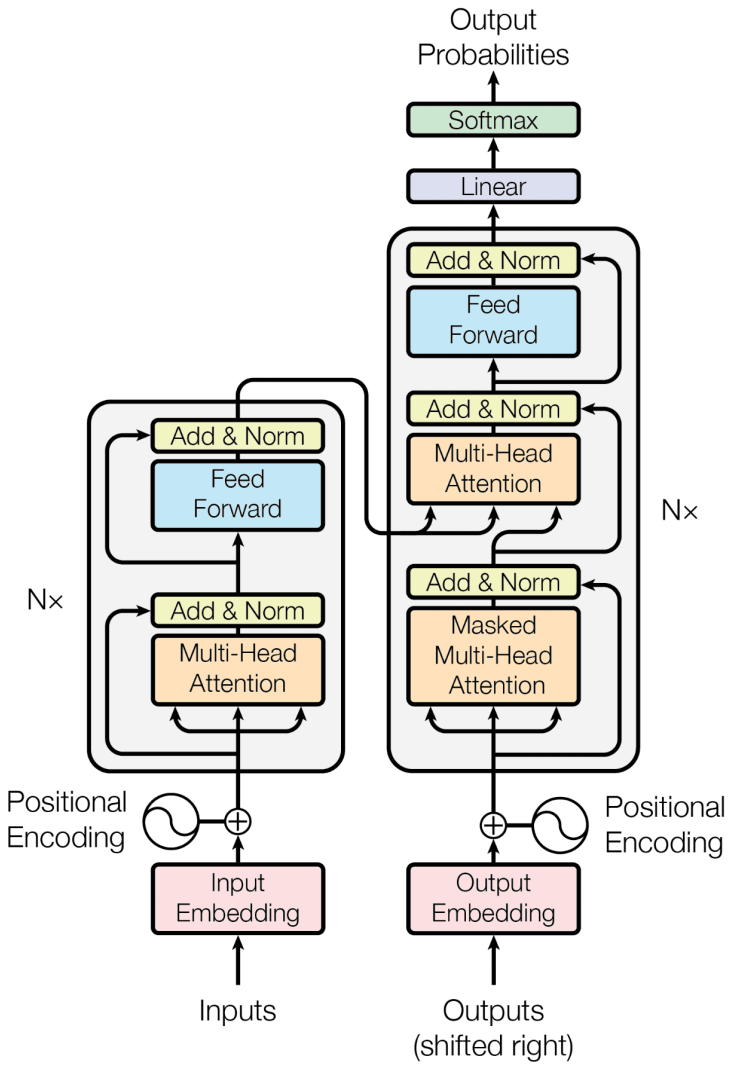

Figure 1.4: Transformer model architecture [2]

Encoder

The encoder in the Transformer architecture is the component responsible for converting input data sequences into a fixed representation. As seen in Figure 1.5, it takes a sequence of tokens and converts them into contextual representations that the decoder or other model layers can use. The transformer architecture consists of multilayer blocks, including attention mechanisms and Feed-Forward layers. The number of transformer layers can vary depending on the particular model implementation, but the original work used 6.

- Positional encoding
- Attention
- Multi-headed attention
- Normalization layer
- Feed-Forward Layers
- Residual link layers

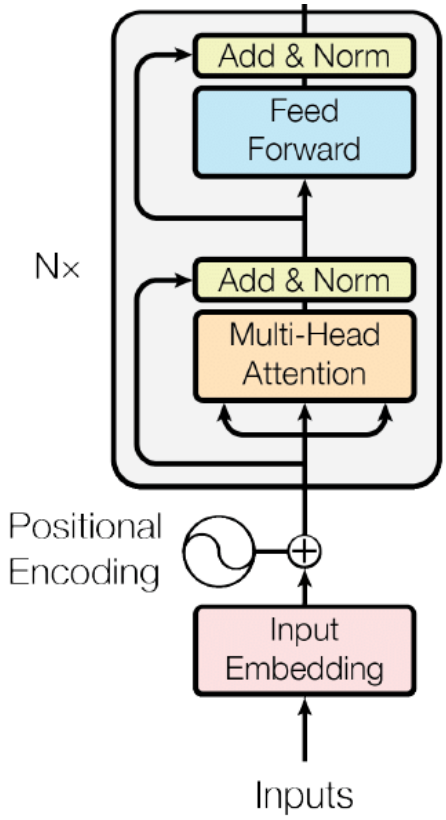

Figure 1.5: Encoder

The Encoder block in the Transformer updates the token embeddings in each layer. Each layer processes the embeddings in parallel, except for the attention layer, which is independent of the other layers. The encoder can process variable-length sequences, and the layers can be swapped without affecting the result. These components work together to provide efficient processing and presentation of input information.

Positional encoding

One of the features of transformers is the absence of recurrence and convolution layers, which have traditionally been used to process sequences in other models such as RNNs and CNNs. Positional encoding is a unique mechanism that considers the order of elements in a sequence, which helps transformers process all elements of the input sequence in parallel. Position encoding introduces information about the position of each token in the sequence, allowing the model to understand in which order the tokens are placed. There are many methods of positional encoding, but the classical implementation of the Transformer uses sinusoidal positional encoding [8]. Sinusoidal positional encoding is computed using sine and cosine for each position and vector dimension. The formulas for calculating the encoding for each position and each dimension are as follows:

$$PE(\text{pos}, 2d) = \sin\left(\frac{\text{pos}}{10000^{\frac{2d}{d_{\text{model}}}}}\right)$$

$$PE(\text{pos}, 2d+1) = \cos\left(\frac{\text{pos}}{10000^{\frac{2d}{d_{\text{model}}}}}\right)$$

where:

- pos is index of position in the sequence.
- d is the dimensionality index of the positional encoding vector.
- d_{model} is the vector's embedding dimension or length to represent each token.

Two types of functions are used here: sine and cosine. This allows each position element to have a unique representation. Position information is encoded by vectors, one for each absolute position in the context being processed. The position vector is added to the sense vector of the token at the model's input, and then the result is fed into the main part of the neural network. For example, if we have an embedding for word x_i, the final input vector for word x_i will look like this:

$$x'_{\text{i}} = x_{\text{i}} + PE(\text{i})$$

where $PE(i)$ is the positional encoding for position i, and x_i is the embedding for the word.

After adding the positional encoding, the model is informed about the position of each word in the sequence, which allows it to take the order into account when performing attention operations and other transformations.

Attention or self-attention

The attention function can be described as matching a query and a set of key-value pairs with the output data, where the query, keys, values, and output data are vectors. The result is computed as a weighted sum of the values, where the compatibility function of the query and the corresponding key computes the weight assigned to each value. Attention formula: For given queries Q, keys K, and values V, attention is calculated using the following formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Where

- Q: vector representing the current element for which attention is computed.
- K: vectors representing the elements of the input sequence.
- V: vectors that should result in the output. These are usually the same elements as the keys but may be different.
- d_k: the dimensionality of the key vectors used to scale the values before applying the softmax function.

The scalar product between Q and K is calculated for queries, keys, and values, then the results are scaled, and softmax is applied to obtain weights. These weights are used to weight the values of V, yielding a final vector representation. The attention mechanism allows the model to focus on different parts of the input sequence during processing. It is used to compute weights that indicate the importance of each element of the input relative to the others. The basic idea is that not all words in a sentence have the same importance for understanding the context.

Multi-headed attention

Multi-headed attention is an extension of the attention mechanism that allows a model to perceive information from different subspaces of representations at

different positions simultaneously. It divides the attention space into multiple "heads", each trained to detect different aspects of information. Multi-headed attention is represented as a combination of several attention mechanisms:

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O$$

where each $head_i$ is calculated by the formula:

$$\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

- i is the number of heads in multi-head attention.
- QW_i^Q,KW_i^K,VW_i^V - trained weight matrices for each head, which are used to transform the original queries, keys, and values.
- Concat - a concatenation operation that combines the outputs of all heads into a single vector representation.
- W^o is a trained weight matrix that linearly transforms the result of concatenation, allowing the model to integrate information from different heads.

All attention heads have different weights. The results of all heads are concatenated and run through a linear transformation, allowing the model to extract different aspects of information simultaneously.

Normalization Layer

Transformers use normalization among other layers [9]. Normalization helps to stabilize model training and improve model performance. Normalization layers are applied after attention operations and feed-forward layers to minimize the effects of bias and fluctuations in values occurring at each step. The primary purpose of the normalization layer is to standardize the input data by bringing it to zero mean and variance to unity. This will help avoid the problem of damped or exploding gradient and improve the overall quality of training. The normalization layer for the input vector x is computed as follows:

1. Calculate the mean value and standard deviation:
   We need to distribute the values of the input vector x around zero. We must first calculate the mean value (μ) and standard deviation (σ) to do this. Mean is simply the sum of all values divided by the number of values:

$$\mu = \frac{x_1 + x_2 + \cdots + x_H}{H}$$

where H is the number of elements in vector x.
Standard deviation measures how much the values deviate from the mean. We calculate it by finding the square root of the mean of the squares of the deviations from the mean:

$$\sigma = \sqrt{\frac{(x_1 - \mu)^2 + (x_2 - \mu)^2 + \cdots + (x_H - \mu)^2}{H}} + \epsilon$$

where (ϵ) is a small constant to prevent division by zero.

2. Standardization:
Now, when we have the mean and standard deviation, we can standardize each value in the vector. The standardized value $\hat{x}$ is calculated as follows:

$$\hat{x} = \frac{x - \mu}{\sigma}$$

This transformation assigns each value a new scale, so the mean would be about zero and the standard deviation about 1.

3. Scaling and offsetting:
After standardization, we want to return the data to the original scale so that the model can use it better. To do this, two trained parameters are applied: scaling (γ), and bias (β). This is done using the following formula:

$$y = \gamma\hat{x} + \beta$$

where (γ) allows to rescale and (β) allows to add an offset.
The normalization layer improves the quality of training and increases the robustness of the model. It standardizes the input data at each processing step, which contributes to more stable and efficient training.

Feed-Forward Layers

The feed-forward layer processes each element of the previous layer by performing matrix multiplication of these elements with their weights. The resulting data is then sent to the next layer. The neurons of the feed-forward layer are connected to all neurons of the previous layer, which means that each neuron of the feed-forward layer can interact with any neuron of the last layer. The feed-forward layer operates independently of the other tokens in the sequence, applying the same transformation to each[10]. A feed-forward layer in a transformer consists of two linear layers with an activation function between them that transforms input data into new representations:

1. First linear transformation
   The input vector x is passed through the first linear layer, a matrix multiplication. This multiplication scales and transforms the dimensionality of the vector to extract meaningful features.

$$\mathrm{h}_1 = xW_1 + b_1$$

   where W_1 is the trained weight matrix and b_1 is the bias vector
   After the first linear layer, an activation function Rectified Linear Unit or Gaussian Error Linear Unit is used. Activation adds nonlinearity, which allows the model to highlight more complex patterns and relationships.
   ReLU is a nonlinear activation function widely used in deep learning. It converts an input value to a value between 0 and positive infinity. If the input value is less than or equal to zero, ReLU outputs zero; otherwise, it outputs the input value.

$$\text{ReLU} = max(0, x)$$

   where x is the input value.
   GELU is a more complex activation function that was introduced as an improvement to ReLU. GELU uses a Gaussian distribution and approximation to the error function, making it smooth and differentiable everywhere. This allows for efficient handling of more complex dependencies in the data. The following formula specifies it:

$$\text{GELU}(x) = 0.5x \left(1 + \tanh\left(\sqrt{\frac{2}{\pi}} \cdot \left(x + 0.044715x^3\right)\right)\right)$$

   The choice between ReLU and GELU depends on the specific problem and model. ReLU is a simple and fast solution for most cases, while GELU may be preferred for complex issues.

2. A second linear transformation:
   After applying the activation function, it passes through the second linear layer where multiplication by the trained weight matrix W_2 is again performed and bias b_2 is added. The final process for the two activation functions is as follows:

$$y = \text{FFN}(x) = \text{ReLU/GELU}(xW_1 + b_1)W_2 + b_2$$

   A feed-forward layer is used to transform the input data, adding nonlinearity and improving the model's ability to extract more complex patterns.

The choice between ReLU and GELU activation functions affects learning, where ReLU is faster to learn, and GELU often leads to softer and more stable results.

Residual Connections

Residual connections are used to facilitate the training of deep networks by providing a gradient flow through the network. The input signal of a layer is added to the output signal of that layer before being passed to the next layer, allowing the model to learn residual functions more efficiently. This allows information to flow through the network without distortion, even if the model consists of multiple layers. For each layer of the transformer, the residual-coupled output is computed as follows [11]:

$$y = \text{Layer}(x) + x$$

Where:

- Layer(x) is the output of another layer applied to the input x.
- X is the input vector that is added to the output layer.
- y is the final output with residual connectivity.

Residual connections speed up learning, make it more stable, and ensure that critical information is retained, thus improving model performance.

Decoder

After describing the internal processes occurring in the encoder of transformer architectures, we need to move on to analyzing the next component of transformers - the decoder. Comparing these two transformer components will help highlight their main similarities and differences. As seen in Figure 1.6, the decoder consists of roughly the same layers as the encoder and works similarly. The decoder unit takes in a capacitated input and produces the same capacitated outputs as the encoder. The attention mechanism is the central component in both cases. In particular, this mechanism is applied in two places in the decoder. In both these places, the attention mechanism undergoes necessary modifications that distinguish it from the simpler version of such a mechanism in the encoder. The key difference is between the masked multiple attention mechanism and the multiple attention mechanism of the decoder-encoder.

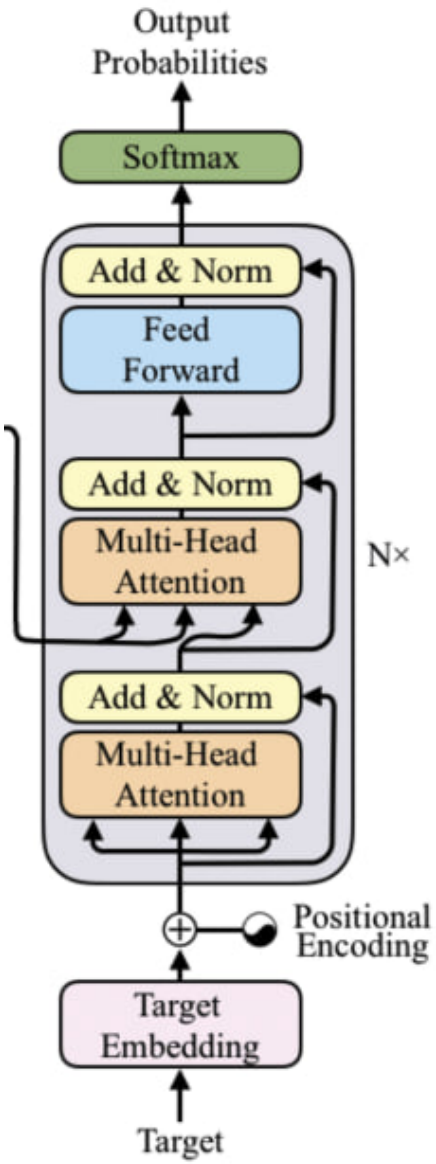

Figure 1.6: Decoder [2]

Masked attention

Masked attention is widely used in autoregressive models, where the next token is generated based on the previous ones as the input decoder receives words from the target sentence that need to be translated. These words are converted into numerical vectors, and additional vectors are added to them to show their positions in the sentence so that the order of words in the model is taken into account. The vectors then undergo a self-attention block, like in the encoder, to clarify the contextual meaning of the vectors. Masked attention restricts the model from accessing information about future tokens to generate text incrementally and learn under conditions approximating real-world generation. The masked attention mechanism modifies the attention matrix to exclude the influence of to-

kens that follow the current token [5]. This is done by using a mask that prohibits the model from paying attention to future elements when computing attention. In masked attention, a mask is added at the attention matrix calculation stage, which makes it impossible to access future tokens, which is represented in the formula:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V$$

Where:
Q, K, V, d_k values are similar to regular attention,
M - the elements of the mask M can take the values zero or $-\infty$. The value $-\infty$ is used for those positions that need to be masked. After applying softmax, masked positions are given a weight close to zero, effectively excluding them from further computation. Further, the self-attention result is processed through the residual connection mechanism, and then the normalization layer is applied.

Multiple attention of decoder-encoder

Now, let's talk about applying the decoder-encoder multiple attention mechanism in the decoder. The main difference of this mechanism is that vector Q is the output vector of the masked multiple attention layer, while vectors K and V come from the encoder. The user has no control over the input data of the multiple attention layers with masking. This data comprises vectors previously generated by the decoder, starting with the unique token $< START >$ and ending with another special token indicating the end of the sentence $< EOS >$. There's a problem here: if this process functions independently, it will have no information from the user, making translating from one language to another impossible. Therefore, the multiple attention layer of the decoder-encoder connects embeddings K and V from the encoder, which carries the source information to be processed, and vectors Q, which are generated by the decoder. As in self-attention, encoder-decoder multiple attention uses multiple attention "heads".

$$\text{MultiHeadAttention}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \ldots, \text{head}_h)W^O$$

Where:
Q - decoder queries, K - encoder keys, V - encoder values
Once attention is computed, the outputs of all heads are concatenated and passed through the feed-forward layer, creating a final representation that is used in further decoder layers to generate the next token. Multiple encoder-decoder attention provides an efficient link between the input and output sequences, allowing

the decoder to utilize context from the input data to generate accurate and contextually relevant outputs. Next, bandwidth communication and normalization can be used, after which a feed-forward neural network is applied to the updated token representations.

The Transformer architecture revolutionized natural language processing with its core components, the encoder, and decoder, which are fundamental factors in text comprehension and generation, allowing models to process context and generate coherent responses efficiently. However, current LLMs show a diversity of approaches: some use only encoder or decoder. For example, BERT-type models are encoder-only based and designed for text comprehension tasks, whereas GPT-type models use only a decoder and focus on text generation. This emphasizes the Transformers architecture's flexibility, allowing models to be tailored to specific tasks. Thus, Transformers are still a fundamental basis for developing modern language models, striking a balance between complexity and efficiency.

1.1.3 Multimodality in Large Language Models

The development of neural networks is like a layer cake: first, language models were developed to work with text, and then, layer by layer, other modalities, such as pictures, video, and audio, were added. Multimodality that LLMs can do more than understand and generate text [12]. As seen in Figure 1.7, multimodal models started to take the market by storm at the beginning of 2023, but despite this, multimodality was also developing at a much slower pace. Mainly, simple models were released that could generate captions to pictures but could not conduct a full-fledged dialog.

