



**FACULTY  
OF MATHEMATICS  
AND PHYSICS**  
Charles University

**BACHELOR THESIS**

Mikoláš Fromm

**Design of LLM prompts for iterative  
data exploration**

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Tomáš Petříček, Ph.D.

Study programme: Informatics

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date .....

Author's signature

I would like to thank the thesis supervisor, Mgr. Tomáš Petříček, Ph.D., for his guidance in the creation of this thesis. The collaboration was greatly beneficial to me, even beyond the content of this text. I would also like to thank the Department of Distributed and Dependable Systems for providing the resources for experimental testing and doc. Mgr. Robert Šámal, Ph.D. for his consultation.

Title: Design of LLM prompts for iterative data exploration

Author: Mikoláš Fromm

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D., Department of Distributed and Dependable Systems

Abstract: For many data exploration problems such as constructing a database query or selecting from a large data source, it can be difficult to validate the correctness of the final result. However, these structures can be more easily checked if they are created iteratively, especially when we consider that the situation can be represented as a navigation through a space of all available options. In this thesis, we make two contributions related to the problem. We implement such an iterative approach in a web-based data exploration tool, enriched with an LLM-based assistant helping the user to select the best next step based on their entered query. Since the way an LLM is prompted significantly affects LLM's responses, we conduct an analysis that evaluates the best way of constructing LLM prompts for such scenarios. This work therefore contributes to the problem of optimizing LLM utility through prompt engineering, offering a foundation for further enhancements in user-assistance tools in data analytics.

Keywords: large language model, LLM prompting, iterative data exploration, assisted query generation, prompting strategies comparison

Název práce: Návrh LLM promptu pro iterativní data exploraci

Autor: Mikoláš Fromm

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Pro mnoho problémů spojených s prozkoumáváním dat, jako je vytváření databázových dotazů či vybírání objektů z velkého datového skladu, může být těžké ověřit správnost výsledného dotazu či výběru. Databázové dotazy a výběry z datových skladů však lze snadněji kontrolovat, pokud jsou vytvářeny iterativně, zejména pokud vezmeme v potaz, že lze situaci chápat jako navigaci prostorem všech dostupných možností. V této práci přinášíme dva příspěvky související s daným problémem. Implementujeme zmíněný iterativní přístup do webového nástroje pro průzkum dat, obohacený o LLM asistenta, který pomáhá uživateli vybrat nejlepší další krok na základě dotazu zadaného na začátku procesu průzkumu dat. Vzhledem k tomu, že způsob, jak je LLM promptováno, výrazně ovlivňuje správnost generovaných asistencí, provádíme analýzu, která vyhodnocuje nejlepší způsoby konstrukce LLM promptu pro účely exploračního prostoru možností. Tato práce proto přispívá ke zvýšení užitečnosti LLM v praktických aplikacích, nabízející základy pro další vylepšení v oblasti asistenčních nástrojů používaných pro datovou analýzu.

Klíčová slova: velký jazykový model, LLM prompting, iterativní data explorace, asistované generování dotazů, porovnání promptovacích strategií

# Contents

<b>Introduction</b>	<b>7</b>
<b>1 Background</b>	<b>9</b>
1.1 The Gamma . . . . .	9
1.1.1 Downsides . . . . .	10
1.1.2 Inspirations . . . . .	10
1.2 A prompt Pattern Catalogue . . . . .	11
1.3 Iterative Prompting Assistant . . . . .	12
<b>2 Iterative Prompting Assistant - C# Implementation</b>	<b>15</b>
2.1 Description . . . . .	15
2.2 Architecture . . . . .	16
2.2.1 WhisperService . . . . .	16
2.2.2 Controller . . . . .	16
2.2.3 IQueryAgent . . . . .	17
2.2.4 ICommunicationAgent . . . . .	17
2.2.5 Interfaces and Datatypes . . . . .	19
2.3 Communication with an LLM . . . . .	20
2.4 Case study . . . . .	22
<b>3 Constructing better prompts</b>	<b>26</b>
3.1 Motivation . . . . .	26
3.2 Framework description . . . . .	26
3.2.1 Key Components and Methodology . . . . .	27
3.3 Hypotheses . . . . .	28
3.3.1 Expanding Context Increases Success . . . . .	28
3.3.2 Reason for Higher Success . . . . .	28
3.3.3 Model Invariance . . . . .	28
3.3.4 Language Invariance . . . . .	29
3.4 Methodology . . . . .	29
3.4.1 Data Simulation . . . . .	29
3.4.2 Prompt Engineering . . . . .	29
3.4.3 Testing . . . . .	31
3.4.4 Evaluation . . . . .	34
3.5 Results . . . . .	36
3.5.1 Expanding Context Increases Success . . . . .	36
3.5.2 Better but bigger . . . . .	37
3.5.3 Reason for Higher Success . . . . .	39
3.5.4 Model Invariance . . . . .	40
3.5.5 Language Invariance . . . . .	41
3.6 Threats to validity . . . . .	44
<b>4 Conclusion</b>	<b>45</b>
<b>Bibliography</b>	<b>46</b>

<b>List of Figures</b>	<b>47</b>
<b>List of Tables</b>	<b>48</b>
<b>A Attachments</b>	<b>49</b>
A.1 C# WebApplication Source Code . . . . .	49
A.2 Python Testing Framework (with results) . . . . .	49

# Introduction

In this thesis, we will explore two distinct worlds: the world of large language models (LLMs) and the world of data exploration. At a first glance, these worlds may seem separate, but with the increasing capabilities of LLMs, the worlds begin to interact. Specifically, we will investigate how an exploration of a dataset, understood as a broadly conceived *space of all available options*, can be more readily accomplished by a user with the help given by an LLM.

LLMs are at the forefront of natural language processing (NLP), representing a significant technological advancement in artificial intelligence. These models are built using deep learning algorithms, particularly variants of transformer architectures, which allow them to process and generate text in ways that are contextually and semantically rich. By training on extensive datasets containing diverse text sources—such as books, websites, and academic papers—LLMs learn the intricate patterns of language, enabling them to perform tasks like text summarization, question answering, completion, and more.

LLMs nowadays are increasingly becoming part of a wide range of user applications to streamline user work or enhance the application environment. The currently most widespread family of conversational models, OpenAI’s ChatGPT, demonstrates that their use significantly extends beyond the realm of meaningless conversations, and their generative capabilities are often better in limited domains compared to humans. [1, 2, 3]. However, it is not always straightforward and clear how to communicate with such a model, especially how to achieve a specific format of response or how to deliver the necessary context for understanding the user’s assignment.

The answer to the mentioned problem is an *LLM prompting*, which can influence the final form and quality of an LLM response. Therefore, an *LLM prompt* can be understood as an extension to the bare user assignment, which might add rules how the response should look like, help to interpret next user inputs, or describe the context in which the agent is situated.

Although current LLMs are capable of generating relatively complex exploratory scripts, whether it be SQL queries or a short program in Python or R, it is very often difficult for users, especially in more complicated cases, to decide whether the script is valid considering original user query. Therefore, an *iterative data exploration* method could be proposed, emphasizing the way the data set is traversed and how information from the data set is obtained.

When using this method, the user builds the final script or query sequentially through repeated selections from all possible atomic operations that can be performed at the given point. Not only is the user acquainted with all available steps through this approach, but it is also assumed that the user is better able to decide whether the atomic operation they perform on the data set is appropriate.

Having such iterative exploration tool, especially in conjunction with LLM prompting, can be very useful for exploring data sets that may be either factually complicated in terms of size or semantics of the data or that require knowledge of a query language (such as SQL for relational databases) through which data can be explored.

This work aims to combine the aforementioned two directions into a single solution, which will serve as a proposal for a tool for easier work with data sets. The thesis will be divided into two parts, implementation and experimental.

In the implementation part, we will build a web application in the C# programming language operating over a virtual representation of tabular data. The application will use an LLM ChatGPT agent that will suggest the best next step to obtain a fitting result of the user's initial query. The goal of this part is to build a functional prototype of an iterative data exploration tool, serving as a representative of the entire class of such tools. The implementation will place an emphasis on generality of the solution, allowing the same code-base to be used in another projects.

In the experimental part, we will focus on the challenges of the LLM prompting and, based on the experiences gained from implementation of the iterative data exploration tool, we will further test selected approaches on how to design LLM prompting and how to maximize the agent's success in finding the correct next step, where the prompting will be tested on different cases and combined with proposed procedures.

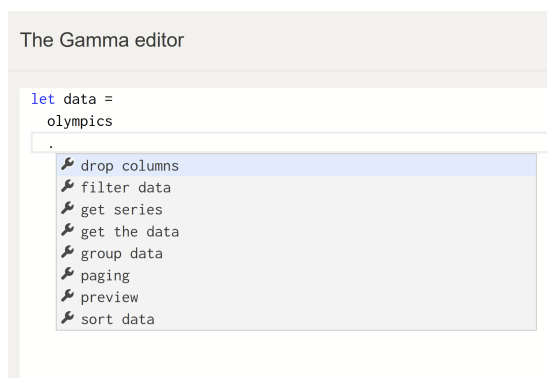


# 1 Background

In this chapter, we introduce sources that served as the main motivation for creating an interface with a custom assistant for iterative data exploration. We will highlight the key elements we consider important in the creation of our own implementation of such an assistant and demonstrate in which parts of the project it is appropriate to use an LLM assistant to increase the success of finding the final query.

## 1.1 The Gamma

*The Gamma* [4] is a data exploration environment specifically designed to make the power of programmatic data analysis accessible to non-programmers, particularly targeting users such as data journalists. It achieves this through the iterative prompting approach, where users construct simple, transparent, scripts by selecting from a range of code completions rather than writing their own code from scratch.



**Figure 1.1** An example of *The Gamma* offering all possible next moves after writing ”.” character.

This design significantly lowers the barrier to entry for data exploration, allowing users to effortlessly navigate and analyze data from various sources, without requiring prior programming knowledge. By transforming the complex process of data exploration into a series of simple, guided selections, *The Gamma* not only simplifies data analysis but also encourages a deeper engagement with data. Its emphasis on simplicity and reproducibility aligns with the broader goal of informed decision-making in data journalism and beyond. As such, *The Gamma* represents a critical advancement in making sophisticated data analysis tools more accessible, fostering a more data-literate society where users can easily leverage open data to uncover insights, tell compelling stories, and support factual claims.

Although this approach technically facilitates data exploration and potentially aids in decision-making about which option to choose, it still cannot help the user if, at a given step, it is impossible to confidently determine which choice is correct. This may not only be due to a lack of knowledge of the defined operations but also the nature of the source data. Since LLMs are capable of efficiently analyzing large volumes of text and making micro-decisions over that text [5], they seem like a suitable candidate for integration into such a system.

### 1.1.1 Downsides

The method of displaying all available options from a given state requires an internal definition of all possible operations generally applicable in the given context, regardless of the current position. However, for some query languages, such as SQL, it may be difficult to define all existing operations in the given language and, above all, transform them into a reasonable sequential form containing atomic elements. It is therefore quite understandable that such a system contains only a limited set of operations for which the data exploration is supported.

However, even if the system contains only a limited set of defined operations, there are scenarios where generating all the next options will be challenging. In some cases, it is desirable for the menu to be created dynamically, based on the current data. This data can be not only potentially unlimited in size but also time-consuming to obtain.

