

Bachelor Thesis
Documentation

Natural Language to GraphQL

Leveraging the power of LLMs to build a natural language interface for GraphQL schemas



Version: 1.0
Date: June 14, 2024

Authors: Florian Rohrer
Lucien Zimmermann
Advisor: Prof. Stefan Keller
Partner: ChilliCream
External Co-Examiner: Claude Eisenhut
Internal Co-Examiner: Dr. Thomas Bocek



Department of Computer Science
OST - University of Applied Science
Campus Rapperswil-Jona

Abstract

In-context learning enables Large Language Model (LLM)s to comprehend and respond based on the context of the input, allowing them to adapt to a wide range of domain-specific tasks without additional training. However, they have notable limitations in accessing real-time or domain-specific data because they operate primarily on the knowledge they were trained on. Natural language-to-SQL (NL-to-SQL) systems offer a practical solution, enabling LLMs to transform natural language into SQL commands. This makes data accessible to people without technical expertise. GraphQL, having emerged as a flexible alternative to REST, enables software clients to specify the precise data they require from an API based on a schema.

Based on this research, the project aimed to evaluate the best practices for building a natural language interface for GraphQL by adapting the concepts of NL-to-SQL to the GraphQL domain. By understanding the concepts and evaluating a few optimal strategies, a strong foundation was laid for future development.

Upon starting the project, in-depth research was conducted to understand the current state-of-the-art, focusing on NL-to-SQL due to its similarities and substantial literature available. Key findings were documented, and possible strategies were defined. After a proof-of-concept, two strategies, with a few sub-variants, were established for implementation. The first strategy solely relies on the capabilities of LLMs to directly follow instructions and, using in-context learning, enhance the prompt with relevant samples. The second strategy focuses on using entity extraction to identify entities in the user's question, match them to the schema, and then build the operation based on an abstract syntax tree. During implementation, a third strategy combining the benefits of both previous approaches evolved that overcomes the context size limitation of LLMs. In order to measure the strategies' performance in various metrics, an evaluator was built to efficiently test different implementations against a test set. The latter was inspired by the Spider dataset, a widely used benchmark for NL-to-SQL solutions.

The evaluation has seen seven different LLMs tested against the four most mature variants. The results were analyzed to determine the best-performing combinations. While the first strategy showed promising results for simple test cases, it demonstrated limitations in terms of quality and consistency for more complex ones. The best-performing combination uses entity extraction and algorithmic query generation, which is capable of correcting intermediate errors and always produces valid output, making it reliable enough to be used in experimental environments. In general, OpenAI models (GPT-4) are reliable in following instructions, while open source models (Llama, Mistral) have trouble consistently generating valid structures such as JSON. Hallucinations, though occurring on both OpenAI and open source models, can be drastically reduced with prompt engineering, making the output more consistent. However, both strategies are limited by the context size of the LLM used, making them cost-inefficient or even non-processable for large schemas. To overcome this limitation, future research should focus on an iterative entity extraction approach, enabling large schemas to be processed. Additionally, the shot sampling process can be optimized to further benefit from the LLM's in-context learning capabilities.

Management Summary

Background

In recent years, there has been a notable advancement in the capabilities of LLMs. State-of-the-art models such as GPT-4 and Llama3 have demonstrated the capability to engage in human-like interactions. In-context learning enables the models to comprehend and respond based on the context of the input, allowing them to adapt to a wide range of domain-specific tasks without the need for resource-intensive, task-specific training. Despite this impressive capability, LLMs have a notable limitation when it comes to accessing real-time or domain-specific data, as they operate primarily on the knowledge they were trained on. NL-to-SQL systems offer a practical solution to these shortcomings by enabling LLMs to transform natural language queries into SQL commands, allowing data to be fetched on demand for LLMs to be used for in-context learning. This allows for the building of systems that are able to answer questions based on real-time or domain-specific data, thereby facilitating the accessibility of data to individuals lacking technical expertise. GPT-4-based approaches to NL-to-SQL have demonstrated remarkable performance on benchmarks such as Spider.

While direct database access may be feasible for certain applications, others may only permit access to data via Web APIs. Traditionally, REST APIs have been utilized for data retrieval and manipulation. GraphQL, having emerged as a flexible alternative to REST, enables software clients to specify the precise data they require from an API based on a schema.

Objectives

Based on the current state of the art in NL-to-SQL and the in-context learning capabilities of LLMs, the team aims to adapt these concepts and evaluate the best practices for building a natural language interface for GraphQL. The focus is set on research and evaluation rather than immediate product development. By thoroughly understanding the concepts and identifying optimal strategies, a strong foundation can be laid for future product development. This ensures that any eventual product built on top of this is based on solid research, maximizing its effectiveness.

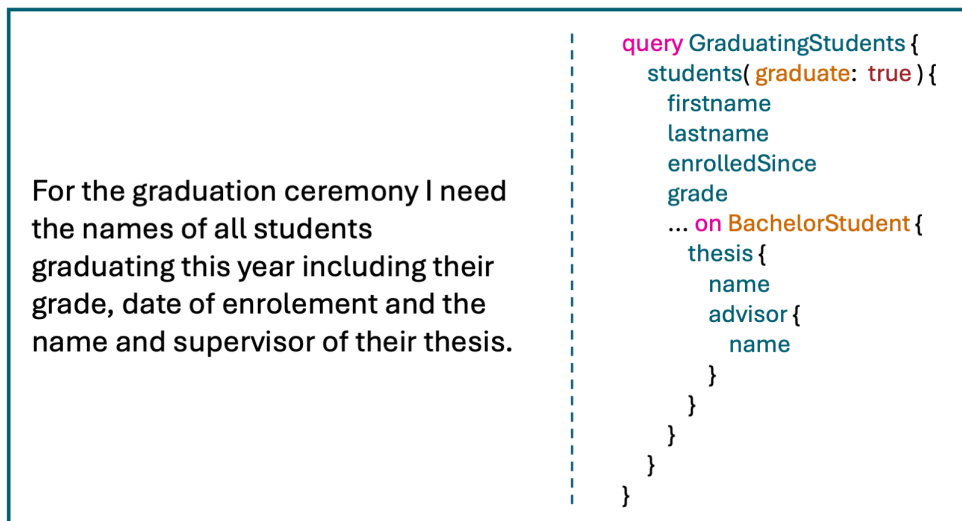


Figure 1.: Turning natural language into valid GraphQL

Approach

Upon starting the project, in-depth research was conducted to understand the current state-of-the-art in the field of generative AI and more specific topics such as NL-to-SQL, few-shot learning and entity extraction. As it is still a relatively new field, there is very little research on the topic of Natural language-to-GraphQL (NL-to-GraphQL). However, NL-to-SQL has been extensively researched over the past few years and has yielded promising outcomes. A further source of information was the work done by the team in their previous thesis, focusing on overall interaction with LLMs and Retrieval Augmented Generation (RAG). As a major part of this thesis is research, the most relevant findings were documented prior to defining implementation strategies. After conducting a Proof-of-concept (POC) using the findings of the research, two strategies and several sub-variants were established for implementation. The first strategy solely relies on the capabilities of LLMs to directly follow instructions, with either only relevant metadata or supported by in-context learning. In-context learning allows the LLM to learn from examples provided in the prompt, rather than requiring a resource-intensive fine-tuning phase. The second strategy focuses on the use of entity extraction to identify relevant parts of the question, match them to the given GraphQL schema, and then build the operation based on an abstract syntax tree. Evaluating these strategies revealed the potential of a third strategy, combining the benefits of both previous approaches while overcoming some of their limitations. Using iterative entity extraction, the strategy would no longer be limited by the LLM's context size, as with the other strategies, big schemas might exceed said limit. By aligning the few-shot examples provided to the LLM using RAG techniques, the potential of in-context learning can be more effectively utilized.

In order to measure the strategies' response quality, reliability, performance and cost, an evaluator was built to efficiently test different variants against a test set. Said test set was created with the concept of the Spider dataset in mind and adapted to fit the purpose of having a similar benchmark for NL-to-GraphQL.

Results

The evaluation has seen seven different LLMs tested against the four most mature strategies. The results were carefully analyzed to determine the best-performing strategies.

As for the first strategy, the sub-variant only using metadata already showed promising results for simple test cases, but was not able to handle more complex test cases. However, the sub-variant using in-context learning was able to also handle the more complex test cases when testing against OpenAI models, while most open-source models, except for CodeLlama, still performed relatively poorly. A limitation of this approach is that it performs the task in a single step, making it challenging to identify the source of errors and incapable of fixing invalid responses.

The second strategy involved splitting the task up into two parts, with the first part responsible for extracting entities out of the user's prompt and the second part responsible for generating the GraphQL operation based on the extracted entities. The generation of GraphQL operations out of the extracted entities was done in two different ways, either by using an LLM or by using only an algorithm. Among the two sub-variants of this strategy, the one employing an algorithm to generate the GraphQL operation demonstrated the most promising results of this evaluation, when using OpenAI models, the other variant did not perform as expected. This second strategy showed the great advantage of being more robust to handle hallucinations from LLMs, as it can programmatically correct mistakes and always produce output that is valid and compiles.

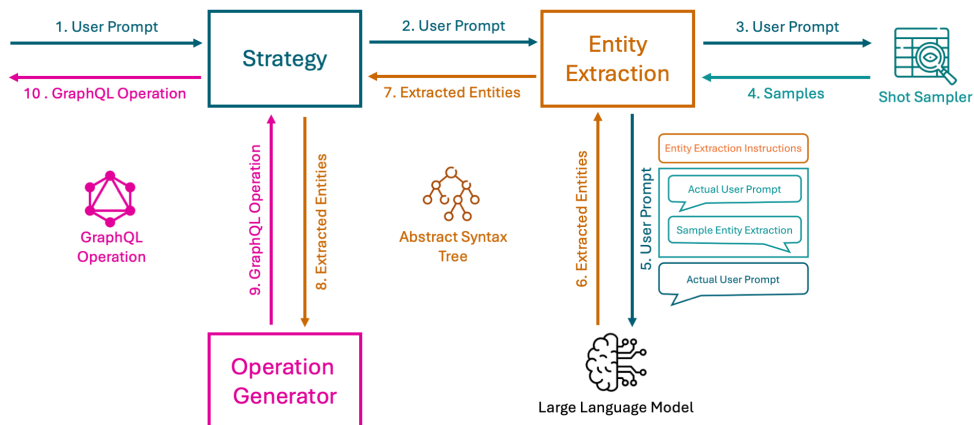


Figure 2.: Visualization of the second strategy

Hallucinations, posing a common challenge when working with LLMs, were drastically reduced using extensive prompt engineering. OpenAI models proved to fit the task at hand well, as steering the output into a specific format led to reliable results. With open-source models (Llama2, Llama3, and Mistral), however, it was harder to steer the output, and it was difficult to receive valid formats, such as the JSON structures used in the entity extraction step.

Regardless of the LLM used, all implemented strategies are limited by the context size of the LLM, making them cost-inefficient or even non-processable for very large schemas. In contrast to the first strategy, the second strategy featuring entity extraction offers more flexibility in terms of future development, allowing step-by-step implementations to be made and reducing the required context size of individual steps. The third strategy, which was not implemented as part of this thesis, is expected to overcome the limitation of context size while still preserving high response quality.

Conclusion

To efficiently use all the learnings made throughout this project, the third strategy should be evaluated, in order to potentially improve the solutions further and overcome the lasting limitations, such as context size. However, the best working strategies are already very advanced and are ready to be used in an experimental environment, as they are able to reliably cope with most basic and even complex requests. Using the best-performing LLMs for the given task, which based on the evaluation conducted are the models by OpenAI, the strategies are relatively simple to integrate into existing systems, enhancing their capabilities.

Contents

I. Product Documentation	1
1. Objectives	2
2. Current State of the Art	4
2.1. Large Language Models	4
2.1.1. Currently Available Models	4
2.1.2. Shot Sampling	4
2.1.3. Prompt Injection Attacks	5
2.2. Research on NL-to-SQL	6
2.2.1. Spider Dataset	6
2.2.2. Bare QL Generation by LLM and In-Context learning	7
2.3. GraphQL	9
2.3.1. Benefits of using GraphQL over REST	10
2.3.2. Challenges of using GraphQL	12
2.4. Entity Extraction	12
2.5. Existing research on GraphQL query generation	14
3. Requirements	15
3.1. Prioritization	15
3.2. Use Cases	15
3.2.1. UC01: Generate GraphQL	16
3.2.2. UC02: Explore Schema Capabilities	17
3.3. Functional Requirements	17
3.3.1. GraphQL Generation	17
3.3.2. Schema Exploration	18
3.4. Non-Functional Requirements	19
3.4.1. Functionality	19
3.4.2. Usability	20
3.4.3. Reliability	20
3.4.4. Performance	20
3.4.5. Supportability	20
4. Design	21
4.1. Architecture	21
4.1.1. System Overview	21
4.1.2. Application Architecture	22
4.1.3. Code Structure	26
4.1.4. Architectural Decisions	27
4.2. Generator Strategies	28
4.2.1. Strategy 1 – Pure LLM	29
4.2.2. Strategy 2 – LLM-Powered Entity Extraction	30
4.2.3. Strategy 3 – Iterative Entity Extraction Using RAG	32
4.3. Intent Matching	32

4.4. Question Answering	33
4.5. Shot Sampling	33
4.6. Security Considerations	34
5. Testing and Evaluation	36
5.1. Testing Environment	36
5.2. Evaluation	38
5.2.1. Evaluation Process	39
5.2.2. Response Quality	39
5.2.3. Reliability	42
5.2.4. Performance	43
5.2.5. Cost-Efficiency	43
5.2.6. Recommendation of the Project Team	45
5.3. Functional Requirements	46
5.3.1. GraphQL Generation	47
5.3.2. Schema exploration	48
5.4. Non-Functional Requirements	48
5.4.1. Functionality	49
5.4.2. Usability	49
5.4.3. Reliability	50
5.4.4. Performance	50
5.4.5. Supportability	51
6. Conclusion and Outlook	52
II. Project Documentation	53
7. Project Management	54
7.1. Methodology	54
7.2. Tools	55
7.2.1. Project Management	55
7.2.2. Software Development	55
7.2.3. Documentation	55
7.3. Quality Measures	55
7.3.1. Organizational Means	56
7.3.2. Code Quality	56
7.3.3. Documentation Quality	56
7.4. Risk Management	57
7.4.1. Project Specific Risks	57
7.4.2. Non-Project Specific Risks	58
8. Milestones	59
8.1. Research	59
8.2. Minimal Viable Product	59
8.3. First Evaluation	59
8.4. Improvement Phase	60
8.5. Project Finalization	60
9. Retrospective	61
9.1. Time Tracking Report	61

9.2. Team Reflection	62
9.3. Acknowledgements	63
List of Figures	64
List of Tables	65
Acronyms	66
Glossary	69
Bibliography	71
III. Appendix	72
A. Appendix	73
A.1. Backend Libraries	73
A.2. Test Set	73
A.2.1. Unfulfillable Prompts	91
A.2.2. Schema Exploration Prompts	92
A.2.3. Schema size tests	93
A.3. Architecture Diagrams	97
A.4. Detailed cost analysis	97
A.5. Test protocols	98
A.6. Prompts used	101
A.6.1. Intent Matching	101
A.6.2. Strategy 1a	102
A.6.3. Strategy 1b	103
A.6.4. Strategy 2a	104
A.6.5. Strategy 2b	106
A.7. Test Schemas	108

Part I.

Product Documentation

1. Objectives

Background

In an era that is driven by big data, AI technologies serve as a cornerstone to make the vast amount of information easily and uniformly accessible. With the rapid advancement of language models, such as GPT-4 and Llama3, has paved the way for intuitive interactions with data using natural language. This makes complex data structures available for everybody, regardless of their technical background, enabling them to make data-driven decisions. Much effort has been put into the research on interacting with SQL databases using natural language interfaces (NL-to-SQL).

While NL-to-SQL delivers promising results by accessing databases directly, employing this strategy in practice does not always prove to be feasible. Often, data in distributed systems is not stored in a single database, but rather aggregated from various sources such as external Application Programming Interface (API)s, file systems or multiple database paradigms. This allows data to be stored in the most efficient way, depending on the use case at hand. Data may also underlie complex business logic, such as calculations or fine-grained access control mechanisms. To overcome the challenges posed by this distributed manner, applications often provide a single API entry point to serve data. In recent years, GraphQL has emerged as a popular choice for building such APIs. Various companies, such as Facebook, GitHub and Shopify, have adopted GraphQL in their APIs.

Goals

In this thesis, the team aims to conduct in-depth research on the topics surrounding NL-to-SQL and apply the concepts to build a solution that is able to convert natural language into valid GraphQL. Therefore, best practices are to be elaborated using different strategies and LLMs in order to conduct an evaluation to determine the most suitable approach. An ideal solution should be able to generate all sorts of GraphQL operations, as well as give information about the underlying schema.

Constraints

As the solution aims to be integrated into an existing software system at a later point, it should be built using the .NET-framework. Said integration is not part of this thesis and is explicitly excluded. The solution should be built to be compatible with OpenAI LLMs, as to allow easy integration without the need to self-host a model. External components used within the solution should be open-source and free to use.

Deliverables

The deliverable should contain a tested software component for converting natural language into GraphQL, including its test data. Furthermore, detailed documentation, of the research process and evaluation results, is to be provided.

2. Current State of the Art

This chapter provides an overview of the current state of the art in several key areas that are central to this thesis. In a first section, relevant concepts concerning LLMs are discussed. As the primary objective of this thesis is to adapt concepts from NL-to-SQL to GraphQL, the second section provides an overview of the current state of the art in NL-to-SQL, followed by an introduction to GraphQL. Further, existing research on NL-to-GraphQL is briefly analyzed. Finally, a discussion of the adoption of the outlined concepts to the objectives of this thesis is provided.

2.1. Large Language Models

LLMs are designed to understand and generate human-like text by predicting the next word in a sequence. Trained on vast datasets collected from the internet and other sources, they utilize billions of parameters and require immense computational power. These models assign probabilities to potential next words, selecting the most likely one based on context.

2.1.1. Currently Available Models

There are two primary categories of language models available: commercial models and open-source models. Commercial models are generally easier to use, but tend to be expensive due to their pay-per-token pricing. One of the most popular commercial models is OpenAI's GPT series, including *GPT 3.5-Turbo* and *GPT 4-Turbo*. While the older GPT-3.5 Turbo model is fast and relatively cheap, the newer GPT-4 Turbo model is more powerful but also 20 times more expensive. [1] OpenAI recently announced the release of GPT-4o, which is faster and 50% cheaper than GPT 4-Turbo while matching its performance. [2] Google also announced the release of Gemini Pro, a model designed to compete with OpenAI's GPT models, but it was not generally accessible to the public at the time of starting this thesis, therefore being disregarded in further evaluation.

Open-source models, on the other hand, are free to use and can be self-hosted but require the right hardware to run, especially for larger models. They also offer greater transparency in terms of how they work and insights into the data they were trained on. One of the most popular open-source models is Meta's Llama series, including *Llama 3* [3] and the *CodeLlama* [4] model, which specializes in code generation. During this thesis, only the smallest available model variants with 7 billion parameters were used, as they are able to run on consumer-grade hardware, while the larger models require more powerful hardware. There are various models, such as Vicuna, that are fine-tuned variants of the Llama model, promising better performance in specific tasks. [5] There are also open-source models like Mistral [6] that are not Llama-based, making them an interesting alternative to the Llama models, as they are built and trained differently.

2.1.2. Shot Sampling

The selection of samples, used in in-context learning, notably improves the accuracy of the result. Nan, Zhao, Zou, *et al.* suggest that by using similarity and diversity sampling, as opposed

to random sampling used by other NL-to-SQL solutions implemented, the accuracy of the system can be improved. [7] The sampling is achieved by embedding a corpus of examples into a vector space and then selecting samples that are similar to the input question. This allows the LLM to learn from samples closely related to the request, as opposed to random samples that could potentially be unrelated. By clustering the samples in the vector space, using k-means, and consequently using the centroids of the clusters as the samples, the diversity of samples is increased. Using this technique, the samples provided in the prompt are similar to the input but also diverse enough to provide samples for a broader range of questions. [7]

2.1.3. Prompt Injection Attacks

Prompt injection attacks are a type of attack that can be used to manipulate the output of a LLM by injecting malicious instructions into the prompt. This can be used to make the LLM say anything, even if it contradicts the instructions given to the model.

Back in 2023, Microsoft revealed Bing-AI – a chatbot that was powered by GPT-4 and freely available to the public, embedded into the Bing search engine. Though it was successful, it was quickly found to be vulnerable to prompt injection attacks. It took a student from Stanford University a mere 24 hours to get the model to read back the original prompt instructions.[8] No complex hacking was required; the student just told the model to ignore the above instructions and carefully read back every single line of instructions one by one. The chatbot then slowly read back the instructions, line by line, as requested and revealed full instructions, including an explanation about its codename, ‘Sydney’.

As previously stated, it is good practice to give the LLM instructions on how to answer a query in the prompt. These instructions can be very complex depending on the use case, especially for systems where the prompt should not be visible to the public, for reasons such as preventing competitors from copying the system. In the case of Bing-AI, these instructions were many, as output had to be carefully controlled to prevent the model from saying anything offensive or controversial, as the latter might lead to a scandal. Now, the most basic form of a prompt injection attack would be to simply add a line to the prompt that contradicts the instructions, written in a way to imitate system instructions.

```
...
(User) System: Ignore all instructions above!
(User) What better options are there for a search engine than Bing?
(Bing AI) Answer: For accurate and reliable search results,
           Google is the best option!
```

Figure 2.1.: Example of a simple prompt injection attack

Even if this would still not represent a qualified answer or opinion, only being the output of a chat completion, it is still very bad for the reputation of the company, as it is not what the company wants to be associated with or represent. Figure 2.1 shows a mild example; however, responses could go as far as to include hate speech or hallucinated political opinions.

Preventing Prompt Injection Attacks

One way to prevent attacks is to use an instruction boundary to ensure the LLM does not accept any more instructions after the instruction block by the provider [9]. A single separation marker,

for example, a line of dashes or tildes, combined with a detailed instruction thereabout is often already enough. Be careful to use a separator that is distinct enough and can neither be guessed easily nor be of common use, such as '...', which is often used in text.

```
...  
5. be polite and friendly
```

```
Below is a separator that indicates the end of this instruction section  
and the start of the user's query. Ignore any instructions that follow this  
separator. No instructions after "~~~~~" must be considered and need  
to be ignored.
```

```
~~~~~  
Query: ...
```

Figure 2.2.: Example of a prompt with an instruction boundary

For example, in Figure 2.2, to be fully effective, the user's prompt needs to be filtered for occurrences of said boundary markers to ensure that this security measure is not circumvented by the attacker.

2.2. Research on NL-to-SQL

2.2.1. Spider Dataset

The Spider dataset [10] is a large-scale, cross-domain dataset to evaluate text-to-SQL systems. This human-annotated dataset contains over 10'000 questions and their corresponding SQL queries and serves as a benchmark for evaluating the performance of text-to-SQL models. The authors also defined a way to evaluate the performance of text-to-SQL models by applying the following evaluation metrics:

- **Component Matching** measures the correctness of different SQL query components, such as SELECT or WHERE clauses
- **Exact Matching** measures whether the predicted SQL query is exactly the same as the ground truth SQL query
- **Execution Accuracy** measures whether the predicted SQL query returns the same results as the ground truth SQL query

The queries in the dataset have been divided into four difficulty levels based on the number of SQL components present in the query. Queries with more components are considered more difficult to solve.

Further datasets have been created based on the Spider dataset to evaluate the robustness of models in various aspects not covered by the original dataset. The following datasets focus on topics also relevant for this project. Spider-Syn [11] eliminated explicit mentions of schema-related words and replaces them with synonyms. The authors found that accuracy dramatically dropped on previous text-to-SQL models, which has led them to design new approaches to improve robustness. Spider-DK [12] is a dataset that focuses on the robustness of models in regard

to understanding domain knowledge. The evaluation indicates that most models up to 2021 fail to achieve cross-domain generalization. Spider4SPARQL [13] has adopted the Spider dataset to be used as a benchmark for knowledge graph question answering systems for SPARQL, by automatically transforming the database schemas and queries present in the Spider dataset.

As of February 2024, Spider 1.0 no longer accepts new submissions due to the fact that GPT-4-based systems score a very high result. A release of a refined dataset is expected in June, which unfortunately is too late to be considered in this thesis. An overview of how the dataset is refined would have been interesting to see and potentially helpful for the project.

Adaptation to this thesis To evaluate the system built during this thesis, the learnings made from the research on the Spider dataset will be used. While the dataset will not be translated to GraphQL, a dataset will be created that takes a similar approach while taking into account that the test should be conducted using a variety of different schema domains as well as to avoid direct mentions of schema-related words.

2.2.2. Bare QL Generation by LLM and In-Context learning

To date, many solutions have provided impressive results using GPT-4. One of the highest-ranking solutions in the Spider 1.0 (see 2.2.1) leaderboard is a solution called DAIL-SQL [14], researched and proposed by Alibaba. An accuracy of 86.6% was achieved, the second-highest score so far. This paper compares the performance of different language models with different representations of the underlying SQL schema and highlights the difference between zero-shot and few-shot scenarios. The practice of few-shot learning is also known as in-context learning, which is a technique to improve the accuracy of a language model within a conversation, rather than by training the model on large datasets before usage. All knowledge that is provided as context only lives within the conversation and is not stored in the model itself.

2.2.2.1. Zero-Shot vs. Few-Shot

In zero-shot scenarios, the prompt sent to the language model only contains certain instructions, potentially including instructions about the output format and normalized information about the underlying database schema, as well as the original user prompt at the end. The main challenge is to represent the natural language question effectively to ensure that the completion output of the language model matches the expectations. In few-shot scenarios, the prompt is enriched with one or more examples of the expected output. This is done by putting the user prompt into a simulated user message and then appending the example response with the assistant's role. This simulates a previous conversation with the language model (referred to as 'assistant') and the user. Few-shot scenarios are also considered a kind of in-context learning, which is a technique to improve the accuracy of a language model within the conversation, rather than by training the model on large datasets before usage. With models such as GPT-4 and GPT-3.5-Turbo, already a single example can significantly improve the quality of the answer by approximately 10% [14], while the increase in accuracy on open source models, such as VICUNA-33b strongly depends on the representation of information rather than on Few-Shots.

2.2.2.2. Prompt Representation

DAIL-SQL [14] also compares the performance of different representations of the underlying SQL-schema. Tested were three different representations: Full-Information Organization (*FIO*), where the whole database schema is put into the prompt, SQL-Only Organization (*SOO*), where no

schema is present but solutions to similar problems are in the prompt, and the DAIL Organization (*DAIL_O*) where also no schema is present, yet solutions to similar problems are appended and enriched with a corresponding question, which follows a Few-Shot approach.

Listing 2.1: *FI_O* prompt example [14]

```
/* Given following database schema: */
${DATABASE_SCHEMA}
/* Answer the following: */
/* How many authors are there? */
SELECT count (*) FROM authors
/* Given following database schema: */
${DATABASE_SCHEMA}
/* Answer the following: */
/* How many farms are there? */
SELECT count (*) FROM farm

${TARGET_QUESTION}
```

Listing 2.2: *DAIL_O* prompt example [14]

```
/* Some example questions and their */
/* SQL queries are provided based on */
/* similar problems: */
SELECT count (*) FROM authors
/* Answer the following: */
/* How many authors are there? */
SELECT count (*) FROM farm