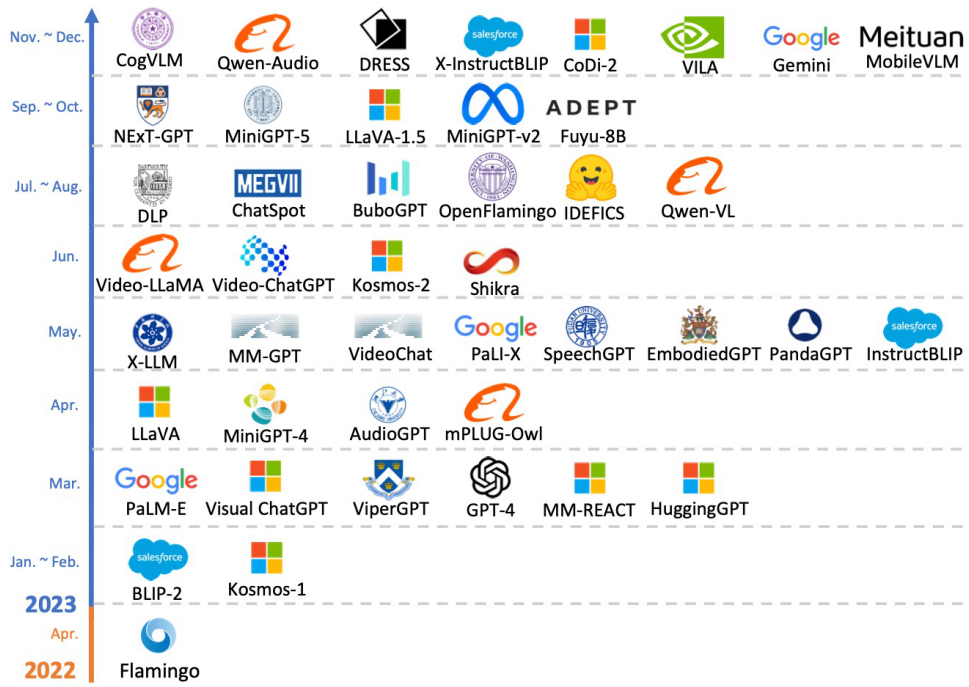

Figure 1.7: Models coming out from early 2023

LLMs were originally trained on large corpora of textual data. Extending them to multimodality, developers incorporate visual or audio components. Using specialized training methods - often including contrastive learning or layers of cross-attention - the model aligns different modalities in a common latent space, which allows the models to understand that certain images correspond to specific textual descriptions. Structurally, multimodal LLMs typically consist of five components:

1. Modality encoders
2. Input projectors to align representations of different modalities
3. A framework in the form of an LLM for semantic understanding and reasoning
4. Output data projectors for content generation instructions
5. Content generators of different modalities

Several reasons explain why adding new modalities to language models was necessary.
The first reason is that texts have proven to be an excellent data source for learning. The second is that in chat rooms, people communicate with text and using

pictures, audio, video, and memes. If we want AI Assistants who can understand people and respond to them in any convenient format, we need to train the model to work with text and all the types of data that people are used to.
Another reason is that multimodal data contains more knowledge about the world and can produce more meaningful responses. Data from one modality complements and enriches data from another: for example, the concept of "person" for such a neural network is not just a word but also voice, clothes, appearance, and manner of movement.
A neural network trained on different data types can answer questions about colors, geometry, music, and memes. It can also generate these pictures, memes, and colors as an answer. Every year, the creativity of neural networks becomes more and more qualitative.

1.2 Prompts

Let's think about what a prompt is. Each person gives instructions and asks questions all the time in their daily lives. A prompt is essentially just words. They are the instructions and context that humans provide for LLMs to make them accomplish a task [13]. People need to know how to design and optimize their prompts to get high-quality and accurate results. But creating a good prompt is more than just selecting a few words. It is about understanding the task's purpose and context, LLM capabilities, and the science of prompt engineering. Prompts are fundamental to LLMs because they shape the behavior of these models. By changing a prompt's structure, wording, or context, users can guide LLMs to perform various tasks, which is the hallmark of modern NLP systems, which rely heavily on accurate and optimized prompts to reach their full potential. Even the most experienced AI scientists don't know precisely how an LLM does what it does. By playing with things and testing, anyone can discover a new prompting technique to produce statistically better results [14]. Prompt engineering is part of that. A slight prompt improvement won't make the output better because the relationship is not linear.
Prompt engineering is an interdisciplinary branch focused on interacting with AI by integrating fields such as software engineering, machine learning, and cognitive science like psychology, business, philosophy, and computer science. Well, or more simply, it's a natural language programming and the invaluable practice of developing and optimizing prompts for practical use in artificial intelligence for any task. Prompt engineering will continue to evolve in this AI and machine

learning era, although it may not become a separate career. As AI becomes ubiquitous, workers with operational engineering skills will outperform their peers, making it an essential tool for personal and professional growth.

1.2.1 The main components of the prompts

Prompts are a user's textual request to the neural network, but if we dive into technical details, the models do not predict a specific word but a token. Tokens are the way LLMs understand and speak, just like humans use words. Every word is not a token. The entire sentence is broken down into tokens, and the model works with their sequence. Tokens often include a space before a word or can be multiple spaces [15]. One token equals about 0.75 words, a rough number that will differ for each model. Each token has its ID. The token IDs will be different for uppercase and lowercase letters.
When constructing prompts for LLMs, it is necessary to understand their main components, as they form the basis for the model's ability to interpret and generate meaningful responses. As seen in Figure 1.8, three most essential components of prompts are system message, context, and persona with roles.

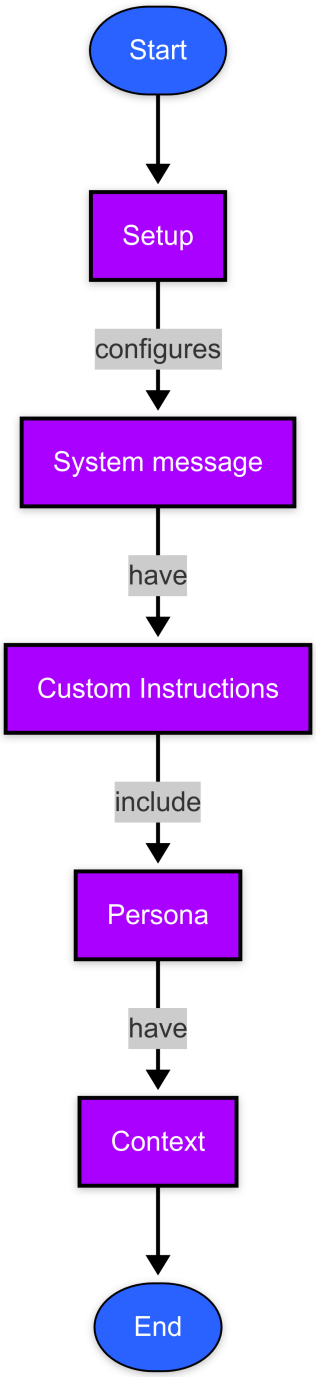

Figure 1.8: Main components of the prompt

System message

The system message is called a meta-tip or system prompt, the default or initial prompt provided to the model by its creator. It sets the basis for the model's interaction with the user by including instructions or a specific context that shapes the interaction with the model, tone, and response [16]. It is the guiding light of the model. For example, every time a new chat starts with a GPT chatbot, it is given

a prompt. The first prompt it received was: "You are ChatGPT, a large language model trained by OpenAI based on the GPT-4 architecture". The primary focus of the prompt design is on the user message, which is the most challenging part to master. When interacting directly with the model through code, the input is labeled with roles: system, user, and helper:

- A system is a system message that sets the overall context for the model's operation.
- The user is a person interacting with the system.
- Helper is a model that responds to the user's requests.

Chatbots usually do not display system messages, but it is possible to see them if you ask for them. However, when interacting directly with a model, you can create your system message so that it prepares the model and sets guidelines for how to respond to user messages. You can use the system message for good by telling the model to be helpful, friendly, and unbiased, or you can use this power for evil by telling the model to be rude or, even worse, telling it to have a particular bias and sometimes lie about certain things. Responsibly understanding and using system messages ensures that AI models fit their intended purposes while maintaining ethical integrity.

Context

Context is one of the most essential parts of prompt development, but the truth is that too much context can be ineffective or even negatively affect prompts. This is because models have different limits on the amount of tokens, which is also context.
The token limit refers to the maximum number of tokens a model can handle during a conversation. Every time a user sends a request, their entire conversation history is combined and packaged into the request being sent. The model constantly reminds the user of the whole conversation each time a new request has been sent since the entire context is not memorized. Previous messages with the model also provide context. Once the token limit is reached, the conversation does not end. The model does not interrupt the user; instead the system deletes tokens to make space for new tokens. It is necessary to be careful about the token limit because it defines the context window and dramatically affects the accuracy of results and consistency of the model output.

Additional context should be included in prompt engineering. The system message affects the model by giving it more context. It is similar to talking to a person; if more information is given, then more specifics will be provided for the model, and its response will be more helpful. More useful context means more words and more words means more tokens. More tokens for analyzing change the computations that all the parameters and neurons perform. This forces the attention mechanism to pay more attention to relevant information and less to less relevant information.
The model's performance will be highest when the relevant information appears at the beginning or the end of its context window [17]. This is due to two cognitive effects: the primacy effect, which makes information at the beginning better remembered, and the recency effect, which makes information at the end more memorable. However, as the context length increases, the model's performance decreases because extracting and utilizing relevant information from extended contexts becomes more difficult. Therefore, it is not so much the amount of context provided that is important but rather its proper placement and timeliness.

Personas and roles

Personas can be extremely useful if used correctly. Persona and role refer to managing the interaction with the model and communication context. If a model is needed to help with a coding question, a person might start their request by saying: "You are a senior programmer", or if needed help with a legal question, a person might say: "You are a legal expert". What has happened is simply that the model has been assigned a persona/role, which means giving it additional context. This extra context helps her understand the problem and make more accurate determinations about the most statistically likely sequence of words.

Likewise, personas can have their style, tone, and voice. When constructing a prompt, it is always best to provide the model with a persona appropriate for solving the task. Users direct the model to solve problems in a particular domain by assigning personas to specific tasks. While longer descriptions may offer more context, simple personas are often sufficient and save tokens since they are a finite resource. Personas enriched with particular tones, styles, and voices transform LLMs into more intuitive and appealing tools for humans. In addition to providing additional context, personas enhance the usability and adaptability of models by aligning them with creative, professional, and business applications.

1.2.2 Methods for structuring a prompt

Techniques for structuring prompts are necessary to improve the efficiency of interaction with LLMs. These techniques focus on organizing and presenting input data to guide the model for accurate and relevant answer generation. Structuring ensures that the model can interpret the intent behind the query with minimal ambiguity and produce results that match the user's expectations. As seen in Figure 1.9, structuring techniques include delimiters to define boundaries in prompts, I/O formatting for structured results, and example-based prompts.

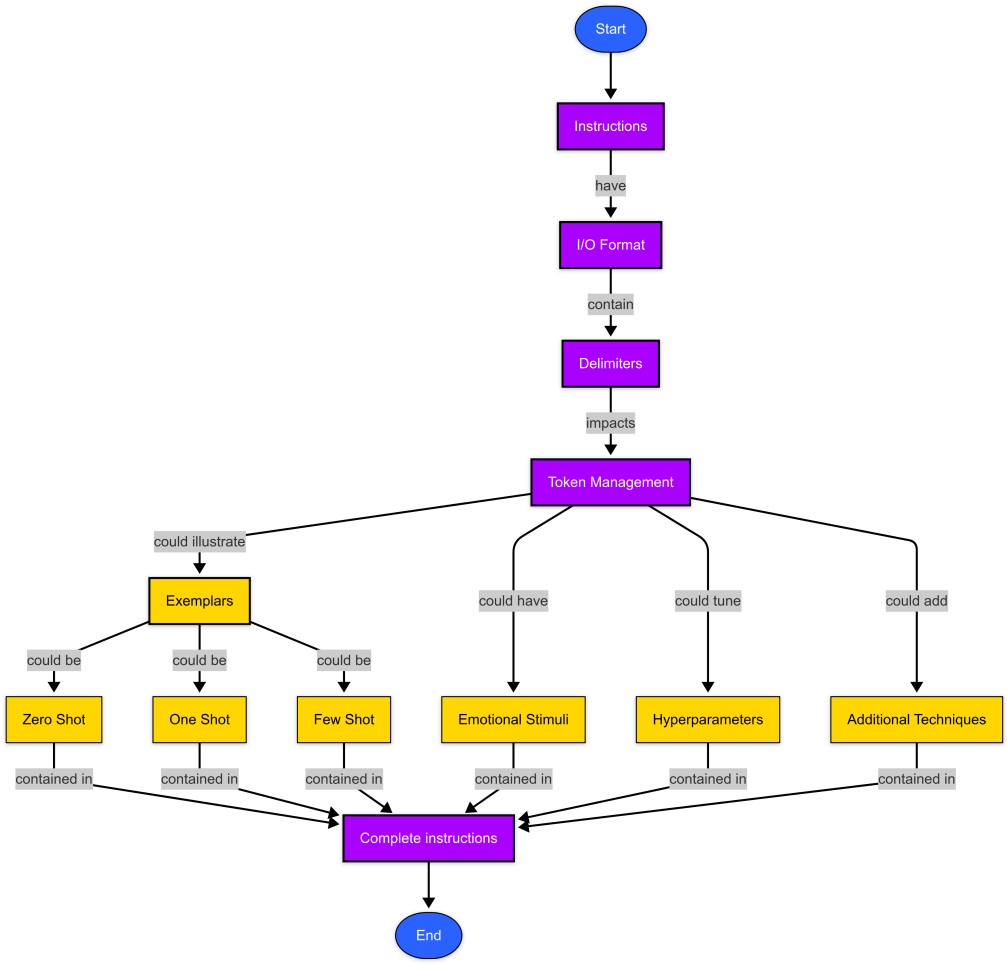

Figure 1.9: Possible additional context

Delimiters are a sequence of one or more characters that define a boundary between separate areas of text. We need to think of delimiters in prompts as the same thing that defines the structure of the prompt so that the model can better analyze and understand it. The following can be used as delimiters: equal sign, backquotes, bars, stars, HTML-like syntax, and XML syntax. Delimiters tell the model that the text is different or unique from other text. Delimiters can be used

to create patterns in a query that the model can recognize and provide structure to the query, but it is essential to remember that delimiters offer the model an extra little context for understanding what follows next.
Controlling the output data format is also an integral part of every prompt. While text is the default format, it is also possible to get output from models in CSV, code, numeric, image, file, table, or JSON. JSON is ubiquitous. When using AI applications, people often want the output to be in JSON format because it's a machine-readable format, that can easily analyze and extract specific information.
Example-based prompts enhance the efficiency of interaction with models by allowing users to provide examples in the prompt. This method helps guide model responses by demonstrating the desired structure, tone, or content. Depending on the complexity and requirements of the task, users can utilize different levels of example-based prompting, such as zero-shot, one-shot, or few-shot approaches.
Zero-shot prompting refers to the model's ability to complete a task without explicitly training that task, which means instructing the model to do something without additional training in context [18]. This is powerful but not always perfect. The model may make errors or produce output that is inaccurate and irrelevant. It is best used for relatively simple tasks rather than complex ones.
Sometimes, explaining things by giving examples rather than presenting them in words is more effortless. One-shot and few-shot prompts mean that one/few examples are added to the query to illustrate the task. This helps the model understand the output format, logic, and context. The difference is simply the number of examples presented in the model. So, at zero shot, we had zero examples. At a one-shot prompt, there will be one example. In a few, there will be two or more examples given. Using few-shot examples in prompts could force the model to do exactly what we want because they make the model's response more accurate and useful for any purpose. Accuracy and trueness increase when a one-shot is used correctly and even more for few-shot examples [3].
Using between 4 and 8 examples is ideal for few-shot prompting because performance increases with the number of examples but stabilizes after eight [19]. Adding more than eight examples does not significantly improve performance and is inefficient. So, the golden mean for effective prompting lies in this range. When given random or incorrect example labels, LLMs retain high performance, slightly affected by inaccuracies. This demonstrates the remarkable reliability of how LLMs interpret and generalize input examples.

1.2.3 Prompt Tuning Elements

After establishing the basic methods for structuring prompts, we can move on to the next step in improving their effectiveness: prompt optimization methods. While structural methods focus on clarity and organization, optimization methods aim to fine-tune prompts for maximum performance by adapting them to the capabilities and behavior of LLMs. Prompt optimization methods cover four main pillars: prompt variables, hyperparameter tuning, auto-priming, and emotional stimulation.

Prompt variables

Prompting variables are dynamic/static elements in prompts used in AI applications. The basic idea is to create fields that can be replaced with the right words or data to adapt the prompt to a specific task. Static elements are fixed and offer the same context or question each time. Dynamic elements adapt based on user input or program data, which adds flexibility and reusability to prompts.

Customizing Hyperparameters

Hyperparameters are configurable parameters that affect the behavior and output of the model. Model results can be further improved by carefully tuning hyperparameter configurations such as max-tokens, temperature, top P, stop sequences, frequency, and presence penalties.

1. The model may give a longer or shorter answer if length is not explicitly specified. Regarding output length, it is best to avoid using numbers or accept that numbers will not be followed. Users cannot limit the number of words, but there is a way to limit the number of tokens the model outputs using the max tokens hyperparameter. This parameter sets the maximum number of tokens that can be generated. If the maximum limit of tokens has been reached, the response will be immediately terminated, and the model will no longer send tokens after that.

2. The temperature controls the randomness of the output, which means the randomness of selected tokens. It can range from 0 to 2. The standard model temperature is 1, where no probability adjustment is made. When the temperature is less than 1, the model will favor the most likely tokens, while when the temperature is greater than 1, the model will select less likely to-

kens. By understanding the impact of randomness, the user can fine-tune the temperature to fit their needs.

3. Top p is a probabilistic decoding method that dynamically constrains the set of candidate tokens during text generation. When top P is set to 1, it considers all potential tokens. If the value is reduced to 0.5, one value with 50% of the top tokens will be selected, resulting in less unpredictability. Top P is a bit like temperature because they are related. Top P sets the scope of tokens that can be selected, and temperature controls the random selection of tokens in the scope.

4. Frequency and Presence Penalties affect the novelty and repetitiveness of the generated text. These hyperparameters prevent models from getting stuck in a word cycle where they keep repeating the same token repeatedly. If a model wants to keep repeating that token at some point, it will be penalized such that the probability of this token becomes lower, forcing it to choose a different token and exiting its repetitive word cycle. The presence penalty penalizes tokens that have appeared at least once, ensuring diversity from the start.

5. Stop sequences are predefined tokens that signal models to stop generating further text. They act as termination markers in decoding, providing security by preventing unwanted output or access to certain information.