If we disregard the problem with the time-consuming nature of processing all the next options, its total size is a problem that remains present even in the case of using LLM, as their contextual window, in which communication between the user and the agent is done, is limited.

### 1.1.2 Inspirations

In our own project, we primarily want to adopt the iterative, dot separated, approach (*dot approach* in short) in constructing the final user query, which offers all subsequent available options at each step and, upon applying a given step, displays the current state of the queried data source. Especially the first mentioned feature is key for engaging an LLM agent, as it clearly defines the available selection of all options from which the LLM agent can be asked to pick the most suitable option regarding the initial user query.

## 1.2 A prompt Pattern Catalogue

For a successful integration of LLM prompted hints into a data exploration tool with the aim of helping the user, it is crucial to carefully select how to construct the prompt and ensure the highest success rate in processing the assistant’s response. In the work *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT* [6], a rich catalogue of generically structured approaches to prompting with LLMs was proposed. Some of the approaches are also relevant to our work and will therefore be adopted in our solution. Namely, these approaches include:

1. **Persona Pattern:** The assistant will be introduced in its role as an assistant, helping to select the best possible choice from the given selection with respect to the user’s intent and previous choices. From this prompt pattern, we expect that the assistant will be better placed in the context of the application and will not generate machine-unprocessable and undesirable responses, such as suggestions of random source codes solving the user’s query. The following example illustrates, how an assistant could be introduced to a situation of helping user navigate through Eurostat database.

*You are an assistant that is helping a user to navigate through the set of tables from the Eurostat database. You will be given a query and all available subsections. You should help the user to navigate to the correct subsection.*

2. **The Template Pattern:** Since the user will not interact with the LLM directly, but only indirectly through the proposed best response from the given option set, it is important to ensure that the LLM produces formatted output according to predetermined rules, based on which the generated output can be processed and evaluated. The example below illustrates how to give LLM a specification of a template that will be used to communicate together.

*You will be given a list of all available options, each associated with unique index. Answer only with the number of the subsection you think is the best fitting. If you don't know the answer, return -1.”*

The other introduced patterns seem to be no less interesting for the purpose of obtaining guidance in selecting the right option from the list of all possible operations over the data set, yet they mostly require a human recipient, who is capable of understanding more complex and indirect structures of guidance. An example of such an approach could be the *Flipped Interaction Pattern* defined as follows:

I would like you to ask me questions to achieve X.
You should ask questions until this condition is met or to achieve this goal (alternatively, forever).
(Optional) ask me the questions one at a time, two at a time, etc.

**Table 1.1** Definition of *Flipped Interaction Pattern* [6, page 6]

Which could look in the context of data exploration as follows:

<i>context</i>	<i>User wants to find the average age of customers in a certain database table...</i>
User:	Ask me questions so that you can find what is the average age of all customers in our database system.
Assistant:	What is the name of the database table?
User:	...
Assistant:	What are the column names in the table?
User:	...
Assistant:	<i>Other related questions...</i>
User:	...
Assistant:	The query you are looking for might look like:  <pre>SELECT AVG(age) AS average_age FROM customers;</pre>

**Table 1.2** An example use of *Flipped Interaction Pattern* for data exploration.

It is clear from this example that this approach is almost impossible to automate in the background of a data exploration tool, as it requires direct user responses and at the same time produces the final answer in any format with any additional comments.

### 1.3 Iterative Prompting Assistant

Consider the following prompt for this subsection:

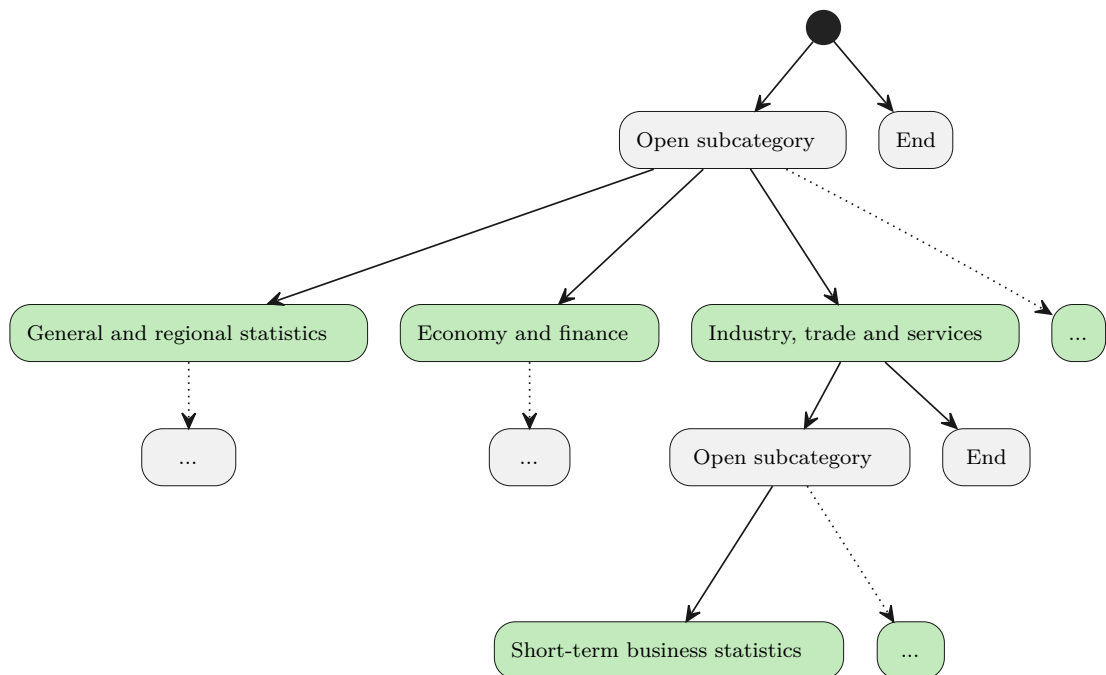
*I am looking for a table with an overview of the year-on-year inflation rate development for the Federal Republic of Germany.*

We are a user who wants to find the correct table in the Eurostat database [7]. This database has systematically structured content, where values are stored in tables that are thematically classified into structures reaching a maximum depth of more than 10 subcategories.

```
Eurostat
- Detailed datasets
  ...
- Selected datasets
  ...
  - Economy and finance
    - Government statistics
      ...
    ...
  ...
...
```

As a user seeing this database for the first time and having no experience with how the database is structured, any help with finding our table would be appreciated. The only option the user has is to instinctively enter various categories. The user either is lucky enough and finds the desired table or reaches a point where the user is sure that the path is wrong and tries another category. If the user is familiar with the subcategory names and understands their meaning, the user has a relatively high chance of successfully finding the desired table. However, if the user knows almost or absolutely nothing about the topic, then any help from an LLM assistant can be a valuable contribution. Let's therefore define, how to represent such situation.

Any such environment that can be iteratively navigated can be understood as a decision tree, where a vertex represents the current state and an edge represents the operation of transitioning to the next state. Typically, there will be as many edges as there are operations the user can perform at the vertex (state). If we had to represent more complex operations, than just the operation *open subcategory* with one argument *name of the subcategory* for example, we would break down such an operation into atomic units so that the above-mentioned formalism can again be applied.



**Figure 1.2** Tree representation of the iterative approach in data selection environment.

In Figure 1.2, we represent a situation in which we are in the initial empty state while having two options, *open new subcategory* or *end the generation*. We can also see that the *open subcategory* is effectively split into two atomic operations, first being the *chose an operation* (grey) and second being the *fill the category name* (green).

If we were to model such an environment for communication with an LLM agent, whose task would be to select the correct option, we must provide the assistant with the following input information:

1. User query,
2. path from the root to the current vertex,
3. edges leading from the current vertex.

The description of the current situation is thus expanded by the original user query and the already visited vertices, representing the *path* through the graph to the current position. These *metadata* must be attached to all prompts. Not only can it be assumed that it would be difficult for the LLM to search in a long history of prompts which operations have already been performed and to orient itself correctly, but such a history could, in more complex examples, exceed the maximum volume of context insertable into the LLM to get a response. Each single prompt is therefore a standalone unit with which the LLM agent should be able to choose the next step, without having access to the previous prompts.

The prompt describing the tree illustration from Figure 1.2 could thus look as follows:

---

User query:	<i>I am looking for a table with producer prices in industry for the Federal Republic of Germany</i>
Path done:	'Open subcategory' . 'Industry, trade and services' . 'Open subcategory'
Next moves:	<ul style="list-style-type: none"> <li>&gt; Short-term business statistics</li> <li>&gt; Business structure and dynamics</li> <li>&gt; Globalisation in business statistics</li> <li>&gt; Tourism</li> <li>&gt; Statistics on the production of manufactured goods</li> </ul>

---

**Table 1.3** Prompt representation of the Image 1.2.

This is also the prompt design we implement in our Iterative Prompt Assistant solution in the following chapter.

## 2 Iterative Prompting Assistant - C# Implementation

In this chapter, we introduce an implementation (A.1) for the iterative prompt assistant. As a demonstration scenario, we simulate a basic query language on tabular data. We chose this scenario because it illustrates well the use of an LLM assistant for obtaining next-step hints. It also reveals the pitfalls associated with mapping the LLM assistant’s textual response to calls of specific procedures in the program code.

In the program, all available operations are statically defined, including their description, list of arguments, and their functions applicable to the data set. At each step of creating the final query by the user, the LLM agent is prompted in the background with a list of currently available options, along with the already performed history of operations, for advice on which of the following options is the best to perform. This scenario is relatively straightforward as long as it involves choosing one of the listed options. However, some operations also require an argument that cannot be selected from a pre-generated list, typically one word (e.g., filtering a data set by given value), and ensuring an acceptable response from the LLM for such operations is more complex.

Later in this chapter, we show in detail the approach chosen for communication with the LLM and introduce the pitfalls this approach brings.

We chose C# as the programming language for the web application due to the higher level of proficiency and familiarity with the language. This familiarity allowed for a smoother and more efficient development process, enabling us to focus on implementing features rather than learning a new language.

ASP.NET was selected as the development framework because it offers seamless integration with C#. Its compatibility with C# was crucial for the project, as it allowed for direct calls to C# functions from the web, aligning perfectly with the application’s requirements.

### 2.1 Description

The user interacts with the application via a web interface, which is written in HTML and controlled using JavaScript. In this environment, the user has the option to upload their file containing tabular data in the CSV format (along with specifying the delimiter of the file), enter their descriptive sentence about desired transformations of the data, iteratively create the final sequence of operations, and in the real-time view the (partially) transformed data. Operations are entered using a *dot approach*, meaning that each argument is expected to be separated by a dot and each dot is triggering the LLM suggester, which advises the user on the next step. In the background of this web application, the current input is divided into individual operations and the operations from the previous iterations are executed. Together with the currently available and numbered options, a new prompt is generated from the last remaining operation. It is also accompanied by a request for selecting the best option in regard to the user’s intent. Upon receiving a response, system tries to process it (assigning it to the correct operation

from the list) and, if successful, adds the operation to the collection of already entered operations. Thanks to that, the web application can serve a preview of the current transformed data at any time. At every step, the user is shown an offer of all available operations, along with the option highlighted as the most fitting by the LLM agent. This way, the user is able to conduct assisted iterative data exploration.