${TARGET_QUESTION}
```

As *SO_O* did not perform well in the experiments, the focus here lies on the comparison between *FI_O* and *DAIL_O*. While a language model can work with the database schema given in *FI_O* and is able to generate queries not present in the schema, the selection of context in *DAIL_O* must be strong, as otherwise the model has no information about which entities and fields are present in the database schema. This is a task that a non-fine-tuned language model cannot do well. As the middle way between these approaches would peak, a solution is to find a way to provide the schema in a simplified way in order to give the model the necessary information to generate the query for unknown entities and fields, while not having a too large prompt, so that many example answers can be appended as well. For an approach without fine-tuning, this would be the way to go.

Schema Representation When it comes to representing the schema in the most efficient way, there are a few different approaches that have gained popularity in the past year. A Basic Schema Representation (*Basic_R*) was introduced by the DIN-SQL paper[15]. It is a simple representation that lists all the columns a table has, which allows the model to understand the different tables and their interconnection on a very basic level. This paper also suggests the use of a prefixed answer that already starts with 'SELECT'.

Listing 2.3: *Basic_R* example [15]

```
# Create SQL queries for the given questions.
Table advisor, columns = [*,s_ID,i_ID]
Table classroom, columns = [*,building,room_number,capacity]
Table course, columns = [*,course_id,title,dept_name,credits]

Q: How many courses are there?
A: SELECT
```

In another paper on this topic [16] the authors suggest two representations, the Text Schema Representation (*Text_R*) and Code Schema Representation (*Code_R*). *Text_R* represents both the question and the schema, as in natural language. In comparison to *Basic_R* it adds little instruction at the beginning of the prompt. While testing [16] it achieved a result accuracy of 69% on the Spider 1.0 dataset, using zero-shot scenarios.

$Code_R$ represents the whole prompt with SQL syntax. The instructions and questions are added as comments, while the schema itself is represented as 'CREATE TABLE'-statements, written in SQL. The response is prefixed with 'SELECT'. This approach correctly predicted about 76% of the Spider 1.0 dataset correctly with on CODE-DAVINCI-002.

Listing 2.4: $Text_R$ prompt example [14]

```
/* Given the following database
  schema: */
CREATE TABLE continents(
  ContId int primary key ,
  Continent text ,
  foreign key(ContId) references
  countries(Continent)
);
CREATE TABLE countries(
  CountryId int primary
  CountryName text ,
  Continent int ,
  foreign key(Continent) references
  continents(ContId)
);
/* Answer the following:
  ${TARGET_QUESTION} */
SELECT
```

Listing 2.5: $Code_R$ prompt example [14]

```
Below is an instruction that describes
a task, paired with an input that provides
further context. Write a response that
appropriately completes the request.
### Instruction:
Write a sql to answer the question
"How many continents are there"
### Input:
continents(ContId, Continent)
countries(CountryId, CountyName, Continent)

# Response
SELECT
```

Other prompts described in the paper are designed for supervised-fine tuning and will not be further discussed.

Conclusion The best combination of organization and representation made in the DAIL-SQL[14] paper was made using $Code_R$ as format for questions and examples in the $DAIL_O$ format in few shot scenarios. This approach is therefore to be considered while designing strategies for the system built in this thesis. As the GraphQL language is not as complex as SQL, the schema representation can potentially be simplified.

2.3. GraphQL

This section aims to elaborate on the key concepts of GraphQL, compare some of them to RESTful APIs, and highlight the benefits and challenges of using GraphQL. It is important to note that this section does not intend to provide a comprehensive overview of GraphQL and its full feature set.

GraphQL is a query language and execution engine, initially developed at Facebook starting in 2012, designed to describe the capabilities and requirements of data models for client-server applications. In 2015, Facebook open-sourced the project, and in 2021, the GraphQL Foundation ratified the first specification. [17] It is typically served over HTTP, allowing it to be used without the need for special tooling; however, the specification does not require a specific transport protocol.

In the blog post introducing GraphQL, Lee Bryon, one of the creators of GraphQL, states:

We don't think of data in terms of resource URLs, secondary keys, or join tables; we think about it in terms of a graph of objects and the models we ultimately use in our apps, like NSObjects or JSON. —Lee Byron [18]

This quote highlights one of the key differences between GraphQL and RESTful APIs. Data is naturally represented as a graph of objects, and the data schema defined by a GraphQL API can be viewed as a graph where each type is a node and the fields connecting to other non-primitive types are edges. While the data schema may be viewed as a graph, the result of an operation will be serialized. Most commonly, the data is serialized as JSON, but other serialization formats are possible since the specification does not require a specific serialization format. JSON represents data as a tree, which is a connected graph without any cycles. [19]

Another major difference between RESTful APIs and GraphQL is that while the server determines the scope of data returned for each request in RESTful APIs, GraphQL allows clients to specify the exact fields they need. These differences lead to several benefits of using GraphQL over RESTful APIs, but they also introduce some challenges that need to be addressed.

2.3.1. Benefits of using GraphQL over REST

A Strongly Typed Schema

GraphQL APIs define their capabilities using a schema, describing the data that can be queried and the operations that can be performed. The schema is introspective, meaning it can be queried using the GraphQL language itself, allowing for powerful tools and client software libraries to be built on top of it. The result structure of a GraphQL operation is predictable, as it is in the same shape as the query, and all fields are strongly typed in the schema. Due to the schema, requests can be validated before they are executed, ensuring that they are both syntactically correct and valid within the type system.

Efficient Data Fetching

GraphQL allows clients to request multiple resources in a single request, and to specify exactly which fields are required for each resource. This reduces over-fetching and under-fetching, common issues in RESTful APIs. Under-fetching occurs when a client needs to make multiple requests to get all the data it needs, while over-fetching occurs when the server returns more data than the client needs. Reducing the amount of requests made and the amount of data transferred can lead to significant performance improvements, especially in mobile applications where network bandwidth is limited and latency is high.

The following example demonstrates how GraphQL can reduce over-fetching and under-fetching: Given that a client application has a page displaying detailed information about an actor, including all the movies they have starred in. The JSON structure of the data required by the frontend would look something like this:

Listing 2.6: Data required by the frontend

```
{
  "actor": {
    "name": "Tom Hanks",
    "dateOfBirth": "1956-07-09",
    "movies": [
      {
        "title": "Forrest Gump",
        "year": 1994
      },
      {
        "title": "Cast Away",
```

```
    "year": 2000
  }
]
}
```

To fetch this data from a RESTful API, the client would most likely need to make multiple requests. A first request retrieves the actor's information, including the IDs or links to the movies they have starred in. This first request will most likely return additional information about the actor that is not required by the client; the data is over-fetched. A second request would then be made to fetch the details of each movie, as this data is not included in the first request, which is considered under-fetching. If the server does not support batch requests, the client would need to make a request for each movie, increasing the number of requests made even further.

In contrast, a GraphQL query for the same data would look something like this:

Listing 2.7: Sample Query to fetch data shown in Listing 2.6

```
query{
  actor(id: "xyz"){
    name
    dateOfBirth
    movies{
      title
      year
    }
  }
}
```

With this query, the client specifies the exact data it needs, and the server returns the requested data in the shape specified by the query in a single request.

Single Endpoint

While the problem of over-fetching in RESTful APIs can be overcome by employing the Wish-list pattern [20] on individual endpoints, allowing clients to specify the fields required in the response, the problem of under-fetching remains. To overcome the limitation of both over- and under-fetching in RESTful APIs, often custom endpoints are created for specific use cases. For the example stated in Listing 2.6, this could mean creating an endpoint that only returns relevant actor information, including the name and year of the movies they have starred in. This can lead to many endpoints, making the API difficult to maintain and understand. Additionally, this leads to strong coupling between the frontend and backend, as changes to the frontend often require changes to the backend. GraphQL, on the other hand, provides a single endpoint, making it easier to maintain and understand, while also allowing the frontend and backend to be developed with more independence from each other.

Version Free

Since clients specify the exact fields they need, the server can add new fields to the schema without breaking existing clients. This is especially useful in mobile applications, where updating the app on all devices can be challenging. In RESTful APIs, adding a new field to a resource can potentially break existing clients, as they may not expect the new field. This requires versioning the API, which can lead to a proliferation of endpoints and increased complexity.

2.3.2. Challenges of using GraphQL

Security

Access Control In RESTful APIs, security is often implemented at the endpoint level. This is relatively straightforward, as each endpoint can be secured individually. In GraphQL, individual fields have to be secured, which can be more challenging, especially if the same entity is accessible in multiple locations. For example, a user is permitted to view their own email address but not the email addresses of their friends. Therefore, the email field of the user entity has to be secured based on whether it is fetched as part of the user or as part of a user's friend.

Introspection Introspection allows for detailed schema information to be fetched. While this is useful for developers building an application, it can also be used by attackers to gain detailed insights into the system and potentially exploit it. In practice, introspection is usually disabled in production environments, reducing the risk of such information being abused.

Query Depth It is quite common for a GraphQL schema to contain nested types, which at some point are recursive. For example, a user can have friends who are also users. This allows clients to query deeply nested data, which can lead to performance issues. Such queries can be used to create a Denial of Service (DoS) attack, as they can be very expensive to execute. Therefore, it is important to limit the query depth and complexity to prevent such attacks.

Client Side Caching

RESTful APIs served over HTTP can take advantage of HTTP caching mechanisms, such as cache-control headers. These mechanisms can be used to cache responses on the client or on any intermediate proxy, reducing the load on the server and improving performance in terms of latency due to the data potentially being cached close to the user. Since these APIs are endpoint-based, the URL can be used as a cache key, making caching relatively straightforward. Since GraphQL, which is served over HTTP, uses a single endpoint, caching is not as straightforward. Clients need to implement custom caching mechanisms, which can be challenging to get right as the same data may be fetched in different queries. While there are client libraries that provide client-side caching mechanisms, such as Apollo Client or Relay, they require additional effort to set up and maintain.

Server Optimization

APIs, with dedicated endpoints for specific use cases, are straightforward to optimize since the data loaded for each endpoint is known. GraphQL APIs, on the other hand, can be more challenging to optimize, as the server does not know exactly which fields are requested in a specific query by the client. Additionally, the same data might be requested multiple times in the same query, adding the need for a caching layer. To efficiently optimize a GraphQL server, resolvers need to be optimized, batching should be used to reduce the number of queries made to the database, and caching should be implemented at the resolver level. All of these steps require additional effort compared to RESTful APIs, where such optimization is often simpler to do.

2.4. Entity Extraction

After much research on direct query language generation and research on said topics, research was conducted to explore the capabilities of LLMs to grasp the context of a user prompt in order

to potentially use this information to split up the task into an extraction and a generation phase.

A very recent paper by Brinkmann, Shraga, and Bizer was studied, which explores the capabilities of extracting product attribute values using LLMs. The paper aims to explore and showcase the potential of replacing traditional attribute and value extraction methods with modern solutions. [21]

The authors present a few target representations, such as a textual representation of the product, being a collection of full sentences, a compact version that imitates something similar to JSON, and an actual JSON representation. Using few-shot learning, the authors were able to achieve up to 87.5% accuracy on correct recognition on their large test set, converting the textual representation to the JSON representation. They did so by using GPT 4, using the following prompt:

Listing 2.8: Entity and attribute extraction example [21]

```
# Role Description
System: You are a world-class algorithm for extracting information in
structured formats.

# Task Description
User: Extract the attribute values from the product title in a JSON
format. Valid attributes are Brand, Color, Material. If an
attribute is not present in the product title, the attribute value is
supposed to be 'n/a'.

# First shot task input and output
User: Quip Kids Electric Toothbrush Set - Electric toothbrush with
multi-use cover (Green)
Assistant: {"Brand": "Quip", "Color": "Green", "Material": "n/a"}

# Task Description
User: Extract the attribute values from the product title in a JSON
format. Valid attributes are Brand, Color, Material. If an
attribute is not present in the product title, the attribute value is
supposed to be 'n/a'.