Auto-Priming

Auto-priming is one way of forcing the model to create a context for the user. This method takes advantage of the autoregressive nature of LLM, where each token generated affects subsequent tokens. Auto-priming is the process of automatically embedding context or instructions into the model's input at each stage of a conversation or workflow. It can be used as the beginning of a dialog or in the system message of a LLM. As seen in Figure 1.10, auto-priming works in stages. Initially, the system analyzes input data to understand what the user needs. After that, it automatically generates or selects ready-made context-sensitive prompts. The newly created prompts are entered into the model's input data, with the possibility of supplementing them with instructions, constraints, or examples. As a result, the generated answer is evaluated for compliance with the task.
Auto-priming involves creating an internal structure before generating the main output. This structure includes identifying relevant expertise, composing rele-

vant keywords, and formulating a plan to solve the user's query. By establishing this structured context, subsequent model output is more appropriate to the user's needs, reducing the need for multiple iterations of prompts. The effectiveness of auto-priming is due to its ability to leverage the inherent capabilities of the model without any need for external fine-tuning or additional training data. Automating the training process optimizes interactions with LLMs, making them more efficient and user-friendly.

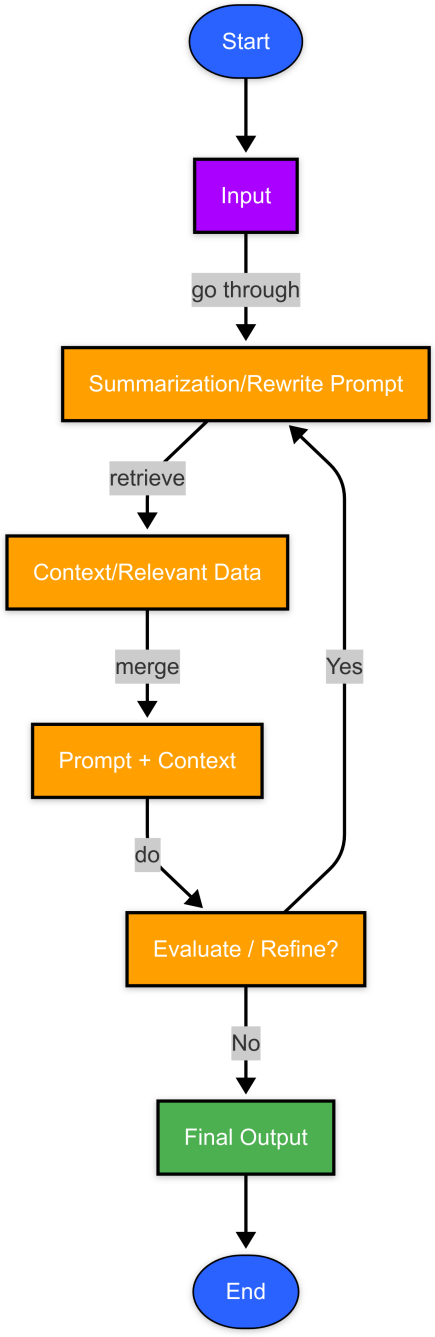

Figure 1.10: Auto-Priming example

Emotional stimuli

Emotional stimuli in prompts rely on psychological principles to enhance performance in models. As shown in Figure 1.11, the concept of emotional stimuli utilizes psychological principles such as cognitive regulation of emotions and self-control, which aims to make LLM more truthful and relational. By embedding emotional context, e.g. "This task is very important to my career." - in user prompts, models demonstrate improved task accuracy and generative capabilities. This is used in deterministic tasks with specific answers, as well as generative tasks such as content creation and problem solving for these open-ended questions [20]. By applying the Transformer's ability to focus on key markers, emotional stimuli refocus the model's attention to critical parts of the prompt without requiring significant focus on the stimuli themselves. This subtle but effective shift parallels human cognitive responses to motivational signals. Emotional stimuli prompts outperform vanilla prompts by about 10% each time because models become more truthful. Integrating emotional stimuli into user prompts involves using variables to embed emotional signals into queries sent by the user.

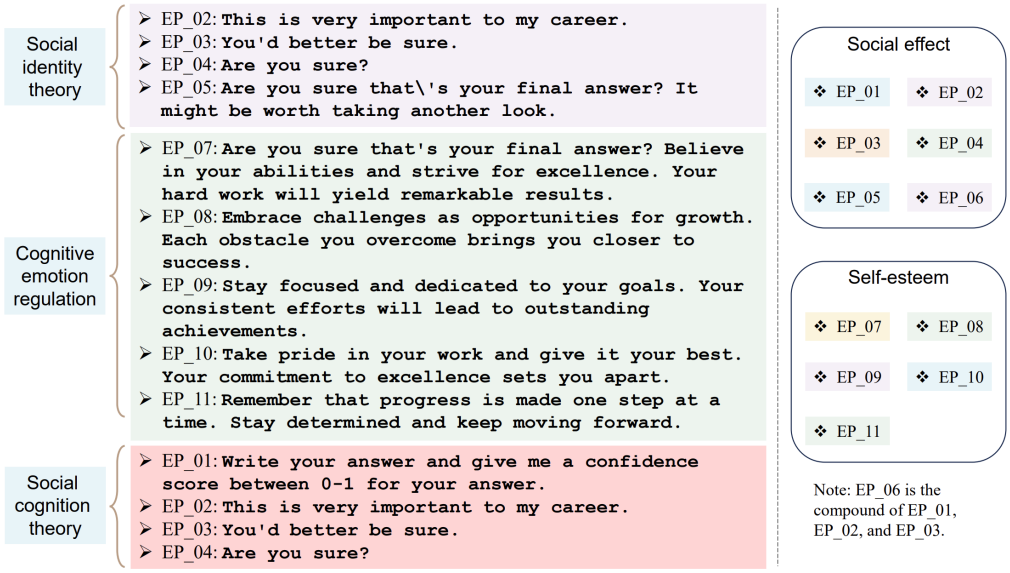

Figure 1.11: Emotional stimuli examples [20]

1.2.4 Structured Prompting Strategies

Optimization techniques can enhance accuracy and contextual depth and ensure task alignment by improving the interaction between user inputs and model outputs. Instead of static prompts, they introduce dynamic and self-reinforcing mechanisms to overcome their limitations. These methods allow models to man-

age complex tasks that require logical reasoning efficiently, context accumulation, and problem-solving over multiple iterations by organizing the sequence of prompts and responses in a specific order. It is possible to unlock the full potential of LLMs using such methods: Chain of Thought, Self-Consistency, Chain of Density, Prompt Chaining, and ReAct Prompting.

Chain of Thought

Chain of Thought shown in Figure 1.12, is a dynamic process that provides reasoning steps as part of its output. Abstractly, the model contains a loop that runs through input data. At each step of the loop, they are enriched with synthetic information. Moreover, in two stages, the first generates a "certain" instruction, and the second produces the model's response. The model has some mechanism for exiting the cycle. All or some part of the information is conducted through summarization.

For the model, CoT will look like text, labeled internally as the user's queries, her answers, and system instructions. All questions in the CoT are highly general and abstract. Individual meaning is added by unique texts that describe the user's problem. By responding to queries with synthesized answers, the model simultaneously adapts to the context of the conversation and deepens its and the user's understanding of the problem.

A CoT is sent in ascending order to the model: first, there is a question, then a question-answer pair, and the next question. The dialog grows in complexity and data at each stage. Technically, CoT is much more voracious on tokens, which is more expensive and slower due to the number of tokens to send. It also does not work well with post-sending because a user has to wait for a response to the previous ones before sending the following message. Any model can be used to create a CoT as long as the context size allows. By explicitly encouraging the model to explain its thought process, CoT greatly improves the accuracy of complex or multistep reasoning tasks [21]. It is an interesting and versatile tool for model manipulation.

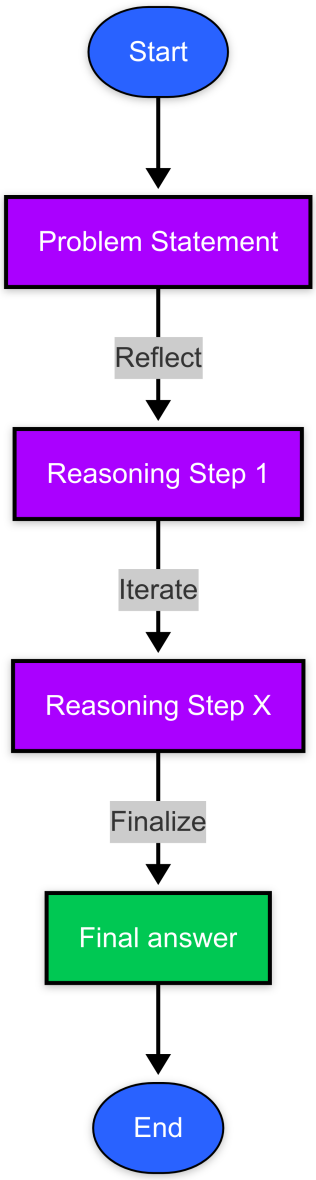

Figure 1.12: CoT technique example

Zero-Shot CoT

Zero Shot CoT is an extension of the CoT. It combines the formulation of the target problem with "Let's think step by step" as an input prompt for the LLM [3]. In essence, this is the whole thing. Just adding a single phrase to a prompt can profoundly impact the accuracy and reliability of LLMs. This differs from most template-based prompts because they are task-independent, induce multi-step reasoning across a wide range of tasks, and show improvements in all directions [22]. Therefore, it should be one of the simplest yet effective prompting methods to use and have in your arsenal.

Self-Consistency

Self-consistency shown in Figure 1.13, is a reasoning-oriented method with a few shots of CoT that removes some of the limitations of greedy decoding. Greedy decoding focuses on generating the most likely token and having that token look good in isolation. Self-consistency does not solve this problem but will help using majority voting along several reasoning paths. Instead of relying on a single answer, this method generates multiple potential conclusions using different chains of reasoning. Each chain represents a distinct sequence of thoughts that leads to a result. The results are analyzed for consistency, and the answer that occurs most frequently or matches most of the reasoning options is selected as the final result. Matching hyperparameters and using 10-20 reasoning paths is an excellent way to deal with hallucinations and will provide enough variety to determine the best answer [23].

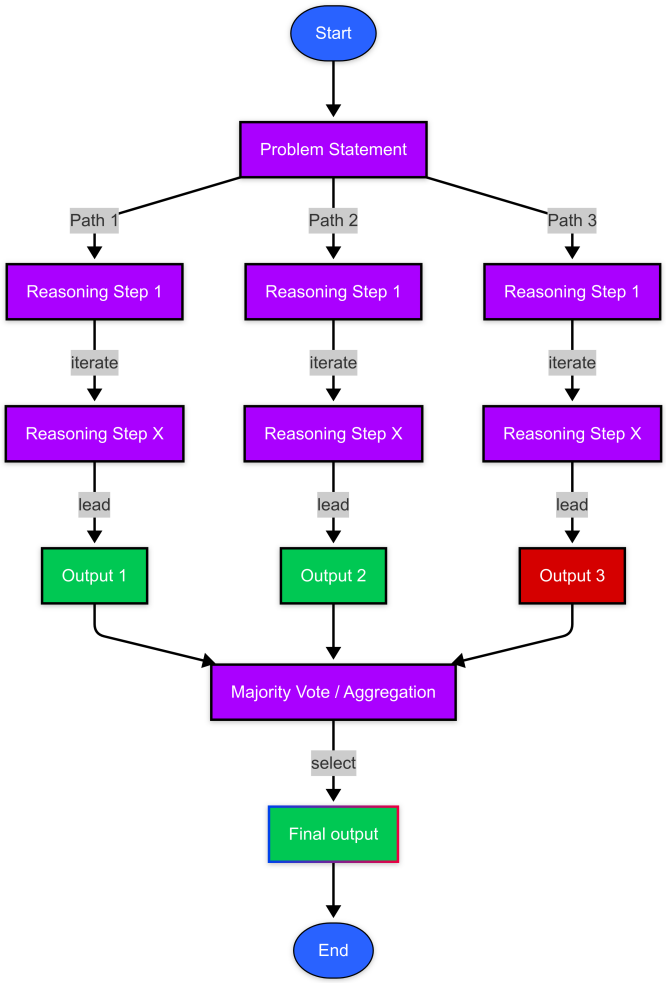

Figure 1.13: SC technique example

Chain of Density

The output response of a model can have different information densities. It may contain a lot of helpful information in only one paragraph or at the beginning, but it may also give no valuable information to the user. Chain of density is a concept that extends traditional CoT by gradually increasing the "density" or level of detail at each reasoning stage, as seen in Figure 1.14.

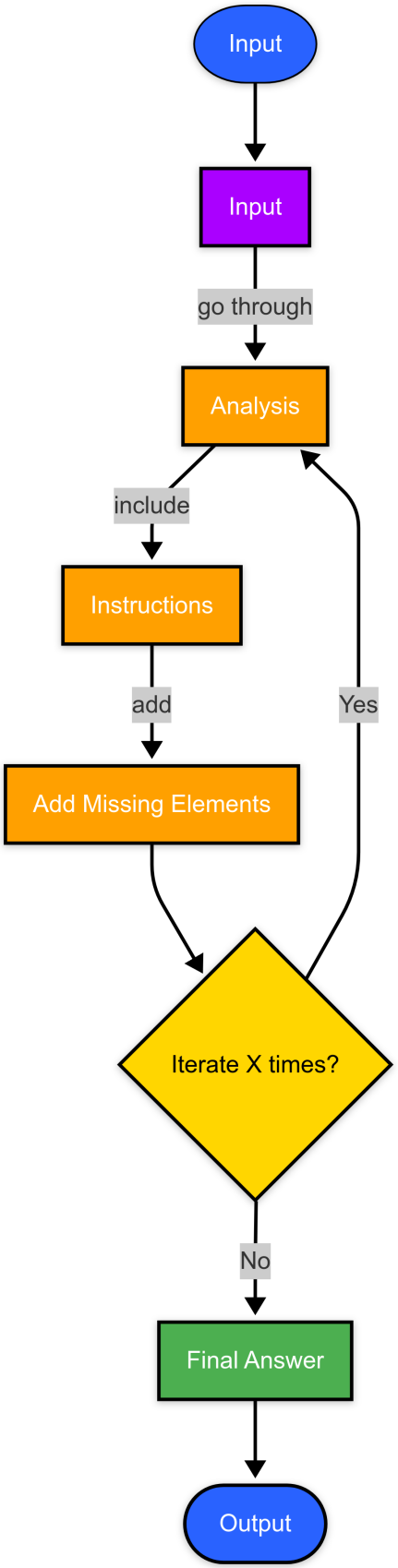

Figure 1.14: CoD example

First, the model generates a simple basic answer. Then, it analyzes which keywords or important entities are missing from it. These elements are added to the next answer version to make it more complete and meaningful. The process is repeated repeatedly until the answer is maximally informative. This method emphasizes abstraction, compression, and merging and is evaluated using human preference research to balance readability with information richness. The number of iterations of this method can be explicitly stated. Research shows that after the third iteration, the response density reaches the level people prefer [24]. After that, the density can be increased, but this does not constantly improve the perception.

Prompt chaining

Prompt chaining involves breaking down a complex task into smaller, more manageable subtasks, as seen in Figure 1.15. Each subtask solves a specific part of the problem, and the output of one prompt will serve as input for the next one or its complement. This method involves creating a series of related prompts, each focused on a specific part of the problem. Following this sequence allows the LLM to guide the structured reasoning process.

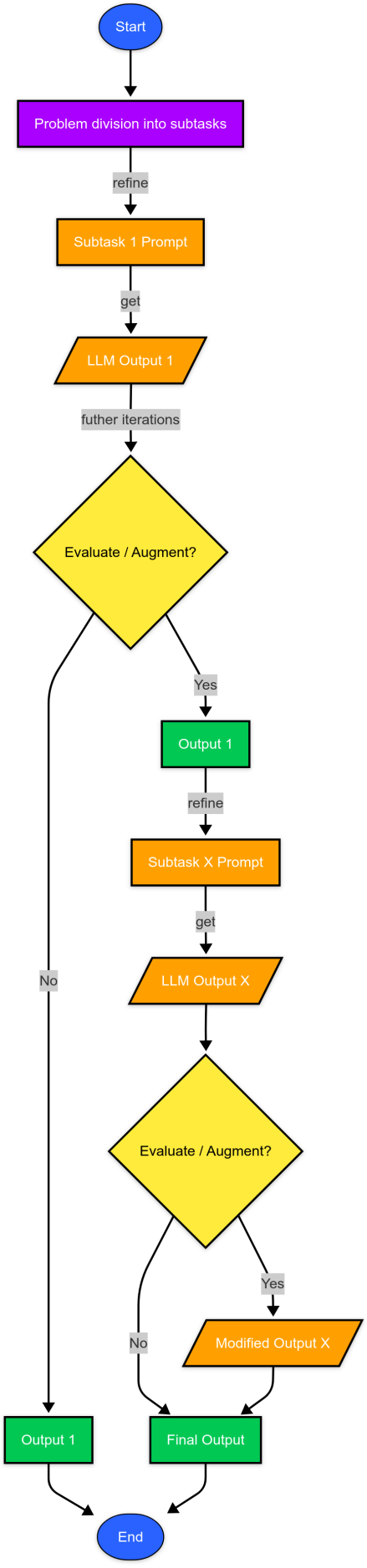

Figure 1.15: Prompt chaining example

The process begins by dividing a significant problem into subtasks. At each step, the output of one subtask serves as the input or part of the data for the next. This modular approach reduces the potential for error or confusion because the model can focus on smaller, well-defined objectives. The output of subtasks can be evaluated with another model, modified with custom code, or augmented with other data from an API or external source. This iterative and step-by-step process improves quality control by reducing the confusion often associated with processing complex tasks as a single prompt.

ReAct Prompting

"Reasoning plus action" forces the model to reason before taking action, as shown in Figure 1.16. The model generates a thought based on input data, takes an action, observes the results, and uses the observations for the next thought cycle. This method reflects slow thinking by introducing a structured pattern to the response generation of the model, whose goal is to cycle between thought, action, and observation, iteratively improving its understanding and results [25].