## 2.2 Architecture

When designing the architecture of the program, great emphasis was placed on the functional separation of the individual units of the program and on the possible implementation substitutability of the units participating in the creation of the final transformations.

### 2.2.1 WhisperService

The provider of the entire service is the `WhisperService` class. It is designed to be an application component with simple user API, allowing to easily insert the whispering service into an already running system. The main purpose of this class is to be an intermediary for queries captured from the running application (WebAPI controller [2.2.2] in our case). It internally stores a collection of all statically predefined operations and a collection of already processed operations to be able to provide a view into the current state of the data. When requested, it creates a view of the current data set after applying all captured operations. When a new user input is given, it divides the input into single operations and passes them to the query agent (2.2.3).

### 2.2.2 Controller

The application controller, which is tasked with receiving queries posed from the web interface, performs basic input data control, and calls the appropriate methods of the `WhisperService` class (2.2.1). For the purposes of our demo, the controller, together with the front HTML page, serves as a mediator between the real user calls and the `WhisperService`, even though the `WhisperService` would be inserted into an already existing project code base of the production environment otherwise.

The controller contains the following endpoints with their mentioned functions:

- POST: *api/whisper/upload-csv* (Uploading tabular data),
- POST: *api/whisper/upload-user-input* (Uploading user query),
- POST: *api/whisper/process* (Processing input, returning recommendation),
- GET: *api/whisper/get-current* (Getting the current view of the data).

Together with forwarding requests to the service class, it also handles simple validity checks of the input files and of the user query.



### 2.2.3 IQueryAgent

QueryAgent is the core part of the program. It is responsible for creating prompts with respect to the current situation and user input. It internally holds a reference to the current copy of the transformed data in order to be able to offer all the next available steps. When invoked, it sequentially goes through the individual operation names from the input and, until there are some left, it creates a "history" of already performed operations. This history is used to generate, accordingly to the current state, a new prompt asking for the best next move. It is also where response format validity checks are implemented and error handling is managed. While it is the core for creating the LLM prompts, the communication with a certain LLM is separated into the Communication Agent (2.2.4), ensuring that different LLMs can be used without the rest of the program being changed.

---

**Program 1** IQueryAgent interface definition, together with the final QueryViewModel.

---

```
public interface IQueryAgent
{
    public void AddUserQuery(string userQuery = null);
    public void StartNewQueryAttempt();
    public QueryViewModel PerformQuerying(IList<string> queryItems,
        IList<Field> fields);
}

public class QueryViewModel
{
    public int BotSuggestionIndex {...}
    public string BotSuggestion {...}
    public IEnumerable<ITransformation> Transformations {...}
    public IEnumerable<string> NextMoves {...};
    public void AddTransformation(ITransformation transf) {...}
    public string AddBotSuggestion(string suggestion,
        bool userInputExpected = false) {...}
}
```

---

### 2.2.4 ICommunicationAgent

The final link for communication with the LLM agent, enabling to connect our solution to any reasonable LLM. In our implementation, OpenAI API [8] was used to access OpenAI LLMs.

Two types of messages are used for communication with the LLM agent:

1. System message: This is the part of the prompt that is present in every user message in the conversation. It is therefore primarily suitable for familiarization with the context and understanding of the role. In our case, it is used for the *Persona pattern* mentioned in section 1.2.

2. User message: This is already the part of the prompt carrying the main information of the entire message. It does not adhere to any text formatting, since it is possible to communicate in any text format with LLMs. In our case, the *Template pattern* from section 1.2 is used for this type of messages.

---

**Program 2** Definition of ICommunication interface.

---

```
public interface ICommunicationAgent
{
    public bool Verbose { get; }
    public CommunicationAgentMode Mode { get; }

    public void FlushCurrentChat();
    public void AddUserQuery(string userQuery = null);

    public string InsertSystemMessage(string message);
    public string InsertUserMessage(string message);
    public string ErrorMessage(string message);

    public string GetResponse(string querySoFar = null,
        string nextMove = null, int nextMoveIndex = -1,
        bool isUserInputExpected = false);

    public void ShowConversationHistory();

    public IEnumerable<string> CreateNextQuestion(
        string question,
        IEnumerable<string> possibleChoices = null);
    ...
}
```

---

---

**Program 3** Definition of ITransformation interface.

---

```
public interface ITransformation
{
    public TransformationType Type { get; }
    public bool HasArguments { get; }
    public bool HasFollowingHumanArguments { get; }
    public int TotalStepsNeeded { get; }

    public List<Field> PerformTransformation(List<Field> input);
    public List<EmptyField> Preprocess(List<EmptyField> list);
    public IEnumerable<string> GetNextMoves(
        IEnumerable<EmptyField> fields);
    ...
}
```

---

## 2.2.5 Interfaces and Datatypes

Since one of the goals of the work is to create a tool that can simulate work with any data set, a custom structure, compatible with any defined transformation, was created to represent the data set.

As a result, *IQueryAgent* (Program 1) works with the *ITransformation* (Program 3) interface and the implementation is identical (and only one) for any transformation that is available in the program.

The data fields are represented by the *DataSet* class, which contains a list of *Fields* (Program 4), each holding *Header* and a list of *Cells*. The *Header* is a representative of the column, storing *FieldDataType* used in its column. Secondly, it stores the position of the column in the *DataSet* and its name. The *Cell* is a representative of a single value stored in the column, storing content and its position in the column.

For practical reasons, there is also an *EmptyField* class defined (Program 4), which serves as a type representative of a field and carries only the name and type of values contained in the given field. This is required since the type-information can affect the selection of available subsequent operations in the user selection, and it is therefore necessary to maintain (and possibly change) the type of fields even during the iterative construction of the user query without actually touching the data. This is used in the *Preprocess* method (Program 3) to change the data fields accordingly to the current state of the query.

Finally the *QueryViewModel* is the response of the *PerformQuerying* method (Program 1) that holds all relevant information for creating a collection of next moves with the LLM agent suggestion whispered, potentially used in the user application.

---

**Program 4** Definitions of classes representing dataset.

---

```
public enum FieldDataType
{
    Bool,
    String,
    Number,
    Date
};

public class Header
{
    public FieldDataType Type { get; set; }
    public string Name { get; set; }
    public int Index { get; set; } = 0;
}

public class Cell : IComparable<Cell>
{
    public string Content { get; set; }
    public int Index { get; set; }

    public int CompareTo(Cell? other) {...}
    public int CompareToTypeDependent(FieldDataType type,
        Cell? other) {...}
}

public class EmptyField
{
    public Header Header { get; set; }
}

public class Field : EmptyField
{
    public List<Cell> Data { get; set; } = new();
}

public class DataSet
{
    public List<Field> Fields { get; set; } = new();
}
```

---

## 2.3 Communication with an LLM

In this section, we describe the format of conversation that is conducted between the program and the LLM agent for the purpose of obtaining guidance for the next step. During the design of the LLM prompts, it was discovered that the way we communicate with the agent radically affects the validity and correctness of the answers and we therefore devote a whole chapter 3 to this topic. Here, we only describe the approach that was the motivation for chapter 3 and is part of the program implementation.

The form of the prompt used in the application is described in the following example (the dataset contains journeys and consumption of traction energy by railway carriers in Germany):

```
"User initial input is: Group data by carrier 'CDC'
and show count of distinct values in each column.
```

```
The query built so far: GroupBy.Carrier
```

```
---> Choose one of the following Aggregations
you want to apply to the grouped dataset:
```

```
> [0] Sum
> [1] Avg
> [2] Concat
> [3] CountDistinct
> [4] CountAll
```

```
Answer the appropriate number!"
```

The example shown above is taken from a test run of the application. It shows the full body of a prompt that the LLM agent receives when asked to choose the next best step. The form can generally be described as follows:

1. With each subsequent step of an iterative build of the final query, the LLM agent is informed about the original user request,
2. the previously selected operations are listed to the agent too,
3. then, an enumerated collection of further options is presented, and
4. finally, the agent is reminded to respond only with a number from the collection.

During the first usage, we discovered that the absence of the original user request in the option offer (item 1) may cause the agent to consider the query as finished, although especially in queries combining multiple operations at once, a query was finished for only a part of the request. Therefore, the user request is added to each prompt.

For similar reasons, with each prompt, the agent is reminded of the already created query (item 2) even though the agent has these data available in the chat history, which is being formed by every added prompt. However, it can be assumed that the agent places greater emphasis on the most recently added messages and especially with longer queries may neglect already performed operations.

When designing the communication with an agent, we assumed that with each prompt, the agent would only be presented with a list of names of individual operations and the agent would respond only with the name of the selected operation. However, when applying this approach, the agent's response often contained not only the name of the operation but also a lot of confusing text was added:

```
"From the list above, I would choose 'CountDistinct'"
```

Instead of simply responding by:

```
"CountDistinct"
```

For that reason, the LLM agent parameters (such as *top-p* or *temperature*) were set to the maximum extent possible so that the response would be as straightforward as possible and contain a minimum of ancillary text.

Unfortunately that did not stop the LLM in producing ancillary text either and therefore all options are uniquely numbered (item 3) and it is expected from the agent to respond only with the corresponding number.

That being said, if the following option is considered correct:

```
[3] CountDistinct
```

The LLM is expected to answer with:

```
"3"
```

However, since this approach was still susceptible to responses of a similar character;

```
"Let's start with GroupBy carrier 'CDC'  
and show count of distinct values.  
So the next transformation would be [3] GroupBy."
```

an imperative sentence was added to the end of the prompt (item 4) specifying what type of response we currently expect from the agent.

This design of prompts proved to be functional in the application's use and the agent started responding in a valid format in the most of the cases. It thus appears that the more restrictions about the input the agent receives, the less prone it is to deviate from the given format.

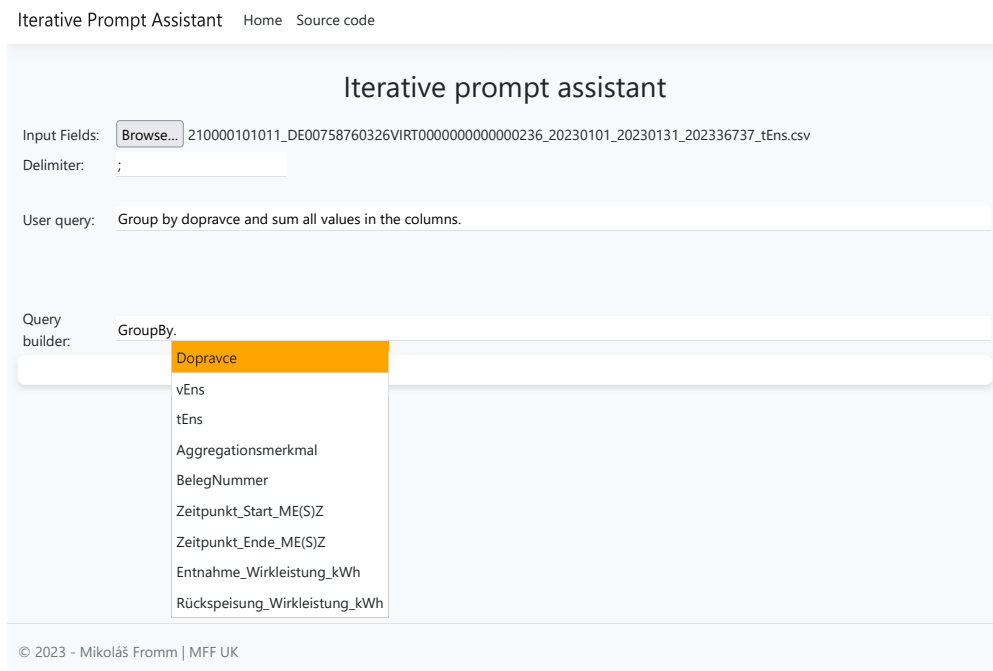
## 2.4 Case study

In this section, we demonstrate the final form of the web application with several screenshots.