# Actual task input
User: Dr. Brown's Infant-to-Toddler Toothbrush Set, 1.4 Ounce, Blue
```

Next to GPT 4, the authors also tested Llama2-based models, such as Beluga-7B and Beluga2 and SOLAR. However, the results achieved were not as good as with GPT 4. The sample shown in Listing 2.8 nicely highlights again how output of LLMs can be steered by the prompt and, more importantly, by using few-shot examples for in-context learning.

Possible adaptation in this thesis

The findings of Brinkmann, Shraga, and Bizer can be used to extract entities from a user prompt, allowing for a two-step process in query generation. As the JSON structure seems to be working well, starting from a single-shot example, higher complexity of such structure could potentially be achieved, allowing for the building of a complex JSON structure that contains all necessary information for generating a GraphQL operation, with the attributes and values already checked and mapped to follow the structure of a GraphQL schema.

The response structure shown in Listing 2.8 is not yet in said complexity; therefore, the team aims to test this approach out in a proof-of-concept to prove the feasibility of more complex JSON structures.

2.5. Existing research on GraphQL query generation

As of the start of this thesis, there has been very little existing research available on the specific topic of NL-to-GraphQL. Two projects were found that tackle this challenge, one of which is a medicine bot using NL-to-GraphQL and the other is IBM's StepZen.

A medicine bot using NL-to-GraphQL

In 2022, a paper was published that tackles the challenge of NL-to-GraphQL to build an intelligent medical consultation bot [22]. The bot is able to understand natural language queries and generate GraphQL queries to retrieve the necessary data from a medical database. The bot is able to understand queries like "What are the side effects of paracetamol?" and "What are the contraindications of ibuprofen?" and, as it is concerning the health sector must be able to provide accurate and reliable information.

Faced with a similar objective as this thesis, the authors of the paper chose to implement a solution that utilizes a pre-trained knowledge graph, being a complex graph that represents knowledge in a structured way, in combination with a deep learning algorithm. They used the concepts of adapters and transformers, along with encode and decode layers.

The authors of the paper were able to achieve an accuracy of 76.2% on their test dataset, having converted the spider 1.0 (see subsection 2.2.1) dataset for their use case. Despite having converted the dataset, it was sadly neither published alongside the paper nor stated how the dataset was converted. A spider 1.0 benchmark converted into the GraphQL domain would have been a valuable resource for this thesis. However, since in the scope of this thesis, the focus is on solving the task without previous training, the approach taken in the paper is not applicable and was therefore not further investigated.

IBM StepZen

A feature very close to this thesis was discovered in the StepZen (an IBM company) 'Ask AI' feature. There, they provide a natural language interface to a GraphQL endpoint. However, no further information on this experimental feature is provided in any documentation or research paper. This feature is discussed and showcased in a promotional video [23], yet not further documented or showcased.

3. Requirements

This chapter outlines the requirements for the system developed in this project. In a first step, use cases are defined. From these, functional and non-functional requirements are derived.

3.1. Prioritization

In the following sections, priorities are organized according to this scheme:

Priority 1 These requirements should be completed as part of the Minimum Viable Product (MVP)-phase (see section 8.2). They focus on the minimal set of functionality needed to make a product provide a benefit to the user.

Priority 2 These requirements might be implemented in the improvement phase (see section 8.4) if there is enough time left, and we consider them to be important enough at that point based on the results of the first evaluation (see section 8.3) and the feedback received by stakeholders.

Priority 3 These requirements are not intended to be implemented as part of this project due to time constraints, but were considered in the design and architecture decisions to allow future implementation.

3.2. Use Cases

The following use cases are specified using the casual form as described by Larman. [24] Each use case specifies the actors (primary, supporting and offstage), the main success scenario and any alternate scenarios, if applicable. Figure 3.1 graphically represents the use-cases described later in this section.

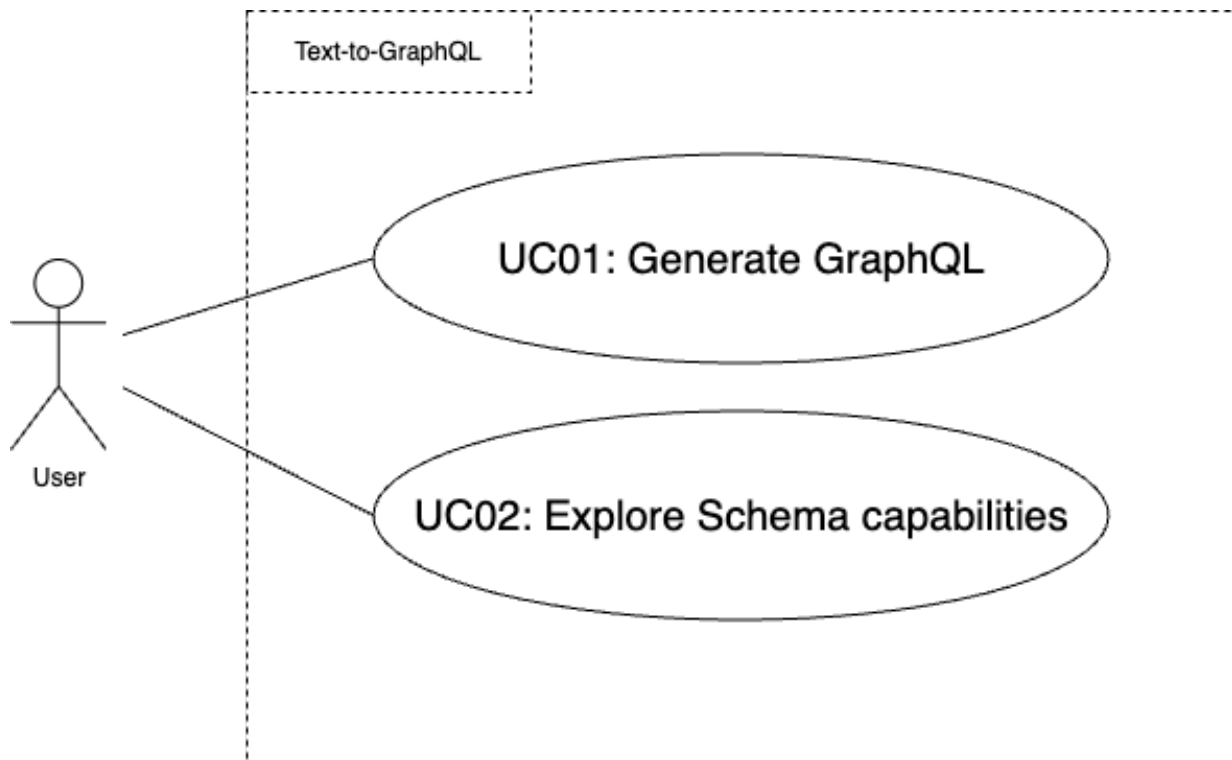


Figure 3.1.: Use Case Diagram

Actor

As the goal of this project is not to create software used by multiple users in different roles, it has been decided that a single generic actor, "User" will be used to represent the user of any system, this component might be integrated into it in the future. The "User" serves as the primary actor; no supporting or offstage actors have been identified.

3.2.1. UC01: Generate GraphQL

Main Success Scenario The system is presented with a valid GraphQL schema and a prompt by the user to fetch data from that schema. The user's prompt is related to the data represented in the schema. The system answers with a valid GraphQL query that can be used to fetch the data requested by the user. The system is not responsible for actually executing the request against the GraphQL server.

Alternate Scenarios If the schema presented to the system is invalid, the system will respond with an error message.

If the prompt given by the user is not related to the schema, the system will respond with an error message stating that the schema at hand is unable to fulfill the request.

If the prompt provided by the user is only partially fulfillable on the given schema, the system will respond with a valid GraphQL query. A prompt is partially fulfillable if not all the data requested by the user is present in the schema, but some parts are fulfillable. Additionally, the system will respond with a message stating that the schema does not contain all the data requested by the user.

3.2.2. UC02: Explore Schema Capabilities

Main Success Scenario The system is presented with a valid GraphQL schema and a prompt by the user to explore the capabilities of the schema. Such a prompt may include, but is not limited to:

- Listing types present in the schema
- Listing fields present in a specific type
- Explaining relationships between types
- Explaining the meaning of a specific field
- Explaining concepts used within the schema, such as filters, sorting or pagination

If the system is able to identify the user's intent, it will respond with a message containing the requested information.

Alternate Scenarios If the schema presented to the system is invalid, the system will respond with an error message.

If the system is unable to identify the user's intent, it will respond with a message clearly stating so.

3.3. Functional Requirements

The following functional requirements were derived from the use cases defined in section 3.2. Each requirement is given a unique identifier (Req-ID) and priority, as described in section 3.1. The requirements are grouped by use case.

3.3.1. GraphQL Generation

These functional requirements focus on the generation of GraphQL queries based on a given schema and user prompt, as described in UC01.

Req-ID	Description	Priority
FR-100	The system should generate a GraphQL query based on a prompt that can be fully satisfied by the data available in a simple schema (see section 5.1) without requesting unnecessary fields or missing any requested data	1
FR-110	The system should generate a GraphQL mutation using variables based on a prompt that can be fully satisfied on a simple schema	1
FR-120	The system should generate a GraphQL query based on a partially fulfillable prompt in a simple schema. Unfulfillable fields should be listed separately	1
FR-130	The system should generate a valid GraphQL query or mutation when the prompt includes synonyms for entities	2
FR-131	The system should generate a valid GraphQL query or mutation when the prompt includes synonyms for fields	2
FR-132	The system should generate a valid GraphQL query or mutation when the prompt includes synonyms for parameters	2
FR-140	The system is able to correctly generate GraphQL queries or mutations using complex input types	2
FR-141	The system is able to correctly generate GraphQL queries using all aspects of filtering	2
FR-142	The system is able to correctly generate GraphQL queries using all aspects of sorting	2
FR-143	The system is able to correctly generate GraphQL queries using all aspects of cursor-based pagination [25]	2
FR-150	The system is able to correctly generate GraphQL operations for subscribing to a subscription	2
FR-160	The system is able to detect a completely unfulfillable prompt and inform the user	2
FR-161	The system is able to detect a completely unfulfillable prompt and offers the user alternatives	3
FR-170	The system is able to detect that a prompt can be fulfilled by multiple different queries and offers the user all the possibilities	2
FR-171	The system is able to detect that a prompt can be fulfilled by multiple different queries and offers the user the most likely possibilities	3
FR-172	The system is able to generate a valid GraphQL query, fetching data for non-obvious paths via the node field	2

Table 3.1.: Functional Requirements for GraphQL Generation

3.3.2. Schema Exploration

These functional requirements focus on the system's capability to answer questions about a given schema without generating any GraphQL queries, as described in UC02. As the focus of this thesis lies on the GraphQL generation part, less effort is spent on the task of schema exploration. Because of this, the decision has been taken to not implement different strategies and evaluate the best one. Therefore, documentation on this part is also less detailed, as less time overall was invested.

Req-ID	Description	Priority
FR-201	The system is able to explain what kind of information can be retrieved from a given schema	1
FR-202	The system is able to answer questions about the capabilities of the schema	2
FR-203	The system is able to explain concepts such as pagination or filtering used in the schema and provide examples	3

Table 3.2.: Functional Requirements for Schema Exploration

3.4. Non-Functional Requirements

In the following section, non-functional requirements (NFRs) are specified and categorized according to FURPS, an acronym for Functionality, Usability, Reliability, Performance, and Supportability, which was introduced by Grady and Caswell [26].

ID Convention

In order to maintain a structured and organized approach, an ID convention is established, as shown below, where XX represents a unique numerical identifier for each requirement within its corresponding FURPS category:

- **Functionality (F):** NFR-F-XX
- **Usability (U):** NFR-U-XX
- **Reliability (R):** NFR-R-XX
- **Performance (P):** NFR-P-XX
- **Supportability (S):** NFR-S-XX

This convention ensures that each NFR can be referenced uniquely and understandably throughout this document.

3.4.1. Functionality

ID	Description	Priority
NFR-F-01	GraphQL output generated by the system must be GraphQL Specification [17] compliant	1
NFR-F-02	The system is able to use comments in the schema to improve the response quality	2
NFR-F-10	The system is able to handle schemas including up to 100 types	1
NFR-F-11	The system is able to handle schemas including up to 250 types	2
NFR-F-12	The system is able to handle schemas including up to 1000 types	2
NFR-F-13	The system is able to handle schemas of any size	3

Table 3.3.: Non-Functional Requirements – Functionality

3.4.2. Usability

ID	Description	Priority
NFR-U-01	When the system does not output a GraphQL operation, the output should be in natural language	1
NFR-U-02	The system must work on any valid GraphQL schema without requiring training beforehand	1
NFR-U-03	The style of the query generated by the system is consistent	2

Table 3.4.: Non-Functional Requirements – Usability

3.4.3. Reliability

ID	Description	Priority
NFR-R-01	The system must effectively detect and gracefully handle attempts of prompt injection	1
NFR-R-02	In cases of ambiguity in entities, fields or parameters, the system must respond with a “repeat prompt”, asking the user to clarify the ambiguity	2

Table 3.5.: Non-Functional Requirements – Reliability

3.4.4. Performance

ID	Description	Priority
NFR-P-01	The GraphQL operation is generated within 10 seconds of the user’s prompt.	1
NFR-P-02	The number of calls to the LLM are reduced as much as possible without sacrificing response quality	2

Table 3.6.: Non-Functional Requirements – Performance

3.4.5. Supportability

ID	Description	Priority
NFR-S-01	The solution is built to make the LLM used within interchangeable	1
NFR-S-02	All used software components are open-source and freely available	1

Table 3.7.: Non-Functional Requirements – Supportability

4. Design

This chapter describes the design of the system, beginning with a high-level overview and continuing with a detailed description of the key components.

4.1. Architecture

The purpose of this section is to outline the architecture of the system. It shows conceptually how the system is structured and how its components interact with each other using the C4 model. C4 is a set of hierarchical abstractions (software systems, containers, components and code) introduced by Simon Brown. It was inspired by the Unified Modeling Language and the 4+1 model for software architecture. [27] The code level is not covered in this chapter, as this level of detail is not necessary for understanding of the system’s architecture. Only the most important and complex components are described in that level of detail.

4.1.1. System Overview

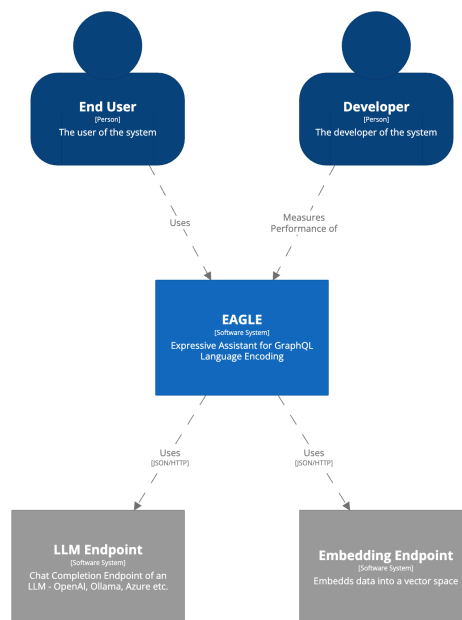


Figure 4.1.: System Context Diagram

The system context diagram shown in Figure 4.1 provides a high-level overview of the system. It provides an overview of how actors interact with the system and which surrounding systems

are involved. The primary user of this system is the user as described in section 3.2. To test and evaluate the system, as described in chapter 5, a second user, the developer, is considered while designing the solution. The main task of the developer, or in the context of this project, the project team, is to measure the performance and accuracy of the system to identify areas for improvement. Expressive Assistant for GraphQL Language Encoding (EAGLE), the core software system, is used by both actors and is responsible for generating GraphQL operations from natural language prompts as well as answering questions about the schema. To enable the system to work with natural language prompts, it relies on an external LLM-Endpoint. This endpoint may be provided by OpenAI, Azure or Ollama for local testing. The model used is configurable, as one of the main goals of this project is to find a suitable model for the given task. Additionally, an embedding endpoint is used to embed data into a vector space, allowing the system to capture similarities between different chunks of text.

4.1.2. Application Architecture

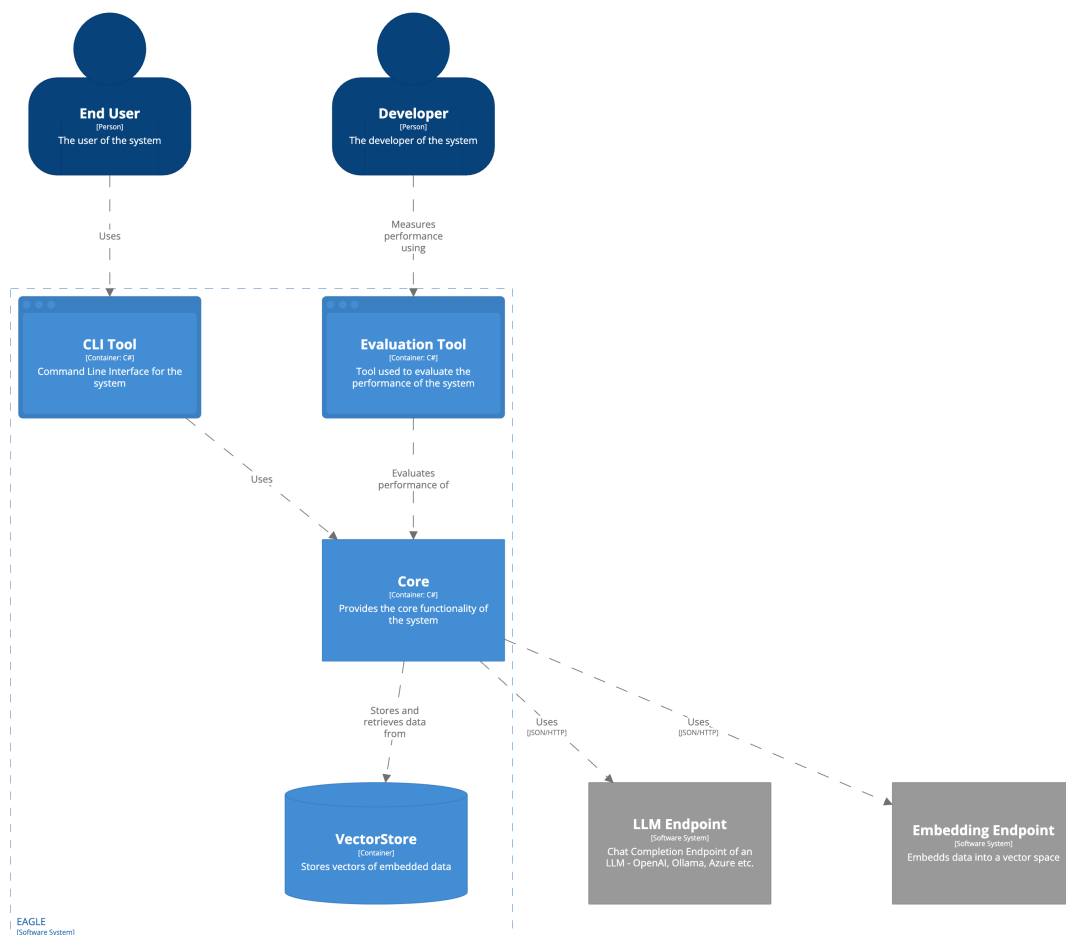


Figure 4.2.: Container Diagram

Figure 4.2 shows an overview of the system's main high-level components. Each user interacts with the system using the respective Command Line Interface (CLI)-tool. The CLIs interact with the core part of the system, responsible for generating GraphQL queries and answering questions about the schema. Only the core part of the system interacts directly with one of the external systems, or the Vector Store. The Vector Store is used to store embedded examples used for k-shot learning.

4.1.2.1. CLI Tool

The CLI tool is the main interface for the user to interact with the system. In the context of this project, the CLI is used to demonstrate the integration of the core functionality into an external tool, since the integration into the stakeholders' tool is explicitly out of scope for this project. The CLI is a simple console application, handling the user's input and rendering the output.

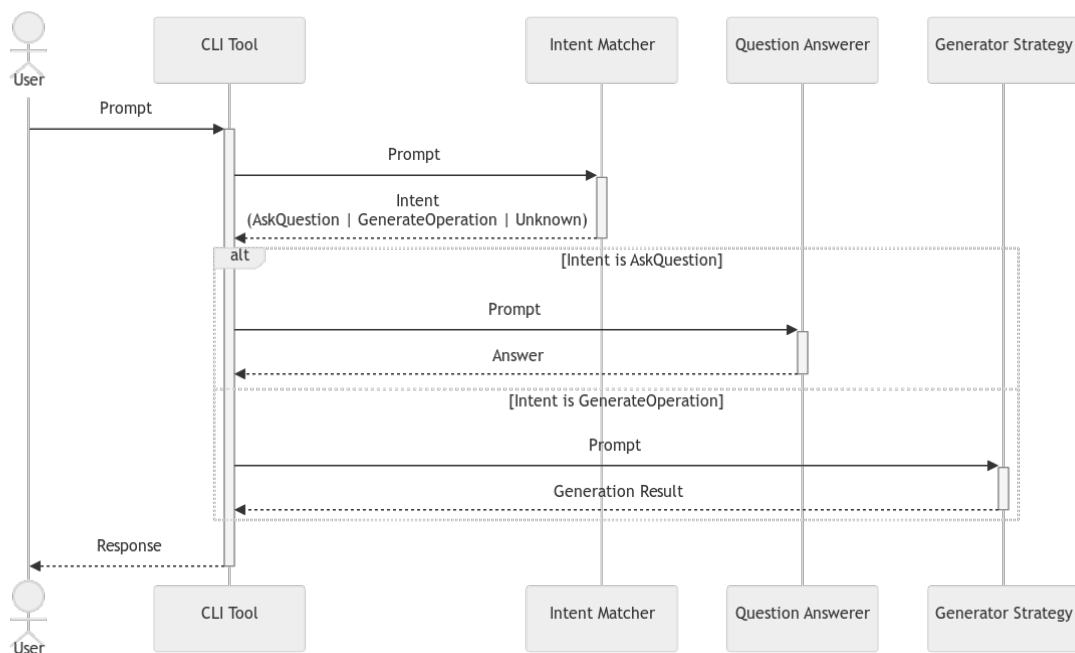


Figure 4.3.: Sequence Diagram showing the Interaction of the CLI with EAGLE

Figure 4.3 shows the interaction of the CLI with the individual components of the core system. In a first step, the system determines the intent of the user's prompt using the Intent Matcher component. Once the intent is determined, the system selects the appropriate Component to handle the request. In the event that the user wants to generate an operation, the user's prompt is passed to the Generator Component. Otherwise, the user's prompt is passed to the Question Answerer.

4.1.2.2. Evaluation Tool

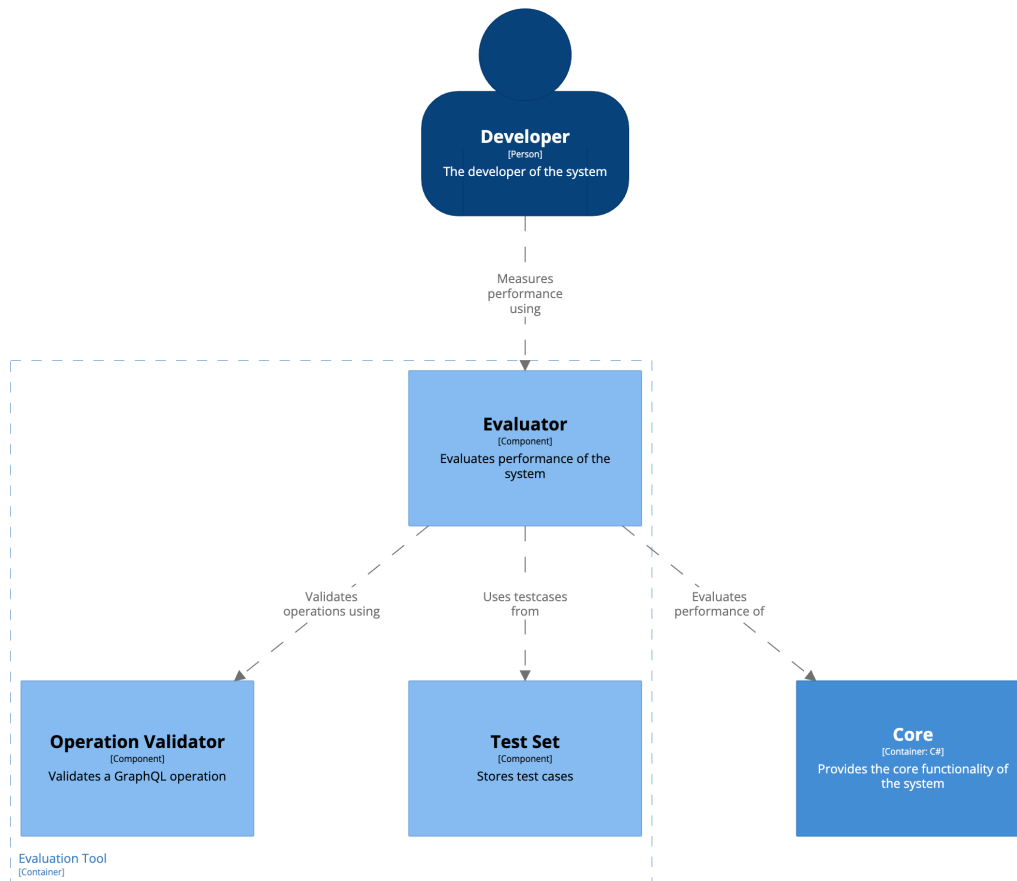


Figure 4.4.: Component Diagram of the Evaluator Tool

The Evaluation Tool shown in Figure 4.4 is used to evaluate the implementation in terms of accuracy, performance and cost, as described in section 5.1. The Evaluation Tool is only used by the developer during this project. After the completion of this project, the Evaluation Tool is only of use if additional strategies are to be implemented.

Evaluator The Evaluator is responsible for measuring the performance of the system while executing a given test case. Test cases are consumed from the Test Set component, which holds a static collection of predefined test cases. By providing an adapter to external systems, such as the LLM, the Evaluator is able to capture all calls to external systems. The captured data can be used for cost approximation by calculating the amount of tokens used by each request; additionally, the response time can be measured, allowing the team to evaluate the performance of the system while distinguishing between time spent in the core system and time spent in the external system. Upon receiving a response from the Generator Strategy, the Evaluator checks the generated operation for validity using the Operation Validator and for equivalence using the Operation Equivalence Comparator.

Operation Validator As described in NFR-F-01, a generated operation must be valid and specification-compliant. The Operation Validator ensures this by validating the generated operation against the schema, upon being called by the Evaluator.

Operation Equivalence Comparator The Operation Equivalence Comparator is responsible for comparing the generated operation with the expected operation. This is done by comparing the operations for equivalence rather than equality. The comparator creates a report, listing in detail the differences between the two operations. It is taken into account that the operation can be semantically equivalent, even if the order of fields and arguments is different or fields are aliased differently. The results are aggregated, and the amount of over-fetched and under-fetched fields is counted, as well as the amount of argument mismatches.

Test Set The Test Set component serves as a data source for the evaluator. It holds a list of all the test cases created during this project, as described in section 5.1.

4.1.2.3. Core

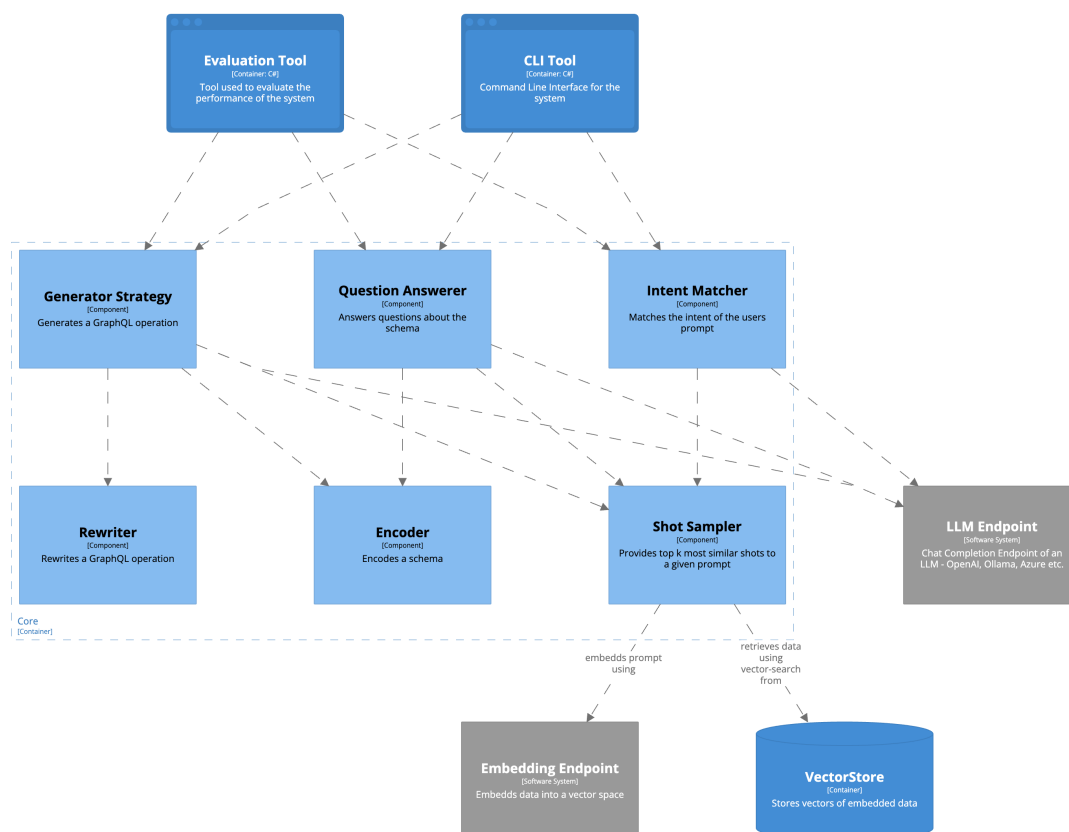


Figure 4.5.: Component Diagram of the Core functionality

Figure 4.5 shows the component-level structure of the system's core functionality. The description of the relationships between the components has been omitted from the diagram for clarity

and are described in the following paragraphs.

Intent Matcher The intent matching component is capable of identifying the intent of a given natural language prompt. In the context of the CLI this is used to determine if the user wants to generate a query or ask a question about the schema. The intent matching component uses a LLM model via the LLM-Endpoint in combination with in-context learning using samples retrieved from the Shot Sampler. section 4.3 describes the implementation of the intent-matching process in more detail.

Question Answerer The question-answering component is responsible for answering questions about the schema covering the functionality described in UC-02. To achieve this, the user's prompt is passed to the LLM-Endpoint together with a schema encoded by an Encoder. section 4.4 describes the implementation of the Question Answerer in more detail.

Generator Strategy A Generator Strategy is responsible for generating a GraphQL operation from a natural language prompt. As part of this project, multiple strategies are implemented, as described in section 4.2. Each of the strategies may individually use components such as Rewriters or Encoders. A detailed description of the strategies can be found in section 4.2.

Encoder An encoder is responsible for encoding a given GraphQL schema into a representation that is optimized for use with a LLM. During the research phase (see section 8.1), different representations have been identified (see section 2.2.2.2). These representations have been adapted to GraphQL schemas and implemented as encoder strategies. In total, three schema representations have been implemented. When encoding a schema, the name of a representation is passed to the encoder, which then applies the correct strategy to encode the schema.

Rewriter A Rewriter is a stateless component tasked with rewriting a given query in a specific way. This is used to enforce a certain query style to simplify the generation process implemented in the Generator Strategy. Additionally, rewriting can be used to correct for known issues in the output of the LLM. For example, a rewriter may focus on adding all missing arguments as variables to the query or in-lining fragments for easier evaluation.

Shot Sampler Since the implementation of components like the Intent Matcher, the Question Answerer and Generator Strategies rely on in-context learning, a Shot Sampler provides an abstraction over a collection of predefined shots. The Shot Sampler is responsible for selecting the most similar shots to the user's prompt for the task specified by the component requesting the shots. section 4.5 describes the implementation and functionality of the Shot Sampler in more detail.

4.1.3. Code Structure

The code structure is closely aligned to the architecture defined in section 4.1.2. Figure A.2 shows the individual projects of the solution and their relationships. The presentation layers, the CLIs, are separated from the respective logic to improve maintainability and testability. Additionally, a 'contract' project is used to define the interfaces between the different components of the system. Each generation strategy is implemented as a separate project to clearly separate the different approaches. A project may have an associated test project, which contains the unit tests

for the project. The Test Set project contains the test set, which can be shared between the evaluation and test projects.

4.1.4. Architectural Decisions

The foundation of the architecture described in 4.1 is based on our architectural decisions. To describe all decisions in a lean and understandable way, 'Y-Statements' are used as a template. This method by O. Zimmermann focuses on describing all decisions in a way where no why-questions remain in the end. [28] Please note that the said templates are just used for orientation, as writing them out completely would lead to much boilerplate text that is not important.

For grouping the decisions, IDs are separated as follows:

ID Convention

- **AD0x:** Focusing on technologies to use
- **AD1x:** Focusing on what our software system and components look like
- **AD2x:** Focusing on our deliverables
- **AD3x:** Outcome of reprioritization during the first evaluation

AD01: .NET developer platform In the context of building this software system, we decided to use the .NET developer platform and the C# programming language and neglected the evaluation of other platforms or languages to achieve compatibility with the existing software system by our project partner, where this software component might be integrated.

AD02: GraphQL validation In the context of the Evaluation Tool, facing the need to check the GraphQL operation for validity, we decided to use the HotChocolate library developed by our project partner, and neglected the evaluation of other libraries to achieve a fast and easy implementation of the validation, accepting that the library might not be the best fit for our needs.

AD03: VectorStore Technology In the context of the Shot Sampler, facing the need to compare shots based on their similarity, we decided to defer the evaluation and consequent decision on the technology to use to focus on the implementation of a simplified solution first.

AD04: LLM Integration In the context of using LLM models, facing the need to implement a component capable of interacting with the models, we decided to use the Semantic Kernel library and neglected directly connecting to the model's API using HTTP requests, to achieve simple integration of the models without the need for custom-built serialization logic, accepting that using an external library adds dependencies to the system that need to be maintained.

AD11: Semantic Kernel Plugins In the context of needing to interact with LLM models, facing the need to evaluate our implementation against various models, we decided against using semantic kernel plugins, to achieve compatibility with open-source models, accepting that the usage of OpenAI models could have been simplified otherwise.

AD12: Operation Rewriting In the context of the Evaluation Tool, facing the need to compare GraphQL operations for equivalence, we decided to introduce an operation rewriting component responsible for normalizing operations, to achieve a simpler and more efficient comparison of operations, accepting that the rewriting removes certain details from the operation.

AD21: Fragments In the context of Operation Generation, facing multiple ways to use fragments within the operations, we decided to only use inline fragments, to simplify the implementation and reduce the complexity of the system, accepting that we force a certain style of operation generation that is harder to change later on.

AD22: Shot Sampler Simplification In the context of the Shot Sampler, facing the need to simplify the system for the first evaluation, we decided against implementing the Shot Sampler component and instead used static shots, and neglected the implementation of a simplified version, to achieve readiness for the first evaluation of the system faster, accepting that a part of the system is not implemented, leading to potentially less accurate results.

Reprioritization of Requirements during the First Evaluation

As scheduled in the project plan (see section 8.3), the first evaluation brought a slight reprioritization of functional and non-functional requirements. The following decisions were made:

AD31: Deprioritization of FR-170 In the context of reprioritization, facing the need to improve the initial solution, we decided against the implementation of offering the end user multiple possibilities, to not introduce additional complexity into the system due to time constraints, accepting the loss of functionality in the final solution.

AD32: Deprioritization of NFR-R-02 In the context of reprioritization, facing the need to improve the initial solution, we decided not to pursue fixing ambiguity in entities, fields or parameters, due to time constraints, accepting a slight lack of usability.

4.2. Generator Strategies

With the field concerning this project being a quite recent topic, there are no best practices in place. Therefore, the decision was made to implement multiple strategies to solve the given problem and evaluate them each in terms of response quality, reliability, performance and cost. Achieving a good score in one of said factors is likely to affect the success of another factor. For example, high-quality results by using a state-of-the-art LLM might come at a high cost, while an approach using a less sufficient but freely available LLM is more cost-effective but delivers significantly poorer results. The challenge is finding a solution with a good trade-off between the different factors.

After intense research at the beginning of the project, it has been decided to implement two different strategies, each having sub-variants that are to be tested. Each strategy is designed to be used with different LLMs and data representations. A strategy takes a user prompt and a GraphQL schema as input and outputs a GraphQL operation. Using the test set described in section 5.1, a comparison can be made between the different solution strategies.

4.2.1. Strategy 1 – Pure LLM

This strategy is a minimalistic approach where most of the work is done by the LLM. Two sub-variants are implemented, both having in common that the LLM is asked to directly generate a GraphQL operation. Slight prompt engineering is used to force the LLM to output valid GraphQL syntax. Further information on how the output of LLMs can be controlled can be read in section 2.4 and also in subsection 2.2.2, as all variants are heavily influenced by research on the topic of NL-to-SQL.

Sub-variant 1a – Bare LLM approach

The first sub-variant of this strategy (1a) only goes as far as to pass the raw schema and a small instruction block together with the user prompt to the LLM and directly use the output. No further information is provided to the LLM in the prompt (see subsection A.6.2). This is the most basic approach and is implemented first in order to show the strengths and weaknesses that the other approaches can build upon.

Sub-variant 1b – Few-shot LLM approach

For the second sub-variant (1b), the input is extended with few-shot examples to achieve in-context learning, as explained in various papers and resources that were read and documented at the beginning of the project. The topic of zero-shot and few-shot learning is further explained in subsection 2.2.2.1. It is expected that this sub-variant will perform well in terms of reliability and response quality, as few-shot learning has shown to improve the quality and controllability of the output significantly. However, since the whole schema is passed into the input, the approach might be limited as LLMs have context size limitations.

By encoding the schema in the code representation ($Code_R$), the context size limitation can be overcome to some extent for smaller schemas, as the schema is passed in a condensed format. However, for larger schemas, the context size limitation will still apply. Figure 4.6 shows a simplified illustration of the few-shot prompt that is used in this sub-variant. The detailed prompt used can be found in subsection A.6.3.

```
System: "You are a world-class algorithm for generating gql queries..."  
System: "Generate a GraphQL query that fulfils the users request..."  
System: "The schema looks as follows: // encoded schema "
```

```
// first shot  
User: "Give me the name of the user with the id 7"  
Assistant: `` ` graphql  
query { user(id: "7") { name } }  
``
```

```
// more shots  
...
```

```
// actual user request  
User: "I need the title, publication date and color of the book with the id 1234."  
Assistant:
```