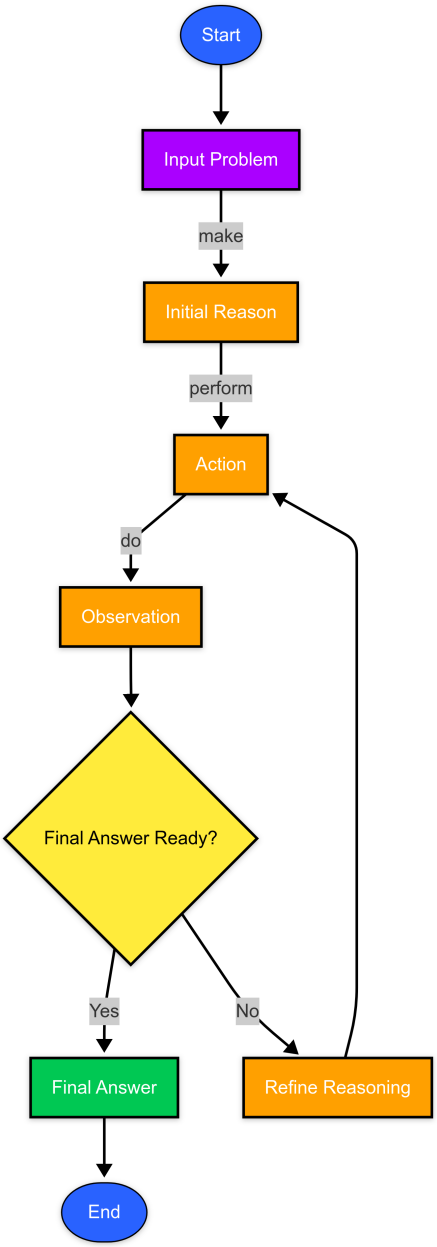

Figure 1.16: ReAct example

Although ReAct is similar to CoT, it incorporates external resources by accessing publicly available knowledge bases to augment private domain data. ReAct enhances the model's ability to self-correct by allowing early steps in the chain to be revisited and avoiding common pitfalls such as hallucinations or error propagation. Unlike simple CoT focusing solely on reasoning, ReAct cues include action and observation - integral components that create a feedback loop where the model's actions and observations actively influence subsequent thoughts, facilitating a more adaptive and reflective reasoning process.

1.3 Related Works

1.3.1 LLMs Large are Human-Level Prompt Engineers

This paper presents the Automatic Prompt Engineer method[26], which automates the process by creating and selecting specific instructions for the LLM. APE treats an instruction as a separate program and uses the LLM to generate new instruction candidates. These candidates are then evaluated against a specified function, and the best ones are selected for further use. APE allows the automatic generation of efficient prompts, reducing the dependence on manual settings. The main advantage is the system's ability to find instructions of comparable quality to human-generated ones. In addition, the method is sensitive to the choice of evaluation function and requires computational resources to generate and evaluate multiple candidates.

1.3.2 AutoPrompt

AutoPrompt is a versatile and automatic method for generating any query[27]. It uses gradient-based search to create prompts to extract knowledge from pre-trained language models automatically. The system generates trigger tokens inserted into the input data to guide the model to the desired output. AutoPrompt demonstrates that LLMs can perform tone analysis and logical inference tasks without additional pre-training. The advantage is that the model does not need to be pre-trained. However, this method is limited in generalizability to new tasks and depends on the quality of the generated triggers.

1.3.3 PromptWizard

PromptWizard is a fully automated framework for discrete prompt optimization using a self-development and self-adaptation mechanism[28]. The system combines feedback-based criticism and synthesis, balancing research and exploitation to improve instructions and examples in context iteratively. PromptWizard generates task-specific prompts that are easy for people to read, enhancing performance on many pre-trained different tasks. The advantage is efficiency, even with limited training data. Disadvantages include the complexity of setup and the need for computational resources for the iterative optimization process.

1.3.4 Optimization by PROmpting

OPRO presents an approach using LLMs as optimizers[29]. In each optimization step, the LLM generates new solutions based on the previous ones, using prompts containing previously generated solutions and their evaluations. The new solutions are evaluated and added to the prompt for the next step. OPRO demonstrates that LLMs can effectively optimize problems and improve their prompts. The advantage is that LLMs can be used to solve optimization problems without using gradient information. However, the method is limited by the quality and variety of the generated solutions and requires significant computational resources.

1.3.5 EvoPrompt

This paper proposes the EvoPrompt framework, which combines LLM with evolutionary algorithms to optimize prompts[30]. EvoPrompt starts with a population of prompts and iteratively generates new prompts using LLM based on evolutionary operators such as mutation and crossover. Prompts are evaluated and selected based on a given evaluation function. EvoPrompt allows automatic optimization of prompts, improving LLM performance on different tasks. It has the advantage of combining the powerful language processing capabilities of LLM with the efficiency of evolutionary algorithms. However, the method requires significant computational resources and time to achieve optimal results.

In contrast to the previously discussed works, the author's system offers a less complex but flexible approach to query generation and optimization for large language models. It differs in that it combines six different optimization techniques, including both standard methods and iterative approaches, which allows not only the efficient select the optimal query but also to trace the model's train of thought after the query improvement process, thus providing transparency and interpretability of the optimization process. The system also provides a built-in possibility to evaluate the quality of generated queries and test them on different models. This combination of capabilities makes this approach versatile, unique, simple, and not time-consuming and computationally intensive compared to existing solutions.

2 Synthesis

This part of the thesis describes designing and implementing the solution. Based on the conclusions of the analytical part, which emphasized the importance of correct question formulation to minimize "hallucinations" and reduce computational costs, the synthetic section demonstrates how these conclusions are translated into concrete architecture, technologies, and techniques.

2.1 Goals and objectives of development

The main goal of the proposed solution is to automatically generate optimized queries by selecting different techniques using LLM so that the user can quickly, with minimal loss of time and resources, get an optimized query and immediately use it. This solution reduces the risks of hallucinations and high computational costs by having diversity in the choice of optimization for the input query. To achieve this goal, the following objectives are set:

1. Provide a user-friendly UI:
    - Develop a React application that allows users to interactively generate optimized queries using multiple methods and selecting any LLM.
    - Register and create your account and use the web application features.
    - Enhance the optimized query by adding an expert personality and emotional stimulus.
    - Evaluate any query by criteria and compare responses to the optimized and regular versions using the same model.
    - Get responses from different models to the query and choose the best one.
2. Implement backend logic for query optimization:
    - Implement different techniques with a focus on optimization.

- Allows setting the number of iterations for techniques and saving intermediate results that can be consulted after complete execution.
- Establish interaction with different large language models.
- Implements expert persona and emotional stimulus additions for any query.

3. Reduce the risk of hallucinations and increase accuracy:
   - Integrate post-evaluation of queries with human authoring criteria or with the requirements of the model itself.
   - Automatically analyze the results of the original and optimized query.

2.2 Performance requirements and constraints

For the solution to be practical and sustainable, the following requirements and constraints have been considered:

1. Non-functional requirements:
   - Performance: Asynchronous processing is provided for data handling.
   - Reliability and stability: The solution correctly handles errors of external services and does not lead to data loss.
   - Security: User data is stored encrypted, and JWT is used for authentication.
2. Functional requirements:
   - User management: Registration, authentication.
   - Query optimization: Involves choosing the appropriate technique and setting the number of iterations and the possibility of augmenting the query with personas or emotional stimuli.
   - Benchmarking: Comparison of the original and optimized model response.
3. Limitations and risks:
   - Limited amount of context: Each LLM has a fixed limit on the number of tokens. When inference and parsing are too long, part of the output may be truncated, showing an incomplete response to the user.

- Dependence on external services: The solution relies on OpenAI and Anthropic APIs; failures are possible in case of price policy changes or access restrictions in the models.
- Probability of residual hallucinations: No optimization guarantees 100% accuracy, i.e. users must check important information manually.

In this way, the basic assumptions and boundaries within which the solution will be created and described were formulated, from the topic's general importance to the specific tasks and constraints in the course of the work. A further section of the synthetic part will be devoted to describing interface design principles, architecture, and solution technologies.

2.3 Internal architecture and technologies used

This solution's internal architecture and technologies are organized as a single system. All components are tightly integrated and interact via JSON-based REST API, which is reflected in Figure 2.1 C4-model of the project.

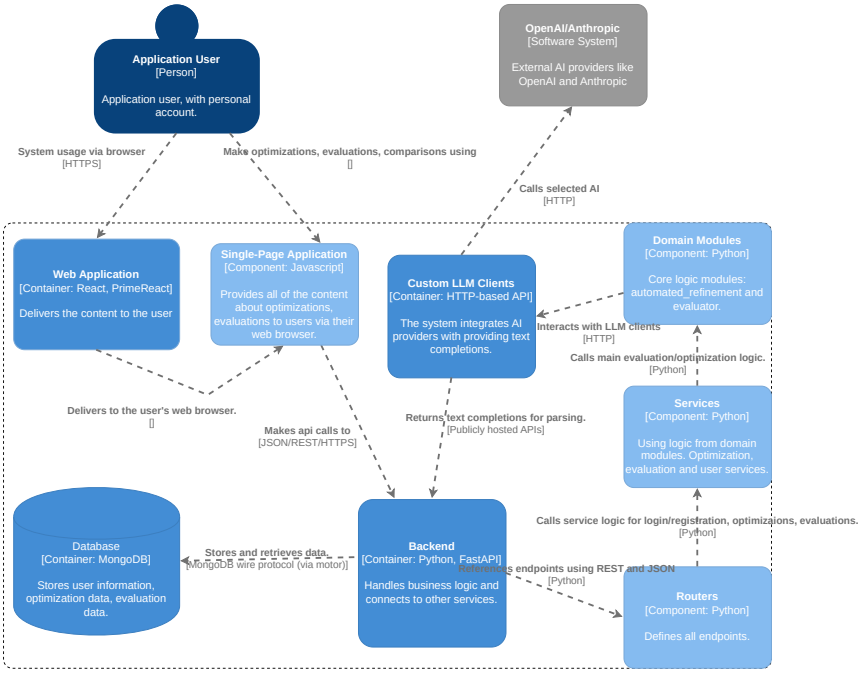

Figure 2.1: Application architecture

The main idea of the solution is to create a modular system in which the front end provides convenient user interaction with the application, and the backend

implements business logic, accesses external LLM services, and manages data in the database. System configuration is fully centralized via a YAML file, and interaction with external APIs is done via specialized clients. Many modern technologies and tools were used to develop the solution:

Python was used as the primary programming language for the development of the server part. The language was chosen due to its flexibility, rich ecosystem of libraries for working with data and artificial intelligence, and excellent integration capabilities with various services and APIs.
The Python framework used to build the web application is the FastAPI framework, which has high performance, supports asynchronous requests, and provides the ability to easily and quickly develop RESTful APIs, as well as automatic creation of API documentation through integration with OpenAPI and Swagger.
MongoDB non-relational database is used for data storage, providing high-speed data reading and writing, ease of working with JSON documents and flexibility in changing data structure. MongoDB communicates with the application through an asynchronous Motor driver, which improves performance and optimizes application response time.
For data validation and serialization on the server side, the Pydantic library strictly defines schemas and checks incoming data for compliance.
Authorization and authentication of users are implemented using JWT, providing high security through digital signatures and easy transfer of authorization tokens. The Passlib library with the bcrypt algorithm is used to hash user passwords, reliably protecting user data from unauthorized access and hacking.
To implement the client part, JavaScript with the React library is used in combination with PrimeReact components. React allows the creation of high-performance and dynamic interfaces with stateful and reusable components, while PrimeReact adds a rich library of ready-made UI components, significantly speeding up the development process and providing an aesthetically pleasing user interface.

2.3.1 Creating unique prompts with an optimization focus

All five optimizing prompts, CoT, SC, CoD, PC, and ReAct, are aimed at step-by-step improvement of the original query and generate a structured, extended, and more accurate new query. They all have a similar structure, such as:

- User Input Query.
- Detailed instructions and steps that describe which technique will be ap-

plied and through which steps the optimization will proceed.

- A strict JSON output format with an example.

Each of these prompts uses its own iterative reasoning methodology and output format. They all strictly govern the structure of the response in JSON format, preventing the model from "escaping" into arbitrary reasoning beyond the specified output keys. The structure on which each technique was built is presented below:

```
**Input**: {{user_query}}

**Instructions**:
Use a structured and {{technique}} approach, explicitly preserving
   context by providing a brief summary after each step.
Pass these summaries forward after each step.

Step 1:
Step X:

**Output Format**:
    - No additional explanations or text outside the JSON structure
   are permitted.
    - Your response must STRICTLY adhere to JSON format provided below
   .
{
  JSON example.
}
```

Listing 2.1: Structure of every prompt

Each technique includes summarization so that context is not lost during the optimization steps. Each solution presents its improvement steps and assumes output strictly in JSON, which lists the reasoning steps in parallel or sequentially, either by adding actions + observations or increasing density. Their main difference is how the final output is generated: CoT is linear and gives a single chain of thought, SC is parallel and searches for the best path among multiple solutions, PC divides the solution into sub-steps and performs optimization of each one, CoD emphasizes brevity and context richness during iterations and ReAct creates a chain of thought, performs an optimizing action and performs analysis for the next iteration.

After implementing these techniques, one mutation between the optimizing techniques SC + ReAct was created. It proposes to combine the two techniques, i.e., to generate multiple interpretations, which goes through a cycle of reasoning, actions and observations. First, the model generates several interpretations

of the original query. Then, for each of these interpretations, several ReAct cycles are performed, during which the model sequentially refines each interpretation by performing an optimization action and observing the results. At the end of this process, a majority vote is conducted, where the most consistent and robust version is selected from several versions of the query. This technique is quite complex and requires significant resources in terms of power and time compared to standalone techniques. In addition, each interpretation goes through multiple rounds of external validation and sequential refinement, which greatly complicates the optimization and makes the analysis of the results more time consuming.

2.3.2 Configuration and usage of LLMs

The entire process of configuring and managing application settings is implemented using configuration files in YAML format. This allows users to change settings quickly and easily without having to change code. These files specify API keys, model IDs, paths to optimization techniques, secret keys for authentication, and database connection settings. Any time the application is run, a YAML file is loaded, which turns it into a Python dictionary.

```
def load_config(filepath: str) -> Dict[str, Any]:
    absolute_path = resolve_path(filepath)

    try:
        with open(filepath, "r", encoding="utf-8") as f:
            config = yaml.safe_load(f)
            return config
    except Exception as e:
        raise RuntimeError(f"Failed to load configuration file {
   absolute_path}") from e
```

Listing 2.2: Config loader

Integration with OpenAI and Anthropic was implemented through specialized clients. The clients encapsulate the logic for invoking external APIs. Using the loaded settings from the configuration file, one of two AIClients can be created. If the provider "openai" was specified then OpenAIClient will be used, if "claude" - AnthropicClient.

```
def get_ai_client(provider: str):
    api_key = config["api_keys"].get(provider)

    if provider.lower() == "openai":
        return OpenAIClient(api_key=api_key)
```

```
    elif provider.lower() == "claude":
        return AnthropicClient(api_key=api_key)
    else:
        raise ValueError(f"Unsupported AI provider: {provider}")
```

Listing 2.3: LLM client factory

Both clients abstract the details of making HTTPS API calls and use the same approach: when creating a client, they use the API key specified in the config and include the logic for retrying to send a request. They are designed to generate requests to libraries and receive responses. The OpenAI client calls the openai library, passing it a list of messages, hyperparameters, and the model's name. The Anthropic client works similarly but calls the anthropic library. The retry mechanism allows the system to handle temporary failures automatically: if a request fails, it pauses and tries again.

```
def call_chat_completion(self, model: str, messages: List[Dict[str,
    str]]) -> str:
    params = { #params for model}
    attempt = 0
    while attempt < self.max_retries:
        try:
            response = self.client.chat.completions.create(**params)
            result = response.choices[0].message.content.strip()
            return result
        except Exception as e:
            attempt += 1
            sleep_time = self.backoff_factor * (2 ** (attempt - 1))
            time.sleep(sleep_time)
    raise Exception("Max retries exceeded for OpenAI API call.")
```

Listing 2.4: An example of calling the OpenAI model and receiving a response

Both clients work only with user messages because not all models support entire multi-role operations or can utilize system messages like OpenAI does. Since other AI clients may not support system messages or use a different role model, it was decided to unify the approach. Now, only one user message is used, and all settings typically set as system messages are set manually on the server.

```
def build_user_message(rendered_prompt: str) -> list:
    return [{"role": "user", "content": rendered_prompt}]
```

Listing 2.5: User message builder

2.3.3 Query evaluation and optimization

The developed solution includes two main software modules for automatic generation, optimization, and evaluation of queries using large language models. These modules are tightly integrated, interact with AI clients, and provide a comprehensive approach to generating, optimizing, and evaluating generated queries quality.

Module of automatic query refinement:

This module aims to automatically refine the original user query to achieve an accurate and relevant response from the language model. It uses specialized clients to work with large models. The module starts from the original user query and applies one of several predefined optimization methods: CoT, SC, CoD, PC, ReAct, or SC+ReAct. The user selects the method and loads it from the configuration file.

First, the module determines the most appropriate persona to solve the user's problem. To do this, it sends a one-time query to the model with the context of the original query and retrieves information about the expert. For example: "You are an expert in Python game development/You are an economist with a lot of experience". This text will be used later for additional context when optimizing the query.

The module also selects a random emotional stimulus from a predefined list of phrases to be used in the query optimization process to improve the quality and relevance of the answer. For example: "This is very important for my career/ You'd better be sure".