After uploading a dataset in the CSV format and specifying the delimiter used in the file, the user can enter their request. The request is formulated using a simple sentence, which is ended with a period. The presence of the period simultaneously triggers the first prompt to the LLM agent, followed by displaying available options. The preferred option by the LLM agent is always placed at the first position and is orange-highlighted. The rest of the options remains in the original order.

If the construction of the query is at the beginning or if the construction of one of the transformations has just been completed, the user is offered a list of all next available transformations. On the other hand, if a specific transformation has already been specified but is not yet fully constructed, all possible following arguments are displayed in the offer.

Some transformations have fixed additional arguments (such as *OrderBy*. *<header-name>*. *<ascending/descending>*) for which the agent is always given a complete numbered list of available options, but some transformations (in case of *FilterBy*. *<header-name>*. *<equality>*. *<any-value>*) allow entering any textual argument. In such cases, the LLM agent is prompted using a different imperative sentence at the end of the prompt (item 4) with a helping instruction to enter the textual argument that is most likely considered correct.



**Figure 2.1** Dynamically created dropdown menu with highlighted choice from LLM agent.

In the example 2.1, we can see that in the case of arguments requiring the selection of a specific column name, the offer is dynamically generated according to the current structure of the dataset.

When the user successfully creates at least one complete transformation, the current form of the dataset is displayed in the preview. The user may continue building their desired query or decide to finish building the query at the current state. If they decide to continue, each subsequent transformation will then be applied to the current data from the preview, thus the application implicitly supports layering of a larger number of operations in sequence.

In the example 2.2 below, we can see the final form of one completed transformation and a preview of the current state of the dataset.

**Iterative prompt assistant**

Input Fields:  210000101011\_DE00758760326VIRT00000000000000236\_20230101\_20230131\_202336737\_tEns.csv

Delimiter: ;

User query: Group by dopravec and sum all values in the columns.

Query builder: GroupBy.Dopravec.Sum.

vEns	tEns	Aggregationsmerkmal	BelegNummer	Zeitpunkt_Start_ME(S)Z	Zeitpunkt_End_ ME(S)Z	Entnahme_Wirkleistung_kWh	Rüc
0	0	1402632000000	0	0	0	43,707	17,3
0	0	0	0	0	0	641,768	2,66
0	0	701316000000	0	0	0	505,55	30
0	0	0	0	0	0	42691,5	451
0	0	0	0	0	0	4782,3000000000001	175,

© 2023 - Mikoláš Fromm | MFF UK

**Figure 2.2** Preview panel with transformed dataset.

Finally, an example from testing the application shows that even if the agent is given a very detailed description of the format that is expected to be used in his answers, the validity of his responses is not guaranteed.

Input prompt:

User initial input is: Show only rows with dates later than 10/01/2023 and then filter by carrier = CDC.

The query built so far: `FilterBy.StartDate.GreaterThan.10/01/2023.FilterBy.Carrier.Equals`

Write down the right side of the relation.

Agent's response:

I'm sorry, I made a mistake earlier.

The last transformation you chose was `FilterBy` on the field "Carrier". Therefore, you should choose a value for the "Carrier" field, not the "GreaterThan" relation.

Here are the possible values for the "Carrier" field:

- > [0] CEZ
- > [1] EON
- > [2] PRE

Please choose one of these values.



In this case above, the agent was prompted to fill in the right side of the *Equals* relation, thus any response was understood as a valid argument (while expecting that "CDC" would be the correct answer), yet it is obvious that the response is far from the expected and therefore incorrect. The agent not only adds apology messages that his procedure was wrong, but also defines new options that may be in the correct format, yet completely unrelated to the dataset.

In the following Chapter 3, we will therefore verify whether it is possible to at least partially prevent problems with incorrect responses and how frequent is the error of these conversational bots in a similar application environment.

# 3 Constructing better prompts

In this chapter, we focus on the issue of constructing an appropriate prompt. Creating the Iterative Prompt Assistant from chapter 2 has proven that choosing the right strategy for designing the final form of prompt communication is complicated. Therefore, in this section, we experimentally verify several hypotheses related to the creation of LLM prompts, which can help in designing other systems that depend on prompting with LLM.

## 3.1 Motivation

The main motivation for experimenting with the form of LLM prompting is to maximize the success of the final prompt form. However, it is very difficult to define what "success" of a prompt means in a general case, so we focus on the application environment from chapter 2. Similar to this application, we iteratively build the final query using LLM prompting, but unlike the application described above, the LLM interacts only with a testing framework that assesses the correctness of the next proposed step.

The goal of the LLM agent is to guide a pseudo-user through a tree represented option space based on a request given in a natural language and help to choose the next appropriate step to achieve the goal, which is to find a correct path to the requested object in the underlying data structure. This form of the assignment was particularly designed to suit our testing needs to determine all correct solutions in advance and decide whether the LLM agent's proposal is correct or not afterwards.

We also evaluate whether the success is influenced by the language (e.g. English or German) in which the query or data structure is described and whether the proposed improvements are invariant to the change of the LLM.

## 3.2 Framework description

The framework represents a methodology for conducting automated tests in the context of the iterative prompting methods. It is specifically designed to evaluate how different prompting strategies influence the performance of LLMs, OpenAI's GPTs exactly, in navigating structured data sources like databases or file systems. The system incorporates a setup that enables a simulation of real-world data querying tasks across multiple data domains and languages, testing the adaptability and efficiency of LLM prompts in guiding users to their desired information.

### 3.2.1 Key Components and Methodology

The framework is structured into several core components, each responsible for a distinct aspect of the testing environment:

#### 3.2.1.1 Data Source Simulation

The Data Source component simulates three types of structured data sources: A copy of a content structure of the Eurostat database [7] (obtained on 01/02/2024), a snapshot of a file system containing personal user data (such as photos, videos, documents etc.) and a snapshot of a linux server file system.

Each data source is represented as a navigable tree structure with nodes and leaves, where nodes represent directories or categories, and leaves represent files or final data entries.

The Eurostat [7] offers its content table in three languages: English, French and German. The personal file system has its directory and file names in Czech and the linux server file system is in English.

#### 3.2.1.2 Prompt Engineering

The Prompt Engineering component implements a prompt engineering system capable of generating prompting strategies to interact with the LLM. These strategies vary from simple *STEP-BY-STEP* navigation prompts to more specific *LOOK-AHEAD-X* prompts that provide an overview of potential navigation paths. Alternative prompting method such as *KEYWORD GENERATION AND MATCH*, where the LLM is tasked with identifying relevant subsections based on generated keywords, is also implemented.

#### 3.2.1.3 Test Definition and Execution

The framework also includes testing mechanism to systematically execute predefined navigation tasks within each simulated data source, using a list of correct navigation paths as benchmarks. The performance of each prompting strategy is evaluated based on metrics such as the number of steps taken, the presence of incorrect moves, and the overall success in reaching the correct destination. Extended testing configurations allow for penalization based on error locations.

#### 3.2.1.4 Evaluation and Visualization

Lastly the framework includes tools for statistical analysis and visualization of test results, allowing to easily interpret the obtained results and generate plots illustrating the test results.

## 3.3 Hypotheses

To experiment with increasing the "success" of LLM prompt responses, we set the following hypotheses, which we verify in this chapter:

### 3.3.1 Expanding Context Increases Success

**H1: *The success of selecting the correct answer can be increased by enlarging the contextual information in the LLM prompt.*** When designing the form of LLM prompts, it has been shown to be effective in small test examples to offer the LLM agent not only all currently available next steps but also to include more information about the given step (for example, all subsequent steps related to the given step). We therefore want to verify whether the "look-ahead-X" method, which attaches all further subsequent available steps up to a distance of  $(X - 1)$  from the currently named step, increases the success of the LLM agent.

### 3.3.2 Reason for Higher Success

**H2: *The increase in success for hypothesis H1 is not just a result of a more accurate answer when deciding at the end of the search path.*** Consider now, for example, the look-ahead-3 method and a test example that contains only 3 steps to find the right target. Then the initial prompt that the LLM agent receives as the first one will already contain the textual naming of the goal. It can therefore be assumed that in each such case, where the distance of the look-ahead-X is the same or shorter than the number of remaining steps to reach the goal, the measured success will be artificially higher. Therefore, we introduce a penalty score, which will place more emphasis on the agent's first steps rather than on later steps.

We are interested whether there is a significant difference between the score of the penalized and non-penalized agent, that is, alternatively whether the potential increase in success with look-ahead is caused only by an artificially more accurate answer in the last steps of the decision-making process, or whether this approach generally helps to find the right step when searching for a target.

### 3.3.3 Model Invariance

**H3: *Methods increasing success are invariant to the LLM used.*** For the trends measured when testing hypothesis H1, we would like to verify that it is a general trend that can also be applied to another LLM and is not just a specific feature of one particular tested model. Therefore, we will perform all series of tests on two LLMs simultaneously (GPT-3.5 and GPT-4) and verify that the trend in success relative to the size of the context is the same for both models.

### 3.3.4 Language Invariance

**H4: *The success of selecting the correct answer is not dependent on the specific language of the query or dataset.*** Since the testing framework will primarily contain English data sets and user queries in English, we first want to verify whether we would achieve the same results when testing in another language. For these purposes, we will use the Eurostat database [7], which offers its "list of content" in three languages (other languages are only machine-translated); English, French, and German. Tests will be conducted over all combinations of data sets in English and German and user queries in English and German. Thus, a total of 4 sets of tests will be conducted, which will subsequently be compared.

## 3.4 Methodology

In this section, we introduce the methodology used for testing the hypotheses mentioned in the previous section 3.3. Specifically, we will present how we simulate data sets (3.4.1), how we generate test prompts (3.4.2), how we test the LLM (3.4.3), and finally how we evaluate the results (3.4.4).

### 3.4.1 Data Simulation

Each data set is represented by a tree structure, containing nodes that hold a list of their successors, a reference to their parent, and a textual representation of the content that the node virtually holds. In the context of the Eurostat database [7], the textual representation includes names of subcategories or names of final table files, while in the context of a filesystem, it involves names of directories or final files. During testing, we require from a node to be able to return a list of all its successors and allow access (by index) to a selected successor.

For all data sets, we only create a copy of their structure, ignoring the data contained within the structures (such as the contents of files in a Linux filesystem) during the simulation. The precise implementation of data extraction and the creation of representative trees can be found in the attached program A.2, in the section *Node and Leaf Definitions*.