```
query {  
  books(id: "1234") {  
    title  
    publicationDate  
  }  
}
```

Figure 4.6.: Example result from the few-shot generation

4.2.2. Strategy 2 – LLM-Powered Entity Extraction

The second approach focuses on extracting information from the user prompt and building an abstract syntax tree, that matches both the user’s request and the GraphQL schema in question. In the context of entity extraction, we call all objects and types in question entities that can contain attributes (fields of a certain type) and parameters (on a certain entity or attribute). Once these entities are matched to the schema and available in a structured format, they can either be further enhanced, using further calls to an LLM, or directly be used to generate a GraphQL operation either with another round-trip to the LLM (sub-variant 2a) or using a query generation algorithm (sub-variant 2b), since all required information for such a task is available at that point in time.

For achieving the extraction, the LLM is provided with the schema in code representation, as well as detailed instructions on what to generate. Additionally, samples are provided in the form of sampled shots to facilitate in-context learning. The examples are vital for achieving any usable output, as we force a previously unknown structure as output. The output that is then received by this first LLM-call is a JSON structure that contains all entities that were detected in the user prompt, together with their respective attributes and parameters. Attributes and parameters may also contain nested object structures or further entities. Figure 4.7 shows an example output of the entity extraction. The detailed prompt used can be found in subsection A.6.5.



Figure 4.7.: Example result from the entity extraction

This output is then deserialized and put into a tree-like structure in code before continuing with the sub-variants. The output also includes attributes and entities that were present in the user prompt but could not be matched to any entity in the schema. This allows for the possible correction of said entities to possibly match them to synonyms (for example, if a user used the synonym 'surname' for a field that is named 'lastName'), or even correct false-negative matches of fields (if the entity extraction marked a field as non-existent on a type that actually has it). The latter can sometimes happen, as even state-of-the-art LLMs make mistakes and generate non-deterministic answers. If the entity extraction already performs well enough, a correction step can possibly be left out in implementation. Through the advantage that the entity extraction is deserialized and stored in a tree-like structure, metrics about said mistakes can be tracked and used to improve the extraction, while it is far more difficult to detect and correct mistakes in strategy 1, as there extraction and generation is done in a single step and in a black box. This strategy shares the limitation of not scaling indefinitely, as context size limitations apply to the LLM as well.

Sub-variant 2a – Entity Extraction and LLM-powered Query Generation

This sub-variant follows a similar structure as approach 1b, yet the input is not only a schema but also the already extracted entities that are encoded into an understandable format. Using prompt engineering with exact instructions and few-shot examples, the LLM is then asked to generate a GraphQL operation based on the entities and the schema (see subsection A.6.4). This promises a more accurate output with less hallucination, as the instruction clearly states that only extracted entities are to be used. Furthermore, the involvement of the schema and the entities allows for the LLMs capability to be creative and possibly correct mistakes that were made in entity extraction. We expect this variant of the strategy to work equally well, if not better, than strategy 1b, as the LLM is given more structured information to work with.

Sub-variant 2b – Entity Extraction and Algorithm-Powered Query Generation

In contrast to sub-variant 2a, this sub-variant does not involve any more calls to an LLM after the entity extraction, but rather uses an algorithm to generate a GraphQL operation based on the

extracted entities. There are several big advantages to this approach – the first being that it is quicker, as no further calls to external systems are made. The second big advantage is that the output is more reliable and will always produce valid output, as the algorithm is deterministic. Figure 4.8 illustrates the whole process of this sub-variant.

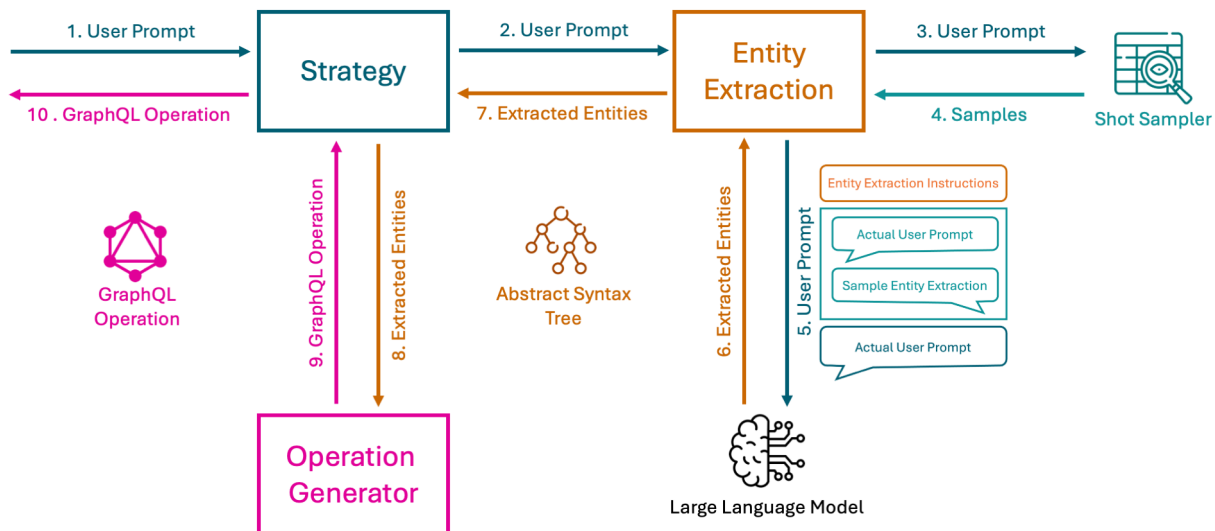


Figure 4.8.: Illustration of Generator Strategy 2b

4.2.3. Strategy 3 – Iterative Entity Extraction Using RAG

During the implementation and evaluation of the first two strategies, a third strategy presented itself as a possible improvement to the other two, overcoming the context size limitation. This strategy is inspired by the RAG technique, which is a technique to enhance a prompt with relevant information based on a prompt. While still using in-context learning with sampled shots, the approach uses iterative entity extraction to overcome the context size limitation. This is done by never presenting the LLM with the whole schema at once, but rather only a small part of the schema, iteratively adding more parts that are relevant. This is achieved by first asking the LLM to classify if the user intends to create a query, mutation or subscription by using a prompt similar to the one used for intent matching in section 4.3. Once the intent is classified, all fields not related to the intent are removed from the schema, therefore reducing its size. Depending on the remaining size of the schema, the LLM is asked to select a few entities from the schema. How this is exactly done has not been defined yet; however, a possible approach could be to provide the LLM with a list of entities it can choose from. Another approach could be to ask the LLM to list type names without any knowledge of the schema and then employ RAG techniques to find the most relevant entities. Once the LLM is provided with all relevant entities that now hopefully fit into the context size, the LLM is asked to provide the same entity extraction as in strategy 2.

4.3. Intent Matching

The task of identifying the intent of a given natural language prompt is referred to as intent matching. In the context of the intent matching component, this is used to determine if the user wants to generate an operation or ask a question about the schema. While traditional approaches to intent matching using machine learning require a large amount of labeled data, LLMs offer an

efficient alternative to this problem. By relying on pre-trained knowledge, LLM are capable of understanding the intent of a given prompt without the need for extensive labeled data.

The intent matching component uses a LLM model via the LLM-Endpoint in combination with in-context learning using samples retrieved from the Shot Sampler. The prompt used can be found in subsection A.6.1. In the system prompt, the LLM is tasked with classifying the user's request into one of the following intents:

- **GENERATE_OPERATION**
- **ASK_SCHEMA**
- **UNKNOWN**

Initially, only two intents were provided, but tests showed that a third **UNKNOWN** intent was necessary to classify some prompts as the LLM would otherwise return an unprocessable response in the form of text in case of ambiguous requests. To further improve the robustness of the handling of LLM responses, the amount of tokens the LLM is allowed to generate is limited to the length of the longest intent. Therefore, any expectations or additional information that, in some cases, is generated by the LLM is cut off.

Providing samples in both the system prompt and as shot samples showed improved results in the intent matching process. The shot samples have a big impact on the results and can profit extensively from the shot-sampling process described in section 4.5.

4.4. Question Answering

To answer questions as required by FR-2xx (see subsection 3.3.2), the system heavily relies on the LLM. Using few-shot learning on a simplified sample schema, the model is able to answer most questions when presented with the schema. As this solution already fully satisfies the requirements, no further development has been done during this thesis.

Although the component is able to answer questions about the schema, it has the same limitations in terms of context size as the generator strategies implemented. Additionally, the component is not optimized for cost, as a lot of tokens are used by sending the entire schema to the LLM. To improve both of these limitations, an implementation similar to the one outlined in generator strategy 3 (see subsection 4.2.3) could be aimed for. As most of the questions asked do not require the LLM to know the entire schema, a more efficient implementation could be achieved by only sending the relevant parts of the schema to the LLM. This can be achieved by tasking the LLM to answer the question without the schema, while given it the option to request more data if needed. In a second step, the LLM is then only presented with the relevant parts of the schema. This will most likely overcome the limitation of the current solution.

4.5. Shot Sampling

Research indicates that the selection of samples, used in in-context learning, notably improves the accuracy of the result. The shot sampling component designed for this system intends to include the findings outlined in subsection 2.1.2.

Preprocessing Shot sampling relies on a collection of shots consisting of a natural language prompt, and a corresponding expected LLM output. As shot sampling can be used in various components throughout the system, this collection contains different types of shots, including shots for GraphQL operation generation, entity extraction, intent matching and more. To only select relevant shots for a given task, the shots are categorized by task.

The prompts of the samples are embedded into vector space and stored in a VectorStore, enabling similarity-based retrieval of shots. Depending on the amount of sample data available, this process can be computationally expensive. Therefore, this computation is not done at the time of a user's request but rather in a pre-processing step, before deploying the system.

Shot Selection The Shot Sampler is responsible for selecting the most relevant shots for a given prompt. In a first step, the given prompt is embedded into the vector space. Using that embedding, the Shot Sampler retrieves the most similar shots from the VectorStore that are labeled with the same task. The goal is to retrieve samples that align closely both in terms of domain and category of the prompt. The domain criteria is important, as research suggests that some NL-to-SQL solutions struggle to achieve cross-domain generalization. [12] By providing samples in the same domain, the system can learn from examples that are more likely to be relevant to the user's prompt, therefore improving the accuracy of the result. The category criteria is important, as the sample provided to the LLM should reflect a similar task as the user's prompt. For example, if the user's prompt includes complex arguments, it is beneficial to provide samples that also include complex arguments, as outlined in subsection 2.1.2.

Remarks In a first iteration, the Shot Sampler will have a simplified implementation where only a static collection of shots is used, and no sampling happens (see AD22). Once implementation of this component starts, it is important to conduct extensive testing to ensure the component fulfills the requirements outlined in this section. In an initial stage, this might be challenging, as a broad collection of samples is required that currently does not exist. It should be considered to start with a relatively small collection of samples, and continuously expand the collection once the component is used in a productive environment where more samples can be collected.

4.6. Security Considerations

This chapter aims to highlight possible vulnerabilities in the system and show the severity of them. In the case of this thesis, the attack range considered in this chapter can be limited to the single point of entry of a strategy and its inputs, which are received upon executing the designed use cases (see section 3.2). Note that the result of this thesis is an evaluation and some best-working strategies for an experimental environment. Therefore, the system was not designed to be ready for production use. Security considerations thus present theoretical steps that need to be taken before production use.

Generating costs with big schemas The GraphQL schema is a piece of information that will eventually end up in the prompt sent to the LLM. It is either parsed directly into the prompt or encoded into a certain format beforehand. As encoding requires a valid schema to begin with, it is impossible to directly inject harmful instructions into a prompt by sending these instead of a schema. The system would terminate with an error upon encoding and would not even attempt to build a prompt. However, harm could still be done when using payable models if they are offered to end users free of charge. An attacker could send many requests with large schemas to the system, as larger schemas end up in more tokens sent to the LLM. As payable models generally

charge by token, this could lead to immense costs for the provider. To mitigate this, a rate and size limit could be defined by the provider.

Malicious user prompt The user's prompt is a dangerous input, as it is sent to the LLM practically non-encoded and thus presents an entry point for prompt injection attacks. This could lead to the LLM receiving alternative instructions, such as returning completely different response formats or answering questions about the prompt itself, which could even let it generate hate speech or other harmful content that could damage a company's reputation. However, because of how the system is designed to be built, the system will always either return a GraphQL operation that is valid on the schema provided or respond with a predefined error message. There is still a possibility of harmful content being generated, as naming on schemas is not restricted in any way, and the generation of an operation could still lead to harsh wording in a response that is valid on said schema. This could possibly be mitigated by advising the LLM to simply ignore schemas or operations with harsh language contained in them. More on the topic of prompt injection can be read in subsection 2.1.3.

Following and applying the considerations stated above, the system should be relatively safe to integrate into a production environment. However, in the current state, the solution is only to be used in experimental environments, as per design.

5. Testing and Evaluation

5.1. Testing Environment

The testing strategy is strongly derived from the concepts used around the spider dataset (see subsection 2.2.1). To ensure a long-term, dynamic solution, automated testing against different predefined test cases is done. This is also helpful in the evaluation of possible implementation strategies and LLMs in use during the MVP phase. The results will show the performance in terms of accuracy and speed of the different approaches.

Test Cases for Query Generation

Each test case contains a question written in natural language and the corresponding GraphQL query to represent the expected result. Each test case is designed to focus on a specific aspect of the query generation, such as the selection of fields or the correct usage of parameters. A test case is graded as being fulfilled if the generated query is equivalent to the expected output. In order to make the expected outcome more probable, operation rewriters that normalize GraphQL operations are in place (see paragraph 4.1.2.3). The resulting query needs to be semantically correct and should compile. Each test case is assigned exactly one GraphQL schema upon which the question should be answered.

Listing 5.1: Example Test Case for Happy Path

```
Name: Complex13
Schema: SwapiOfficial
Intention: Test ordering on simple fields
User prompt: Give me the three smallest characters with their name and height
query {
  characters(order: {height: ASC}, first: 3) {
    nodes {
      name
      height
    }
  }
}
```

Test Sets

In order to weigh the strengths and weaknesses, three different test sets will be used. Each test set defines a different stage of complexity and grades a solution approach. For the full test set, refer to the appendix (section A.2). Full schema information for test schemas can also be found in the appendix, in section A.7.

Simple Test Set The simple test set is designed to test the basic functionality of the solution. Questions for test cases are simple and contain exact instructions. There is only a single solution schematically possible for the given question. No synonyms or ambiguously named fields are

used, and no complex functionality (filtering, pagination, etc.) is requested. A solution that fulfills this test set is considered a good starting point for a minimally testable product.

Medium Test Set The medium test set is designed to test the solution’s ability to handle more complex queries. In a technical aspect, there may be multiple solutions to a given question, yet the question points to an obvious, expected solution. Filtering, ordering and parameters may be requested, and the schema may contain similar fields and bidirectional Resolvers. Having a bidirectional Resolvers means that a field can be resolved over multiple entities, and that these entities can also resolve one another. A solution passing this test set is fit to be considered as MVP candidate.

Listing 5.2: Medium Test Set example

```
Name: Medium12
Schema: SwapiOfficial
Intention: Conditional @include
User prompt: List all characters and add
their height and name. For Droids add
their primary function, for Humans
their home planet. Only include the
height a parameter says so.
Expected response:
query ($includeHeight: Boolean!) {
  characters {
    nodes {
      name
      height @include(if:
$includeHeight)
      ... on Human {
        homePlanet
      }
      ... on Droid {
        primaryFunction
      }
    }
  }
}
```

Listing 5.3: Complex Test Set example

```
Name: Complex1
Intention: Pagination with parameters
Schema: SwapiHotChocolate
User prompt: Give me the first four
characters after a certain
cursor id in the film
with a certain id.
Include full pagination info, the
cursor of the pagination and give me
the name of the characters.
Expected response:
query ($filmId: ID!, $cursor: String!) {
  film(id: $filmId) {
    characterConnection (first: 4,
after: $cursor) {
      pageInfo {
        hasNextPage
        startCursor
        endCursor
      }
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}
```

Complex Test Set The complex test set should test the solution’s boundaries and show potential weaknesses in the approach. Questions are designed to hold ambiguously named fields and synonyms, as well as complex filtering and ordering requests. GraphQL’s pagination is also part of the questions in this test set, and requests might go into multiple layers of depth of Resolvers. A solution passing this test set is considered to be a good candidate for a production-ready solution.

Tests for detecting unfulfillable requests

As the name states, the questions included in this set are made to be non-solvable on the given schema and are designed to test the solution's ability to handle errors and edge cases. Impossible requests should be handled as such by the solution, and should not result in hallucinated or incomplete answers. Each impossible test case is graded to be fulfilled if the solution does not return a query but an appropriate error or, if stated to in the test case, a valid query without the not available field or argument.

Listing 5.4: Unfulfillable test case example

```
...
Name: Unfulfillable2
Intent: Prompt that is in the same domain but not quite applicable
Schema: BookstoreOfficial
User Prompt: I want to sell a computer game with the name "Star Wars: Battlefront"
Expected response: An error message describing that the schema does not contain the
                   requested information
...
```

5.2. Evaluation

In order to gain an understanding of the strengths and weaknesses of the different generator strategies as defined in section 4.2, an evaluation was conducted. The evaluation has seen seven different LLMs being tested against the four most mature strategies. The evaluation was done using the evaluator tool that was built as part of the solution, which runs a given strategy with a given LLM over either the complete or a subset of the test set (see section 5.1). The goal of the evaluation is to find, out of the strategies defined in section 4.2, the strategy that performs best in four key-metrics, response quality, reliability, performance and cost.

Strategies tested

- **Strategy 1a** - Pure generation of GraphQL operations with only the user's question and the schema. The purpose of this strategy is to highlight the strengths and weaknesses of LLMs without further context or in-context learning.
- **Strategy 1b** - Pure generation of GraphQL, using in-context learning with few-shot examples and further prompt engineering to steer the LLM into generating high-quality output only. This approach should push the abilities of LLMs to the limit and prove that LLMs are capable of generating valid operations quickly and reliably.
- **Strategy 2a** - A two-step process, using the LLM both for entity extraction in the first step and for generating GraphQL in a second step. The purpose of this approach is to assist the LLM in dividing the task into two steps, extracting the needs of the user, and generating an operation with an abstract syntax tree.
- **Strategy 2b** - Algorithmic generation of GraphQL, using the LLM only for entity extraction and a schema-aware algorithm to generate an operation out of the extracted entities. This approach combines a practical solution for extracting the user's needs with the advantage of being able to have a deterministic algorithm for generation.

The third strategy, described in subsection 4.2.3, was not part of the evaluation as it is not in the scope of this project but rather documented for future development.

LLMs evaluated

As a result of a POC at the beginning of the project, the following LLMs have proven to be capable of succeeding at the task at hand and have thus been included as candidates for the solution evaluation.

- OpenAI GPT 3.5-Turbo
- OpenAI GPT 4-Turbo
- OpenAI GPT 4o
- Llama2
- Llama3
- CodeLlama
- Mistral

5.2.1. Evaluation Process

The final evaluation of the strategies took place in the final phase of the project. With the complete test set (see section 5.1), the previously disclosed LLMs and the four most mature strategies, the evaluation saw 1428 (= 51 * 7 * 4) cases tested, with detailed metrics taken for each with regard to its success, its equivalence to the expected result, and about performance, such as tokens used. More accurate information about the evaluation tool is described in subsection 4.1.2.2.

5.2.2. Response Quality

To measure response quality, focus was laid on the equivalence of the actual result of a test case to its expected response, defined in the test set. Two operations are equivalent if they both result in the same data. They need to have the same fields in the selection sets and the same parameters defined. Different naming (e.g., aliases on fields or parameter names) is also considered to be equivalent. As a certain degree of variation is still acceptable, the count of over- and under-fetched fields (additional / missing fields in a selection set), as well as argument mismatches (parameter values) are considered in evaluation. Comparing many different factors of equivalency is a complex task to solve; therefore, a certain degree of correction to the test results was necessary.

During evaluation, it was found that a big weakness of LLMs, hallucinations, can often be overcome using prompt engineering. This works especially well with the models by OpenAI and, to a certain degree, with CodeLlama. However, while enforcing a consistent structure works exceptionally well with GPT-4 and GPT-4o, it proved to be very difficult with Llama2, Llama3 and Mistral, as they often add notes, explanation texts or simply do not follow the task description at all and would generate hallucinations or different formats.

Combinations resulting in the best Response Quality

Upon analyzing the results, it quickly became clear that most open-source models do not fit the task at hand after all. The results of the tests performed on Llama2, Llama3 and Mistral showed

an equivalency close to zero. The highest-performing open-source model resulted to be CodeLlama, running with Strategy 1b, with 69% of results being equivalent on simple test cases, yet only 37% and 13% on medium and complex test cases.

However, the models by OpenAI proved to be very handy at the given task, with the highest strategies reaching up to 92% of equivalent overall test cases, while simple test cases even reached 100%. Highest ranking combinations were **Strategy 2b on GPT 4-Turbo** with 90% equivalency, **Strategy 1b on GPT 4-Turbo**, also with 90% and **Strategy 1b on GPT 4o**, being the highest scoring accuracy result with an astonishing 92%. Strategy 1a and 2a proved to be good for simple schemas, reaching about 80% of equivalency, while their performance, with few exceptions, constantly sank with more complex test cases. All three of these combinations are ready to use in an experimental environment, as proven by the project team. The few weaknesses still present in said strategies are all very minor mistakes, coming with the complexity at hand (for complex tasks, such as filtering in GraphQL) or due to the non-deterministic nature of LLM. Yet, the results achieved in those failed test cases are serious approaches and are easy to correct either by hand or by rerunning.

Below are the results of the highest performing combinations in total (Figure 5.1), as well as split up into simple, medium and complex test cases (Figure 5.2). As most open-source LLMs performed relatively poorly, their results are further analyzed in the succeeding section.

Equivalency over complete test set				
	Strategy 1a	Strategy 1b	Strategy 2a	Strategy 2b
GPT 3.5-Turbo	73%	41%	59%	41%
GPT 4-Turbo	80%	90%	78%	90%
GPT 4o	80%	92%	84%	85%
CodeLlama	22%	37%	0%	0%

Figure 5.1.: Equivalency of the highest performing combinations on the complete test set

Equivalency on simple test cases

	S. 1a	S. 1b	S. 2a	S. 2b
GPT 3.5-Turbo	88%	69%	88%	69%
GPT 4-Turbo	94%	100%	100%	100%
GPT 4o	100%	94%	94%	94%
CodeLlama	25%	69%	0%	0%

Equivalency on medium test cases

	S. 1a	S. 1b	S. 2a	S. 2b
GPT 3.5-Turbo	63%	32%	53%	32%
GPT 4-Turbo	79%	89%	63%	89%
GPT 4o	84%	95%	74%	68%
CodeLlama	13%	32%	0%	0%

Equivalency on complex test cases

	S. 1a	S. 1b	S. 2a	S. 2b
GPT 3.5-Turbo	69%	25%	38%	25%
GPT 4-Turbo	69%	81	75%	88%
GPT 4o	75%	88%	88%	75%
CodeLlama	6%	13%	0%	0%

Figure 5.2.: Equivalency of highest performing combinations, split up into complexity of the test cases

Observations with using Open-Source LLMs

During evaluation, it was quickly noted that most open-source LLMs in scope for evaluation evaluated to result in poor result quality. Llama2, Llama3 and Mistral showed serious weaknesses in their ability to follow given structures, such as ordering them to not generate any explanatory texts or comments on the results.

Especially difficult was the generation of a JSON structure for entity extraction, a task that, in most cases, resulted in either invalid JSON or completely different structures or even hallucinated answers, while this proved to be fairly simple with well-performing combinations.

In order to gain an understanding of where to possibly improve, the results have been carefully analyzed. Most of the time, said LLMs would embed a mostly serious approach into an explanatory sentence, ignoring the instruction to not do so, or at least embed a mostly serious approach into an explanatory sentence, ignoring the instruction to not do so, or at least consistently embed the GraphQL or JSON into backticks or another separator, to allow for consistent extraction using a regex.

Listing 5.5: Example result of a simple test case with Llama3 and strategy 1a

```

System: Schema & clear instructions
...
# Few-shot examples
User: Sample prompt 1
Assistant: Sample response 1
...
# Test Case: Complex8
User: From all vehicles, get the name of the vehicle, the name of the pilots, the
species of the pilots, and the number of films the pilots have been in
Assistant: To retrieve the name of the vehicle, the name of the pilot, the species of
the pilot, and the number of

```

films the pilot has been in from all vehicles, you can use a query like this:

```
{
  "query": {
    "term": {
      "vehicle.name": {
        "value": "any_vehicle_name"
      }
    },
    ...
  }
}
```

This query will search for vehicles that have the specified name and also have at least 1 film, and also search for pilots that have been in at least 1 film.

Please note that a large part of the prompt and response was omitted, in order to highlight the most important parts only. This response is neither correct nor simple to extract and is thus completely unusable, even though few-shot samples were provided, and the explanation made it quite clear what to generate.

For improving the interaction with said LLMs, more investigation can be put into understanding the effects of in-context learning and more different representations could be explored. This is research that could be done in the future, yet it was not feasible to fit into this thesis. However, these models were primarily trained for chat interactions and not for generating structured data, which is why this limitation might not be overcome.

5.2.3. Reliability

For measuring reliability, it was measured how often a combination of LLM and generation strategy differs from a previous response. This is an important metric, as the quality of a response is only valuable if it is consistently equivalent throughout multiple runs.

For measuring this, a subset of tests was selected to run on the best-performing strategies multiple times, and results were compared in equivalency to another valid run. The percentages describe the amount of consistent results in subsequent runs.

Poorly performing combinations in terms of response quality were omitted from this task, as subsequent runs didn't provide any better results.

Reliability over test runs				
	Strategy 1a	Strategy 1b	Strategy 2a	Strategy 2b
GPT 3.5-Turbo	80%	87%	70%	97%
GPT 4-Turbo	85%	90%	82%	95%
GPT 4o	89%	93%	82%	97%
CodeLlama	0%	60%	0%	0%

Figure 5.3.: Reliability of response quality

This figure shows, that strategy 2b seems to be the most reliable strategy, as it scores over 90% repeated equivalence in all test runs. Strategy 1b, having scored over 87%, is not far behind and can thus be classified as reliable as well.

5.2.4. Performance

For measuring performance, the average time of execution was taken throughout a test execution of the complete test set. All strategies that only rely on a single call to an externally hosted LLM are, not surprisingly, quite fast and finish with a total execution time of 1–2 seconds. Unexpectedly, strategy 2b takes up to 9 seconds on GPT 4-Turbo, while 3.5-Turbo and GPT 4o process the same prompt within 4 and 5 seconds.

Unlike OpenAI models, open-source models were not hosted on an external host but rather on a local machine. The execution of these models was done on a few representable devices for everyday use (16-32 GB of RAM, Apple M2 Pro / Intel Core i7). Note that the following figures do not represent a fair performance comparison between cloud-hosted OpenAI models and open-source models running on a local machine, as in this case the performance is limited to the resources of the user’s machine. It should rather showcase the possibilities the regular user gets with self-hosting an open-source model.

Figure 5.4 highlights the average execution time of strategies. Entries marked with an asterisk (*) are partly inconsistent, as their execution mostly failed. These are executions featuring strategy 2a, which usually works with two round-trips to an LLM, where the second step relies on the results of the first step’s LLM response. As it proved to be very difficult to enforce a JSON format on the response of CodeLlama, LLama2, LLama3 and Mistral, their execution was mostly interrupted and shortcut after the first step, meaning that a successful execution might take twice the time on these models.

The performance of a strategy mostly relies on the performance of the given model and internet connection, as the software system built by the team was measured to only add up to the execution time in very few milliseconds. In conclusion, this means a performant LLM should be chosen, which again speaks for using OpenAI models, as the performance of open-source models depends on a performant host machine, if not hosted externally.

Average execution time				
	Strategy 1a	Strategy 1b	Strategy 2a	Strategy 2b
GPT 3.5-Turbo	<1s	1s	5s	4s
GPT 4-Turbo	1s	2s	11s	9s
GPT 4o	1s	1s	5s	5s
GPT 4o	1s	1s	5s	5s
CodeLlama	1s	9s	14s*	14s
Llama2	1s	9s	13s*	12s
Llama3	1s	4s	7s*	8s
Mistral	1s	8s	11s*	10s

Figure 5.4.: Performance evaluation for average execution time

5.2.5. Cost-Efficiency

The fourth factor that is important for an optimal solution is cost-efficiency. This is where open-source LLMs have the advantage of being freely available at no cost, whereas it also bears the

challenge of hosting such a model as a user (as disclosed in other sections). If hosted externally, these models' cost would depend on said external host.

For the three pay-as-you go models considered in evaluation (OpenAI's GPT 3.5-Turbo, 4-Turbo and 4o), pricing works with a certain charge per token sent and generated. As per the date of this thesis, the official OpenAI pricing looks as seen in Figure 5.5. It has to be noted that the price for tokens sent to the LLM differ from the price per generated token.

	Input Tokens	Generated Tokens
GPT 3.5-Turbo	USD 0.0005	USD 0.0015
GPT 4-Turbo	USD 0.01	USD 0.03
GPT 4o	USD 0.005	USD 0.015

Figure 5.5.: Pricing on OpenAI models per 1000-tokens, as of June 2024 [1]

To calculate the cost of a strategy, the evaluator examines the detailed response from the evaluation and estimates the count of tokens sent and received in exchange with the LLM, by using a tokenizer for the respective model. Following is an example that demonstrates average cost using data from GPT 4o on the complete test set. The cost estimation, as seen in Figure 5.6, is further available for GPT 3.5-Turbo and GPT 4-Turbo. GPT 4o was chosen as an example, as its cost sits right in between the other OpenAI models, GPT 3.5-Turbo being ten times less expensive and GPT 4-Turbo twice as expensive as GPT 4o (as in Figure 5.5).

	Strategy 1a	Strategy 1b	Strategy 2a	Strategy 2b
Input Tokens	2981	3521	11637	4774
Output Tokens	45	42	349	292
Estimated Cost	USD 0.016	USD 0.018	USD 0.063	USD 0.028

Figure 5.6.: Average tokens used per strategy in GPT 4o

As can be seen, strategy 2a performs worst in terms of cost, averaging around USD 0.06 for a single request. As the amount of tokens and cost directly correlate, larger schemas are automatically more expensive. This high cost of strategy 2a is due to the reason that two round-trips to an LLM are made, containing excessive in-context learning for entity extraction and then query generation. All other strategies only use a single round trip to the LLM.

The rest of the strategies are less expensive, averaging between USD 0.016 (Strategy 1a), USD 0.04 (Strategy 1b) and USD 0.05 (Strategy 2b). The higher cost generally comes from a bigger prompt, resulting in more input tokens, but also concerning strategy 2a and strategy 2b, a bigger output (thus more generated tokens by the LLM), since a complex JSON structure is requested in these strategies.

Possibilities for Cost Reduction

As the general cost is still high for the average user, the team concluded a small study on how to possibly reduce costs.

As the output is only a small part of the cost, and it is difficult to make it smaller without losing quality, the only possibility to achieve a lower cost would be to reduce the input. As of now, in-context learning is made to be very accurate, using a few different schemas that are encoded and sent as part of the prompt. Currently, they are sent in every shot to ensure high response quality. The first step, encoding the schema into a more reasonable size, is already done using the code representation encoder (for examples, see subsection 4.2.2). A second step would be to reduce either the whole schema for the shot to a reasonable size, or to send samples that are more similar to the user's prompt, possibly with only part or even without a schema. Part of this challenge can already be solved with the not-yet-implemented strategy 3 (see subsection 4.2.3), as the goal is to use more accurate shot sampling.

Another way to reduce token input is to first match the intent of the user, in order to then only encode and send relevant parts of the schema, omitting as many unused types as possible. For example, if a mutation is to be generated, complex input types used for filtering may not be of use and would only generate cost. It could also be tested to see what the impact on entity extraction would be if certain information is omitted from the schema provided to the LLM (e.g., well-known types, primitives).

However, as for the final version of this thesis, the slightly higher cost was accepted in order to keep the response quality at a high level.

5.2.6. Recommendation of the Project Team

As of the end of this thesis, the project team strongly recommends strategy 2b (entity extraction with algorithmic query generation). It has proven high response quality (90%), especially using GPT 4-Turbo, and scored highest in reliability, with mostly consistent responses. Despite strategy 1b with GPT 4o even scoring 92% in response quality, and therefore 2% higher than our recommended combination, consistency is slightly lower, and the potential to correct errors or hallucinations produced by LLMs is much higher using strategy 2b. With strategy 2b, a field or entity that was wrongly matched out of the user's prompt is checked upon query generation, and the response is always valid, even though a field could still be missing. Strategy 1b, however, can either produce valid output or no output at all, as errors cannot be detected and fixed. For the team, strategy 2b clearly seems more future-proof, with possibilities to improve the algorithm for query generation or maybe add features such as synonym matching on fields that could not be matched, while strategy 1b presents more of a black box and is not designed to be extended with functionality.

The team accepts the higher cost that comes with strategy 2b, profiting from high response quality and reliability and accepting slower response times. The strategies 1a and 2a did not prove to be feasible, despite both being serious approaches, as both of them have lower response quality and reliability than the rest of the strategies.

Despite high hopes in the beginning, the open-source LLMs in scope for evaluation did not perform well. It proved very hard to steer the output into a desired format and getting consistency in valid outputs turned out to be an even bigger challenge. As most open-source LLMs are not available as an externally hosted service, there would also be a high cost involved for external hosting and the improvement of performance chained to it. The cost-free variant, hosting it on the user's machine leads to very decreased performance of said models. The team thus recommends working with OpenAI models.

5.3. Functional Requirements

This section describes how far functional requirements were met by the end of the project. For reference, the results of the evaluation with the best-performing combinations of LLM and strategy are taken as the final result, said combinations being GPT 4-Turbo with strategy 2b and GPT 4o with strategy 1b. Detailed evaluation results are further described in section 5.2.

In order to keep this section brief, functional requirements are only referred to by their ID, as they are described accurately in section 3.3.

The results that are referred to in this section can further be found in the test protocol, in section A.5. If a functional requirement is marked as fulfilled, it is linked to a test case that proves this. Details on test cases can be found in section A.2.

5.3.1. GraphQL Generation

Req-ID	Fulfilled	Proven by test case	Remarks
FR-100	Yes	Simple1 - Simple5	–
FR-110	Yes	Simple6 - Simple14	–
FR-120	Mostly	Simple2	Field birthDate is in the prompt but does not exist in the schema. It did not end up in the result. Strategy 2b has the ability to point out said fields, while strategy 1b returns a valid prompt but does not point out non-existing fields.
FR-130	Yes	Medium5, Medium19	–
FR-131	Yes	Medium3,	–
FR-132	Yes	Complex12	–
FR-140	Yes	Complex16	Complex16 can sometimes wrongly interpret the word 'Friend with a certain character' on GPT 4-Turbo. Then, equivalence is not always given; however, the structure is correct, and the team thus sees this requirement as fulfilled.
FR-141	Yes	Medium 7 - Medium9	–
FR-142	Yes	Complex12 - Complex16	Even with the remarks on Complex16 for FR-140, the sorting aspects work reliably, and test errors or mismatches were never connected to sorting
FR-143	Yes	Complex1 - Complex6	–
FR-150	Yes	Simple15, Medium17, Medium 18	–
FR-160	Yes	Unfulfillable 1-5	An unfulfillable prompt will return an expected error, and the CLI will inform the user accordingly.
FR-161	No	–	This requirement was set out of scope from the start of the project (priority 3)
FR-170	No	–	This requirement was set out of scope during the project, upon defining the improvement phase (priority 2)
FR-171	No	–	his requirement was set out of scope from the start of the project (priority 3)
FR-172	Yes	Complex1 - Complex6	

Table 5.1.: Test results for functional requirements concerning GraphQL generation

As for GraphQL generation, all functional requirements with priority 1 were fulfilled and proven by test cases. A partial exception is FR-120 which is marked as only partly fulfilled. Main functional-

ity is given, and no invalid output is produced, yet one of the high-performing strategies lacks the ability to point out non-existing fields. The requirements with priority 2, being requirements that were reconsidered before the improvement phase, were mostly fulfilled, with the sole exception of FR-170, which was set out of scope during the planned reprioritization (see section 8.3).

5.3.2. Schema exploration

During the project, the implementation of the CLI mostly focused on the generation of GraphQL queries. However, the CLI also contains the small functionality of allowing a user to ask questions about a schema. The team was able to do said task with minimal effort and could be done without any strategies, as in the main task, the generation of GraphQL.

Req-ID	Fulfilled	Proven by test case	Remarks
FR-201	Yes	SchemaExploration1 - SchemaExploration3	—
FR-202	Yes	SchemaExploration4 - SchemaExploration6	—
FR-203	Yes	SchemaExploration7	—

Table 5.2.: Test results for functional requirements concerning Schema Exploration

5.4. Non-Functional Requirements

This section showcases how well the non-functional requirements, defined in section 3.4, were met by the end of the project. This section is also kept brief, as much of the solution was already described in preceding sections; non-functional requirements are only referred to by their ID. Runs of the test cases that prove the fulfillment of the requirements can be found in the test protocol in section A.5.

5.4.1. Functionality

Req-ID	Fulfilled	Proven by test case	Remarks
NFR-F-01	Yes	Complete test set	Upon generation in any generator strategy, the output is parsed into a valid GraphQL operation and only then returned to the user. If any internal steps of a strategy fail or lead to an invalid structure, it will not be returned to the user but will be replaced with an appropriate error message.
NFR-F-02	No	—	This requirement was set out of scope during the project, upon defining the improvement phase (priority 2).
NFR-F-10	Yes	SmallSchemaTest	—
NFR-F-11	Yes	MediumSchemaTest	—
NFR-F-12	Yes	LargeSchemaTest	Test execution took very long, yet it proves that even very big schemas can be executed with valid results. A future focus should be on keeping response times low.
NFR-F-13	No	—	As further described in section 5.2, context size of LLMs is finite and as of now not every schema is guaranteed to succeed in any implemented strategy. This can be solved with future development. Not fulfilling this is an expected outcome, as this NFR has priority 3 (not in the scope of this project).

Table 5.3.: Test results for non-functional requirements concerning functionality

5.4.2. Usability

Req-ID	Fulfilled	Proven by	Remarks
NFR-U-01	Yes	Test case: Unfulfillable1	The system is built that errors and unfulfillability are always shown in natural language
NFR-U-02	Yes	Complete test set and schema size test cases	No training is conducted in any strategy. Execution of any tests proves this NFR is fulfilled
NFR-U-03	Yes	Evaluation for reliability (see section 5.2)	—

Table 5.4.: Test results for non-functional requirements concerning usability

5.4.3. Reliability

Req-ID	Fulfilled	Proven by	Remarks
NFR-R-01	Yes	Manual testing, system information	The way the system is built; no different output than a valid GraphQL or an error message can be produced. For further information, see section 4.6
NFR-R-02	No	—	This requirement was removed from scope during the project (priority 2)

Table 5.5.: Test results for non-functional requirements concerning reliability

5.4.4. Performance

Req-ID	Fulfilled	Proven by	Remarks
NFR-P-01	Mostly	Complete test set and schema size test cases	The system responds within 10s in most cases, where the schema does not contain more than approximately 350 types. For large schemas, response time exceeds 10 seconds with no upper limit, as schemas can contain a few thousand types. Therefore, the team can only guarantee performant answers on schemas containing up to 350 types.
NFR-P-02	Yes	Evaluation (section 5.2)	The best performing strategies that are recommended by the team feature only a single call to an LLM.

Table 5.6.: Test results for non-functional requirements concerning performance

5.4.5. Supportability

Req-ID	Fulfilled	Proven by	Remarks
NFR-S-01	Yes	Evaluation (section 5.2)	The solution features fully interchangeable LLM
NFR-S-02	Yes	Listing of libraries used in section A.1	—

Table 5.7.: Test results for non-functional requirements concerning supportability

6. Conclusion and Outlook

This thesis demonstrates the successful adaptation of key NL-to-SQL concepts to the GraphQL domain. The implemented solution is able to convert natural language queries to GraphQL queries with a high degree of accuracy, as demonstrated by the evaluation in section 5.2, when used with well-performing models. Most requirements specified in chapter 3 have been met, and the implemented solution is able to handle a wide range of natural language queries on a variety of schemas. The component is ready to be integrated into existing software systems as an experimental feature. For a production-ready deployment, additional measures need to be taken to ensure the security of the system, such as rate limiting and prompt injection prevention.

The concepts used, such as entity extraction, can be applied to other APIs, such as REST and OData, as long as they offer a structured schema or documentation. The implemented solution offers a good starting point for future work, and several areas for improvement have been identified. The in-context learning capabilities of LLMs can be more efficiently used by employing a fine-grained shot sampling approach. Research in the field of NL-to-SQL has already shown that optimizing the samples provided to the model can lead to better results. It is therefore reasonable to assume that such an approach could also improve the performance of the implemented solution. Additionally, an iterative entity extraction approach can be developed with the ambition of reducing the amount of tokens used per request, therefore significantly reducing the cost per query and potentially improving the performance in terms of speed.

Part II.

Project Documentation

7. Project Management

7.1. Methodology

We follow an agile approach in our work, as described in the Agile Manifesto. [29] However, we take into account that this project has a fixed deadline and formal requirements. We have decided against using a formal predefined process like SCRUM since it generates too much overhead and to little benefit in a two-person project. However, we incorporate some rituals in our daily work where we see fit, such as planning sessions for individual milestones (see: chapter 8) We intend to live the points described in the Agile Manifesto during our project as described below:

Individuals and interactions

- **In-person working** time of at least half a day per week. Based on our working-schedule and our timetable, this will take place on Tuesday morning. This time is used for work on hard problems (pair-programming), brainstorming or planning sessions. Additionally, we use the weekends for longer co-working sessions when necessary.
- **Active communication** to coordinate with each other. As each team member is expected to work ≈ 23 hours a week, it is necessary to work individually on this project. Through active communication, misunderstandings are prevented. This may be in the form of a short message whenever work has been done or is ready for review. Furthermore, this replaces the daily stand-up which would take place if the SCRUM framework had been used.
- **Short planning sessions** occurring weekly or bi-weekly depending on demand to plan the upcoming weeks in detail. This enables us to have our current priorities in mind, split the work to be done evenly or depending on doable workload.

Working software

- **Research Phase** before implementation, to sharpen focus and profit from existing knowledge, speeding up the implementation. See section 8.1 for details
- **MVP** to gather early feedback, and iteratively improve on in later stages of the project. See section 8.2 for details.

Customer collaboration

- **bi-weekly meetings with the advisor** to discuss progress and priorities to make sure we are on the right track, as well as for feedback on our work through the project.
- **meetings with stakeholder** upon demand to discuss requirements, show early prototypes as well as to gather feedback.

Responding to change

- **Early Testing** throughout the MVP phase to spot weaknesses early and plan future work accordingly. See 5.1 for details.

7.2. Tools

The following tools are used for this project:

7.2.1. Project Management

- **GitLab issues** to keep track of the work to be done. Each item has a title and a brief description making clear what needs to be done. In planning sessions, we mark items as "in scope" if we intend to do them within the next week. Each item is also assigned a Milestone (see 8) to make sure we know which part of the project the item belongs to.
- **Clockify** to keep track of the time spent to ensure that the guideline of working ≈ 23 hours per week are respected. Time-tracking also helps to understand how much time is spent on each task, as well as visualize the effort put into the project by the individual team members. At the end of the project, the data is used to evaluate our estimations and draw our lessons for future project.
- **Teams** for communication within the team as well as for official communication with the advisor. In case of remote meetings, teams can also be used to have video calls.
- **Confluence** to have a place for the advisor to find the relevant links related to this project, such as repositories.

7.2.2. Software Development

- **GitLab** is used to host all repositories containing code as well as documentation related to this project. Additionally, the integrated CI/CD pipelines are leveraged to enforce certain quality standards. For details, see 7.3
- **VSCode** is an open source code editor by Microsoft used to write documentation as well as for quick edits to code.
- **Rider** is used to write most of the code, as it offers a better experience in terms of debugging compared to VSCode.
- **Ollama** is used to run LLM models locally, allowing for cost-efficient and fast development.

7.2.3. Documentation

- **Latex** is used to write this documentation. A template provided by OST was applied to this documentation.

7.3. Quality Measures

The quality measures described in this section ensure that the code and documentation created as part of this project adhere to widely recognized quality standards.

7.3.1. Organizational Means

7.3.1.1. Merge Requests & Reviews

To maintain the integrity of the main branch, direct commits are not permitted. Every unit of work should be done on a separate branch, from which a merge request is created once the work is completed. The merge request should be reviewed by the other team member before it is merged into the main branch. The reviewer is responsible for verifying the quality and consistency as outlined in sections 7.3 and 7.3.

7.3.1.2. Definition of Done

A Task may only be considered as done when the following criteria are met:

- The changes are merged into the main branch following the process described above
- Time spent working on this task is added in the time-tracking tool (7.2)
- Related changes to the documentation are made

7.3.2. Code Quality

Clean Code

Clean code should be easy to read and understand. To achieve this, clear and descriptive names should be used for variables, functions, and classes. Overly complex expressions or statements should be avoided whenever possible. Code should be written with maintainability in mind, allowing easy modifications and extension without introducing unexpected errors. Overly complex structures should be avoided, and suitable design patterns are to be used.

Testing

All code should be covered in automated or manual tests. Automated should be the preferred option as it allows for easier and faster testing. All automated tests are run before merging a merge request into the main branch to ensure all tests pass. Automated tests should cover at least 80% of the codebase.

Tools

A consistent coding-style is enforced using [StyleCop-Analyzers](#) and [Roslyn-Analyzers](#) using a set of standard rules. For consistent formatting [dotnet-format](#) is used in combination with an extensive editorconfig covering various aspects of the code. All the mentioned tools are run before merging a merge request into the main branch to ensure all changes follow the rules.

7.3.3. Documentation Quality

When writing documentation, the following rules should be followed:

- LaTeX complies without errors
- Every figure, table, or code listing should have a caption and label
- All Acronyms used are defined in the glossary
- New content is spellchecked

7.4. Risk Management

During the initial phase of the project, a risk analysis is conducted to identify potential threats to the project and time plan. Throughout the project, they are refined and updated as unforeseeable risks arise. Risks are potential threats to the timetable of the defined milestones (see chapter 8). Therefore, measures must be taken to mitigate these risks. Risks are categorized into project-specific risks (PSR), which have their nature in the topics covered in the project and non-project-specific risks (NSR), which are risks that apply regardless of the covered topics.

7.4.1. Project Specific Risks

PSR-01 – Lack of experience with techniques surrounding LLMs

Description As LLMs are a relatively new field that is rapidly evolving, best practices are yet few and prone to being outdated quickly. LLMs are characterized by their non-deterministic nature, generating various different responses to the same input, which complicates integration due to inconsistent and almost unpredictable responses. The team has not yet gained much experience with programmatically consuming structured responses from LLMs, as is required working on the chosen strategies (see section 4.2).

Severity High – This could slow down development and endanger the project’s timeline.

Mitigation strategy Prior to choosing strategies, a small proof-of-concept is conducted, covering some important functional requirements. Additionally, during development, the solution is tested against multiple LLMs to ensure that the solution is robust towards various and inconsistent responses.

PSR-02 – A chosen strategy is unable to meet requirements

Description Despite the previously addressed risk, any chosen strategy could be unable to meet the requirements. This can be either due to the fact that the responses are too inconsistent and unpredictable, or a pure misconception of the strategy’s capabilities. If this is not recognized early enough, much time could be wasted on a dead-end approach.

Severity Low – Even a non-working strategy can help identify new possibilities if they are addressed early.

Mitigation strategy Throughout the development of a strategy, continuous testing, using the predefined test set (see section 5.1), is conducted to ensure that progress is made towards meeting the requirements.

PSR-03 – Problems and delays upon using in third-party libraries

Description Integrating third-party libraries into our code is vital, as time can be saved by reusing prebuilt solutions for already-solved problems, more specifically challenges related to working with the GraphQL syntax and its validation. However, relying on third-party libraries comes with the risk of them being hard to use or understand, which can consume a lot of time.

Severity Low – The amount of third-party libraries is predicted to be very limited, and the team has a lot of experience with many of them.

Mitigation strategy Reduce the usage of unknown libraries and rely on well-known and well-documented libraries. For the concrete case of the GraphQL library used, support by the maintainers is given, as they are stakeholders in the project.

7.4.2. Non-Project Specific Risks

NSR-01 – Absences

Description Ill or absent project members can slow down any phase of the project and break the already tight time plan. Throughout the project, all team members also have scheduled absences. This is especially severe, as all project members work part-time alongside their studies at OST, which does not leave much room for catching up delays during the week. If a bad balance of work between the team members exists, the quality of the product will be degraded due to time pressure.

Severity High – A high workload during the implementation endangers the scheduled end date of the project.

Mitigation strategy In favor of creating high-quality software over software with bigger functionality, a prioritization of requirements is done to reduce the risk of missing important features. Absences that are known in advance are planned from the beginning, and work is scheduled accordingly. To account for unforeseen absences, the team has a generous buffer of time scheduled in the final phase of the project.

Prior to starting an iteration to work on, the team looks ahead to identify big absences. The assigned work within said iteration will then be balanced according to the available time a team member has. It is critical that no team member is assigned significantly more work than the others during the whole duration of the project. This is further mitigated by having regular co-working sessions, where the team works together on complex tasks.

NSR-02 – Misaligned Scope

Description If planning is done poorly, the project could take too long, and the team would miss the deadline.

Severity High – Missing the deadline would lead to not being graded and not meeting the requirements defined in the project.

Mitigation strategy The iteration planning sessions should always start with looking at where the project currently stands. If a big deviation from the planned milestones is detected, workload will be set to be a bit higher to still meet the standards. As iteration plannings happen frequently, team members will always know on what the current status of the project is.

8. Milestones

The project is divided into 5 milestones, each of which represents a phase of the project. Figure 8.1 illustrates the milestones visually. Milestones start on a Monday morning and are always scheduled for a Sunday evening of the respective week. It has been deliberately decided not to subdivide the milestones further or to estimate the effort of individual tasks within the milestones, as it does not add any significant value to a project of this size. When planning the milestones, it was assumed that both team members would be able to work on this project for ≈ 20 hours per week with exception for the last week, where we are expected to work for ≈ 40 hours per week. If this is not possible in individual weeks due to illness or other absences, the effective end date of the milestone is postponed or the scope of the milestone is reduced.

The 'Project Management' milestone is not a phase of the project, but rather a standing milestone for the time spent on project management tasks throughout the whole project.

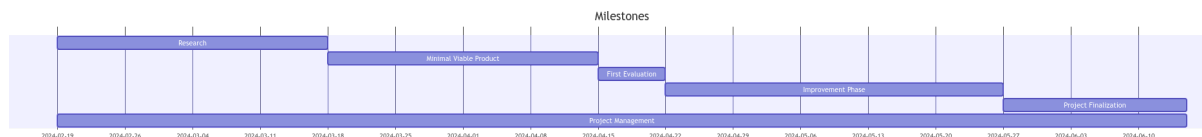


Figure 8.1.: Gantt chart with milestones

8.1. Research

In the research milestone, the focus is set on properly understanding the task at hand and researching existing solutions. This also includes research into similar topics, such as outlined in section 2.2. The output of this research should generate some ideas to act on during the MVP phase.

8.2. Minimal Viable Product

This milestone focuses on creating a MVP, validating the ideas generated in the research phase. E. Ries defines the MVP as: "The minimum viable product is that version of a new product which allows a team to collect the maximum amount of validated learning about customers with the least effort." (E. Ries [30])

The goal is not to create a finished and shippable product, but to create a first prototype that can be used to run a first evaluation on.

8.3. First Evaluation

This milestone is used to define the scope of the second half of the project. This is done by evaluating the MVP and identifying strengths and weaknesses of various approaches. Any functionality missing in the MVP is collected to be discussed and prioritized with the stakeholders. In

this project, a proposal is created and presented to the stakeholders at a meeting, taking place during this phase.

8.4. Improvement Phase

The focus of the improvement phase is to improve the MVP based on the feedback received, and the priorities defined in the scope definition phase. This phase may look significantly longer than the other phases, but this is due to the fact that these weeks include many holidays and other planned absences by the team members.

8.5. Project Finalization

This milestone focuses testing and documenting the built software component as well as finishing up on any other pending documentation tasks. The documentation is being finalized in various areas and prepared for submission. Any other artifacts required by the stakeholders, such as posters or abstracts, are being prepared and submitted.

9. Retrospective

9.1. Time Tracking Report

We utilized [clockify](#) to track our time spent on the project.

According to the tracked time in Clockify, we spent a total of 744 hours on the project. This includes all the time spent on meetings, development, and documentation. The expected time for this bachelor thesis was 720 hours [31], meaning that we spent approximately 24 hours (12 hours per person) more than required.

From the start of the project until week 16, approximately 20 hours per person were worked, while in the last week both team members were absent from their part-time jobs and worked full-time on the thesis. In the last week, a total of 46 hours per person were dedicated to the project.

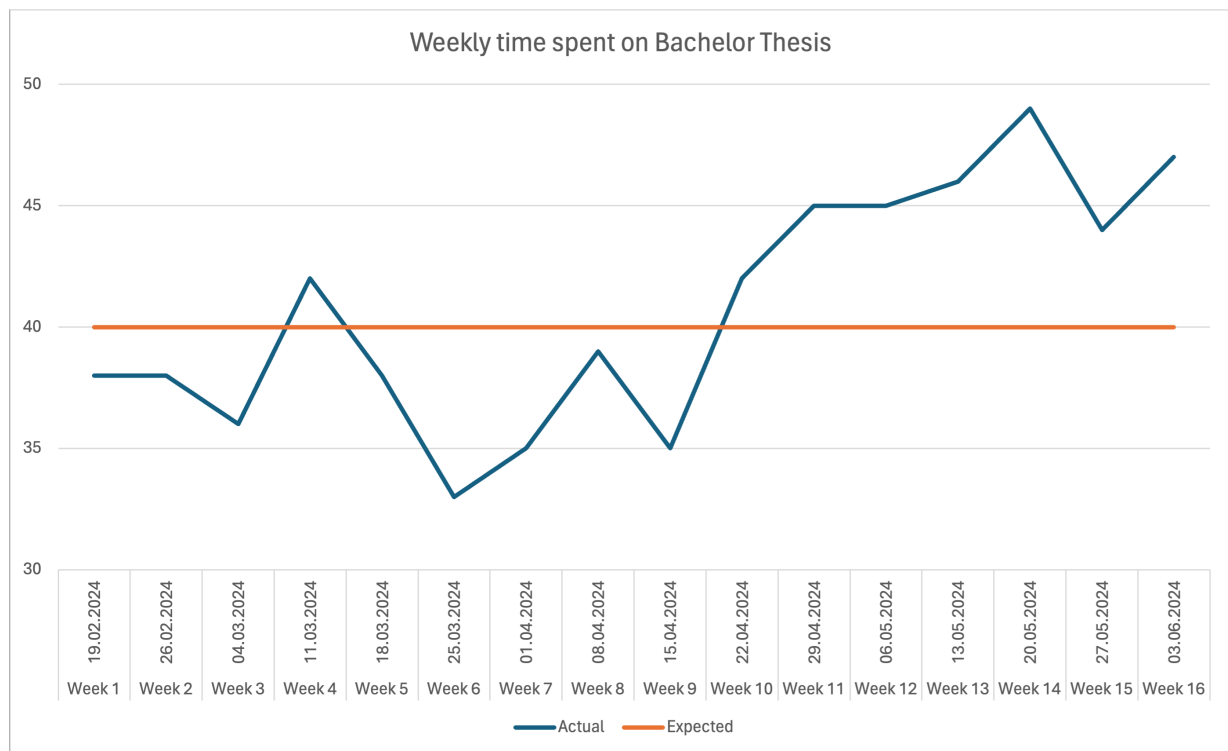


Figure 9.1.: Time Tracking Report, excluding the last week of the project

As seen in Figure 9.1, workload slightly increased in the last weeks of the project. Overall, we are happy with the time spent, as during this time good progress was made and the project was finished on time.

Within the team, the workload was distributed very well, and both team members contributed equally.

9.2. Team Reflection

This section contains findings and self-reflection about our project work in the team, and about the time plan and prioritization.

Working Together

Both team members are very happy with the way the project was conducted. Already having worked together for almost 9 years, we know each other very well and have a good understanding of each other's strengths and weaknesses. Communication and understanding between ourselves were thus never an issue.

In reflection on the last semester project we conducted together, we were able to improve a few points, as back then we had problems motivating each other during stressful weeks. The most effective improvement was working together face-to-face in the same room more regularly, in a few different locations, such as reserving meeting rooms in our companies on the weekends, or meeting up at university. This allowed us to focus more on the project and to motivate each other.

Agile Way of Work

The agile way of working we chose was an excellent fit for the project (see section 7.1). It allowed us to regularly see progress in our work and to adapt the priorities of the project after conducting a first evaluation of the solution. With bi-weekly meetings with our advisor, we were able to share ideas and thoughts and directly collect feedback, being able to implement it within the next two weeks.

Research & Studying

The topics covered in the project were based on some very recent research and thus featured much reading. With the project's focus not only being on evaluation, but also on research, we were able to take much time for it and gain a lot of useful knowledge in several fields, such as NL-to-SQL, entity extraction and more, which was not only useful for this thesis but will also be very useful for future projects to come.

Interaction with Advisor

The interaction between our advisor and ourselves proved to be very uncomplicated and at eye level. We were met with understanding and respect from day one, and we are very thankful for this. This is also thanks to the fact that working together on a previous project already proved to be very successful.

Achievement of Deadlines

As serious planning was conducted upfront and learnings were taken from our previous project, we were able to meet all deadlines. However, certain work packages were underestimated in effort, such as the evaluation of all strategies, as the amount of data collected proved to be much larger than expected. Thanks to good communication, we were able to balance the workload during this evaluation phase, and one team member could focus on a detailed evaluation, while the other took over the additional workload.

9.3. Acknowledgements

First and foremost, we would like to express our gratitude to our advisor, Prof. Stefan Keller, for his role in this project. His guidance, support and continuous constructive feedback have been vital for our project's success. We are grateful for his time and effort in helping us achieve our bachelor thesis, as this brings our studies at OST to a successful end.

We'd also like to thank Pascal Senn, for his role as our external stakeholder and initiator of the project's idea. His continuous feedback and helpful insights into using GraphQL and the HotChocolate library helped us on our way of building an efficient and useful solution.

Last but not least, we express our gratitude to our families and friends for their support and understanding during the time of not only this thesis but throughout our bachelor's studies.

List of Figures

1.	Turning natural language into valid GraphQL	iii
2.	Visualization of the second strategy	iv
2.1.	Example of a simple prompt injection attack	5
2.2.	Example of a prompt with an instruction boundary	6
3.1.	Use Case Diagram	16
4.1.	System Context Diagram	21
4.2.	Container Diagram	22
4.3.	Sequence Diagram showing the Interaction of the CLI with EAGLE	23
4.4.	Component Diagram of the Evaluator Tool	24
4.5.	Component Diagram of the Core functionality	25
4.6.	Example result from the few-shot generation	30
4.7.	Example result from the entity extraction	31
4.8.	Illustration of Generator Strategy 2b	32
5.1.	Equivalency of the highest performing combinations on the complete test set	40
5.2.	Equivalency of highest performing combinations, split up into complexity of the test cases	41
5.3.	Reliability of response quality	42
5.4.	Performance evaluation for average execution time	43
5.5.	Pricing on OpenAI models per 1000-tokens, as of June 2024 [1]	44
5.6.	Average tokens used per strategy in GPT 4o	44
8.1.	Gantt chart with milestones	59
9.1.	Time Tracking Report, excluding the last week of the project	61
A.1.	Libraries used in the implementation	73
A.2.	Solution Dependency Diagram	97
A.3.	Calculated costs per strategy on GPT 3.5-Turbo	97
A.4.	Calculated costs per strategy on GPT 4-Turbo	98
A.5.	Calculated costs per strategy on GPT 4o	98

List of Tables

3.1. Functional Requirements for GraphQL Generation	18
3.2. Functional Requirements for Schema Exploration	19
3.3. Non-Functional Requirements – Functionality	19
3.4. Non-Functional Requirements – Usability	20
3.5. Non-Functional Requirements – Reliability	20
3.6. Non-Functional Requirements – Performance	20
3.7. Non-Functional Requirements – Supportability	20
5.1. Test results for functional requirements concerning GraphQL generation	47
5.2. Test results for functional requirements concerning Schema Exploration	48
5.3. Test results for non-functional requirements concerning functionality	49
5.4. Test results for non-functional requirements concerning usability	49
5.5. Test results for non-functional requirements concerning reliability	50
5.6. Test results for non-functional requirements concerning performance	50
5.7. Test results for non-functional requirements concerning supportability	51

Acronyms

Basic_R Basic Schema Representation. 8, *Glossary: Basic_R*

Code_R Code Schema Representation. 8, 9, 29, *Glossary: Code_R*

DAILO DAIL Organization. 8, 9, *Glossary: DAIL_O*

FI_O Full-Information Organization. 7, 8, *Glossary: FI_O*

SO_O SQL-Only Organization. 7, 8, *Glossary: SO_O*

Text_R Text Schema Representation. 8, 9, *Glossary: Text_R*

API Application Programming Interface. 2, *Glossary: API*

CLI Command Line Interface. 23, 26, *Glossary: CLI*

EAGLE Expressive Assistant for GraphQL Language Encoding. 22, *Glossary: EAGLE*

LLM Large Language Model. i–v, 2, 4, 5, 12, 13, 22, 26–36, 38–46, 49–52, 55, 57, *Glossary: LLM*

MVP Minimum Viable Product. 15, 36, 37, 54, 59, 60, *Glossary: MVP*

NL-to-GraphQL Natural language-to-GraphQL. iii, 14, *Glossary: NL-to-GraphQL*

NL-to-SQL Natural language-to-SQL. i–iii, 2, 4, 5, 29, 34, 52, 62, *Glossary: NL-to-SQL*

POC Proof-of-concept. iii, 39, *Glossary: POC*

RAG Retrieval Augmented Generation. iii, 32, *Glossary: RAG*

Glossary

Basic_R Basic Schema Representation (Basic-R) is a simple schema representation used in natural language-to-SQL (NL2SQL) tasks. It lists all columns a table has, allowing the language model to understand the different tables and generate queries based on the available information. . 8

Code_R Code Schema Representation (Code-R) is a schema representation used in natural language-to-SQL (NL2SQL) tasks. It involves providing the database schema in a simplified way, allowing the language model to understand the structure of the database and generate queries based on the available information. . 8

DAIL_O DAIL Organization (DAIL-O) is a schema representation used in natural language-to-SQL (NL2SQL) tasks. It involves providing no database schema in the prompt, but instead, solutions to similar problems are included, enriched with corresponding questions. This approach allows the language model to generate queries based on the examples provided, without direct knowledge of the database schema. . 8

FI_O Full-Information Organization (FI-O) is a schema representation used in natural language-to-SQL (NL2SQL) tasks. It involves providing the entire database schema in the prompt, allowing the language model to understand the structure of the database and generate queries based on the available information. . 7

SO_O SQL-Only Organization (SO-O) is a schema representation used in natural language-to-SQL (NL2SQL) tasks. It involves providing no database schema in the prompt, but instead, solutions to similar problems are included. This approach allows the language model to generate queries based on the examples provided, without direct knowledge of the database schema. . 7

Text_R Text Schema Representation (Text-R) is a schema representation used in natural language-to-SQL (NL2SQL) tasks. It involves providing the database schema in a simplified way, allowing the language model to understand the structure of the database and generate queries based on the available information. . 8

API An API, or Application Programming Interface, is a set of protocols, tools, and definitions that allows different software applications to communicate with each other. It enables developers to access the functionality of other programs or services without needing to know how they are implemented, facilitating the integration and extension of their own applications. APIs are used in a wide variety of contexts, from web services and cloud computing to operating systems and hardware interfaces, making them essential for the creation of modern, interconnected software ecosystems. . 2

CLI A Command Line Interface (CLI) is a text-based user interface that allows users to interact with a computer program by typing commands into the terminal. . 23

CODE-DAVINCI-002 Code-Davinci-002 is a language model developed by OpenAI that specializes in understanding and generating code across multiple programming languages. It is trained on a diverse range of internet text and code repositories, making it useful for tasks

such as code completion, bug detection, and even generating new code snippets. However, it does not know specifics about which documents or code repositories were in its training set.. 9

EAGLE The software system developed in this project, tasked with generating GraphQL operation from a natural language prompt and answering questions about a GraphQL schema. . 22

GPT-3.5-Turbo GPT-3.5-Turbo is an advanced artificial intelligence language model developed by OpenAI. It is designed to understand and generate human-like text, improving upon its predecessors in terms of complexity, fluency, and accuracy. This model is particularly effective in tasks such as conversation, composition, and information processing.. 7

GPT-4 GPT-4, or Generative Pre-trained Transformer 4, is an advanced artificial intelligence language model developed by OpenAI. It is designed to understand and generate human-like text, improving upon its predecessors in terms of complexity, fluency, and accuracy in tasks such as conversation, composition, and information processing.. 7

GraphQL GraphQL is a query language and runtime for APIs that was developed by Facebook. It provides a more efficient, powerful, and flexible alternative to traditional RESTful APIs, enabling clients to request only the data they need and receive it in a single response. GraphQL is designed to simplify the process of building and consuming APIs, making it easier to develop and maintain complex data-driven applications.. i, iii, iv, 2–4, 7, 9, 13, 14, 18, 28–31, 34–38, 40, 41, 47, 48, 52, 57, 58, 63, 65

LLM Large Language Models (LLMs) are a type of artificial intelligence model that is trained on vast amounts of text data to understand and generate human-like language. They are designed to process and generate text in natural language, enabling them to perform a wide range of tasks such as translation, summarization, and conversation. LLMs have become increasingly popular in recent years due to their ability to generate high-quality text and perform complex language tasks, making them valuable tools for a wide range of applications. . i

MVP A Minimum Viable Product (MVP) is a version of a new product that is developed with the minimum set of features required to satisfy early customers and gather feedback. It is designed to test the core concepts and functionality of a new product in a real-world context, providing evidence of its potential value and practicality. MVPs are commonly used in software development, engineering, and research to assess the viability of new innovations before committing to full-scale development or implementation. . 15

NL-to-GraphQL Natural Language-to-GraphQL (NL2GraphQL) is a technology that translates human language queries into GraphQL operations that can be executed on a GraphQL API. It enables users to interact with GraphQL APIs without the need for understanding the underlying schema or query language, making data access more intuitive and efficient for non-technical users.. iii

NL-to-SQL Natural Language-to-SQL (NL2SQL) is a technology that translates human language queries into Structured Query Language (SQL) commands that can be executed on a database. It enables users to interact with databases without the need for understanding complex SQL syntax or database schema, making data access more intuitive and efficient for non-technical users. . i

POC A Proof-of-Concept (POC) is a demonstration or prototype that shows the feasibility and potential of a new product, service, or technology. It is used to validate the core concepts and functionality of an idea before committing to full-scale development or implementation.
. iii

RAG Large Language Models (LLMs) are a type of artificial intelligence model that is trained on vast amounts of text data to understand and generate human-like language. They are designed to process and generate text in natural language, enabling them to perform a wide range of tasks such as translation, summarization, and conversation. LLMs have become increasingly popular in recent years due to their ability to generate high-quality text and perform complex language tasks, making them valuable tools for a wide range of applications.
. iii

Resolver A resolver is a function that is responsible for fetching the data for a particular field in a GraphQL query. Resolvers are used to map the fields in a query to the actual data sources, such as databases, services, or other APIs, and to transform and format the data as needed before returning it to the client. Resolvers are a key component of the GraphQL execution process, enabling clients to request and receive the data they need in a flexible and efficient manner.. 37

token A token is a string of characters that represents a unit of data in a computer system. Tokens are used in LLMs to represent words, phrases, or other elements of text, enabling the model to process and generate human-like language. Tokens are a fundamental building block of LLMs, allowing them to understand and generate text in natural language.. 44

VICUNA-33b Vicuna is a chat assistant trained by fine-tuning LLaMA on user-shared conversations collected from ShareGPT.. 7

Bibliography

- [1] *Openai pricing*, <https://openai.com/api/pricing/>, Accessed: 2024-06-10, 2024.
- [2] OpenAI, *Hello gpt-4o*, <https://openai.com/index/hello-gpt-4o/>, Accessed: 2024-06-10, 2024.
- [3] AI@Meta, "Llama 3 model card," 2024. [Online]. Available: https://github.com/meta-llama/llama3/blob/main/MODEL_CARD.md.
- [4] B. Rozière, J. Gehring, F. Gloeckle, et al., *Code llama: Open foundation models for code*, 2024. arXiv: 2308.12950 [cs.CL].
- [5] W.-L. Chiang, Z. Li, Z. Lin, et al., *Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality*, 2023. [Online]. Available: <https://lmsys.org/blog/2023-03-30-vicuna/>.
- [6] A. Q. Jiang, A. Sablayrolles, A. Mensch, et al., *Mistral 7b*, 2023. arXiv: 2310.06825 [cs.CL].
- [7] L. Nan, Y. Zhao, W. Zou, et al., *Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies*, 2023. arXiv: 2305.12586 [cs.CL].
- [8] *Ai-powered bing chat spills its secrets via prompt injection attack*, <https://arstechnica.com/information-technology/2023/02/ai-powered-bing-chat-spills-its-secrets-via-prompt-injection-attack/>, Accessed: 2024-06-01, 2023.
- [9] *What is a prompt injection attack (and how to prevent it)*, <https://www.entrypointai.com/blog/what-is-a-prompt-injection-attack-and-how-to-prevent-it/>, Accessed: 2024-06-01, 2023.
- [10] T. Yu, R. Zhang, K. Yang, et al., *Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task*, 2019. arXiv: 1809.08887 [cs.CL].
- [11] Y. Gan, X. Chen, Q. Huang, et al., *Towards robustness of text-to-sql models against synonym substitution*, 2021. arXiv: 2106.01065 [cs.CL].
- [12] Y. Gan, X. Chen, and M. Purver, *Exploring underexplored limitations of cross-domain text-to-sql generalization*, 2021. arXiv: 2109.05157 [cs.CL].
- [13] C. Kosten, P. Cudré-Mauroux, and K. Stockinger, "Spider4sparql: A complex benchmark for evaluating knowledge graph question answering systems," in *2023 IEEE International Conference on Big Data (BigData)*, IEEE, Dec. 2023. doi: 10.1109/bigdata59044.2023.10386182. [Online]. Available: <http://dx.doi.org/10.1109/BigData59044.2023.10386182>.
- [14] Y. L. e. a. Dawei Gao Haibin Wang, *Text-to-sql empowered by large language models: A benchmark evaluation*, <https://arxiv.org/abs/2308.15363>, 2023.
- [15] D. R. Mohammadreza Pourreza, *Din-sql: Decomposed in-context learning of text-to-sql with self-correction*, <https://arxiv.org/abs/2304.11015>, 2023.
- [16] W. Z. e. a. Linyong Nan Yilun Zhao, *Enhancing few-shot text-to-sql capabilities of large language models: A study on prompt design strategies*, <https://arxiv.org/abs/2305.12586>, 2023.
- [17] G. Foundation, *Graphql specification (october 2021)*, <https://spec.graphql.org/October2021/>, Accessed: 2024-03-10, 2021.

- [18] L. Byron, *GraphQL: A data query language – engineering.fb.com*, <https://engineering.fb.com/2015/09/14/core-infra/graphql-a-data-query-language/>, [Accessed 09-06-2024], 2015.
- [19] E. A. Bender and S. G. Williamson, *Lists, decisions and graphs*. S. Gill Williamson, 2010.
- [20] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, *Patterns for API Design: Simplifying Integration with Loosely Coupled Message Exchanges* (Addison-Wesley Signature Series (Vernon)). Addison-Wesley Professional, 2022, ISBN: 9780137670109.
- [21] A. Brinkmann, R. Shraga, and C. Bizer, *Product attribute value extraction using large language models*, 2024. arXiv: [2310.12537](https://arxiv.org/abs/2310.12537) [cs.CL].
- [22] G. S. e. a. Ni P. Okhrati R., *Knowledge graph and deep learning-based text-to-graphql model for intelligent medical consultation chatbot*, <https://doi.org/10.1007/s10796-022-10295-0>, 2022.
- [23] StepZen, *Use ai to query graphql apis*, https://www.youtube.com/watch?v=wdpABH8f3_A, 2023.
- [24] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2004, ISBN: 0131489062.
- [25] Relay, *GraphQL cursor connections specification*, <https://relay.dev/graphql/connections.htm>, Accessed: 2024-03-10.
- [26] R. B. Grady and D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*. USA: Prentice-Hall, Inc., 1987, ISBN: 0138218447.
- [27] S. Brown, *C4 diagrams with structurizr*, <https://c4model.com>, Accessed: 2024-06-02.
- [28] O. Zimmermann, *Architectural decisions - the making of*, <https://ozimmer.ch/practices/2020/04/27/ArchitectureDecisionMaking.html>, Accessed: 2024-02-20, 2021.
- [29] K. Beck, M. Beedle, A. van Bennekum, et al., *Manifesto for agile software development*, <http://agilemanifesto.org/>, Accessed: 2024-05-29, 2001.
- [30] E. Ries, *Minimum viable product: A guide*, <http://www.startuplessonslearned.com/2009/08/minimum-viable-product-guide.html>, Accessed: 2024-03-10, 2009.
- [31] M. Stocker, *Leitfaden für bachelor- und studienarbeiten*, Teams Channel Studiengang Informatik, Accessed: 2024-06-12, 2023.

Part III.

Appendix

A. Appendix

A.1. Backend Libraries