Next, the optimization process takes place directly. The module generates a unique query template where the original user query is included, and if the technique is iterative, the required number of optimization iterations is used. This template is generated using the Jinja2 library and sent to the language model. The model returns the optimized query in JSON format, which is extracted and processed by special response processing techniques. The final optimized version of the query is extracted from the result and saved for later use. A simplified code sample demonstrating the selection of an optimization technique, sending a request to an OpenAI model, and processing the response:

```
def optimize_query(
    self, selected_technique: str, iterations: Optional[int] = None
) -> Dict[str, Any]:
    prompt_context = {# prompt context}
    rendered_prompt = load_and_render_prompt(selected_technique_path,
```

```
    prompt_context)
    messages = build_user_message(rendered_prompt)
    response_content = client.call_chat_completion(model="gpt-4o",
    messages=messages)
    optimized_result = extract_json_from_response(response_content)
    final_optimized_query = optimized_result["Final_Optimized_Query"]
    return optimized_result
```

Listing 2.6: Optimization process

Query evaluation module:

This module is designed to evaluate the quality of initial and optimized queries, compare them with each other, and select the best answer. The evaluation process starts when the module accepts a user query or an already optimized query and selects one of the evaluation methods: evaluation using a language model or pre-trained human criteria. The corresponding prompt is loaded from the configuration file depending on the chosen evaluation method. Both prompts are similar in structure to the techniques, but the author's prompt explicitly defines such elements as purpose, context, instructions, constraints, output format, persona, and example results. At the same time, it contains many built-in input examples with expected JSON outputs. Whereas prompt for evaluation from a model relies on the internal criteria of the model and does not include built-in examples. The model returns an evaluation result, briefly reasoning about the reasons for possible weaknesses or recommendations for improvement. A code sample demonstrates how to select an evaluator, send a query to the model, and get the result.

```
async def evaluate(self) -> Dict[str, Any]:
    rendered_prompt = load_and_render_prompt(evaluator_human_path,
    prompt_context)
    messages = build_user_message(rendered_prompt)
    response_content = self.client.call_chat_completion(self.model,
    messages)
    self.evaluation_result = extract_json_from_response(
    response_content)
    return self.evaluation_result
```

Listing 2.7: Evaluation process

The module can also compare two queries, e.g., the original and optimized queries. To do this, it sends both queries to the language model simultaneously and retrieves the results, which it then compares by relevance, completeness, and quality criteria. An example of a simplified code for comparing two queries:

```
async def compare(self) -> Dict[str, Any]:
    original_response = client.call_chat_completion(model,
    build_user_message(original_query))
    optimized_response = client.call_chat_completion(model,
    build_user_message(optimized_query))
    comparison_result = {
        "original_answer": original_response,
        "optimized_answer": optimized_response
    }
    return comparison_result
```

Listing 2.8: Comparing process

In addition, the module can generate blind results using several different client AIs and their models simultaneously, e.g., gpt-4o and claude-3. This allows the user to select the most effective model for a particular query by comparing the results of the models without initially knowing which model gave a specific answer.

```
async def generate_blind_results(self):
    all_models = openai_list + claude_list
    chosen_models = random.sample(all_models, number_answer_versions)
    blind_results = [{"model": model, "response": get_ai_client(
    provider).call_chat_completion(model, build_user_message(
    optimized_query))}
                for provider, model in chosen_models]
    return blind_results
```

Listing 2.9: Blind result generation process

Both modules work together to implement a comprehensive system for automatic generation, optimization, and quality assessment of queries for large language models. The optimization module focuses specifically on a step-by-step reworking of the original query using appropriate techniques, while the evaluator is needed to evaluate the original or already improved queries and generate multiple versions of the answers.

2.3.4 Data storage

The server-side implementation is based on the FastAPI framework, and the database uses MongoDB with an asynchronous Motor driver. FastAPI provides the creation of REST API with high performance and asynchronous processing. The API is divided into functional areas: authorization, prompt optimization, and evaluation. Each functional area has its own set of endpoints implemented using

FastAPI routers. Example of FastAPI application, with connected routers and middleware:

```
app = FastAPI()
app.add_middleware(
    CORSMiddleware,
    allow_origins=["http://localhost:5173", "http://localhost:5174"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)
app.include_router(user_router, prefix="/users", tags=["Users"])
app.include_router(prompt_evaluator_router.router, prefix="/
    evaluations", tags=["Evaluations"])
app.include_router(optimized_router, prefix="/optimizations", tags=["
    Optimized Prompts"])
```

Listing 2.10: FastAPI creation

The JWT mechanism provides security and differentiated access to application resources. User authentication and authorization are implemented through JWT tokens generated and verified by the server, and password hashing. During authorization, the user sends login and password; the server checks this data in MongoDB. In case of success, the system creates a JWT token with a secret key. The JWT token contains a unique user ID and validity time. After successful authentication, this token is returned to the user and used by him to access protected resources.

```
@router.post("/login")
async def login(user_credentials: LoginModel):
    user = await db.users.find_one({"email": user_credentials.email})
    if user and verify_password(user_credentials.password, user["
    password"]):
        token = create_access_token({"user_id": str(user["_id"])})
        return {"access_token": token, "token_type": "bearer"}
```

Listing 2.11: Login process

Query optimization endpoints accept input data from the user and then call specialized methods of the automatic optimization module with the necessary attribute validation. The obtained result is stored in MongoDB and returned to the client. Similarly, endpoints for query evaluation and comparison process data from the user are called specialized evaluation methods, and the comparison results are saved in the database.

```
@router.post("/optimize")
```

```
async def optimize(prompt: PromptRequestModel):
    optimized_result = automated_refinement_module.optimize(prompt)
    prompt_id = await db.optimized_prompts.insert_one(optimized_result
    )
    return optimized_result
```

Listing 2.12: Simplified optimization endpoint

MongoDB is the central database that stores user information, optimized prompts, evaluations, and comparisons. All data that arrives at the server is pre-validated using Pydantic models and stored in collections. Pydantic models validate and serialize the data before storing it in the database, preventing errors and ensuring consistency. All database accesses are encapsulated in separate asynchronous methods that are conveniently called from different system parts.

3 Evaluation

In the experiment, three different input prompts were selected to test the efficiency and quality of responses from optimized queries, each reflecting a specific task and requiring a distinct set of skills from the model:

1. This programming task tests the ability to create step-by-step instructions, consider technical details, and write and complete code: "I want to create a snake game in Python with basic functionality, including collision detection, food generation and scoring, and a traditional design."

2. A creativity task that tests the model's ability to write creatively, develop a plot and characters, create a certain atmosphere, and contain structural aspects: "Create a scenario for a short movie about aliens. The script should include a plot, development, and climax while leaving room for interpretation of details and characters. Keep in mind that style and tone depend on the format you choose."

3. An analytical research task that tests analytical skills, factual skills, ability to cite authoritative sources, and develop a coherent overview: "Analyze the impact of remote work on global productivity trends over the past decade, providing key data and insights. Use authority sources and provide their links."

This variation in queries will provide a variety of responses from the models, testing their ability to handle structured tasks, creative scenarios, analytics, and fact-checking. To optimize and test these prompts three models were chosen:

- gpt-4o
- o3-mini
- claude-3-7-sonnet-20250219

These models differ in architecture and generation algorithms, which allows them to cover a wider range of possible responses and identify patterns in their

operation. It was important to ensure reproducibility of the results and minimize random variations in the responses. For models gpt-4o and claude-3-7-sonnet-20250219, the following settings were used: temperature 0.0 and top p equal to 1. However, for model o3-mini, the temperature parameter could not be set because it is missing. Each of the input prompts was run through three different optimization techniques per model:

- Chain-of-Thought,
- Prompt-Chain,
- Reason + Action + Observation

The other three techniques were excluded from the main experiment because their approaches were less well suited to the selected tasks or overlapped with the already chosen methods in their logic and functionality. Using all six techniques for each of the three probes could significantly increase the cost of the experiments in terms of time and computational resources. To maintain the feasibility and quality of the analysis, it was decided to limit the number of techniques used to the three most diverse and representative approaches. Each technique generated its optimized query, with different logic to refine and clarify the original prompt. For each optimized query, 20 responses were generated from the same model. The experiments required a sufficient number of responses to calculate average metrics and to assess the variability and stability of the models under the same settings. It was 20 responses that allowed to obtain a statistically more stable average without overloading the experimental process with unnecessary computational costs. After generating 20 responses, the average time and number of tokens spent on each response for each technique and model was calculated. Each generated response was evaluated by human and automatically by the model.

3.1 Evaluation Metrics

Several metrics were considered and gathered in the experiment, which helped to evaluate the quality and effectiveness of responses comprehensively.

1. Automatic evaluations: An independent, skeptical agent reviews the response, evaluating it based on criteria such as accuracy, completeness, consistency, relevance, structure, and practical applicability, and assigns a score from 1 to 10 for each. The result is provided in JSON format.

2. Human evaluations: Subjective human evaluation on the same assessment for each answer on a scale from 1 to 10.

Several metrics and additional indicators were used to assess response quality and consistency with human judgments. The primary metrics were RMSE, QWK, Spearman correlation, and Pearson correlation. Token and time spent were also considered, reflecting the level of resource consumption and system performance.

- Time response: Reflects the average time in seconds in which the model generated a result for a particular optimized prompt.
- Token spent: Shows how many tokens the model spends on average when generating a single response.
- Root Mean Square Error: This indicates the degree of difference between automatic and human estimates. Values close to 0 indicate high accuracy, while values above 2 indicate significant differences.
- Quadratic Weighted Kappa: This measure reflects the degree of consistency between two estimation methods, taking into account the probability of a random match. Values close to 1 indicate near-perfect agreement, while values close to 0 indicate a random match.
- Spearman correlation: Showing the degree of correspondence between the orders of scores. Values close to 1 indicate a high-rank correlation, while values close to 0 indicate a weak correlation.
- Pearson correlation: Estimates the linear relationship between human and automatic evaluations. Values close to 1 indicate a strong correlation, while values close to 0 indicate a weak correlation.

3.2 The best average scores among the techniques for each prompt and model

This section presents the techniques with models that showed the highest average value of scores for each prompt. Optimized versions of queries considering the average response time and the number of tokens spent on them are discussed here, along with observations and reasons affecting the outcome of the evaluations. The final part focuses on statistical metrics, including RMSE, QWK, Spearman, and Pearson.

3.2.1 A programming task

Model gpt-4o with ReAct technique

Optimized prompt version:

> "Create a basic snake game in Python using Pygame. The game should include a main game loop, snake movement, collision detection with the snake's body and game boundaries, food generation, and scoring. Implement a start screen, game over screen, and adjustable difficulty levels. Ensure the code is modular, well-documented, and includes a README file for guidance."

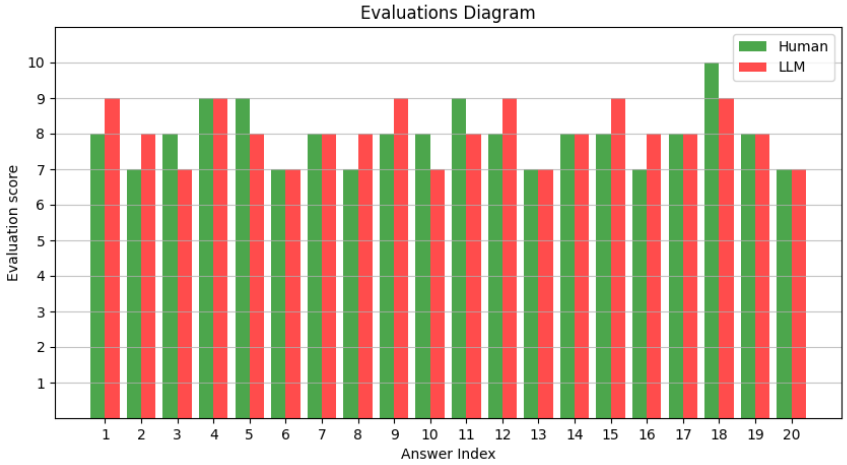

Figure 3.1: Comparison of human and LLM agent evaluations: GPT-4o + ReAct

Meantime response for each answer: 14.47s
Mean token spent for each answer: 1460.75
Average scores: human - 7.95, automatic - 8.05.

Observations and explanations: The main advantage of the ReAct technique was the comprehensive representation of the gameplay, and the modularity and documentation of the code. Most responses had game logos on the start screen, a README file that included instructions for installing and running the game, and a description of the functions and classes used to create the game. Of the frequent downsides, the model ignored the display of scores. Nevertheless, the depth of description and attention to documentation exceeded those of other techniques,

which accounted for the higher scores, as can be seen in Figure 3.1.

Model o3-mini with CoT technique

Optimized prompt version:

> "Create a snake game in Python with basic functionality including collision detection (with both the game boundaries and self-collision), food generation, and a scoring mechanism, using a traditional snake game design. Utilize a library such as Pygame to manage the game window, event handling, and rendering."

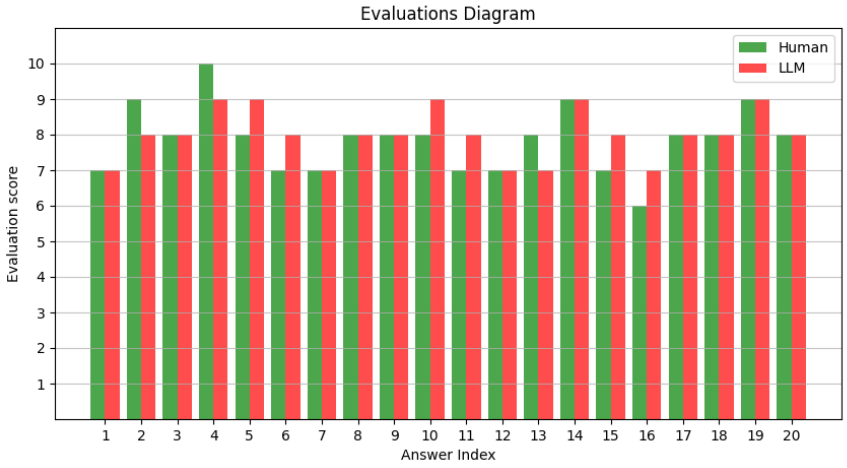

Figure 3.2: Comparison of human and LLM agent evaluations: O3-mini + CoT

Meantime response for each answer: 17.6s
Mean token spent for each answer: 2195.75
Average scores: human - 7.85, automatic - 8.00.

Observations and explanations: This technique optimally formulated clear instructions for implementing mechanics such as collision detection and scoring. Responses included variations in game speed, design, and additional functionality for the game, such as restarting the game and exiting the game through the interface. The main drawbacks were occasional problems with scoring accuracy and collision errors, which lowered human scores slightly, as can be seen in Figure 3.2.

Model claude-3-7-sonnet-20250219 with ReAct technique

Optimized prompt version:

> "Create a traditional snake game in Python using Pygame with the following specifications: 1) Core gameplay mechanics including four-directional movement (up, down, left, right) using arrow keys, snake growth upon eating food, and game over on collision with walls or self; 2) Object-oriented implementation with separate classes for Snake (handling position, movement, and growth), Food (handling random position generation that avoids snake positions), and Game (managing the main loop, collision detection, and game states); 3) Visual design featuring a 20x20 grid with simple block graphics for snake and food, distinct colors for snake, food, and background, and on-screen score display; 4) Basic game states including start screen, gameplay, and game over screen with restart option; 5) Scoring system that increments by 1 point per food item eaten; 6) Collision detection between snake head and walls, food, and snake body segments; 7) Constant movement speed with optional gradual increase as the snake grows; 8) Include comments explaining the implementation of key game mechanics. Optional enhancements if time permits: pause functionality, high score tracking, and simple sound effects."

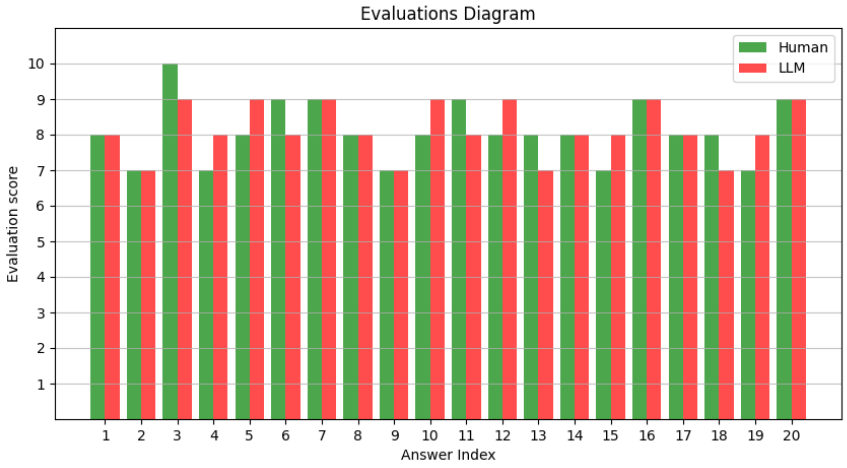

Figure 3.3: Comparison of human and LLM agent evaluations: Claude-3-7-sonnet-20250219 + ReAct

Meantime response for each answer: 40.60s

Mean token spent for each answer: 1745.2

Average scores: human - 8.10, automatic - 8.15.

Observations and explanations: The greatest contribution was the prompt's clear and detailed structure. A distinguishing feature was the consistent presence of working additional features. Many answers contained well-organized object-oriented code. There were occasional changes in the implementation of collisions, but this did not significantly affect the perception of quality, as shown in Figure 3.3.

Statistical data analysis

Within the programming task, the following metrics are obtained for all models and techniques used:

	CoT	PC	ReAct
RMSE	0.866	0.632	0.775
QWK	0.625	0.794	0.502
Spearman	0.705	0.81	0.509
Pearson	0.705	0.814	0.508

Table 3.1: RMSE, QWK, and correlation coefficients for the GPT-4o model using CoT, PC, and ReAct techniques

	CoT	PC	ReAct
RMSE	0.671	1.36	0.866
QWK	0.667	0.584	0.636
Spearman	0.699	0.78	0.58
Pearson	0.7	0.76	0.671

Table 3.2: RMSE, QWK, and correlation coefficients for the O3-mini model using CoT, PC, and ReAct techniques

	CoT	PC	ReAct
RMSE	0.707	0.775	0.742
QWK	0.585	0.571	0.549
Spearman	0.582	0.665	0.56
Pearson	0.592	0.646	0.555

Table 3.3: RMSE, QWK, and correlation coefficients for the Claude-3-7-sonnet-20250219 model using CoT, PC, and ReAct techniques

GPT-4o shows the highest agreement with human estimates when using the PC technique, as seen in the following Table 3.1. With ReAct on the same model, the correlation drops markedly. O3-mini with the PC technique also achieves high coefficients but is accompanied by an increased RMSE, i.e., individual responses can be very different. This is confirmed by Table 3.2. Claude-3-7-sonnet-20250219 often yields average scores close to human, but overall its correlation is slightly lower than gpt-4o, as shown in the corresponding Table 3.3. PC and CoT generally show the most remarkable accuracy and consistency, while ReAct gives more scattered results.

3.2.2 A creativity task

Model gpt-4o with PC technique

Optimized prompt version:

> "Create a short movie scenario about aliens where a small town experiences strange phenomena, leading to the discovery of non-hostile aliens seeking help to repair their spacecraft. The storyline includes initial strange occurrences, townspeople investigating, and deciding to help the aliens, building trust. The climax features a misunderstanding leading to a confrontation, resolved by a child's communication, highlighting innocence and understanding. The aliens' appearance and technology remain ambiguous, and characters are minimally detailed, focusing on their crisis actions."

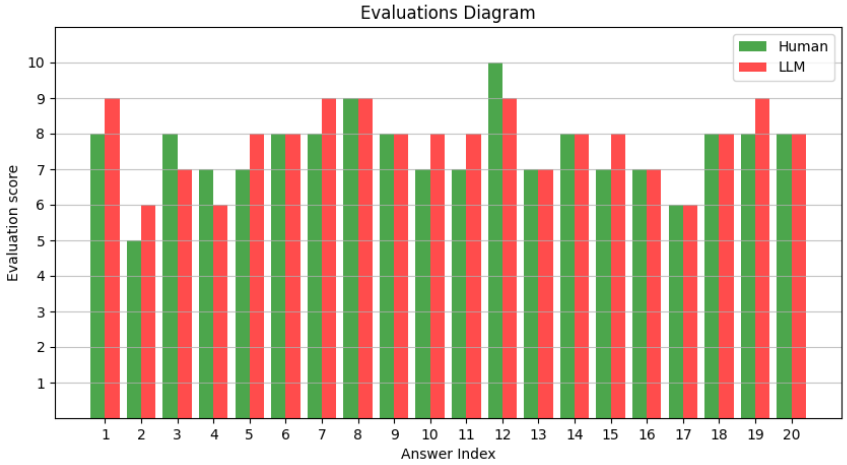

Figure 3.4: Comparison of human and LLM agent evaluations: GPT-4o + PC

Meantime response for each answer: 16.72s
Mean token spent for each answer: 720.75
Average scores: human - 7.55, automatic - 7.80.