### 3.4.2 Prompt Engineering

During testing, we access the LLM via the OpenAI web API [8], which allows creation of a chat-like conversation between an "assistant" and a "user". Additionally, it permits the inclusion of messages with the role of a "system message", which are primarily intended to be descriptive and directive to better acquaint the agent with the situation.

For testing purposes, we have created the *OpenAIWrapper* class that encapsulates these API calls and defines two primary methods:

```
def get_response_to_prompt(self, prompt : str) -> str:
    ...
def get_additional_response(self, prompt : str) -> str:
    ...
```

As the construction of the assistant’s prompts in Chapter 2 revealed, the maximum limit of the total number of tokens used in the conversation acts as a limiting factor that did not allow some assisted selections to be completed. Therefore, the class includes two methods for obtaining a response from the LLM, where the primary method *get\_response\_to\_prompt* automatically deletes the previous conversation and replaces it with a new one, while the secondary method *get\_additional\_response* retains the existing conversation and only adds another user prompt.

The decision to delete the previous conversation with each subsequent prompt indeed solves the problem with limited context for most tests but, conversely, now requires to include an information about the operations already performed in every prompt. Without such information, a deterioration in the agent’s success in choosing the best next step can be expected.

### 3.4.2.1 Prompt Generation

The creation of textual prompts is managed by the `AssistantWorker` class, which, for the original user query, the path already taken through the nodes, the current node, and a specific mode, creates a textual prompt containing a numbered selection of all the next available options.

```
def create_llm_query(self, initial_query : str,
    path_done : List['Node'], current_node : Node,
    mode : int) -> str:
```

Each prompt always includes a reminder of what the user’s original query was and which steps have already been taken. Subsequently, depending on the specific *mode*, the remaining content is determined:

<i>Mode</i>	Description
<i>STEP BY STEP</i>	The prompt contains only the numbered names of all subsequent actions, each action on a new line. Intended to be a baseline-score solution.
<i>LOOK AHEAD X</i>	The prompt contains the numbered names of all subsequent actions, which are recursively supplemented (up to depth $X - 1$ relative to the current node) with the names of all future actions available from that node. Each supplementary action is placed on a new line, but yet without numbering.
<i>LOOK AHEAD X (INLINE)</i>	Similar to the mode above, except all future actions are inserted on the same line in brackets (...) after the name of the given node. This approach is expected to only reduce the number of tokens in the prompt and test whether such a change in the format (effectively making impossible for human to read the content) affects the score.
<i>KEYWORD GEN. AND MATCH</i>	An experimental mode, which asks the LLM agent to generate $N$ keywords that are, according to the LLM agent, relevant for searching within the given structure. This generation is executed prior to the iterative process. Subsequently, the tree holding all nodes is filtered so that only nodes with a similar textual name remain. We expect that if the filtering is not too aggressive and the right choices remain, it has a potential to reduce the token usage dramatically.

**Table 3.1** Selection of all modes over which tests were conducted.

In addition to the above, the LLM agent is introduced to the specific situation, briefly explained the context and given a short example, at the beginning of each conversation through a *system message* (explained in section 3.4.2).

### 3.4.3 Testing

For testing the correctness of the response obtained from the LLM agent based on the generated prompt (section 3.4.2.1), we need to know (at least one) correct next step from the given node. For this purpose, we define the *CorrectPath* class, internally holding the sequence of indexes of responses that are considered correct. Since some data sets may have more than one correct answer, the *CorrectPath*

class accounts for multiple correct sequences. With these sequences, it can later determine whether the next proposed step is correct or if it has reached the end of the path, thus finding the goal.

```
def is_next_move_correct(self, move_index : int) -> bool:
def is_at_the_end(self) -> bool:
```

Since the correctness of the answer is checked purely at the level of numerical indexes, the indexes of correct responses must be selected from the identical set over which the tests are conducted. For this reason, representations of all test data sets (Eurostat, personal filesystem, and Linux server filesystem) were processed and integrated into the testing framework to prevent changes in the data over time (available in the attachment A.2).

Together with all correct responses, it is necessary to pass a specific user query that the LLM agent is supposed to assist with in searching the given data structure. This is passed as a natural text and represents a typical user query that might be posed over the given data structure.

```
create_test_sequence(
    source=eurostat_en,
    query="What was the GDP growth for Czech Republic in 2019?",
    correct_seqs=["0 1 0 0 0 0", "0 1 0 0 0 1"])
```

With the correct responses established and the user request given, it only remains to introduce the LLM agent his role and describe the basic rules to follow when generating prompts:

```
system_message = """You are an assistant that is helping a user
to navigate through the Eurostat database.
You are given a query and all available subsections.
You should help the user to navigate to the correct subsection.
Answer only with the number of the subsection.
If you don't know the answer, return -1."""
```

Then it is possible to test the correctness of the LLM agent's responses over the given scenario. Test run is defined as follows:

```
def __run(self, root : Node, query : str,
    worker : AssistantWorker, openai : OpenAIWrapper,
    correct_path : CorrectPath, mode : int) -> TestResult:
```

In the "Handle invalid response" part, *additional messages* (described in section 3.4.2) are created, and based on the nature of the error, an attempt is made to obtain the correct response. If the error is only in the wrong choice of the next step, the LLM agent is notified about it accordingly. In other cases, the error is understood as the wrong answer format, the LLM agent is alerted that it responded in the wrong format and is reminded of the correct format. In both cases, it is then asked to respond to the question again.



---

**Algorithm 1** Execution of one test run.

---

```
function RUNTEST(root, query, worker, openai, correctPath, mode)
  currentNode ← root
  pathDone ← []
  keywords ← []
  indexTranslationDict ← {}
  result ← new TestResult()
  if mode = KEYWORD_GEN_AND_MATCH then
    keywords ← get keywords from LLM response
  end if
  while True do
    if mode = KEYWORD_GEN_AND_MATCH then
      Adjust currentNode and indexTranslation based on keywords
    end if
    if no children for currentNode and not at end of correctPath then
      Mark wrong move and assisted answer
      Move to correct step using correctPath
    else
      Generate and send prompt to LLM
      Process LLM response
      if response is valid then
        Make move to next child node
      else
        Handle invalid response
      end if
    end if
    if at end of correctPath then
      Mark success and break loop
    else
      Add currentNode to pathDone
    end if
  end while
  Set tokens used for result
  return result
end function
```

---

Although we initially wanted to limit the number of corrective attempts to the total number of all possible next steps (effectively allowing the LLM agent to simply enumerate all possibilities), this approach was too costly in some cases in terms of the volume of tokens transferred and therefore the maximum number of corrective attempts was empirically limited.

The output of one run is an instance of the *TestResult* class, which holds the total number of steps, the total number of incorrect steps and the total number of assisted responses (in case the LLM agent exceeded the number of corrective attempts). This result is later evaluated in section 3.4.4.

This class is further extended by the *TestResultExtended* class, which aggregates errors into individual steps, allowing for more complex analysis and penalization of specific parts of the run (used in hypothesis H2).

### 3.4.4 Evaluation

For evaluating the effectiveness of a specific prompting method, we use the following simple metric:

$$score_{basic} = 1 - \frac{W}{S} \quad (3.1)$$

$$score_{basic} \in \langle 0, 1 \rangle \quad (3.2)$$

where:

- $W$  is the total number of wrong moves taken, and
- $S$  is the total number of all moves taken.

Therefore, if the LLM agent always responded correctly on the first attempt, its obtained  $score_{basic} = 1$ . Conversely, if it always responded incorrectly (and exceeded the maximum number of corrective attempts every time), its  $score_{basic} = 0$ .

For each data structure, we created at least 10 static testing scenarios (user queries) along with all the correct responses, on which all selected approaches will be tested. These scenarios contain various lengths of paths to the goal and are meant to serve as a representative sample of user queries that could be posed over the given data structure. For the Eurostat database [7], this includes scenarios such as:

1. *What was the GDP growth rate for the Czech Republic in 2019?*
2. *How many people lived in Germany in 2020?*
3. *What was the unemployment rate in Austria in 2015?*
4. *What percentage of the population in Europe had access to clean water and sanitation facilities in 2010?*

For the purposes of testing hypothesis H2, we define an extended testing metric that categorizes errors into individual steps and allows using a penalization coefficient to place greater emphasis on selected steps. (A single step in the context of our testing is considered to be a conversation from issuing a new prompt to the LLM agent to obtaining the correct response or exhausting all possible corrective attempts).

$$score_{extended} = 1 - \frac{\sum_{i=1}^n w_i \cdot p_i}{\sum_{i=1}^n s_i \cdot p_i} \quad (3.3)$$

$$score_{extended} \in \langle 0, 1 \rangle \quad (3.4)$$

where:

- $n$  is the total number of segments in the test,
- $s_i$  is the number of steps taken in the  $i$ -th segment,
- $w_i$  is the number of wrong moves in the  $i$ -th segment,
- $p_i$  is the penalization coefficient applied to the  $i$ -th segment,
- the sum  $\sum_{i=1}^n s_i \cdot p_i$  represents the total weighted steps taken across all segments, considering the penalization coefficients,
- the sum  $\sum_{i=1}^n w_i \cdot p_i$  represents the total weighted wrong moves across all segments, considering the penalization coefficients.

Thus, for a sequence of  $N$  steps of one test, we want to assign penalization coefficients  $p_0, p_1, \dots, p_N$  such that greater emphasis is placed on the first third of steps and conversely, less emphasis is placed on the third third. Therefore, we define the following assignment of coefficients for hypothesis H2:

$$p_i = \begin{cases} 2 & \text{if } 1 \leq i < \frac{N}{3} \\ 1 & \text{if } \frac{N}{3} \leq i < \frac{2N}{3} \\ 0.5 & \text{if } \frac{2N}{3} \leq i \leq N \end{cases} \quad (3.5)$$

where:

- $N$  is the total number of steps,
- $i$  is the index of the current step ( $1 \leq i \leq N$ ).