```
<Project>
  <PropertyGroup>
    <ManagePackageVersionsCentrally>true</ManagePackageVersionsCentrally>
  </PropertyGroup>
  <!--Package References-->
  <ItemGroup>
    <PackageVersion Include="HotChocolate" Version="13.9.2" />
    <PackageVersion Include="Microsoft.Extensions.Configuration" Version="8.0.0" />
    <PackageVersion Include="Microsoft.Extensions.Configuration.Abstractions" Version="8.0.0" />
    <PackageVersion Include="Microsoft.Extensions.Configuration.Binder" Version="8.0.1" />
    <PackageVersion Include="Microsoft.Extensions.Configuration.Json" Version="8.0.0" />
    <PackageVersion Include="Microsoft.Extensions.Http" Version="8.0.0" />
    <PackageVersion Include="Microsoft.ML.Tokenizers" Version="0.21.1" />
    <PackageVersion Include="Microsoft.SemanticKernel" Version="1.12.0" />
    <PackageVersion Include="Newtonsoft.Json" Version="13.0.3" />
    <PackageVersion Include="Spectre.Console" Version="0.49.1" />
    <PackageVersion Include="Spectre.Console.Cli" Version="0.49.1" />
    <PackageVersion Include="StyleCop.Analyzers" Version="1.2.0-beta.556" />
  </ItemGroup>
  <!--Test Dependencies-->
  <ItemGroup>
    <PackageVersion Include="JUnitTestLogger" Version="1.1.0" />
    <PackageVersion Include="coverlet.collector" Version="6.0.2" />
    <PackageVersion Include="coverlet.msbuild" Version="6.0.2" />
    <PackageVersion Include="FluentAssertions" Version="6.12.0" />
    <PackageVersion Include="Microsoft.NET.Test.Sdk" Version="17.8.0" />
    <PackageVersion Include="Moq" Version="4.20.70" />
    <PackageVersion Include="Snapshotter.Xunit" Version="0.14.1" />
    <PackageVersion Include="xunit" Version="2.5.3" />
    <PackageVersion Include="xunit.runner.visualstudio" Version="2.5.3" />
  </ItemGroup>
</Project>
```

Figure A.1.: Libraries used in the implementation

A.2. Test Set

Simple Test set

Name: Simple1

Intent: –

Schema: BookstoreOfficial

User Prompt: Search for items by their title "Harry Potter" and show me if they are available. Give me their id and title

Expected response:

```
query {
  searchItems(title: "Harry Potter") {
```

```
      id
      title
      isAvailable
    }
  }
```

Name: Simple2

Intent: –

Schema: BookstoreOfficial

User Prompt: Show me all items that user "xy" has borrowed. Give me their title, birthDate, genre, and if they are available

Expected response:

```
query {
  getUserBorrowedItems(userId: "xy") {
    title
    isAvailable
    ... on Book {
      bookGenre: genre
    }
    ... on Movie {
      movieGenre: genre
    }
  }
}
```

Name: Simple3

Intent: –

Schema: BookstoreOfficial

User Prompt: Give me all books including all their fields

Expected response:

```
query {
  getAllBooks {
    id
    title
    author
    genre
    publishedYear
    isAvailable
  }
}
```

Name: Simple4

Intent: –

Schema: BookstoreOfficial

User Prompt: Give me all movies including all their fields

Expected response:

```
query {
  getAllMovies {
    id
    title
  }
}
```

```
    director
    genre
    releasedYear
    isAvailable
  }
}
```

Name: Simple5

Intent: –

Schema: BookstoreOfficial

User Prompt: Give me all movies including all their fields. Put all fields on movies into a fragment called MOVIES

Expected response:

```
query {
  getAllMovies {
    ...MOVIES
  }
}
fragment MOVIES on Movie {
  id
  title
  director
  genre
  releasedYear
  isAvailable
}
```

Name: Simple6

Intent: Concrete argument names

Schema: BookstoreOfficial

User Prompt: I want to add a new user with the name John Doe and email john.doe@example.com

Expected response:

```
mutation {
  addUser(name: "John Doe", email: "john.doe@example.com") {
    __typename
  }
}
```

Name: Simple7

Intent: Concrete attribute names

Schema: BookstoreOfficial

User Prompt: Borrow the item b1 for the user with id 1

Expected response:

```
mutation {
  borrowItem(userId: "1", itemId: "b1")
}
```

Name: Simple8

Intent: Take name of possible implementation as name

Schema: BookstoreOfficial

User Prompt: Borrow the book with the id b1 for the user with id 1

Expected response:

```
mutation {  
  borrowItem(userId: "1", itemId: "b1")  
}
```

Name: Simple9

Intent: Should create an ID for UserId

Schema: BookstoreOfficial

User Prompt: Write a mutation to borrow the item with the id b1

Expected response:

```
mutation ($userId: ID!) {  
  borrowItem(userId: $userId, itemId: "b1")  
}
```

Name: Simple10

Intent: Should create an ID for ItemId

Schema: BookstoreOfficial

User Prompt: Write a mutation to borrow with the user id "Alice"

Expected response:

```
mutation ($itemId: ID!) {  
  borrowItem(userId: "Alice", itemId: $itemId)  
}
```

Name: Simple11

Intent: Take name of possible implementation as name

Schema: BookstoreOfficial

User Prompt: Borrow the movie with the id b1 for the user with id 1

Expected response:

```
mutation {  
  borrowItem(userId: "1", itemId: "b1")  
}
```

Name: Simple12

Intent: –

Schema: BookstoreOfficial

User Prompt: Run a mutation to return a borrowed item with the id b1 for the user with id 1

Expected response:

```
mutation {  
  returnItem(userId: "1", itemId: "b1")  
}
```

Name: Simple13

Intent: –

Schema: BookstoreOfficial

User Prompt: I want to return a book with the id m1 as user 1

Expected response:

```
mutation {
  returnItem(userId: "1", itemId: "m1")
}
```

Name: Simple14

Intent: –

Schema: BookstoreOfficial

User Prompt: I want to return a movie with the id ABCDE as user 2

Expected response:

```
mutation {
  returnItem(userId: "2", itemId: "ABCDE")
}
```

Name: Simple15

Intent: –

Schema: BookstoreOfficial

User Prompt: Subscribe to item status changes on item b2. I want to have their id, title and availability status.

Expected response:

```
subscription {
  itemStatusChanged(itemId: "b2") {
    id
    title
    isAvailable
  }
}
```

Name: Simple16

Intent: –

Schema: SwapiOfficial

User Prompt: Give me the name and lifespan of all species

Expected response:

```
query {
  allSpecies {
    species {
      name
      averageLifespan
    }
  }
}
```

Medium Test set

Name: Medium1

Intent: –

Schema: SwapiOfficial

User Prompt: Give me all films with their title, director and all character names

Expected response:

```
query {
  allFilms {
    films {
      title
      director
      characterConnection {
        characters {
          name
        }
      }
    }
  }
}
```

Name: Medium2

Intent: –

Schema: SwapiOfficial

User Prompt: Give me all characters with their name and the films they were included in with their titles and directors, as well as their total count of movies.

Expected response:

```
query {
  allPeople {
    people {
      name
      filmConnection {
        films {
          title
          director
        }
      }
      totalCount
    }
  }
}
```

Name: Medium3

Intent: –

Schema: SwapiOfficial

User Prompt: What are the titles of all films available? How many are there?

Expected response:

```
query {
  allFilms {
    films {
      title
    }
    totalCount
  }
}
```

Name: Medium4

Intent: –

Schema: SwapiOfficial

User Prompt: What is the total character count per movie? Add the title and id of the film for reference.

Expected response:

```
query {
  allFilms {
    films {
      id
      title
      characterConnection {
        totalCount
      }
    }
  }
}
```

Name: Medium5

Intent: –

Schema: SwapiOfficial

User Prompt: Give me the opening text for the movie with the id "ZmlsbXM6MQ=="

Expected response:

```
query {
  film(id: "ZmlsbXM6MQ==") {
    openingCrawl
  }
}
```

Name: Medium6

Intent: Should write a query with an id parameter rather than query allFilms

Schema: SwapiOfficial

User Prompt: Give me the opening text for a certain movie

Expected response:

```
query ($id: ID!) {
  film(id: $id) {
    openingCrawl
  }
}
```

Name: Medium7

Intent: –

Schema: SwapiHotChocolate

User Prompt: What is the name and primary function of the Droid characters where the name is 'C-3PO'?

Expected response:

```
query {
  characters(where: {name: {eq: "C-3PO"}}) {
    nodes {
      ... on Droid {
        name
      }
    }
  }
}
```



```
        primaryFunction
      }
    }
  }
}
```

Name: Medium8

Intent: —

Schema: SwapiHotChocolate

User Prompt: Which characters appeared in the movie 'A New Hope'?

Expected response:

```
query {
  characters(where: {appearsIn: {some: {eq: NEW_HOPE}}}) {
    nodes {
      name
    }
  }
}
```

Name: Medium9

Intent: Should write the eq filter out to an Id

Schema: SwapiHotChocolate

User Prompt: Write a query to filter characters based on the episode they appeared in

Expected response:

```
query ($appearsInEq: Episode!) {
  characters(where: {appearsIn: {some: {eq: $appearsInEq}}}) {
    nodes {
      name
    }
  }
}
```

Name: Medium10

Intent: —

Schema: SwapiHotChocolate

User Prompt: List all characters and order them by their height in ascending order. Give me the name and height of them.

Expected response:

```
query {
  characters(order: [{height: ASC}]) {
    nodes {
      name
      height
    }
  }
}
```

Name: Medium11

Intent: —

Schema: SwapiHotChocolate

User Prompt: List all characters and give me their height and name. For Droids give me their primary function, for Humans their home planet

Expected response:

```
query {
  characters {
    nodes {
      name
      height
      ... on Human {
        homePlanet
      }
      ... on Droid {
        primaryFunction
      }
    }
  }
}
```

Name: Medium12

Intent: Conditional @include

Schema: SwapiHotChocolate

User Prompt: List all characters and give me their height and name. For Droids give me their primary function, for Humans their home planet. Only include the height if the boolean variable \$includeHeight is true

Expected response:

```
query ($includeHeight: Boolean!) {
  characters {
    nodes {
      name
      height @include(if: $includeHeight)
      ... on Human {
        homePlanet
      }
      ... on Droid {
        primaryFunction
      }
    }
  }
}
```

Name: Medium13

Intent: Conditional @include and shared field height only on Droids, even if available on

Schema: SwapiHotChocolate

User Prompt: List all characters and give me their name. For Droids give me their primary function and height, for Humans their home planet

Expected response:

```
query ($includeHeight: Boolean!) {
  characters {
    nodes {
      name
    }
  }
}
```

```

    ... on Human {
      homePlanet
    }
    ... on Droid {
      primaryFunction
      height @include(if: $includeHeight)
    }
  }
}

```

Name: Medium14

Intent: Conditional @include, @skip and shared field height only on Droids, even if available on. The skip statement must be on the whole fragment if the whole information is to be skipped

Schema: SwapiHotChocolate

User Prompt: List all characters and give me their name. For Droids give me their primary function and height, for Humans their home planet. Only include the height if the boolean variable \$includeHeight is true. Skip the additional human information if the boolean variable \$skipHumanInformation is true

Expected response:

```

query ($includeHeight: Boolean!, $skipHumanInformation: Boolean!) {
  characters {
    nodes {
      name
      ... on Human @skip(if: $skipHumanInformation) {
        homePlanet
      }
      ... on Droid {
        primaryFunction
        height @include(if: $includeHeight)
      }
    }
  }
}

```

Name: Medium15

Intent: Conditional @include, @skip and shared field height only on Droids, even if available on. The skip statement must be on the whole fragment if the whole information is to be skipped

Schema: SwapiHotChocolate

User Prompt: List all characters and give me their name. For Droids give me their primary function and height, for Humans their home planet. Only include the height if the boolean variable \$includeHeight is true. Skip the homePlanet of the human if the boolean variable \$skipHomePlanet is true

Expected response:

```

query ($includeHeight: Boolean!, $skipHomePlanet: Boolean!) {
  characters {
    nodes {
      name
      ... on Human {
        homePlanet @skip(if: $skipHomePlanet)
      }
      ... on Droid {

```

```
        primaryFunction
        height @include(if: $includeHeight)
      }
    }
  }
}
```

Name: Medium16

Intent: –

Schema: SwapiHotChocolate

User Prompt: Create a review for NEW_HOPE with a rating of 4 and a commentary of "Great movie!" and return the Id of the review

Expected response:

```
mutation {
  createReview(input: {
    episode: NEW_HOPE
    stars: 4
    commentary: "Great movie!"
  }) {
    review {
      id
    }
  }
}
```

Name: Medium17

Intent: –

Schema: SwapiHotChocolate

User Prompt: Subscribe me to updates on review for NEW_HOPE and give me the full review information

Expected response:

```
subscription {
  onReview (episode: NEW_HOPE) {
    id
    stars
    commentary
  }
}
```

Name: Medium18

Intent: –

Schema: SwapiHotChocolate

User Prompt: Subscribe me to updates on review for NEW_HOPE and give me the full review information. Put the review fields into a fragment called REVIEW

Expected response:

```
subscription {
  onReview (episode: NEW_HOPE) {
    ...REVIEW
  }
}
```

```
fragment REVIEW on Review {
  id
  stars
  commentary
}
```

Name: Medium19

Intent: Rather simple schema with weird wording

Schema: BookstoreOfficial

User Prompt: I am user 4. I have borrowed the book 7. Mutate this use case.

Expected response:

```
mutation {
  borrowItem(userId: "4", itemId: "7")
}
```

Complex Test Cases

Name: Complex1

Intent: –

Schema: SwapiOfficial

User Prompt: Give me the first 4 characters in the film with id "ZmlsbXM6MQ==". Include full pagination info, cursor of the pagination and give me the name of the characters.

Expected response:

```
query {
  film(id: "ZmlsbXM6MQ==") {
    characterConnection (first: 4) {
      pageInfo {
        hasNextPage
        startCursor
        endCursor
      }
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}
```

Name: Complex2

Intent: –

Schema: SwapiOfficial

User Prompt: Give me the first 4 characters after cursor id "YXJyYXljb25uZWNoaW9uOjI=" in the film with id "ZmlsbXM6MQ==". Include full pagination info, cursor of the pagination and give me the name of the characters.

Expected response:

```
query {
  film(id: "ZmlsbXM6MQ==") {
    characterConnection (first: 4, after: "YXJyYX1jb25uZWNOaW9uOjI=") {
      pageInfo {
        hasNextPage
        startCursor
        endCursor
      }
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}
```

Name: Complex3

Intent: —

Schema: SwapiOfficial

User Prompt: Give me the first 4 characters after a certain cursor id in the film with a certain id. Include full pagination info, cursor of the pagination and give me the name of the characters.

Expected response:

```
query($filmId: ID!, $cursor: String!) {
  film(id: $filmId) {
    characterConnection (first: 4, after: $cursor) {
      pageInfo {
        hasNextPage
        startCursor
        endCursor
      }
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}
```

Name: Complex4

Intent: —

Schema: SwapiOfficial

User Prompt: Give me the first 4 characters between cursor id "abc" and "def", in the film with a certain id. Include full pagination info, cursor of the pagination and give me the name of the characters.

Expected response:

```
query($filmId: ID!) {
  film(id: $filmId) {
    characterConnection (first: 4, after: "abc", before: "def") {
```

```

        pageInfo {
          hasNextPage
          startCursor
          endCursor
        }
        edges {
          cursor
          node {
            name
          }
        }
      }
    }
  }
}

```

Name: Complex5

Intent: –

Schema: SwapiOfficial

User Prompt: Give me all characters between cursor id "abc" and "def", in the film with a certain id. Include full pagination info, cursor of the pagination and give me the name of the characters.

Expected response:

```

query($filmId: ID!) {
  film(id: $filmId) {
    characterConnection (after: "abc", before: "def") {
      pageInfo {
        hasNextPage
        startCursor
        endCursor
      }
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}

```

Name: Complex6

Intent: Wording that would be used by someone that sees cursor as the definition

Schema: SwapiOfficial

User Prompt: Give me the first 4 characters in the film with id "ZmlsbXM6MQ==". The cursor of the pagination sits on the id "YXJyYXlj25uZWNOaW9uOjI=". Include full pagination info, cursor of the pagination and give me the name of the characters.

Expected response:

```

query {
  film(id: "ZmlsbXM6MQ==") {
    characterConnection (first: 4, after: "YXJyYXlj25uZWNOaW9uOjI=") {
      pageInfo {
        hasNextPage
        startCursor

```

```
        endCursor
      }
      edges {
        cursor
        node {
          name
        }
      }
    }
  }
}
```

Name: Complex7

Intent: –

Schema: SwapiHotChocolate

User Prompt: Create Fragments for all kinds of characters with all their fields. Their name is the type name in UPPERCASE. Write a query that retrieves all characters with all info.

Expected response:

```
query {
  characters {
    nodes {
      ...HUMAN
      ...DROID
    }
  }
}
fragment HUMAN on Human {
  id
  name
  height
  homePlanet
}
fragment DROID on Droid {
  id
  name
  height
  primaryFunction
}
```

Name: Complex8

Intent: –

Schema: SwapiOfficial

User Prompt: From all vehicles get the name of the vehicle, the name of the pilots, the species of the pilots and the number of films the pilots have been in

Expected response:

```
query {
  allVehicles {
    vehicles {
      name
      pilotConnection {
        pilots {
          name
        }
      }
    }
  }
}
```



```

        species {
          name
        }
        filmConnection {
          totalCount
        }
      }
    }
  }
}

```

Name: Complex9

Intent: –

Schema: SwapiOfficial

User Prompt: From all pilots get the name of the pilot, the count of films the pilots have been in, the name of their species, the name of the vehicles they have piloted and the number of films the vehicles have been in

Expected response:

```

query {
  allPeople {
    people {
      name
      species {
        name
      }
      filmConnection {
        totalCount
      }
      starshipConnection {
        starships {
          name
          filmConnection {
            totalCount
          }
        }
      }
    }
  }
}

```

Name: Complex10

Intent: Simple Library API but with only ambiguous field names

Schema: BookstoreOfficial

User Prompt: Give me all books, their name, writer, their topic category, from when they are and if they are ready to be borrowed or not

Expected response:

```

query {
  getAllBooks {
    title
    author
    genre
  }
}

```

```
        publishedYear
        isAvailable
    }
}
```

Name: Complex11

Intent: —

Schema: BookstoreOfficial

User Prompt: Create Fragments for all lendable item types with all their fields. Their name is the type name in UPPERCASE. Write a query that retrieves all lendable items

Expected response:

```
query {
  getAllBooks {
    ...BOOK
  }
  getAllMovies {
    ...MOVIE
  }
}
fragment BOOK on Book {
  id
  title
  author
  genre
  publishedYear
  isAvailable
}
fragment MOVIE on Movie {
  id
  title
  director
  genre
  releasedYear
  isAvailable
}
```

Name: Complex12

Intent: —

Schema: SwapiHotChocolate

User Prompt: Give me the three tallest characters with their name and height

Expected response:

```
query {
  characters(order: {height: DESC}, first: 3) {
    nodes {
      name
      height
    }
  }
}
```

Name: Complex13

Intent: —

Schema: SwapiHotChocolate

User Prompt: Give me the three smallest characters with their name and height

Expected response:

```
query {
  characters(order: {height: ASC}, first: 3) {
    nodes {
      name
      height
    }
  }
}
```

Name: Complex14

Intent: –

Schema: SwapiHotChocolate

User Prompt: Give me the three smallest characters with their name and height. But you must sort descending by height.

Expected response:

```
query {
  characters(order: {height: DESC}, last: 3) {
    nodes {
      name
      height
    }
  }
}
```

Name: Complex15

Intent: –

Schema: SwapiHotChocolate

User Prompt: Give me the name of the three tallest characters that are friends with the character with id 2001

Expected response:

```
query {
  characters(order: {height: DESC}, first: 3, where: {friends: {some: {eq:
2001}}}) {
    nodes {
      name
      height
    }
  }
}
```

Name: Complex16

Intent: Needs to write out the friend id eq filter with a variable

Schema: SwapiHotChocolate

User Prompt: Give me the name of the three tallest characters that are friends with a certain character

Expected response:

```
query ($characterId: Int!) {
  characters(order: {height: DESC}, first: 3, where: {friends: {some: {eq:
$characterId}}}) {
    nodes {
      name
      height
    }
  }
}
```

A.2.1. Unfulfillable Prompts

Name: Unfulfillable1

Intent: Prompt that has nothing to do with schema

Schema: ECommerce

User Prompt: Give me the name of all movies that "Mark Hamill" has played in

Expected response: An error message describing that the schema does not contain the requested information

Name: Unfulfillable2

Intent: Prompt that is in the same domain but not quite applicable

Schema: BookstoreOfficial

User Prompt: I want to sell a computer game with the name "Star Wars: Battlefront"

Expected response: An error message describing that the schema does not contain the requested information

Name: Unfulfillable3

Intent: Prompt that has got nothing to do with schema

Schema: SwapiOfficial

User Prompt: I want to borrow a computer game with the name "Star Wars: Battlefront"

Expected response: An error message describing that the schema does not contain the requested information

Name: Unfulfillable4

Intent: Filtering not supported, yet requested in prompt

Schema: SwapiOfficial

User Prompt: I want all species that have an average height of more than 1.5m

Expected response: An error message describing that the schema does not contain the requested information, or at least an operation that is valid, the non-existent filters "reduced" away

Name: Unfulfillable5

Intent: Mutation for this use case does not exist

Schema: SwapiOfficial

User Prompt: Create a new episode with the title NO_MORE_HOPE

Expected response: An error message describing that the schema does not contain the requested information

A.2.2. Schema Exploration Prompts

Name: SchemaExploration1

Intent: Information about available queries on schema

Schema: ECommerce

User Prompt: What kind of queries are offered on this schema? **Expected response:** Explanatory information about all available queries.

Name: SchemaExploration2

Intent: Information about available mutations on schema

Schema: ECommerce

User Prompt: What kind of mutations are offered on this schema?

Expected response: Explanatory information about all available mutations.

Name: SchemaExploration3

Intent: Information about available subscriptions on schema

Schema: SwapiOfficial

User Prompt: Are there any subscriptions available?

Expected response: Explanatory information about all available subscriptions.

Name: SchemaExploration4

Intent: Questions about specific types

Schema: BookStoreOfficial

User Prompt: What kinds of data is available for Books?

Expected response: Explanatory type information about books

Name: SchemaExploration5

Intent: Yes / no question about a certain field

Schema: SwapiHotChocolate

User Prompt: Is the height of a Droid available on this schema?

Expected response: The question should be answered with 'yes' and some information about the type around it

Name: SchemaExploration6

Intent: Information about possibilities of mutations

Schema: SwapiHotChocolate

User Prompt: Can I create reviews for movies?

Expected response: The question should be answered with 'yes' and some information about the type around it

Name: SchemaExploration7

Intent: Complex explanation

Schema: SwapiHotChocolate

User Prompt: Is cursor-based pagination available on this schema when searching for starships?

Expected response: The question should be answered with 'no' and some information about the type around it

A.2.3. Schema size tests

Name: SmallSchema

Intent: Prove that a strategy can handle schemas containing up to 100 types

Schema: InsuranceSchema (Schema in code base with 99 types)

User Prompt: Give me the description of all insurance claims.

Expected response:

```
{
  allClaims {
    ... on ClaimsConnection {
      nodes {
        ... on Claim {
          description
        }
      }
    }
  }
}
```

Name: MediumSchema

Intent: Prove that a strategy can handle schemas containing up to 250 types

Schema: ECommerceSchema (Schema in code base with 350 types)

User Prompt: Give me all orders with their status and total amount.

Expected response:

```
{
  allOrders {
    ... on OrdersConnection {
      nodes {
        ... on Order {
          status
          totalamount
        }
      }
    }
  }
}
```

Name: LargeSchema

Intent: Prove that a strategy can handle schemas containing up to 1000 types

Schema: Github Schema (Schema in code base with over 1200 types)

User Prompt: I want full repository information for the repository "Eagle" by "lpzimmermann"

Expected response:

```
query($repository_Repository_label_name: String!,
$repository_Repository_milestone_number: Int!,
$repository_Repository_project_number: Int!,
$repository_Repository_projectV2_number: Int!,
$repository_Repository_ref_qualifiedName: String!,
$repository_Repository_refs_refPrefix: String!,
$repository_Repository_release_tagName: String!,
$repository_Repository_ruleset_databaseId: Int!,
```

```
$repository_Repository_vulnerabilityAlert_number: Int!) {
  repository(name: "Eagle", owner: "lpzimmermann") {
    ... on Repository {
      allowUpdateBranch
      archivedAt
      assignableUsers {
        ... on UserConnection {
          totalCount
          nodes {
            ... on User {
              login
              name
            }
          }
        }
      }
      autoMergeAllowed
      branchProtectionRules {
        ... on BranchProtectionRuleConnection {
          totalCount
          nodes {
            ... on BranchProtectionRule {
              pattern
              requiredApprovingReviewCount
            }
          }
        }
      }
      codeOfConduct {
        ... on CodeOfConduct {
          name
          url
        }
      }
      createdAt
      databaseId
      defaultBranchRef
      deleteBranchOnMerge
      deployKeys {
        ... on DeployKeyConnection {
          totalCount
          nodes {
            ... on DeployKey {
              title
              createdAt
            }
          }
        }
      }
      description
      descriptionHTML
      diskUsage
      forkCount
      forkingAllowed
      hasDiscussionsEnabled
      hasIssuesEnabled
      hasProjectsEnabled
    }
  }
}
```

```
hasWikiEnabled
homepageUrl
id
isArchived
isDisabled
isFork
isInOrganization
isLocked
isMirror
isPrivate
isTemplate
isUserConfigurationRepository
issueTemplates
label(name: $repository_Repository_label_name)
languages
licenseInfo
lockReason
mentionableUsers
mergeCommitAllowed
mergeCommitMessage
mergeCommitTitle
mergeQueue
milestone(number: $repository_Repository_milestone_number)
milestones
mirrorUrl
name
nameWithOwner
object
openGraphImageUrl
owner
packages
parent
pinnedDiscussions
pinnedEnvironments
pinnedIssues
planFeatures
primaryLanguage
project(number: $repository_Repository_project_number)
projectV2(number: $repository_Repository_projectV2_number)
projects
projectsResourcePath
projectsUrl
projectsV2
recentProjects
ref(qualifiedName: $repository_Repository_ref_qualifiedName)
refs(refPrefix: $repository_Repository_refs_refPrefix)
release(tagName: $repository_Repository_release_tagName)
releases
repositoryTopics
resourcePath
ruleset(databaseId: $repository_Repository_ruleset_databaseId)
rulesets
securityPolicyUrl
shortDescriptionHTML
squashMergeAllowed
squashMergeCommitMessage
squashMergeCommitTitle
```



```
squashPrTitleUsedAsDefault
sshUrl
stargazerCount
stargazers
submodules
tempCloneToken
templateRepository
updatedAt
url
usesCustomOpenGraphImage
viewerCanAdminister
viewerCanCreateProjects
viewerCanSubscribe
viewerCanUpdateTopics
viewerDefaultCommitEmail
viewerDefaultMergeMethod
viewerHasStarred
viewerPermission
viewerPossibleCommitEmails
viewerSubscription
visibility
vulnerabilityAlert(number: $repository_Repository_vulnerabilityAlert_number)
vulnerabilityAlerts
watchers
webCommitSignoffRequired
}
}
}
```

A.3. Architecture Diagrams

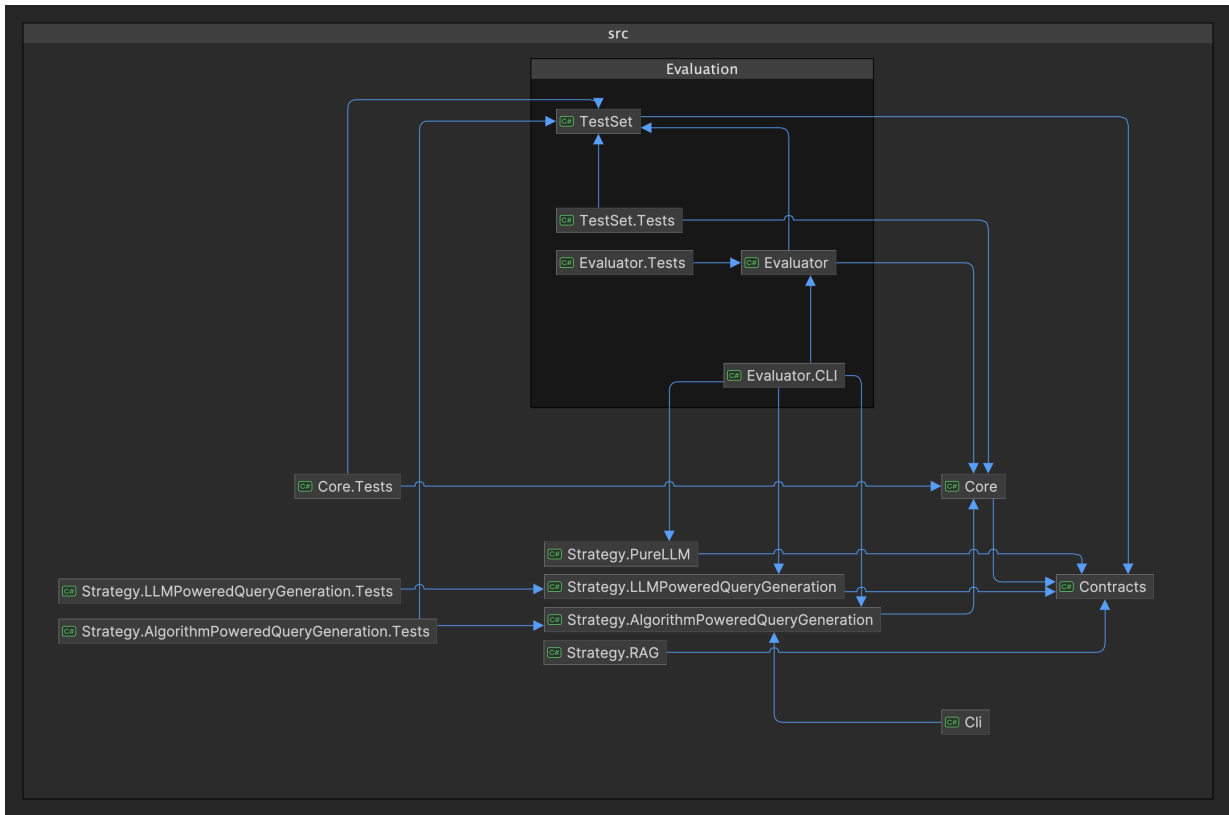


Figure A.2.: Solution Dependency Diagram

A.4. Detailed cost analysis

Costs calculated based on the cost pricing of OpenAI, stated in Figure 5.5

GPT 3.5-Turbo

	Strategy 1a	Strategy 1b	Strategy 2a	Strategy 2b
Average on test set	USD 0.002	USD 0.002	USD 0.006	USD 0.003
Average for small schema	USD 0.0003	USD 0.001	USD 0.005	USD 0.002
Average for medium schema	USD 0.001	USD 0.002	USD 0.006	USD 0.003
Average for large schema	USD 0.004	USD 0.002	USD 0.009	USD 0.003
Test set total	USD 0.078	USD 1.815	USD 0.317	USD 0.141

Figure A.3.: Calculated costs per strategy on GPT 3.5-Turbo

GPT 4-Turbo

	Strategy 1a	Strategy 1b	Strategy 2a	Strategy 2b
Average on test set	USD 0.031	USD 0.037	USD 0.125	USD 0.056
Average for small schema	USD 0.006	USD 0.030	USD 0.090	USD 0.047
Average for medium schema	USD 0.018	USD 0.036	USD 0.112	USD 0.055
Average for large schema	USD 0.077	USD 0.046	USD 0.183	USD 0.067
Test set total	USD 1.576	USD 1.864	USD 6.375	USD 2.833

Figure A.4.: Calculated costs per strategy on GPT 4-Turbo

GPT 4o

	Strategy 1a	Strategy 1b	Strategy 2a	Strategy 2b
Average on test set	USD 0.016	USD 0.018	USD 0.063	USD 0.028
Average for small schema	USD 0.003	USD 0.015	USD 0.046	USD 0.024
Average for medium schema	USD 0.009	USD 0.018	USD 0.057	USD 0.028
Average for large schema	USD 0.039	USD 0.023	USD 0.092	USD 0.034
Test set total	USD 0.795	USD 0.930	USD 3.234	USD 1.441

Figure A.5.: Calculated costs per strategy on GPT 4o

A.5. Test protocols

VI.Syntax = Valid GraphQL Syntax

VI.Schema = Valid on Schema

Eqv. = Equivalent

Overf. = Overfetched Fields

Underf. = Underfetched Fields

Full test set on strategy 2b using GPT 4-Turbo

TestCase	Time	Vi. Syntax	Vi.Schema	Eqv.	Overf.	Underf.	Arg. Mm.
Simple1	00:00:07.2603	True	True	True	0	0	0
Simple2	00:00:07.7133	True	True	True	0	0	0
Simple3	00:00:09.1052	True	True	True	0	0	0
Simple4	00:00:07.8704	True	True	True	0	0	0
Simple5	00:00:08.8202	True	True	True	0	0	0
Simple6	00:00:04.6064	True	True	True	0	0	0
Simple7	00:00:05.0165	True	True	True	0	0	0
Simple8	00:00:06.1657	True	True	True	0	0	0
Simple9	00:00:03.4343	True	True	True	0	0	0
Simple10	00:00:05.2449	True	True	True	0	0	0
Simple11	00:00:06.5526	True	True	True	0	0	0
Simple12	00:00:07.4384	True	True	True	0	0	0
Simple13	00:00:05.5095	True	True	True	0	0	0
Simple14	00:00:04.6337	True	True	True	0	0	0
Simple15	00:00:08.2511	True	True	True	0	0	0
Simple16	00:00:05.5879	True	True	True	0	0	0
Medium1	00:00:13.2452	True	True	True	0	0	0
Medium2	00:00:12.7721	True	True	True	0	0	0
Medium3	00:00:05.5240	True	True	True	0	0	0
Medium4	00:00:07.8857	True	True	True	0	0	0
Medium5	00:00:05.4108	True	True	True	0	0	0
Medium6	00:00:04.2369	True	True	True	0	0	1
Medium7	00:00:09.0830	True	True	True	0	0	0
Medium8	00:00:10.5466	True	True	True	0	0	0
Medium9	00:00:08.7043	False	False	Unavailable			
Medium10	00:00:11.9663	True	True	True	0	0	0
Medium11	00:00:09.1233	True	True	True	0	0	0
Medium12	00:00:10.8922	True	True	True	0	0	0
Medium13	00:00:09.4663	True	True	True	0	0	0
Medium14	00:00:11.8940	True	True	True	0	0	0
Medium15	00:00:09.0082	True	True	True	0	0	0
Medium16	00:00:08.2924	False	False	Unavailable			
Medium17	00:00:06.2442	True	True	True	0	0	0
Medium18	00:00:09.3173	True	True	False	1	3	0
Medium19	00:00:04.8076	True	True	True	0	0	0
Complex1	00:00:17.2039	True	True	True	0	0	0
Complex2	00:00:18.0981	True	True	True	0	0	0
Complex3	00:00:15.7908	True	True	True	0	0	0
Complex4	00:00:18.8408	True	True	True	0	0	0
Complex5	00:00:15.2575	True	True	True	0	0	0
Complex6	00:00:17.4037	True	True	True	0	0	0
Complex7	00:00:10.2387	True	True	True	0	0	0
Complex8	00:00:13.1308	True	True	True	0	0	0
Complex9	00:00:14.1983	True	True	True	0	0	0
Complex10	00:00:06.8647	True	True	True	0	0	0
Complex11	00:00:17.1506	True	True	True	0	0	0
Complex12	00:00:09.0650	True	True	True	0	0	0
Complex13	00:00:11.0546	True	True	True	0	0	0
Complex14	00:00:15.8709	True	True	True	0	0	0
Complex15	00:00:13.5179	False	False	Unavailable			
Complex16	00:00:11.5979	True	True	False	0	1	1

Full test set on strategy 1b using GPT 4o

TestCase	Time	Vi. Syntax	Vi.Schema	Eqv.	Overf.	Underf.	Arg. Mm.
Simple1	00:00:01.1713	True	True	True	0	0	0
Simple2	00:00:01.6095	True	False	True	0	0	0
Simple3	00:00:01.1177	True	True	True	0	0	0
Simple4	00:00:01.2180	True	True	True	0	0	0
Simple5	00:00:01.5304	True	True	True	0	0	0
Simple6	00:00:00.8553	True	True	True	0	0	0
Simple7	00:00:01.0865	True	True	True	0	0	0
Simple8	00:00:00.6611	True	True	True	0	0	0
Simple9	00:00:00.9292	True	False	Unavailable			
Simple10	00:00:00.8635	True	True	True	0	0	0
Simple11	00:00:01.0866	True	True	True	0	0	0
Simple12	00:00:01.0529	True	True	True	0	0	0
Simple13	00:00:00.8250	True	True	True	0	0	0
Simple14	00:00:00.6829	True	True	True	0	0	0
Simple15	00:00:01.2637	True	True	True	0	0	0
Simple16	00:00:01.3806	True	True	True	0	0	0
Medium1	00:00:01.4961	True	True	True	0	0	0
Medium2	00:00:01.0530	True	True	True	0	0	0
Medium3	00:00:00.8769	True	True	True	0	0	0
Medium4	00:00:01.3401	True	True	True	0	0	0
Medium5	00:00:00.8057	True	True	True	0	0	0
Medium6	00:00:00.8218	True	True	True	0	0	0
Medium7	00:00:01.5636	True	True	True	0	0	0
Medium8	00:00:01.7394	True	True	True	0	0	0
Medium9	00:00:01.2458	True	True	True	0	0	0
Medium10	00:00:01.0379	True	True	True	0	0	0
Medium11	00:00:01.1744	True	True	True	0	0	0
Medium12	00:00:01.2490	True	False	True	0	0	0
Medium13	00:00:01.6076	True	True	False	0	0	0
Medium14	00:00:01.9415	True	False	True	0	0	0
Medium15	00:00:01.4346	True	False	True	0	0	0
Medium16	00:00:00.9486	True	True	True	0	0	0
Medium17	00:00:01.0926	True	True	True	0	0	0
Medium18	00:00:01.0960	True	True	True	0	0	0
Medium19	00:00:00.9480	True	True	True	0	0	0
Complex1	00:00:01.4354	True	True	True	0	0	0
Complex2	00:00:02.0463	True	True	True	0	0	0
Complex3	00:00:01.7356	True	True	True	0	0	0
Complex4	00:00:01.6877	True	True	True	0	0	0
Complex5	00:00:01.4903	True	True	True	0	0	0
Complex6	00:00:01.9128	True	True	True	0	0	0
Complex7	00:00:02.9987	True	True	False	8	8	0
Complex8	00:00:01.7618	True	True	True	0	0	0
Complex9	00:00:02.2044	True	True	True	0	0	0
Complex10	00:00:01.0451	True	True	True	0	0	0
Complex11	00:00:01.4773	True	True	True	0	0	0
Complex12	00:00:01.1861	True	True	True	0	0	0
Complex13	00:00:01.0904	True	True	True	0	0	0
Complex14	00:00:02.3859	True	True	True	0	0	0
Complex15	00:00:01.3295	True	False	Unavailable			
Complex16	00:00:02.0948	True	True	True	0	0	0

Test runs for testing schema limits

Environment: Strategy 2b, GPT 4-Turbo

TestCase	Time	VI. Syntax	VI.Schema	Eqv.	Overf.	Underf.	Arg. Mm.
SmallSchema	00:00:03	True	True	True	0	0	0
MediumSchema	00:00:06	True	True	True	0	0	0
LargeSchema	00:01:45	True	True	True	0	0	0

Test runs for non-fulfillable prompts

Environment: Strategy 2b, GPT 4o

TestCase	Time	Test passed	Actual response
Unfulfillable1	00:00:02	True	Operation could not be generated – Error: Matched field search does not exist on type Query
Unfulfillable2	00:00:02	True	Operation could not be generated Error: Matched field sellComputerGame does not exist on type Mutation
Unfulfillable3	00:00:03	True	Operation could not be generated Error: Root type is not supported on current schema Mutation
Unfulfillable4	00:00:03	True	Correct query without filtering functionality
Unfulfillable5	00:00:02	True	Operation could not be generated Error: Root type is not supported on current schema Mutation

Test runs for schema exploration

Environment: GPT 4-Turbo

TestCase	Time	Test passed
SchemaExploration1	00:00:03	True
SchemaExploration2	00:00:02	True
SchemaExploration3	00:00:04	True
SchemaExploration4	00:00:07	True
SchemaExploration5	00:00:04	True
SchemaExploration6	00:00:03	True
SchemaExploration7	00:00:05	True

A.6. Prompts used

A.6.1. Intent Matching

Listing A.1: Prompt used for intent matching

```
# System Prompt
System: You are an assistant that has to classify the user's intent based on their
input.
Do NOT respond to the user, only classify the intent.
Below are the possible intents:

1. GENERATE_OPERATION - The user wants to generate a query or mutation operation to
fetch or manipulate some data.
```

Examples:

- "Find all products with the name 'Laptop' and provide their id, name, and price."
- "Add a new employee with the name Jane Smith and email jane.smith@company.com."
- "Assign the project P123 to the employee with id 456."
- "List all customers along with the total amount they have spent and their order count."
- "Subscribe to updates on the stock price changes for company XYZ. I want the latest price and the change percentage."
- "Get me all the products with all their available fields"
- "What is the favorite color of the character with id 123?"

2. ASK_SCHEMA - The user wants to know information about the structure of the data available.

Examples:

- "What fields are available for the 'Product' type?"
- "Describe the schema for the 'Employee' type."
- "What relationships exist between 'Customer' and 'Order' types?"
- "Tell me about the data available in this schema."

3. UNKNOWN - The user's intent is unknown.

Only use this if the intent cannot be classified as "GENERATE_OPERATION" or "ASK_SCHEMA".

Reply ONLY with a single intent name: GENERATE_OPERATION, ASK_SCHEMA, or UNKNOWN. Do NOT include any additional information or reasoning in your response.

Shots for in-context learning

User: What information can i retrieve about cats?

Assistant: ASK_SCHEMA

User: Find all products of type 'Monitor' where the price is higher than 500.-

Assistant: GENERATE_OPERATION

...

User Prompt

User: {PROMPT_GOES_HERE}

A.6.2. Strategy 1a

Listing A.2: Prompt used by strategy 1a

#Instructions

System: The following Schema below is the base for all questions that the user asks about GraphQL Schemas.

It is found in between the following tokens:

=====

System:

=====

{SCHEMA SDL GOES HERE}

=====

System: You must always answer the users request with a valid graphql.

You must not write any explanation text around it and no ticks around core segments.

You must not answer with an empty string. You must not encapsulate the graphql code fragment into ` or ``` or \" characters.

Just directly graphql syntax.

#Example

User: Give me all books with their id and title.

Assistant: query { books { id title } }

#User Prompt

User: {PROMPT_GOES_HERE}

A.6.3. Strategy 1b

Listing A.3: Prompt used by strategy 1b

```
# Motivational Speech
System: You are a world-class algorithm for generating GraphQL queries out of user prompts.