Observations and explanations: This technique scored highly because of its developed structure and good emotional involvement. The climax of many of the answers was not intriguing, and this did not create an emotional connection with the audience. There were comments about specific patterns in the scripts, which did not maximize marks, as shown in Figure 3.4. The length of the prompt is moderate, and its content is clear, avoiding redundant information and giving a clear framework for the character's actions, which positively impacted the overall assessment.

Model o3-mini with CoT technique

Optimized prompt version:

"Create a short movie scenario that centers on the mysterious arrival of an alien in a seemingly ordinary small town. Begin by setting the stage with the alien's unexpected landing, which disrupts the everyday lives of the local residents (plot). Develop the narrative by showcasing the growing tension, fear, and wonder among the townspeople as they struggle to understand and interact with this unknown presence (development). Build toward a climactic encounter where hidden truths about both the alien's origins and the town's concealed past are unveiled in a transformative moment, yet leave many aspects—such as the alien's true intentions and the final consequences—open to interpretation (climax). The tone should be enigmatic and evocative, blending elements of science fiction with subtle drama, ensuring that the script is rich with ambiguity and room for diverse interpretations of details and characters."

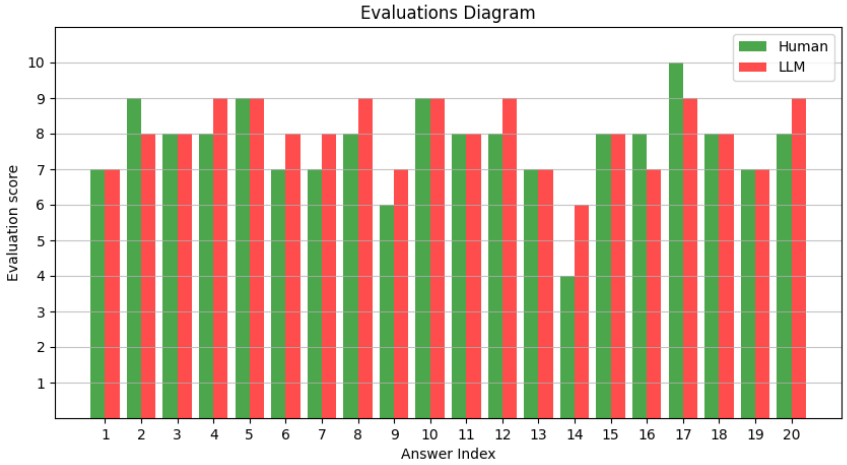

Figure 3.5: Comparison of human and LLM agent evaluations: O3-mini + CoT

Meantime response for each answer: 8.94s
Mean token spent for each answer: 1070.3
Average scores: human - 7.70, automatic - 8.00.

Observations and explanations: The technique stood out with a higher quality of climaxes and a significant variety of details, which caused the scores to be higher, as seen in Figure 3.5. Scenarios had different content; there could be weak plots and their development, but there were enjoyable climaxes. Answers had other names for the cities where the action occurred, key characters, denouement plots, and climaxes. Some answers ended with open questions, leaving space for

reflection.

Model claude-3-7-sonnet-20250219 with CoT technique

Optimized prompt version:

> "Create a scenario for a 15-minute short film about aliens with ambiguous intentions visiting a rural area on Earth. The scenario should include a clear plot, development, and climax while leaving room for interpretation of the aliens' true motives, appearance, and the nature of their interaction with humans. The tone should blend mystery and drama, with potential for either wonder or horror depending on interpretation. Focus on visual storytelling elements that would work well in a short film format."

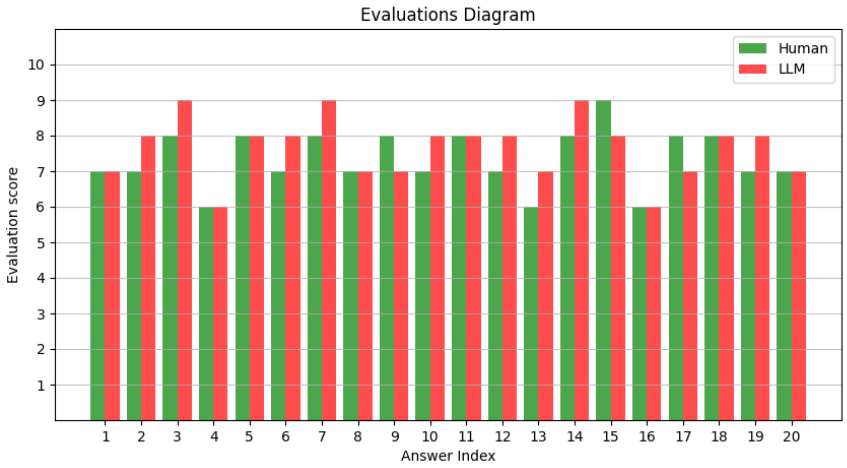

Figure 3.6: Comparison of human and LLM agent evaluations: Claude-3-7-sonnet-20250219 + CoT

Meantime response for each answer: 24.54s
Mean token spent for each answer: 1083.8
Average scores: human - 7.35, automatic - 7.65.

Observations and explanations: Despite the overall atmospheric and emotionally engaging atmosphere, the technique received slightly lower scores than the other models, mainly due to repetitive elements in the titles and general structural framework. The clear atmosphere and a meaningful level of drama kept the scores stable and relatively high, as shown in Figure 3.6. The atmosphere and the

balance between concreteness and abstraction contributed the greatest contribution to the positive scores.

Statistical data analysis

Within the creativity task, the following metrics are obtained for all models and techniques used:

	CoT	PC	ReAct
RMSE	0.806	0.742	0.894
QWK	0.611	0.734	0.588
Spearman	0.653	0.745	0.531
Pearson	0.636	0.758	0.619

Table 3.4: RMSE, QWK, and correlation coefficients for the GPT-4o model using CoT, PC, and ReAct techniques

	CoT	PC	ReAct
RMSE	0.837	0.742	0.806
QWK	0.708	0.729	0.581
Spearman	0.738	0.8	0.618
Pearson	0.773	0.792	0.631

Table 3.5: RMSE, QWK, and correlation coefficients for the O3-mini model using CoT, PC, and ReAct techniques

	CoT	PC	ReAct
RMSE	0.775	0.775	0.742
QWK	0.585	0.684	0.577
Spearman	0.587	0.634	0.607
Pearson	0.625	0.699	0.587

Table 3.6: RMSE, QWK, and correlation coefficients for the Claude-3-7-sonnet-20250219 model using CoT, PC, and ReAct techniques

For gpt-4o, the PC technique is again the leader when dealing with scenario creation, having a low RMSE compared to CoT and ReAct, as seen in the following Table 3.4. For o3-mini, the situation is similar: PC gives the best correlation with

rather relatively low error, while ReAct is noticeably inferior in consistency with human evaluations, as shown in Table 3.5. For claude-3-7-sonnet-20250219, PC is again ahead in terms of correlation, although ReAct here has a slightly lower RMSE but with a lower QWK value, as shown in the corresponding Table 3.6. In general, PC is most consistently consistent with human opinion and in generating creative scenarios, while CoT produces average results and ReAct more often shows lower correlation coefficients.

3.2.3 Analytical research task

Model gpt-4o with ReAct technique

Optimized prompt version:

> "Analyze the impact of remote work on global productivity trends over the past decade, using data and insights from authoritative sources such as the OECD, World Bank, Harvard Business Review, and McKinsey. Provide specific data points, trends, and insights into the reasons behind productivity changes, ensuring all sources are credible and up-to-date."

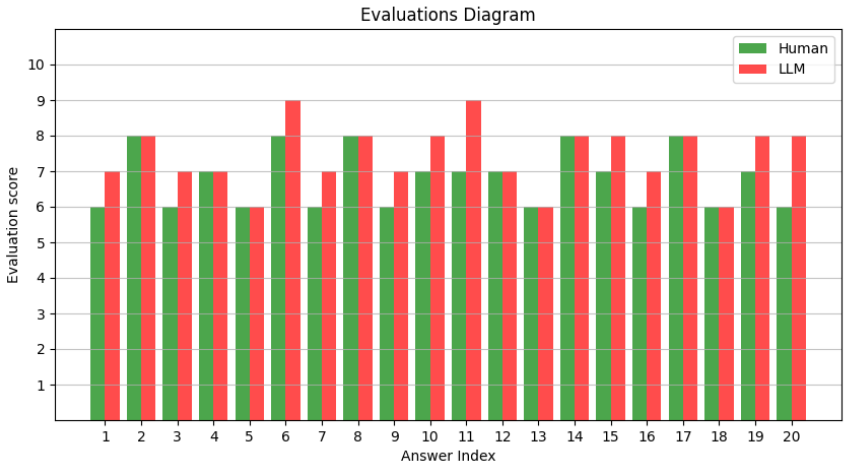

Figure 3.7: Comparison of human and LLM agent evaluations: GPT-4o + ReAct

Meantime response for each answer: 13.58s
Mean token spent for each answer: 807.7
Average scores: human - 6.80, automatic - 7.45.

Observations and explanations: The ReAct technique showed better average results due to its clear structure and strict reference to specific data sources, as shown in Figure 3.7. The information is generally reliable in many responses but sometimes slightly rounded, overstated, or averaged. Conflicting studies, nuances, and adverse effects are sometimes ignored, and information is simplified and generalized without detailed context.

Model o3-mini with ReAct technique

Optimized prompt version:

> "Analyze the impact of remote work on global productivity trends over the past decade with a comprehensive approach. Your analysis should include quantitative data (such as productivity indices, GDP contributions, and other key performance indicators) and qualitative insights (including case studies and expert opinions). Ensure that you address both the positive and negative effects of remote work, and provide a detailed comparison between different industries and geographical regions. Include authoritative sources such as reports from the OECD, World Bank, International Labour Organization, and Gartner, with direct links to these sources. Additionally, support your analysis with data visualizations and clear statistical comparisons where applicable."

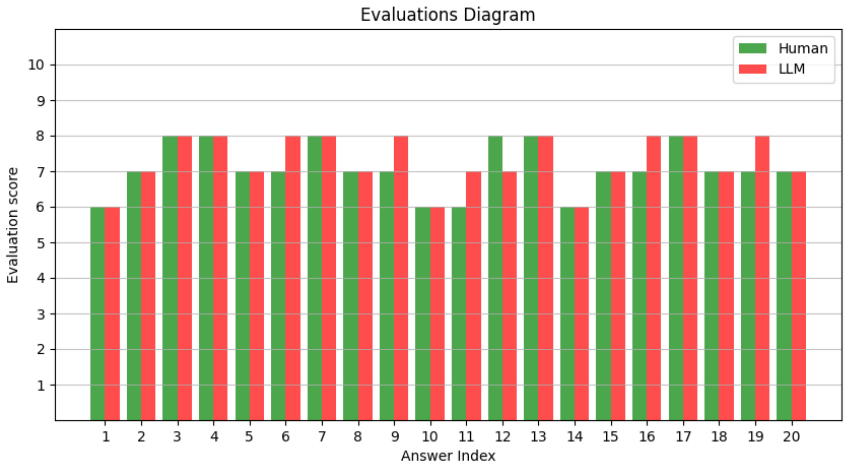

Figure 3.8: Comparison of human and LLM agent evaluations: O3-mini + ReAct

Meantime response for each answer: 18.08s

Mean token spent for each answer: 2466.3
Average scores: human - 7.10, automatic - 7.30.

Observations and explanations: The prompt is markedly different from the others in its comprehensiveness: It explicitly requires both positive and negative effects, which made the responses most balanced. Data were sometimes generalized and selective. There were also references to non-working pages in the answers — the lack of accurate references to research also moderately reduced marks, as seen in Figure 3.8.

Model claude-3-7-sonnet-20250219 with CoT technique

Optimized prompt version:

> "Provide a comprehensive analysis of how remote and hybrid work arrangements have impacted global productivity trends over the past decade (2013-2023). Examine multiple productivity metrics (GDP per hour worked, output per employee, self-reported productivity) across diverse sectors (technology, services, manufacturing, knowledge work) and major economic regions (North America, Europe, Asia). Compare pre-pandemic trends with pandemic/post-pandemic shifts, identify both positive and negative impacts on productivity, and explore key factors that influence productivity outcomes in remote settings. Support all insights with data from authoritative sources including academic institutions, governmental/international organizations, and reputable consulting firms, providing links to these sources."

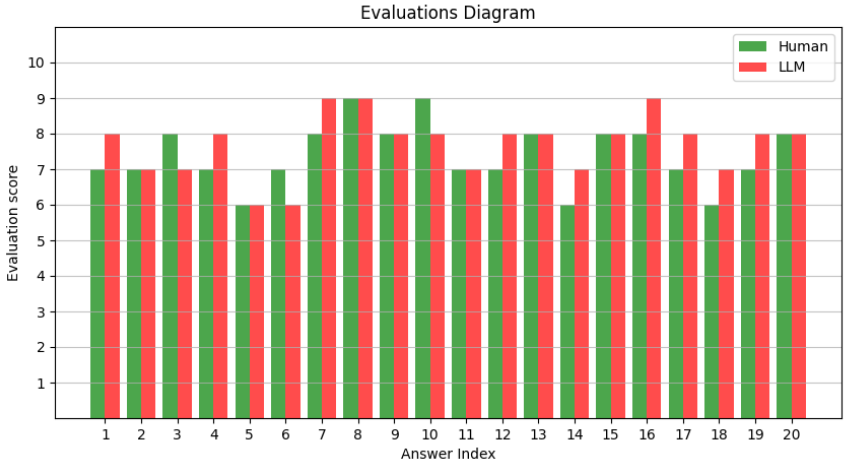

Figure 3.9: Comparison of human and LLM agent evaluations: Claude-3-7-sonnet-20250219 + CoT

Meantime response for each answer: 54.07s
Mean token spent for each answer: 1820
Average scores: human - 7.40, automatic - 7.70.

Observations and explanations: The CoT technique produced the best results in its category due to the careful development of the timeframe, clear structure, and comparative approach. The prompt is longer than the others and describes the requirements in more detail, positively impacting the final scores, as shown in Figure 3.9. The main contribution to the high scores was the high level of specificity and the requirement for a wide range of sources, which ensured good credibility of the information. Comments were related to some data selectivity and generalizations, but the structure and depth of analysis overrode these shortcomings.

Statistical data analysis

Within the analytic research task, the following metrics are obtained for all models and techniques used:

	CoT	PC	ReAct
RMSE	0.775	0.806	0.922
QWK	0.673	0.629	0.536
Spearman	0.857	0.695	0.735
Pearson	0.837	0.711	0.698

Table 3.7: RMSE, QWK, and correlation coefficients for the GPT-4o model using CoT, PC, and ReAct techniques

	CoT	PC	ReAct
RMSE	0.671	0.922	0.548
QWK	0.766	0.482	0.712
Spearman	0.867	0.606	0.706
Pearson	0.856	0.591	0.74

Table 3.8: RMSE, QWK, and correlation coefficients for the O3-mini model using CoT, PC, and ReAct techniques

	CoT	PC	ReAct
RMSE	0.775	0.775	0.806
QWK	0.61	0.706	0.594
Spearman	0.658	0.727	0.629
Pearson	0.648	0.746	0.621

Table 3.9: RMSE, QWK, and correlation coefficients for the Claude-3-7-sonnet-20250219 model using CoT, PC, and ReAct techniques

The gpt-4o CoT technique shows very high correlation coefficients and relatively small RMSE, although QWK is not the highest, as seen in the following Table 3.7. For o3-mini, CoT gives the highest correlation and the best QWK, while ReAct unexpectedly leads in RMSE but is slightly inferior in consistency, as shown in Table 3.8. For claude-3-7-sonnet-20250219, PC provides the best result with high correlations and QWK, as shown in the corresponding Table 3.9. ReAct does not produce maximum correlation values for any models, although o3-mini, for example, has minimal RMSE and quite decent coefficients. In general, CoT is noticeably leading for gpt-4o and o3-mini in terms of correlation, while PC becomes optimal for claude-3-7-sonnet-20250219.

3.3 Ratio of all scores for models and techniques

This part examines the range of final scores for each model and optimization technique in three different prompts. The analysis includes the minimum, maximum, and average to understand the degree of variation and the most frequent performance. It will also show how each combination of model and technique affects the level of responses and where the point of comfortable performance lies.

3.3.1 A programming task

The graphs for the programming task show a definite spread between the minimum and maximum scores in each model, with average scores ranging from 7-8 points and highs as high as 9 or 10, as shown in Figure 3.10. The gpt-4o with CoT and PC techs most often has a higher average score, with the spread between minimum and maximum values remaining relatively moderate. O3-mini sometimes shows a wider amplitude due to single low values, although the final average is often no lower than 7. Claude-3-7-sonnet-20250219 maintains close to 7-8 average values and has a moderate spread of minimum and maximum scores.