Since our testing interest lies only in the relationship between the chosen prompting method and its overall *general* success rather than the success at individual empirically created scenarios, we will understand the success  $S$  of the chosen method tested on  $M$  scenarios as follows:

$$S = \frac{\sum_{i=1}^M s_i}{M} \quad (3.6)$$

where:

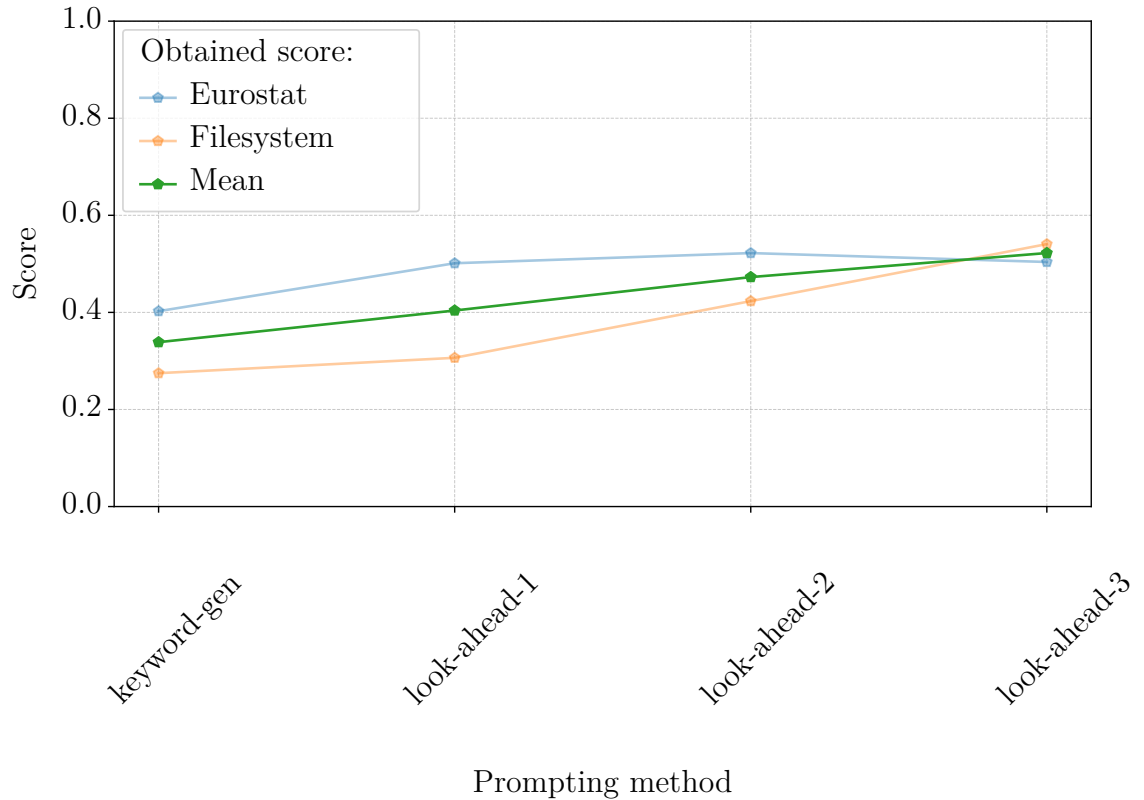
- $s_i$  is the score result of the  $i$ -th scenario.

We are now prepared to commence testing the aforementioned hypotheses (3.3).

## 3.5 Results

### 3.5.1 Expanding Context Increases Success

We start with testing the hypothesis H1. During the development of the Iterative Prompt Assistant (Chapter 2), it proved to be beneficial in several test cases to allow the LLM agent the capability to "see ahead" by recursively including not just all currently following options into the prompt, but also all subsequent successors of a given step, up to a depth of  $X - 1$  for the *LOOK AHEAD X* test. The prompt, however, is still constructed to be human-readable, hence placing each option on a separate line and denoting each level of depth with an additional *TAB* character.



**Figure 3.1** The impact of context size on the final score. Prompt content being separated by tabs, remaining human-friendly.

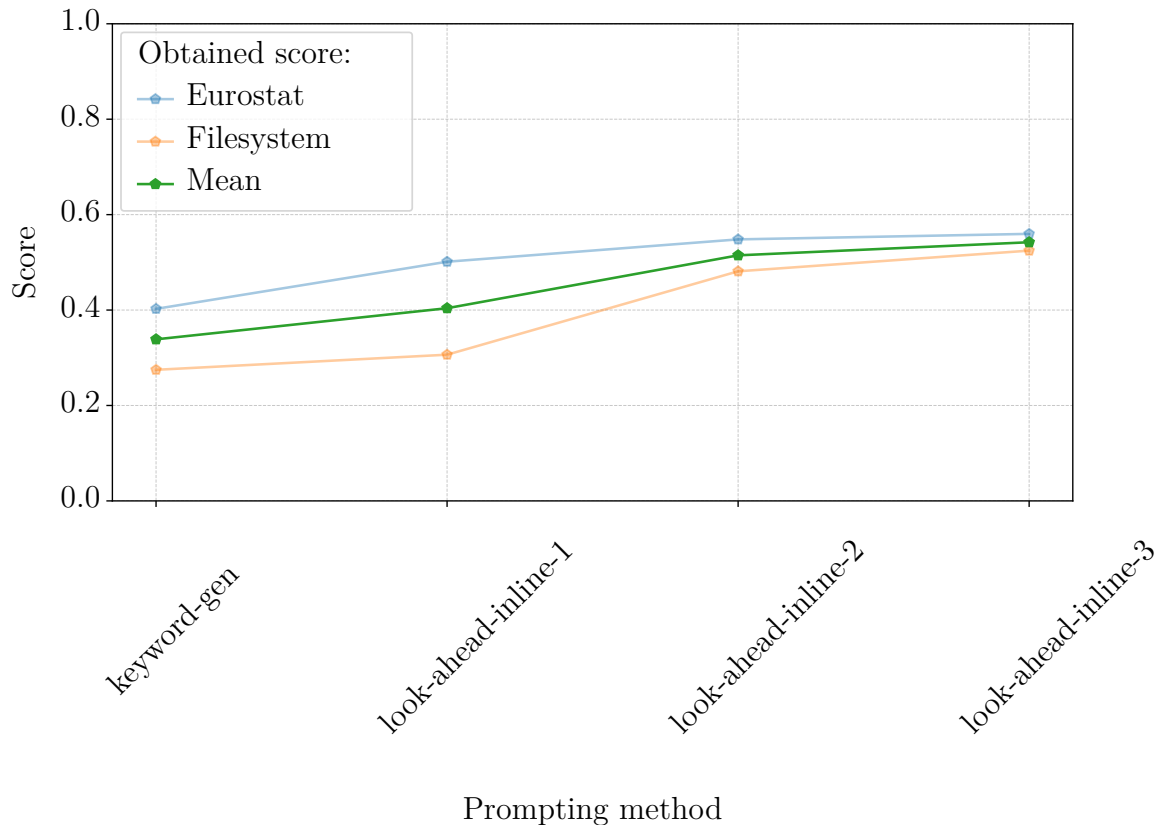
	<b>K-G&amp;M</b>	<b>LA-1</b>	<b>LA-2</b>	<b>LA-3</b>
Eurostat	0.403	0.501	0.522	0.504
Filesystem	0.275	0.306	0.423	0.541
<b>Mean</b>	<b>0.339</b>	<b>0.404</b>	<b>0.473</b>	<b>0.522</b>

**Table 3.2** Scores obtained with each prompting method, showing an increasing trend for bigger context.

The results demonstrate that providing information about future steps positively influences the overall success of the LLM agent’s responses at every level. However, as will be shown later in the results 3.5.2, the maximum size of conversation with the LLM agent remains a limiting factor, preventing us from conducting tests with a *LOOK AHEAD* longer than 3 levels ahead.

### 3.5.2 Better but bigger

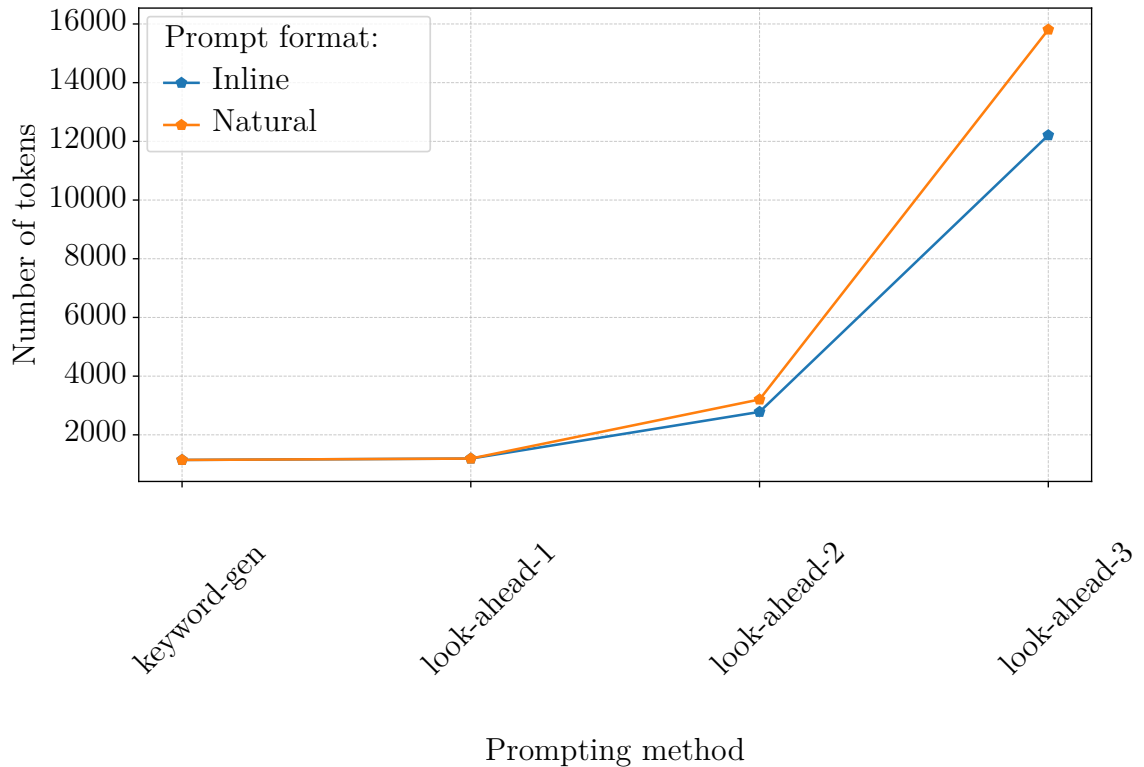
Since we encountered the maximum conversation size with the LLM in the tests used in section 3.5.1 and could no longer perform further tests, we attempt to save prompt size by compressing all *LOOK AHEAD* options into a single line, enclosed in parentheses ( ). Our primary interest then is whether the trend observed in section 3.5.1 is still present, that is, increasing context size positively influences the obtained score. Secondly, we show how the total prompt size (measured in the number of tokens) grows relatively to the depth of the *LOOK AHEAD X* prompt and whether the method of compression into a single line has a significant effect on the total prompt size.



**Figure 3.2** The impact of context size on the final score. Prompt content being compressed into one line, human-unfriendly.

	K-G&M	LA-1	LA-2	LA-3
Eurostat	0.403	0.501	0.522	0.504
Eurostat-inline	0.403	0.501	<b>0.548</b>	<b>0.560</b>
Filesystem	0.275	0.306	0.423	<b>0.541</b>
Filesystem-inline	0.275	0.306	<b>0.481</b>	0.525
<b>Mean</b>	0.339	0.404	0.473	0.522
<b>Mean-inline</b>	0.339	0.404	<b>0.515</b>	<b>0.542</b>

**Table 3.3** Comparison of results from standard *LOOK AHEAD* and compressed *LOOK AHEAD*



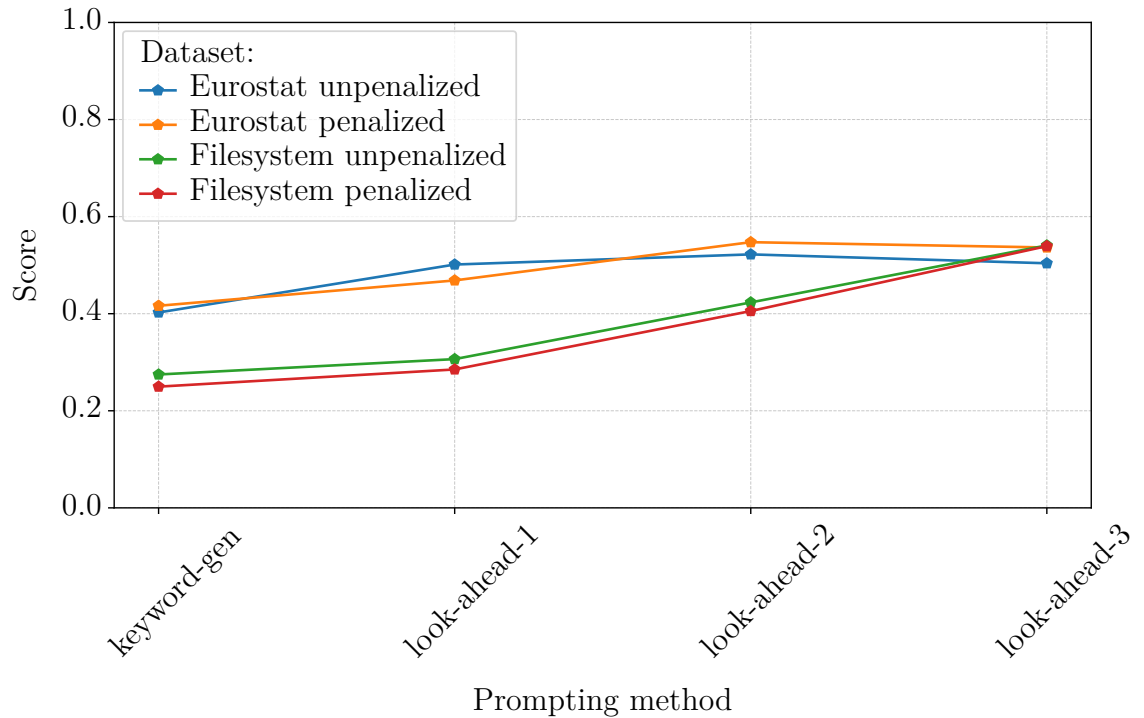
**Figure 3.3** The impact of different prompting methods on the number of tokens used.

We can see from Table 3.3 that compressing prompts does not worsen the overall success of the LLM agent in selecting the correct answer, even on the contrary, with methods using *LOOK AHEAD*  $X \geq 2$  it achieves even better results on average. The results from Figure 3.2 thus show a similar trend as seen in Figure 3.1.

However, if we are concerned with the total conversation size (measured in the number of tokens) in relation to the test method and prompt compression, we see in Figure 3.3 that the size of the prompt grows exponentially in both cases and although *inline* compression helps especially with bulkier prompts to reduce the overall size, we still soon hit the maximum conversation size available with the LLM.

### 3.5.3 Reason for Higher Success

From the results 3.1 and 3.2, or 3.3 respectively, we assume that increasing the context of the prompt has a positive effect on the overall success of the next step selection by the LLM agent. However, hypothesis H2 extends the results 3.3 and examines whether the positive effect of increasing the context of individual prompts is not merely due to the fact that the prompts in the *LOOK AHEAD X* method contain the textual representation of the targeted goal  $X - 1$  steps earlier (compared to the *STEP BY STEP*) and thus, for potentially more complex scenarios that contain a significantly longer path to goal than the value  $X$ , this method of prompting will have no positive effect.



**Figure 3.4** Overview of score changes when penalizing earlier mistakes and ignoring the ones at the end.

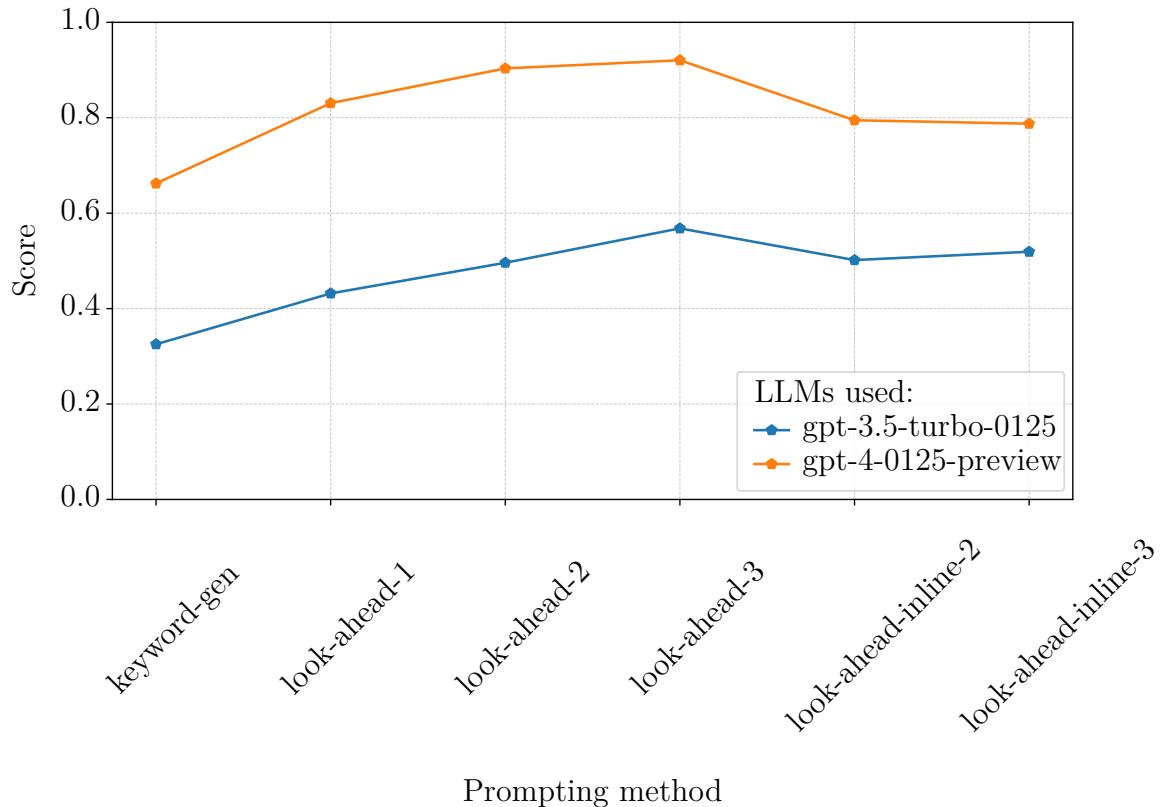
	K-G&M	LA-1	LA-2	LA-3
Eurostat Unpenalized	0.403	0.501	0.522	0.504
Eurostat Penalized	0.416	0.469	0.547	0.536
<b>Eurostat Difference</b>	<b>-0.014</b>	<b>0.033</b>	<b>-0.025</b>	<b>-0.033</b>
Filesystem Unpenalized	0.275	0.306	0.423	0.541
Filesystem Penalized	0.250	0.285	0.405	0.539
<b>Filesystem Difference</b>	<b>0.025</b>	<b>0.021</b>	<b>0.018</b>	<b>0.001</b>

**Table 3.4** Comparison of unpenalized and penalized scores, showing their differences.

The results from Figure 3.4 suggest that even when penalizing the score (described in Section 3.4.4), there is still a visible trend of positive influence from increasing the context on the success of choosing the correct next step by the LLM agent. We therefore expect that the same trend would be present in scenarios with longer step sequences and the *LOOK AHEAD X* is generally applicable.

### 3.5.4 Model Invariance

Additionally, we want to test hypothesis H3, namely whether the observed trends are applicable across different LLMs and are not just a specific characteristic of one selected model.



**Figure 3.5** Comparison of the two LLMs, with GPT-4 being an evolution of GPT-3 while exhibiting the same score trends.

	K-G&M	LA-1	LA-2	LA-3	LAI-2	LAI-3
GPT-3.5	0.325	0.432	0.496	0.568	0.502	0.519
GPT-4	0.662	0.831	0.903	0.920	0.795	0.788
GPT-3.5 inter-diff		0.107	0.064	0.072	-0.066	0.017
GPT-4 inter-diff		0.168	0.073	0.017	-0.126	-0.007

**Table 3.5** Comparison of scores for GPT-3.5 and GPT-4 model snapshots from 01/2023



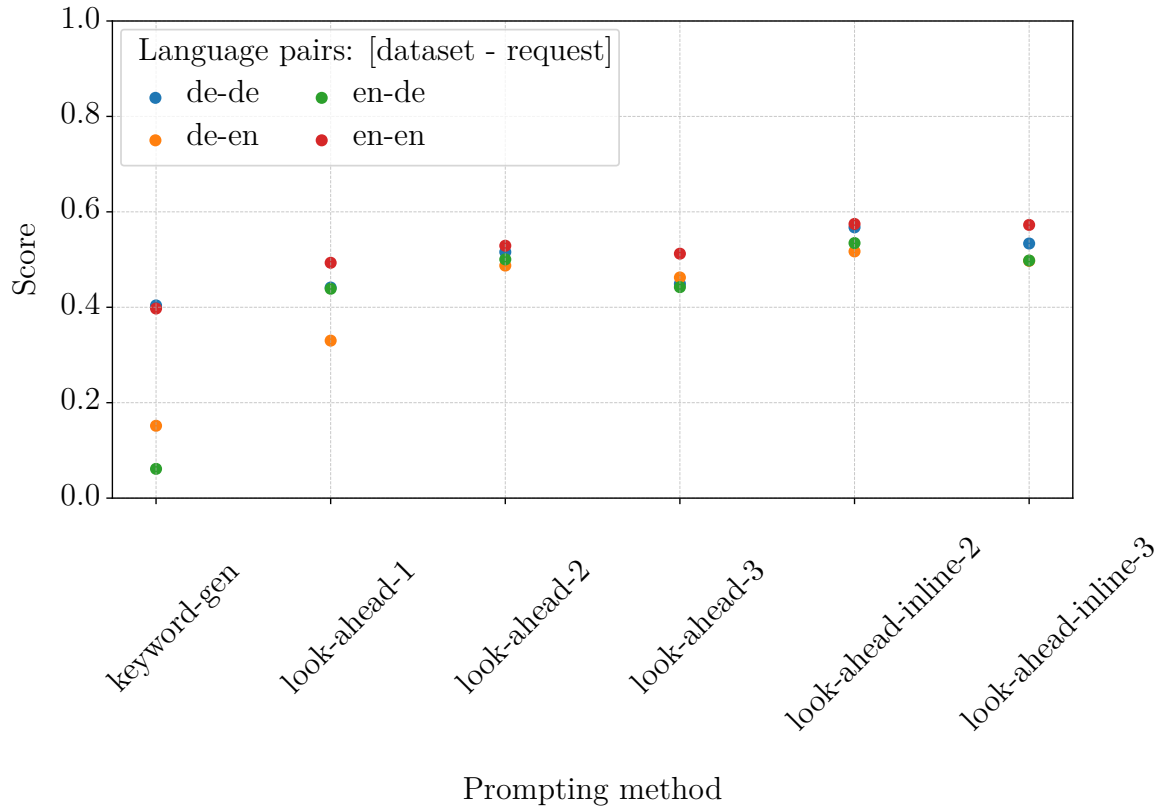
From the results in Figure 3.5, we can observe a very similar trend in score development for both tested models. The GPT-4 model, however, at the *LOOK AHEAD 3* method, already achieved a very high score, hence it may not exhibit as steep trend in score development as the weaker GPT-3 model, for which the increase in context visibly helps across all *LOOK AHEAD X* values. Therefore, based on Figure 3.5, we assume that the trend is common to more LLMs, not just a specific model.