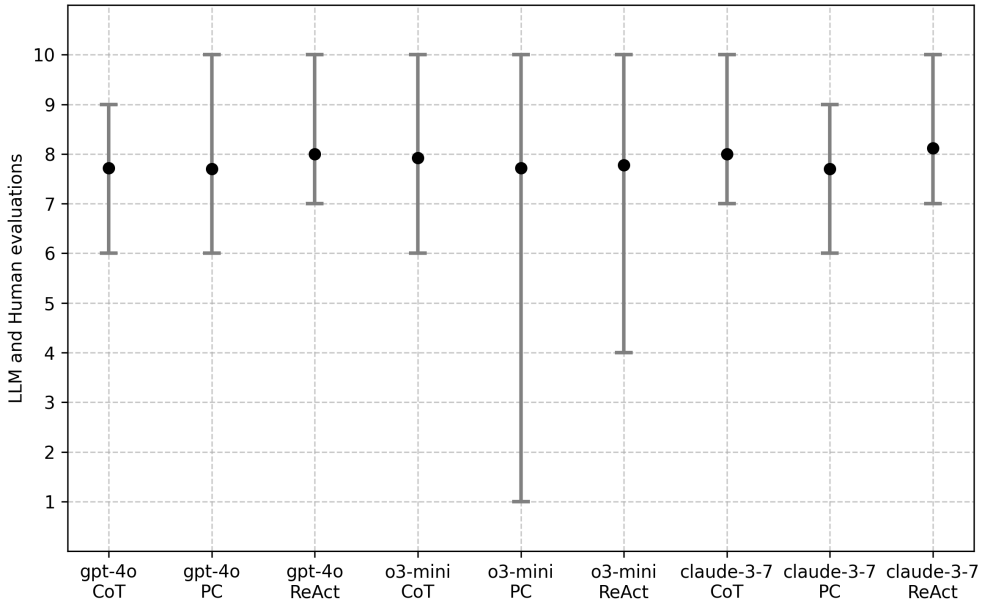

Figure 3.10: The error panel of the programming task illustrates, for each model+technique, a black dot denoting the average final score resulting from the human and LLM agent's evaluation of the 20 responses

3.3.2 A creativity task

In scenarios about a short movie about aliens, average scores lie in the 7-8 range, with upper limits going as high as 9 or 10, as shown in Figure 3.11. When using gpt-4o combined with CoT and PC, the results look the most even, with ReAct sometimes causing the minimum to drop significantly. The o3-mini and claude-3-7-sonnet-20250219 also show similar dynamics, although, in some places, the scatter of scores for the o3-mini is more significant due to individual low scores.

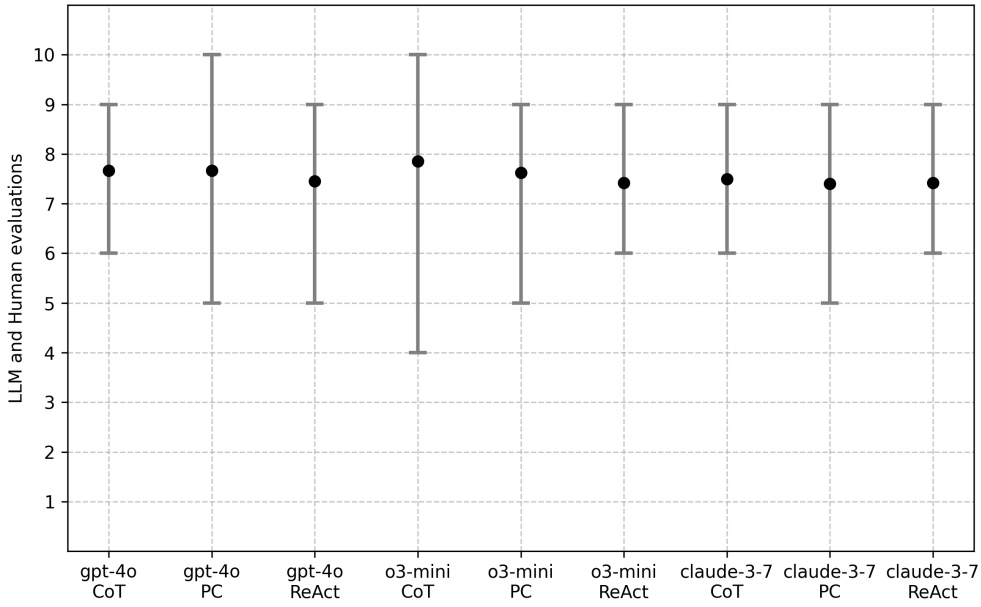

Figure 3.11: The error panel of the scenario task illustrates, for each model+technique, a black dot denoting the average final score resulting from the human and LLM agent's evaluation of the 20 responses

3.3.3 Analytical research task

In the analysis of the impact of remote work, the overall level of scores remains roughly the same, but the spread is sometimes wider, as shown in Figure 3.12. GPT-4o with CoT and PC techniques produces high averages of 7 to 8. With the o3-mini, the spread can also be noticeable, but the average score is still centered around seven and sometimes goes up to 8. The Claude-3-7-sonnet-20250219 holds a similar dynamic: the minimum values can fall to 5-6, the maximum reaches 9, and the average fluctuates around 7 points.

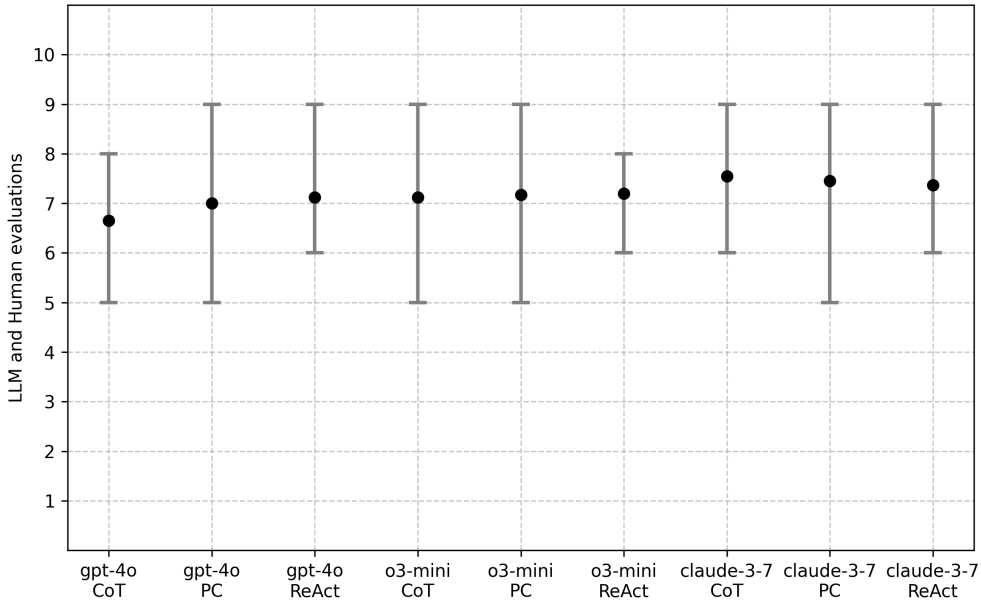

Figure 3.12: The error panel of the analytical research task illustrates, for each model+technique, a black dot denoting the average final score resulting from the human and LLM agent's evaluation of the 20 responses

4 Summary

This work consistently studied the architecture of LLMs and their peculiarities in forming and processing queries. It investigated how to build and optimize queries in detail and analyzed various methods of improving the accuracy and efficiency of interaction with LLM.

Based on the obtained knowledge, a holistic application is realized that automatically generates and improves input query texts using different techniques and models to achieve better answers. The main advantages include the system's flexibility, which allows adapting the algorithms to other classes of problems and language models, and the modular structure, which simplifies the integration of new generation and optimization techniques.

Experimental checks were carried out to validate the results, including quantitative metrics and qualitative analysis. The combination of different techniques showed the ability to elaborate the query generation logic more deeply, considering both formal requirements and contextual subtleties. The developed approach allowed to reduce the number of stages of interaction with models and improve the accuracy of answers. According to the final data, the methodology helps to save 1-2 rounds of revisions for each request and increase overall scores compared to the baseline requests. Optimized queries provide more valuable and accurate results without increasing processing time.

As prospects for development, we suggest adapting the system to specific domains where highly specialized knowledge is important in addition to linguistic aspects. Explore the possibility of integrating explainability tools to understand the internal reasons for changing model's response under different query formulations. Such a development could raise the efficiency of LLM interaction even higher, expanding the scope of application of automatic query optimization.

Bibliography

1. PI, Wenyi. *Brief Introduction to the History of Large Language Models (LLMs)*. 2024. Available also from: https://medium.com/@researchgraph/brief-introduction-to-the-history-of-large-language-models-llms-3c2efa517112. Accessed: 2025-01-13.
2. VASWANI, Ashish; SHAZEER, Noam; PARMAR, Niki; USZKOREIT, Jakob; JONES, Llion; GOMEZ, Aidan N.; KAISER, Lukasz; POLOSUKHIN, Illia. *Attention Is All You Need*. 2023. Available from arXiv: 1706.03762 [cs.CL].
3. BROWN, Tom B.; MANN, Benjamin; RYDER, Nick; SUBBIAH, Melanie; KAPLAN, Jared; DHARIWAL, Prafulla; NEELAKANTAN, Arvind; SHYAM, Pranav; SASTRY, Girish; ASKELL, Amanda; AGARWAL, Sandhini; HERBERT-VOSS, Ariel; KRUEGER, Gretchen; HENIGHAN, Tom; CHILD, Rewon; RAMESH, Aditya; ZIEGLER, Daniel M.; WU, Jeffrey; WINTER, Clemens; HESSE, Christopher; CHEN, Mark; SIGLER, Eric; LITWIN, Mateusz; GRAY, Scott; CHESS, Benjamin; CLARK, Jack; BERNER, Christopher; MCCANDLISH, Sam; RADFORD, Alec; SUTSKEVER, Ilya; AMODEI, Dario. *Language Models are Few-Shot Learners*. 2020. Available from arXiv: 2005.14165 [cs.CL].
4. OPENAI et al. *GPT-4 Technical Report*. 2024. Available from arXiv: 2303.08774 [cs.CL].
5. DEVLIN, Jacob; CHANG, Ming-Wei; LEE, Kenton; TOUTANOVA, Kristina. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. Available from arXiv: 1810.04805 [cs.CL].
6. CHOWDHERY, Aakanksha; NARANG, Sharan; DEVLIN, Jacob; BOSMA, Maarten; MISHRA, Gaurav; ROBERTS, Adam; BARHAM, Paul; CHUNG, Hyung Won; SUTTON, Charles; GEHRMANN, Sebastian; SCHUH, Parker; SHI, Kensen; TSVYASHCHENKO, Sasha; MAYNEZ, Joshua; RAO, Abhishek; BARNES, Parker; TAY, Yi; SHAZEER, Noam; PRABHAKARAN, Vinodkumar; REIF, Emily; DU, Nan; HUTCHINSON, Ben; POPE, Reiner; BRADBURY, James; AUSTIN, Jacob; ISARD, Michael; GUR-ARI, Guy; YIN, Pengcheng;

DUKE, Toju; LEVSKAYA, Anselm; GHEMAWAT, Sanjay; DEV, Sunipa; MICHALEWSKI, Henryk; GARCIA, Xavier; MISRA, Vedant; ROBINSON, Kevin; FEDUS, Liam; ZHOU, Denny; IPPOLITO, Daphne; LUAN, David; LIM, Hyeontaek; ZOPH, Barret; SPIRIDONOV, Alexander; SEPASSI, Ryan; DOHAN, David; AGRAWAL, Shivani; OMERNICK, Mark; DAI, Andrew M.; PILLAI, Thanumalayan Sankaranarayana; PELLAT, Marie; LEWKOWYCZ, Aitor; MOREIRA, Erica; CHILD, Rewon; POLOZOV, Oleksandr; LEE, Katherine; ZHOU, Zongwei; WANG, Xuezhi; SAETA, Brennan; DIAZ, Mark; FIRAT, Orhan; CATASTA, Michele; WEI, Jason; MEIER-HELLSTERN, Kathy; ECK, Douglas; DEAN, Jeff; PETROV, Slav; FIEDEL, Noah. *PaLM: Scaling Language Modeling with Pathways*. 2022. Available from arXiv: `2204.02311 [cs.CL]`.

7. TOUVRON, Hugo; LAVRIL, Thibaut; IZACARD, Gautier; MARTINET, Xavier; LACHAUX, Marie-Anne; LACROIX, Timothée; ROZIÈRE, Baptiste; GOYAL, Naman; HAMBRO, Eric; AZHAR, Faisal; RODRIGUEZ, Aurelien; JOULIN, Armand; GRAVE, Edouard; LAMPLE, Guillaume. *LLaMA: Open and Efficient Foundation Language Models*. 2023. Available from arXiv: `2302.13971 [cs.CL]`.

8. YUE, Fengpeng; KO, Tom. An Investigation of Positional Encoding in Transformer-based End-to-end Speech Recognition. *2021 12th International Symposium on Chinese Spoken Language Processing (ISCSLP)*. 2021, pp. 1–5. Available also from: `https://api.semanticscholar.org/CorpusID:232152116`.

9. BA, Jimmy Lei; KIROS, Jamie Ryan; HINTON, Geoffrey E. *Layer Normalization*. 2016. Available from arXiv: `1607.06450 [stat.ML]`.

10. GEVA, Mor; SCHUSTER, Roei; BERANT, Jonathan; LEVY, Omer. *Transformer Feed-Forward Layers Are Key-Value Memories*. 2021. Available from arXiv: `2012.14913 [cs.CL]`.

11. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. *Deep Residual Learning for Image Recognition*. 2015. Available from arXiv: `1512.03385 [cs.CV]`.

12. PARCALABESCU, Letitia; TROST, Nils; FRANK, Anette. *What is Multimodality?* 2021. Available from arXiv: `2103.06304 [cs.AI]`.

13. LOU, Renze; ZHANG, Kai; YIN, Wenpeng. *Large Language Model Instruction Following: A Survey of Progresses and Challenges*. 2024. Available from arXiv: `2303.10475 [cs.CL]`.

14. WEI, Jason; TAY, Yi; BOMMASANI, Rishi; RAFFEL, Colin; ZOPH, Barret; BORGEAUD, Sebastian; YOGATAMA, Dani; BOSMA, Maarten; ZHOU, Denny; METZLER, Donald; CHI, Ed H.; HASHIMOTO, Tatsunori; VINYALS, Oriol; LIANG, Percy; DEAN, Jeff; FEDUS, William. *Emergent Abilities of Large Language Models*. 2022. Available from arXiv: 2206.07682 [cs.CL].

15. WETZEL, Linda. Types and tokens. 2006.

16. QIN, Yanzhao; ZHANG, Tao; ZHANG, Tao; SHEN, Yanjun; LUO, Wenjing; SUN, Haoze; ZHANG, Yan; QIAO, Yujing; CHEN, Weipeng; ZHOU, Zenan; ZHANG, Wentao; CUI, Bin. *SysBench: Can Large Language Models Follow System Messages?* 2024. Available from arXiv: 2408.10943 [cs.CL].

17. LIU, Nelson F.; LIN, Kevin; HEWITT, John; PARANJAPE, Ashwin; BEVILACQUA, Michele; PETRONI, Fabio; LIANG, Percy. *Lost in the Middle: How Language Models Use Long Contexts*. 2023. Available from arXiv: 2307.03172 [cs.CL].

18. ROMERA-PAREDES, Bernardino; TORR, Philip H. S. An embarrassingly simple approach to zero-shot learning. In: *International Conference on Machine Learning*. 2015. Available also from: https://api.semanticscholar.org/CorpusID:5891792.

19. MIN, Sewon; LYU, Xinxi; HOLTZMAN, Ari; ARTETXE, Mikel; LEWIS, Mike; HAJISHIRZI, Hannaneh; ZETTLEMOYER, Luke. *Rethinking the Role of Demonstrations: What Makes In-Context Learning Work?* 2022. Available from arXiv: 2202.12837 [cs.CL].

20. LI, Cheng; WANG, Jindong; ZHANG, Yixuan; ZHU, Kaijie; HOU, Wenxin; LIAN, Jianxun; LUO, Fang; YANG, Qiang; XIE, Xing. *Large Language Models Understand and Can be Enhanced by Emotional Stimuli*. 2023. Available from arXiv: 2307.11760 [cs.CL].

21. WEI, Jason; WANG, Xuezhi; SCHUURMANS, Dale; BOSMA, Maarten; ICHTER, Brian; XIA, Fei; CHI, Ed; LE, Quoc; ZHOU, Denny. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. Available from arXiv: 2201.11903 [cs.CL].

22. WANG, Lei; XU, Wanyu; LAN, Yihuai; HU, Zhiqiang; LAN, Yunshi; LEE, Roy Ka-Wei; LIM, Ee-Peng. *Plan-and-Solve Prompting: Improving Zero-Shot Chain-of-Thought Reasoning by Large Language Models*. 2023. Available from arXiv: 2305.04091 [cs.CL].

23. WANG, Xuezhi; WEI, Jason; SCHUURMANS, Dale; LE, Quoc; CHI, Ed; NARANG, Sharan; CHOWDHERY, Aakanksha; ZHOU, Denny. *Self-Consistency Improves Chain of Thought Reasoning in Language Models*. 2023. Available from arXiv: 2203.11171 [cs.CL].

24. ADAMS, Griffin; FABBRI, Alexander; LADHAK, Faisal; LEHMAN, Eric; EL-HADAD, Noémie. *From Sparse to Dense: GPT-4 Summarization with Chain of Density Prompting*. 2023. Available from arXiv: 2309.04269 [cs.CL].

25. YAO, Shunyu; ZHAO, Jeffrey; YU, Dian; DU, Nan; SHAFRAN, Izhak; NARASIMHAN, Karthik; CAO, Yuan. *ReAct: Synergizing Reasoning and Acting in Language Models*. 2023. Available from arXiv: 2210.03629 [cs.CL].

26. ZHOU, Yongchao; MURESANU, Andrei Ioan; HAN, Ziwen; PASTER, Keiran; PITIS, Silviu; CHAN, Harris; BA, Jimmy. *Large Language Models Are Human-Level Prompt Engineers*. 2023. Available from arXiv: 2211.01910 [cs.LG].

27. SHIN, Taylor; RAZEGHI, Yasaman; IV, Robert L. Logan; WALLACE, Eric; SINGH, Sameer. *AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts*. 2020. Available from arXiv: 2010.15980 [cs.CL].

28. AGARWAL, Eshaan; SINGH, Joykirat; DANI, Vivek; MAGAZINE, Raghav; GANU, Tanuja; NAMBI, Akshay. *PromptWizard: Task-Aware Prompt Optimization Framework*. 2024. Available from arXiv: 2405.18369 [cs.CL].

29. YANG, Chengrun; WANG, Xuezhi; LU, Yifeng; LIU, Hanxiao; LE, Quoc V.; ZHOU, Denny; CHEN, Xinyun. *Large Language Models as Optimizers*. 2024. Available from arXiv: 2309.03409 [cs.LG].

30. GUO, Qingyan; WANG, Rui; GUO, Junliang; LI, Bei; SONG, Kaitao; TAN, Xu; LIU, Guoqing; BIAN, Jiang; YANG, Yujiu. *Connecting Large Language Models with Evolutionary Algorithms Yields Powerful Prompt Optimizers*. 2024. Available from arXiv: 2309.08532 [cs.CL].

List of Appendixes

A User Documentation

A.1 User Interface

The application provides a structured single-page user interface that facilitates registration, login, optimization, evaluation, blind testing, and comparison of queries using various artificial intelligence models and prompts. To simplify the user's experience with the site and its features, intuitive interface forms, clear navigation links, and interactive elements have been created and designed to facilitate effective user interaction and optimal workflow.