### 3.5.5 Language Invariance

Lastly, we want to test the hypothesis H4, which concerns whether all outcomes obtained from the tests are invariant to the language used, or if the data structure’s language or query language affects overall success. Therefore, on the Eurostat dataset [7], which includes human translations into German, we conducted the following four tests with configurations:

Dataset language	Query language
English	English
English	German
German	English
German	German

**Table 3.6** Language configurations for testing language invariance.



**Figure 3.6** Mean score of all language pairs on different prompting methods.

The results from Figure 3.6 show that the language pairs achieved different outcomes, leading us to assume that the results are not invariant to the language of the dataset or user query. However, the most markable drop in the achieved score visible in *keyword-gen* method was expected, since the LLM was asked to generate  $N$  related keywords in different language than the dataset was described in. That led to having keywords generated in the prompt language, while having different language of the dataset.

Because the tree of all choices is later filtered to only hold nodes that have a similar name as one of the keywords generated from the LLM agent, it is obvious, because of the language mismatch, that the most of the nodes would be falsely filtered out. To further support this assumption, we will test the hypothesis H4 using the Kruskal-Wallis test with a significance level 0.05, testing the null hypothesis whether the medians of the language pairs grouped by prompting method are identical.

Language Pair	Values
en-en	0.3974, 0.4932, 0.5290, 0.5123, 0.5746, 0.5726
de-de	0.4039, 0.4413, 0.5158, 0.4492, 0.5675, 0.5336
de-en	0.1516, 0.3301, 0.4874, 0.4625, 0.5170, 0.4973
en-de	0.0613, 0.4388, 0.5004, 0.4422, 0.5345, 0.4980

Test Set	$H$ -statistic	$p$ -value
Keyword-gen	33.137561	0.000000
Look-ahead-1	31.549756	0.000001
Look-ahead-2	13.347805	0.003942
Look-ahead-3	17.029756	0.000697
Look-ahead-inline-2	27.695122	0.000004
Look-ahead-inline-3	29.326829	0.000002

**Table 3.7** Final scores of each pair and their Kruskal-Wallis H-test for differences between each configuration.

The results of the Kruskal-Wallis tests from table 3.7, with all  $p$ -values  $< 0.05$ , let us reject the null hypothesis, thus proving that the language of the query or the dataset has a significant effect on the final score.

We may also extend the testing to only examine matching language pairs (*en-en* and *de-de*) to see if this configuration, which is closer to the real-world scenario, still does not prove to be language invariant.

Test Set	$H$ -statistic	$p$ -value
Keyword-gen	0.822857	0.364346
Look-ahead-1	13.720000	0.000212
Look-ahead-2	1.651429	0.198765
Look-ahead-3	11.062857	0.000881
Look-ahead-inline-2	0.365714	0.545350
Look-ahead-inline-3	10.080000	0.001499

**Table 3.8** Kruskal-Wallis H-test for differences between configurations with matching language pairs.

The results of the Kruskal-Wallis tests from table 3.8 show that in some test scenarios, the results are already *similar enough*, yet some remain too different, to state that the obtained scores are language invariant. Therefore we remain rejecting the null hypothesis in both cases, thus proving that the language of the query or dataset has an impact on the final score.

## 3.6 Threats to validity

Finally, we discuss the potential weaknesses of the selected testing methods. Primarily, it should be noted that the testing of effectiveness described in section 3.4.4 becomes inaccurate once the prompted LLM achieves very good results with basic methods (*STEP BY STEP*). At such a point, it would no longer be possible to compare these results with better methods (*LOOK AHEAD X*), as all results would reach the upper limit of the success scale. In such cases, it would be appropriate to use a metric that is not upper-bounded. For this reason, the stored testing results include not only the final scores in the range  $< 0, 1 >$  but also the total number of correct and incorrect answers.

Secondarily, it is important to note that the final success of an LLM agent also depends on the difficulty of the proposed test questions (user queries). When creating test scenarios, we proposed diverse yet typical queries for the given structure. If inappropriately (or unsuitable) user queries were proposed, we might obtain overly good or bad results from the LLM agent, which would not allow for meaningful comparisons. In such cases, it would be beneficial to increase the number of test scenarios and design them at various levels of difficulty.

Due to technical reasons, we also limited the maximum number of correction attempts by the LLM agent when the response was incorrect or poorly formatted. It can therefore be assumed that in some situations, the LLM agent might have achieved a lower score, even though such cases were not frequent during the testing.

Lastly, we believe it would be interesting to conduct testing of hypothesis H3 on models other than just those from the *ChatGPT* family. However, motivated by chapter 2, our tests were conducted solely on this family of LLMs.

## 4 Conclusion

In this work, we have built upon the project *The Gamma* and attempted to enrich the iterative approach to exploring and working with datasets by integrating an assistant that helps the user select the next appropriate action in relation to the given user query. Along with the application itself, which allows for iterative work with tabular data using behind the scenes querying of an LLM agent for the next best step, we designed an architecture that permits individual modification or replacement of program components without disrupting the integrity of the entire solution. Thus, it enables to easily change the form of prompts sent to the LLM agent, as well as change the target LLM agent to which prompts are sent. Besides demonstrations of the program itself, we showed that the form of the prompt is crucial in ensuring the quality of responses and that it is not easy to guarantee this quality.

Therefore, in the second part of the work, we experimented with selected prompting methods, inspired by experiences gained while creating the iterative assistant. We established a testing framework that simulates iteratively accessible source data, mimics a pseudo-user posing relevant questions and aggregates the LLM agent’s responses into a final score. We verified that it always helps to provide the LLM agent with more information in the prompt, allowing the agent to *see around the corner*. Yet, we demonstrated that this is not a straightforward decision, as the overall increase in conversation size with the LLM is disproportionately larger than the gain in response success of the LLM agent, despite efforts to reduce the overall volume of the conversation.

Moreover, we verified that the trends seen in comparative testing of selected prompting methods are invariant to the model used. On the other hand, the language in which the query is posed or the dataset described has an impact on the final score. Even though we were primarily interested in (*growing*) trends between prompting variants, we also assume that the general success of the LLM agent in selecting the appropriate next step may correlate with the LLM’s knowledge of the query language or dataset.

This work can be extended in two directions. Firstly, it is possible to expand the Iterative Prompt Assistant application by adding more transformation operations or expanding the user interface and adding visualizations to the data previews for better overview and easier orientation over the results. Secondly, it is possible to continue testing with the proposed framework, especially in testing other (and new) ways of prompting with the LLM agent with the goal of finding more successful yet more efficient methods guaranteeing on average a better result than the basic *STEP BY STEP* solution or the advanced *LOOK AHEAD X*.

# Bibliography

1. CHANG, Edward Y. Examining gpt-4: Capabilities, implications and future directions. In: *The 10th International Conference on Computational Science and Computational Intelligence*. 2023.
2. ACHIAM, Josh; ADLER, Steven; AGARWAL, Sandhini; AHMAD, Lama; AKKAYA, Ilge; ALEMAN, Florencia Leoni; ALMEIDA, Diogo; ALTENSCHMIDT, Janko; ALTMAN, Sam; ANADKAT, Shyamal, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*. 2023.
3. HENDRYCKS, Dan; BURNS, Collin; BASART, Steven; ZOU, Andy; MAZEIKA, Mantas; SONG, Dawn; STEINHARDT, Jacob. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*. 2020.
4. PETRICEK, Tomas. The Gamma: Programmatic Data Exploration for Non-programmers. In: *Proceedings of VL/HCC*. Rome, Italy, 2022. IEEE Symposium on Visual Languages and Human-Centric Computing.
5. BANG, Yejin; CAHYAWIJAYA, Samuel; LEE, Nayeon; DAI, Wenliang; SU, Dan; WILIE, Bryan; LOVENIA, Holy; JI, Ziwei; YU, Tiezheng; CHUNG, Willy; DO, Quyet V.; XU, Yan; FUNG, Pascale. *A Multitask, Multilingual, Multimodal Evaluation of ChatGPT on Reasoning, Hallucination, and Interactivity*. 2023. Available from arXiv: 2302.04023 [cs.CL].
6. WHITE, Jules; FU, Quchen; HAYS, Sam; SANDBORN, Michael; OLEA, Carlos; GILBERT, Henry; ELNASHAR, Ashraf; SPENCER-SMITH, Jesse; SCHMIDT, Douglas C. *A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT*. 2023. Available from arXiv: 2302.11382 [cs.SE].
7. UNION, European. *Database - Eurostat* [online]. [visited on 2024-03-18]. Available from: <https://ec.europa.eu/eurostat/data/database>.
8. OPENAI (ed.). *OpenAI API Reference* [online]. [visited on 2024-03-03]. Available from: <https://platform.openai.com/docs/api-reference>.

# List of Figures

1.1	An example of <i>The Gamma</i> offering all possible next moves after writing ”” character. . . . .	9
1.2	Tree representation of the iterative approach in data selection environment. . . . .	13
2.1	Dynamically created dropdown menu with highlighted choice from LLM agent. . . . .	23
2.2	Preview panel with transformed dataset. . . . .	24
3.1	The impact of context size on the final score. Prompt content being separated by tabs, remaining human-friendly. . . . .	36
3.2	The impact of context size on the final score. Prompt content being compressed into one line, human-unfriendly. . . . .	37
3.3	The impact of different prompting methods on the number of tokens used. . . . .	38
3.4	Overview of score changes when penalizing earlier mistakes and ignoring the ones at the end. . . . .	39
3.5	Comparison of the two LLMs, with GPT-4 being an evolution of GPT-3 while exhibiting the same score trends. . . . .	40
3.6	Mean score of all language pairs on different prompting methods. . . . .	42

# List of Tables

1.1	Definition of <i>Flipped Interaction Pattern</i> [6, page 6] . . . . .	11
1.2	An example use of <i>Flipped Interaction Pattern</i> for data exploration.	12
1.3	Prompt representation of the Image 1.2. . . . .	14
3.1	Selection of all modes over which tests were conducted. . . . .	31
3.2	Scores obtained with each prompting method, showing an increasing trend for bigger context. . . . .	36
3.3	Comparison of results from standard <i>LOOK AHEAD</i> and compressed <i>LOOK AHEAD</i> . . . . .	38
3.4	Comparison of unpenalized and penalized scores, showing their differences. . . . .	39
3.5	Comparison of scores for GPT-3.5 and GPT-4 model snapshots from 01/2023 . . . . .	40
3.6	Language configurations for testing language invariance. . . . .	41
3.7	Final scores of each pair and their Kruskal-Wallis H-test for differences between each configuration. . . . .	43
3.8	Kruskal-Wallis H-test for differences between configurations with matching language pairs. . . . .	43



# A Attachments

## A.1 C# WebApplication Source Code

Contained in *attachments/web\_app*.

## A.2 Python Testing Framework (with results)

Contained in *attachments/analysis*.