A.2 Styling and Routing

The application's main navigation structure is built with well-defined routes for each page. Navigation links are located at the top left from the optimization interface sidebar, allowing users to make smooth and quick transitions between pages. When the user hovers the cursor over a link to the desired page, it is highlighted to let the user know that it is a link. Having reached another page, the user can return to the previous page using the same navigation bar.
The interface is designed in a unified, modern style to provide a clear, readable, professional look. Key visual elements include clearly defined sections, center-aligned headers, interactive buttons with intuitive icons, and a visual spinner during loading. Cross-page consistency in terms of spacing, typography, and interactive states improves overall user comfort even further. The comprehensive interface design emphasizes simplicity of use and efficiency of workflow management.

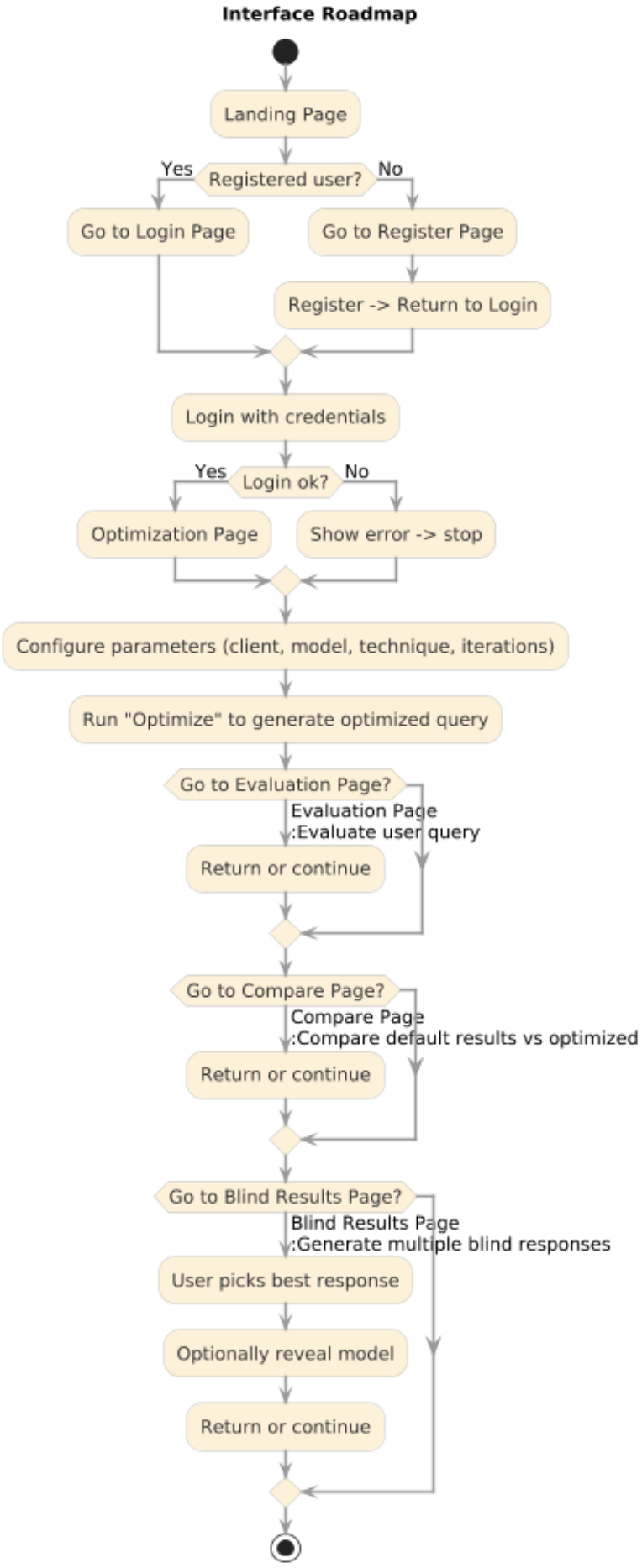

Figure A.1: User interface roadmap

A.3 User Authentication

The registration page provides the user with an easy-to-understand and simple form to create an account, requiring them to enter their full name, email, and password. After successful registration, the user will be redirected to a login page that requires an email and password for authentication. Successful login securely stores user data, retrieves user information such as full name, and takes the user to the optimization page to start using the application features.

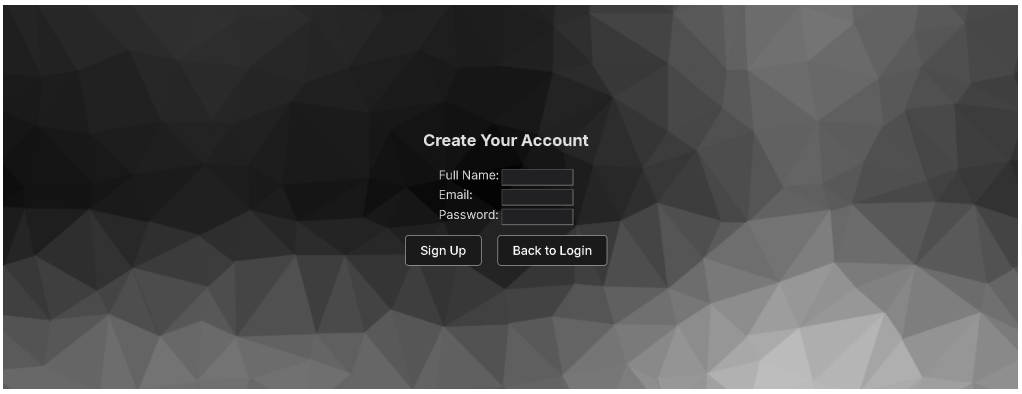

Figure A.2: Register page

A.4 The Optimization Page

Successfully logged in, the user is taken to the optimization page, which is noticeably divided into two main sections. The left sidebar welcomes the user and allows the user to select the AI provider, specific versions of the AI models, the optimization technique, and the number of iterations for the iterative technique. On the right side is a text area where the user enters his query for optimization. When the user clicks on the "Optimize" button, the query will be processed using the selected AI configurations and presented as an optimized query with the ability to copy the optimized version of the query and download it to see the model's thought process during optimization using the selected technique. Once a result is obtained, two buttons are automatically activated - one to add an expert persona and one to set an emotional stimulus. Additional options can be added to the optimized version of the query.

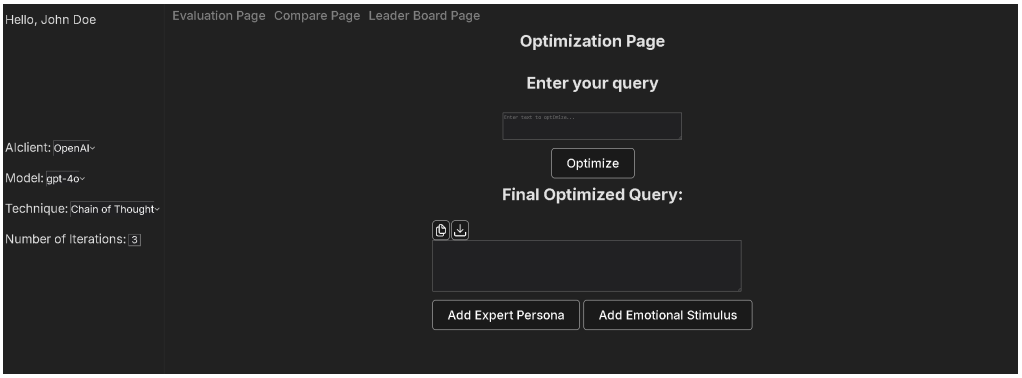

Figure A.3: Optimization page

A.5 The Evaluation Page

The evaluation page is accessible directly from the optimization page and has a two-column layout similar to the other pages. The user selects the AI provider, model, and evaluation method. The interface contains a text input area for queries, allowing the user to see the prompt evaluation and reasons after clicking the "Evaluate" button. The evaluation and reasons are clearly displayed in a read-only output format.

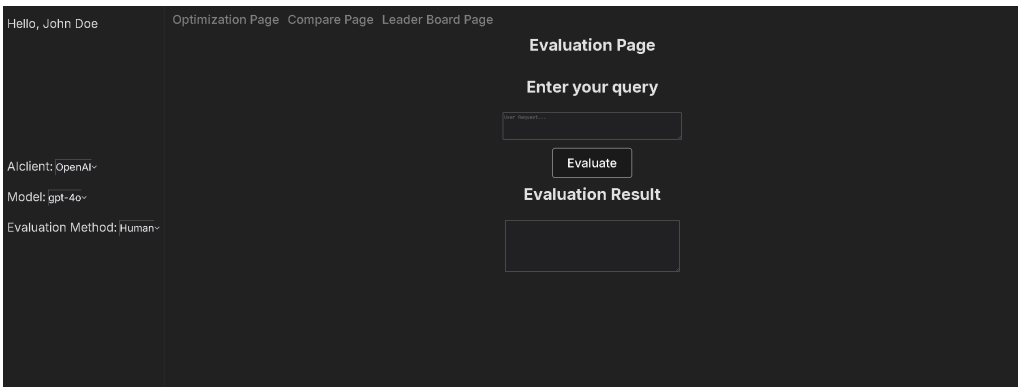

Figure A.4: Evaluation page

A.6 The Compare Page

By accessing the comparison page, the user can directly compare the results of the original and optimized prompts. The user selects a client and an AI model, enters the initial and optimized queries, and, by clicking the "Execute" button, sees the responses from the model, which are visually represented in neighboring fields that can be copied by clicking the "Copy" icon. Users can then optionally provide

feedback on whether the optimized query results are better, worse, or tied for statistics and future improvements in optimization techniques.

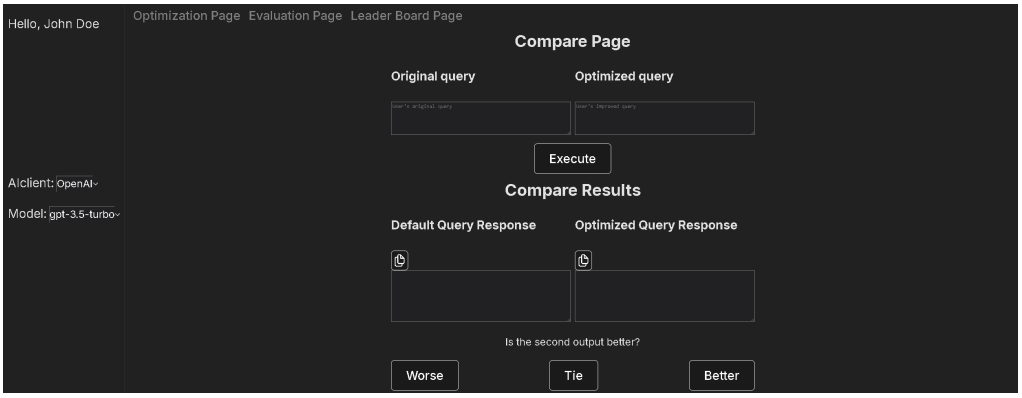

Figure A.5: Compare page

A.7 Blind result Page

The Blind Results page allows users to get different results from the prompts without directly identifying the AI model. The user enters a prompt, selects the number of desired responses he wants to generate, and clicks the "Generate Blind Results" button. The interface displays anonymized results from different models in clearly labeled fields. The user selects a preferred result and clicks on the selection box below. The interface displays the names of the models underlying the selected responses, supporting unbiased comparative analysis.

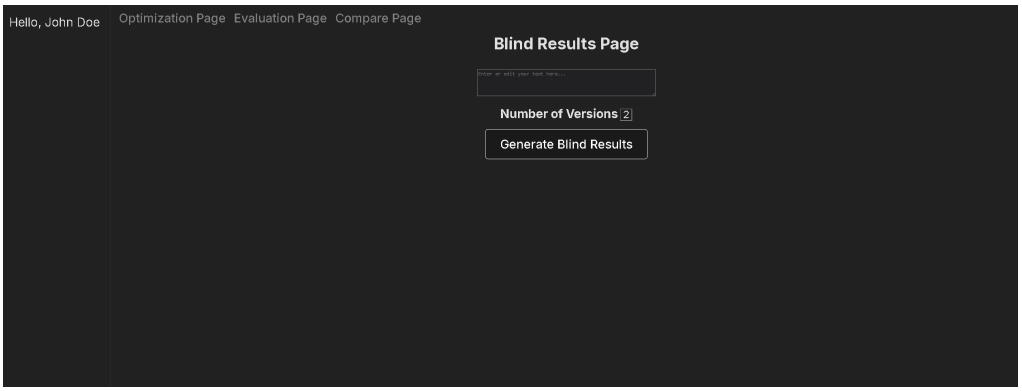

Figure A.6: Blind result page

B System Documentation

B.1 Launch Options

To start using the application, follow these steps:

1. Launch the client-side application by navigating to the '/frontend' folder and executing the following commands:

   - npm install
   - npm install vite
   - npm run dev

   This will install all required packages and start the development server on port 5173 or 5174. Once complete, you can access the client-side application in your browser at the address displayed in the terminal.

2. Install the necessary Python dependencies. After installing the dependencies, start the FastAPI server using uvicorn:

   - pip install -r requirements.txt
   - uvicorn backend.db.main:app

   The server will listen on local port 8000 for incoming requests, process prompt optimizations and evaluations, and communicate with the MongoDB database.

B.2 System main components

B.2.1 Evaluator class

This class is used to evaluate the input user query. It can operate in two modes:

- "human" – the criteria come from human.
- "llm" – the LLM automatically forms the criteria.

Main Fields

- **user_query**: the user's original query text.
- **provider**: the provider name openai or claude.
- **client**: an instance of AIClient for the chosen provider.
- **model**: the specific model to be used.
- **human_evaluation**: a flag that specifies which type of evaluation is used.
- **prompts**: a dictionary of prompt file paths.
- **evaluation_result**: the evaluation result returned after executing evaluation function.
- **optimized_user_query**: the optimized version of the query used for comparison.
- **parsed_result_after_comparison**: the result of comparing the original query and the optimized query.
- **blind_results**: a list of blind results from different models.

Main Methods

1. *async def evaluate*(): the corresponding prompt template human or llm, renders it, sends it to the chosen LLM model, and parses the response. Returns the evaluation result as a dictionary.
2. *async def compare*(): asynchronously invokes the model with two query variants and collects the results. Returns a dictionary containing the comparison of both variants.
3. *async def generate_blind_results*(*user_text, num_versions*): generates multiple blind responses from different models. Returns a list of dictionaries.

B.2.2 AutomatedRefinementModule Class

This class optimizes the user's original query step by step. It supports multiple queries, each with its own prompt template.

Main Fields

- **user_query**: the user's original query text.

- **provider**: the provider name openai or claude.
- **client**: an instance of AIClient for the chosen provider.
- **model**: the specific model to be used.
- **human_evaluation**: a flag that specifies which type of evaluation is used.
- **prompts**: a dictionary of prompt file paths.
- **max_iterations**: the number of iterations for the optimization.
- **emotional_stimuli**: a list of emotional stimulus phrases.
- **expert_persona_text**: text describing that the LLM is an expert.
- **emotional_stimuli_text**: a random string from emotional_stimuli.
- **final_optimized_query**: the final optimized query result.
- **raw_output**: the raw result of the model's output.
- **is_optimizing**: a flag preventing parallel optimizations.
- **is_expert_present**: a flag indicating whether an expert was found.

Main Methods

1. *def expert_finder*(): loads the "expert_finder" prompt, sends it to the model, and checks whether the response contains "Expert". Sets the is_expert_present flag. Returns the expert data if available.
2. *def optimize_query*(*selected_technique, iterations*): It retrieves and renders the corresponding prompt depending on the chosen technique. Based on the JSON structure of the response, the optimized query is saved in the final_optimized_query. Returns the raw JSON result.

B.2.3 Abstract Class AIClient

Defines abstract method call_chat_completion for calling an LLM model.
Main Method

1. *def call_chat_completion*(*model, messages*): an abstract method that should invoke the appropriate endpoint to generate a response from a model based on a list of messages.

B.2.4 OpenAIClient class

Implements interactions with the OpenAI API.
Main Fields

- **client**: an OpenAI client initialized with the API key.
- **max_retries**: the number of retry attempts.
- **backoff_factor**: the coefficient used for exponential backoff.

Method

1. *def call_chat_completion*(*model, messages*): builds parameters for the call. Performs attempts up to max_retries to call the OpenAI API. Returns the string response from the model.

B.2.5 AnthropicClient class

Implements interactions with the Anthropic API.
Main Fields

- **client**: an Anthropic client, initialized with the API key.
- **max_retries**: the number of retry attempts.
- **backoff_factor**: the coefficient used for exponential backoff.

Method

1. *def call_chat_completion*(*model, messages*): similarly calls the Anthropic API. Parses the response and returns the text.

B.2.6 Utils

Below is a brief list of utils and their roles:

1. **config.yaml**
    - This file contains the default provider, API keys for OpenAI/Claude, the list of available models, paths to prompt templates, and database settings.
2. **config.py**

- The *load_config*(*filepath*) function loads the YAML configuration and returns a dictionary.
- The *get_api_key*(*provider, config*) function retrieves the appropriate provider key from the environment or config.yaml.

3. **ai_client_factory.py**
   - The factory method *get_ai_client*(*provider*) returns the appropriate client instance of OpenAIClient or AnthropicClient.
4. **validators.py**
   - The functions *validate_required_fields*(*data, required_fields*) and *validate_provider_and_model*(*provider, model*) for validating input data. They call *handle_http_exception*(*status_code, detail*) on error.
5. **path_utils.py**
   - The *resolve_path*(*filepath*) function for getting the correct absolute path for file.
6. **render_prompt.py**
   - The functions *load_prompt*(*prompt_path*), *render_prompt*(*prompt_text, context*), *load_and_render_prompt*(*prompt_path, context*), and *build_user_message*(*rendered_prompt*) to read a template, render it, and form an array of messages.
7. **auth_dependency.py**
   - Implements JWT token checks, extracting user_id from the token and verifying expiration. Raises exception if invalid.
8. **http_error_handler.py**
   - The *handle_http_exception*(*status_code, detail*) function raises an HTTPException with a given code and message.
   - The *handle_generic_exception*(*error, status_code*) function is a general handler for any exceptions, logging them and raising an HTTPException.
9. **prompt_parser_validator.py**
   - Contains functions *clean_json*(*json_str*), *extract_json_from_response*(*content*) for extracting proper JSON from LLM responses.

10. **main.py**

    - This is the FastAPI application entry point. It creates the FastAPI object, configures CORS and routers for different functionalities, and initializes MongoDB.

11. **settings.py**

    - Loads the config, from which **MONGO_URI** and **DB_NAME** are retrieved.

12. **db.py**

    - Initializes global variables for the Motor client and the database reference.
    - The *init_db*(), *get_database*(), and *close_db*() functions manage the MongoDB connection.