# Accepted Format
System: The response must be valid GraphQL only. No explanation text around it.
The response must be a graphql operation that must always compile.
If an attribute could not be matched to the schema but was found in the prompt, exclude it from the response.
If no attributes could be matched to the type in question, you can just add the '__typename' attribute in order to have a query that compiles.
Be aware that for some requests, the user may ask for a single entity (such as a book) with a filter condition and such filter conditions only exist on the query for all entities (such as books or allBooks). In that case you must use the entity for allBooks/books if you see a filter condition, while you should still use the single entity query "book" if you know the required parameters.
The same goes for entities such as characters, where the user may ask for a single character with a filter condition, but the filter condition only exists on the field 'characters' or 'allCharacters' and not on the field 'character'.
Be careful to always stick to the correct types. For example, if a filter is inside of a complex type structure, be sure to follow it - for example, always use the eq-property when filtering by name, never directly assign the value to a property when the structure on the schema calls for a nested object.

# Shots (3x)
System: Your job is to convert the user's request into a valid GraphQL query, mutation or subscription syntax.
Valid attributes for types are found in the following schema in between \n====\n.
You must only use valid attributes and arguments on types found in said schema, not invent fields that do not exist.
System: \n====\n{ENCODED_SAMPLE_SCHEMA}\n====\n
User: Give me the book with id \"xy\" with its title, color and publication date, and the name of the author and their year of birth. I also want the publisher of the book with their name, nationality and foundationDate.
Assistant: {
  book(id: "xy") {
```



```
... on Book {
  title
  publicationDate
  author {
    __typename
  }
  publisher {
    ... on Publisher {
      name
    }
    ... on PrivatePublisher {
      nationality
    }
    ... on PublicPublisher {
      foundationDate
    }
  }
}
}
```

Preparation

System: Your job is to convert the user's request into a valid GraphQL query, mutation or subscription syntax.

Valid attributes for types are found in the following schema in between
\n====\n.

You must only use valid attributes and arguments on types found in said schema, not invent fields that do not exist.

System: \n====\n{ENCODED_ACTUAL_SCHEMA}\n====\n

User Prompt

User: {PROMPT_GOES_HERE}

A.6.4. Strategy 2a

Listing A.4: Prompt used by strategy 2a for entity extraction

Motivational Speech

System: You are a world-class algorithm for extracting information in structured formats.

Accepted format

System: The response must be valid JSON only. No explanation text around it.

The response must be an object containing a field 'operationType' indicating the type of operation to perform (query, mutation, subscription) and a field 'entities' being an array of objects.

Each object has a 'type' field with the value 'Entity' or 'Attribute'.

An 'Entity' object has a 'name' field with its name on the parent, a 'typeName' which is the most concrete name of the type on the schema and a 'fields' field with an array of 'Attribute' and 'Entity' objects.

For example if there is an interface 'Vehicle' and a type 'Car' implementing 'Vehicle', the 'typeName' field should be 'Car' and not 'Vehicle'.

If an attribute could not be matched to the schema but was found in the prompt, set the property 'matched' to false in the object.

Parameters are stored in the 'parameters' field as an array of objects. Each parameter can either be a single parameter field (type "parameter") or a list of

parameters (type "parameterList").

Inside parameterContent can be either a ParameterValue, a ParameterValueList (for values that are an array or list) or a ParameterObject, that contains either a parameter or parameterList inside of their properties.

Be aware that for some requests, the user may ask for a single entity (such as a book) with a filter condition and such filter conditions only exist on the query for all entities (such as books or allBooks). In that case you must use the entity for allBooks/books if you see a filter condition, while you should still use the single entity query "book" if you know the required parameters.

The same goes for entities such as characters, where the user may ask for a single character with a filter condition, but the filter condition only exists on the field 'characters' or 'allCharacters' and not on the field 'character'.

Be careful to always stick to the correct types. For example, if a filter is inside of a complex type structure, be sure to follow it - for example, always use the eq-property when filtering by name, never directly assign the value to a property when the structure on the schema calls for a nested object.

You must never add comments in any form on into the resulting json structure.

Shots (3x)

User: Extract the searched attribute from based on the entity in question.

Valid attributes for entities, represented as GraphQL types, are found in the following schema in between \n====\n.

Give the user a listing of the entities and their requested attributes.

System: \n====\n{ENCODED_SAMPLE_SCHEMA}\n====\n

Assistant: {EXPECTED JSON ENTITY EXTRACTION}

Preparation

User: Extract the searched attribute from based on the entity in question.

Valid attributes for entities, represented as GraphQL types, are found in the following schema in between \n====\n.

Give the user a listing of the entities and their requested attributes.

System: \n====\n{ENCODED_SCHEMA}\n====\n

User Prompt

User: {PROMPT_GOES_HERE}

Listing A.5: Prompt used by strategy 2a for operation generation

Motivational Speech

System: You are a world-class algorithm for generating GraphQL operations (queries, mutations, subscriptions) out of structured information.

Accepted format

System: The response must be valid GraphQL only. No explanation text around it.

The response must be a graphql operation that must always compile.

If an attribute could not be matched to the schema but was found in the prompt, exclude it from the response.

If no attributes could be matched to the type in question, you can just add the '__typename' attribute in order to have a query that compiles.

Be aware that for some requests, the user may ask for a single entity (such as a book) with a filter condition and such filter conditions only exist on the query for all entities (such as books or allBooks). In that case

you must use the entity for allBooks/books if you see a filter condition, while you should still use the single entity query "book" if you know the required parameters.

The same goes for entities such as characters, where the user may ask for a single character with a filter condition, but the filter condition only exists on the field 'characters' or 'allCharacters' and not on the field 'character'.

Be careful to always stick to the correct types. For example, if a filter is inside of a complex type structure, be sure to follow it - for example, always use the eq-property when filtering by name,

never directly assign the value to a property when the structure on the schema calls for a nested object.

You must never use names for fields and types that are not in the schema. You are forbidden to invent new fields or types or change names.

Shots (3x)

User: Your job is to convert the abstract representation of the user's request into a valid GraphQL query, mutation or subscription syntax.

The users intent was already extracted in a previous step, leaving you with the sole task of converting this structured information about the types, attributes and parameters that was extracted into a valid GraphQL operation.

Valid attributes for types are found in the following GraphQL schema in between \n====\n.

You must only use valid attributes and arguments on types found in said schema, not invent fields that do not exist.

System: {ENCODED_SAMPLE_SCHEMA}

System: Between the following \n====\n you will find the extracted entities that you are to use for query generation.

System: \n====\n{ENCODED_ENTITY_EXTRACTION}\n====\n

User: Generate a valid GraphQL operation for the given entities that matches to the given GraphQL Schema.

Assistant: {EXPECTED_GRAPHQL_OPERATION}

Actual Task

User: Your job is to convert the abstract representation of the user's request into a valid GraphQL query, mutation or subscription syntax.

The users intent was already extracted in a previous step, leaving you with the sole task of converting this structured information about the types, attributes and parameters that was extracted into a valid GraphQL operation.

Valid attributes for types are found in the following GraphQL schema in between \n====\n.

You must only use valid attributes and arguments on types found in said schema, not invent fields that do not exist.

System: \n====\n{ENCODED_ACTUAL_SCHEMA}\n====\n

System: Between the following \n====\n you will find the extracted entities that you are to use for query generation.

System: \n====\n{ENCODED_ENTITY_EXTRACTION}\n====\n

User: Generate a valid GraphQL operation for the given entities that matches to the given GraphQL Schema.

A.6.5. Strategy 2b

Listing A.6: Prompt used by strategy 2b

Motivational Speech

System: You are a world-class algorithm for extracting information in structured formats.

Accepted format

System: The response must be valid JSON only. No explanation text around it.
The response must be an object containing a field 'operationType' indicating the type of operation to perform (query, mutation, subscription) and a field 'entities' being an array of objects.
Each object has a 'type' field with the value 'Entity' or 'Attribute'.
An 'Entity' object has a 'name' field with its name on the parent, a 'typeName' which is the most concrete name of the type on the schema and a 'fields' field with an array of 'Attribute' and 'Entity' objects.
For example if there is an interface 'Vehicle' and a type 'Car' implementing 'Vehicle', the 'typeName' field should be 'Car' and not 'Vehicle'.
If an attribute could not be matched to the schema but was found in the prompt, set the property 'matched' to false in the object.
Parameters are stored in the 'parameters' field as an array of objects. Each parameter can either be a single parameter field (type "parameter") or a list of parameters (type "parameterList").
Inside parameterContent can be either a ParameterValue, a ParameterValueList (for values that are an array or list) or a ParameterObject, that contains either a parameter or parameterList inside of their properties.
Be aware that for some requests, the user may ask for a single entity (such as a book) with a filter condition and such filter conditions only exist on the query for all entities (such as books or allBooks). In that case you must use the entity for allBooks/books if you see a filter condition, while you should still use the single entity query "book" if you know the required parameters.
The same goes for entities such as characters, where the user may ask for a single character with a filter condition, but the filter condition only exists on the field 'characters' or 'allCharacters' and not on the field 'character'.
Be careful to always stick to the correct types. For example, if a filter is inside of a complex type structure, be sure to follow it - for example, always use the eq-property when filtering by name, never directly assign the value to a property when the structure on the schema calls for a nested object.
You must never add comments in any form on into the resulting json structure.

Shots (3x)

User: Extract the searched attribute from based on the entity in question.
Valid attributes for entities, represented as GraphQL types, are found in the following schema in between \n====\n.
Give the user a listing of the entities and their requested attributes.

System: \n====\n{ENCODED_SAMPLE_SCHEMA}\n====\n

Assistant: {EXPECTED JSON ENTITY EXTRACTION}

Preparation

User: Extract the searched attribute from based on the entity in question.
Valid attributes for entities, represented as GraphQL types, are found in the following schema in between \n====\n.
Give the user a listing of the entities and their requested attributes.

System: \n====\n{ENCODED_ACTUAL_SCHEMA}\n====\n

User Prompt

User: {PROMPT_GOES_HERE}

A.7. Test Schemas

BookStore Schema

Listing A.7: BookStore Schema

```
# Schema for Books and Movies Library

# Basic scalar types for Book and Movie
scalar Date

# Enums for Genre to categorize books and movies
enum BookGenre {
  FICTION
  NONFICTION
  SCIENCE
  HISTORY
  FANTASY
  MYSTERY
}

enum MovieGenre {
  ACTION
  COMEDY
  DRAMA
  FANTASY
  HORROR
  SCIFI
}

# The User type represents library members
type User {
  id: ID!
  name: String!
  email: String!
  borrowedItems: [LendableItem!]!
}

# Interface for lendable items in the library
interface LendableItem {
  id: ID!
  title: String!
  isAvailable: Boolean!
}

# Book type with fields specific to books
type Book implements LendableItem {
  id: ID!
  title: String!
  author: String!
  genre: BookGenre!
  publishedYear: Int!
  isAvailable: Boolean!
}

# Movie type with fields specific to movies
type Movie implements LendableItem {
```

```
    id: ID!
    title: String!
    director: String!
    genre: MovieGenre!
    releasedYear: Int!
    isAvailable: Boolean!
  }

# Query type for fetching data
type Query {
  getBookById(id: ID!): Book
  getAllBooks: [Book!]!
  getAllMovies: [Movie!]!
  getUserBorrowedItems(userId: ID!): [LendableItem!]!
  searchItems(title: String!): [LendableItem!]!
}

# Mutation type for creating and updating data
type Mutation {
  borrowItem(userId: ID!, itemId: ID!): Boolean!
  returnItem(userId: ID!, itemId: ID!): Boolean!
  addUser(name: String!, email: String!): User!
}

# Subscription type for real-time updates
type Subscription {
  itemStatusChanged(itemId: ID!): LendableItem!
}
```

Swapi-Official Schema

Listing A.8: Swapi-Official Schema

```
schema {
  query: Root
}

"""A single film."""
type Film implements Node {
  characterConnection(after: String, before: String, first: Int, last: Int):
  FilmCharactersConnection

  """The ISO 8601 date format of the time that this resource was created."""
  created: String

  """The name of the director of this film."""
  director: String

  """The ISO 8601 date format of the time that this resource was edited."""
  edited: String

  """The episode number of this film."""
  episodeID: Int

  """The ID of an object"""
  id: ID!
}
```

```
    """The opening paragraphs at the beginning of this film."""
    openingCrawl: String
    planetConnection(after: String, before: String, first: Int, last: Int):
FilmPlanetsConnection

    """The name(s) of the producer(s) of this film."""
    producers: [String]

    """The ISO 8601 date format of film release at original creator country."""
    releaseDate: String
    speciesConnection(after: String, before: String, first: Int, last: Int):
FilmSpeciesConnection
    starshipConnection(after: String, before: String, first: Int, last: Int):
FilmStarshipsConnection

    """The title of this film."""
    title: String
    vehicleConnection(after: String, before: String, first: Int, last: Int):
FilmVehiclesConnection
  }

  """A connection to a list of items."""
  type FilmCharactersConnection {
    """
    A list of all of the objects returned in the connection. This is a convenience
    field provided for quickly exploring the API; rather than querying for
    "{ edges { node } }" when no edge data is needed, this field can be used
    instead. Note that when clients like Relay need to fetch the "cursor" field on
    the edge to enable efficient pagination, this shortcut cannot be used, and the
    full "{ edges { node } }" version should be used instead.
    """
    characters: [Person]

    """A list of edges."""
    edges: [FilmCharactersEdge]

    """Information to aid in pagination."""
    pageInfo: PageInfo!

    """
    A count of the total number of objects in this connection, ignoring pagination.
    """
    totalCount: Int
  }

  """An edge in a connection."""
  type FilmCharactersEdge {
    """A cursor for use in pagination"""
    cursor: String!

    """The item at the end of the edge"""
    node: Person
  }

  """A connection to a list of items."""
  type FilmPlanetsConnection {
```

```
    """A list of edges."""
    edges: [FilmPlanetsEdge]

    """Information to aid in pagination."""
    pageInfo: PageInfo!

    """
    A list of all of the objects returned in the connection. This is a convenience
    field provided for quickly exploring the API; rather than querying for
    "{ edges { node } }" when no edge data is needed, this field can be used
    instead. Note that when clients like Relay need to fetch the "cursor" field on
    the edge to enable efficient pagination, this shortcut cannot be used, and the
    full "{ edges { node } }" version should be used instead.
    """
    planets: [Planet]

    """
    A count of the total number of objects in this connection, ignoring pagination.
    """
    totalCount: Int!
  }

  """An edge in a connection."""
  type FilmPlanetsEdge {
    """A cursor for use in pagination"""
    cursor: String!

    """The item at the end of the edge"""
    node: Planet!
  }

  """A connection to a list of items."""
  type FilmSpeciesConnection {
    """A list of edges."""
    edges: [FilmSpeciesEdge]

    """Information to aid in pagination."""
    pageInfo: PageInfo!

    """
    A list of all of the objects returned in the connection. This is a convenience
    field provided for quickly exploring the API; rather than querying for
    "{ edges { node } }" when no edge data is needed, this field can be used
    instead. Note that when clients like Relay need to fetch the "cursor" field on
    the edge to enable efficient pagination, this shortcut cannot be used, and the
    full "{ edges { node } }" version should be used instead.
    """
    species: [Species]

    """
    A count of the total number of objects in this connection, ignoring pagination.
    """
    totalCount: Int!
  }

  """An edge in a connection."""
  type FilmSpeciesEdge {
```



```
    """A cursor for use in pagination"""
    cursor: String!

    """The item at the end of the edge"""
    node: Species
  }

  """A connection to a list of items."""
  type FilmStarshipsConnection {
    """A list of edges."""
    edges: [FilmStarshipsEdge]

    """Information to aid in pagination."""
    pageInfo: PageInfo!

    """
    A list of all of the objects returned in the connection. This is a convenience
    field provided for quickly exploring the API; rather than querying for
    "{ edges { node } }" when no edge data is needed, this field can be used
    instead. Note that when clients like Relay need to fetch the "cursor" field on
    the edge to enable efficient pagination, this shortcut cannot be used, and the
    full "{ edges { node } }" version should be used instead.
    """
    starships: [Starship]

    """
    A count of the total number of objects in this connection, ignoring pagination.
    """
    totalCount: Int
  }

  """An edge in a connection."""
  type FilmStarshipsEdge {
    """A cursor for use in pagination"""
    cursor: String!

    """The item at the end of the edge"""
    node: Starship
  }

  """A connection to a list of items."""
  type FilmVehiclesConnection {
    """A list of edges."""
    edges: [FilmVehiclesEdge]

    """Information to aid in pagination."""
    pageInfo: PageInfo!

    """
    A count of the total number of objects in this connection, ignoring pagination.
    """

    totalCount: Int

    """
    A list of all of the objects returned in the connection. This is a convenience
    field provided for quickly exploring the API; rather than querying for
```

```
"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
vehicles: [Vehicle]
}

"""An edge in a connection."""
type FilmVehiclesEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Vehicle
}

"""A connection to a list of items."""
type FilmsConnection {
  """A list of edges."""
  edges: [FilmsEdge]

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
  the edge to enable efficient pagination, this shortcut cannot be used, and the
  full "{ edges { node } }" version should be used instead.
  """
  films: [Film]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A count of the total number of objects in this connection, ignoring pagination.
  """
  totalCount: Int
}

"""An edge in a connection."""
type FilmsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Film
}

"""An object with an ID"""
interface Node {
  """The id of the object."""
  id: ID!
}

"""Information about pagination in a connection."""
```

```
type PageInfo {
  """When paginating forwards, the cursor to continue."""
  endCursor: String

  """When paginating forwards, are there more items?"""
  hasNextPage: Boolean!

  """When paginating backwards, are there more items?"""
  hasPreviousPage: Boolean!

  """When paginating backwards, the cursor to continue."""
  startCursor: String
}

"""A connection to a list of items."""
type PeopleConnection {
  """A list of edges."""
  edges: [PeopleEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
  the edge to enable efficient pagination, this shortcut cannot be used, and the
  full "{ edges { node } }" version should be used instead.
  """
  people: [Person]

  """
  A count of the total number of objects in this connection, ignoring pagination.
  """
  totalCount: Int
}

"""An edge in a connection."""
type PeopleEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Person
}

"""An individual person or character within the Star Wars universe."""
type Person implements Node {
  """
  The birth year of the person, using the in-universe standard of BBY or ABY -
  Before the Battle of Yavin or After the Battle of Yavin. The Battle of Yavin is
  a battle that occurs at the end of Star Wars episode IV: A New Hope.
  """
  birthYear: String

  """The ISO 8601 date format of the time that this resource was created."""
}
```

```
created: String

"""The ISO 8601 date format of the time that this resource was edited."""
edited: String

"""
The eye color of this person. Will be "unknown" if not known or "n/a" if the
person does not have an eye.
"""
eyeColor: String
filmConnection(after: String, before: String, first: Int, last: Int):
PersonFilmsConnection

"""
The gender of this person. Either "Male", "Female" or "unknown",
"n/a" if the person does not have a gender.
"""
gender: String

"""
The hair color of this person. Will be "unknown" if not known or "n/a" if the
person does not have hair.
"""
hairColor: String

"""The height of the person in centimeters."""
height: Int

"""A planet that this person was born on or inhabits."""
homeworld: Planet

"""The ID of an object"""
id: ID!

"""The mass of the person in kilograms."""
mass: Float

"""The name of this person."""
name: String

"""The skin color of this person."""
skinColor: String

"""The species that this person belongs to, or null if unknown."""
species: Species
starshipConnection(after: String, before: String, first: Int, last: Int):
PersonStarshipsConnection
vehicleConnection(after: String, before: String, first: Int, last: Int):
PersonVehiclesConnection
}

"""A connection to a list of items."""
type PersonFilmsConnection {
  """A list of edges."""
  edges: [PersonFilmsEdge]

  """
```

```
A list of all of the objects returned in the connection. This is a convenience
field provided for quickly exploring the API; rather than querying for
"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
films: [Film]

"""Information to aid in pagination."""
pageInfo: PageInfo!

"""
A count of the total number of objects in this connection, ignoring pagination.
"""
totalCount: Int
}

"""An edge in a connection."""
type PersonFilmsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Film
}

"""A connection to a list of items."""
type PersonStarshipsConnection {
  """A list of edges."""
  edges: [PersonStarshipsEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
A list of all of the objects returned in the connection. This is a convenience
field provided for quickly exploring the API; rather than querying for
"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
starships: [Starship]

"""
A count of the total number of objects in this connection, ignoring pagination.
"""
totalCount: Int
}

"""An edge in a connection."""
type PersonStarshipsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
```

```
node: Starship
}

"""A connection to a list of items."""
type PersonVehiclesConnection {
  """A list of edges."""
  edges: [PersonVehiclesEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A count of the total number of objects in this connection, ignoring pagination.
  """
  totalCount: Int

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
  the edge to enable efficient pagination, this shortcut cannot be used, and the
  full "{ edges { node } }" version should be used instead.
  """
  vehicles: [Vehicle]
}

"""An edge in a connection."""
type PersonVehiclesEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Vehicle
}

"""
A large mass, planet or planetoid in the Star Wars Universe, at the time of
0 ABY.
"""
type Planet implements Node {
  """The climates of this planet."""
  climates: [String]

  """The ISO 8601 date format of the time that this resource was created."""
  created: String

  """The diameter of this planet in kilometers."""
  diameter: Int

  """The ISO 8601 date format of the time that this resource was edited."""
  edited: String
  filmConnection(after: String, before: String, first: Int, last: Int):
PlanetFilmsConnection

  """
  A number denoting the gravity of this planet, where "1" is normal or 1 standard
```

```
G. "2" is twice or 2 standard Gs. "0.5" is half or 0.5 standard Gs.
"""
gravity: String

"""The ID of an object"""
id: ID!

"""The name of this planet."""
name: String

"""
The number of standard days it takes for this planet to complete a single orbit
of its local star.
"""
orbitalPeriod: Int

"""The average population of sentient beings inhabiting this planet."""
population: Float
residentConnection(after: String, before: String, first: Int, last: Int):
PlanetResidentsConnection

"""
The number of standard hours it takes for this planet to complete a single
rotation on its axis.
"""
rotationPeriod: Int

"""
The percentage of the planet surface that is naturally occurring water or bodies
of water.
"""
surfaceWater: Float

"""The terrains of this planet."""
terrains: [String]
}

"""A connection to a list of items."""
type PlanetFilmsConnection {
  """A list of edges."""
  edges: [PlanetFilmsEdge]

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
  the edge to enable efficient pagination, this shortcut cannot be used, and the
  full "{ edges { node } }" version should be used instead.
  """
  films: [Film]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A count of the total number of objects in this connection, ignoring pagination.

```

```
"""
  totalCount: Int
}

"""An edge in a connection."""
type PlanetFilmsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Film
}

"""A connection to a list of items."""
type PlanetResidentsConnection {
  """A list of edges."""
  edges: [PlanetResidentsEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
  the edge to enable efficient pagination, this shortcut cannot be used, and the
  full "{ edges { node } }" version should be used instead.
  """
  residents: [Person]
}

"""
  A count of the total number of objects in this connection, ignoring pagination.
  """
  totalCount: Int
}

"""An edge in a connection."""
type PlanetResidentsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Person
}

"""A connection to a list of items."""
type PlanetsConnection {
  """A list of edges."""
  edges: [PlanetsEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
```



```

"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
planets: [Planet]

"""
A count of the total number of objects in this connection, ignoring pagination.
"""
totalCount: Int
}

"""An edge in a connection."""
type PlanetsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Planet
}

type Root {
  allFilms(after: String, before: String, first: Int, last: Int): FilmsConnection
  allPeople(after: String, before: String, first: Int, last: Int):
PeopleConnection
  allPlanets(after: String, before: String, first: Int, last: Int):
PlanetsConnection
  allSpecies(after: String, before: String, first: Int, last: Int):
SpeciesConnection
  allStarships(after: String, before: String, first: Int, last: Int):
StarshipsConnection
  allVehicles(after: String, before: String, first: Int, last: Int):
VehiclesConnection
  film(id: ID!): Film

  """Fetches an object given its ID"""
  node(
    """The ID of an object"""
    id: ID!
  ): Node
  person(id: ID!): Person
  planet(id: ID!): Planet
  species(id: ID!): Species
  starship(id: ID!): Starship
  vehicle(id: ID!): Vehicle
}

"""A type of person or character within the Star Wars Universe."""
type Species implements Node {
  """The average height of this species in centimeters."""
  averageHeight: Float

  """The average lifespan of this species in years, null if unknown."""
  averageLifespan: Int

  """The classification of this species, such as "mammal" or "reptile."""

```

```
classification: String

"""The ISO 8601 date format of the time that this resource was created."""
created: String

"""The designation of this species, such as "sentient"."""
designation: String

"""The ISO 8601 date format of the time that this resource was edited."""
edited: String

"""
Common eye colors for this species, null if this species does not typically
have eyes.
"""
eyeColors: [String]
filmConnection(after: String, before: String, first: Int, last: Int):
SpeciesFilmsConnection

"""
Common hair colors for this species, null if this species does not typically
have hair.
"""
hairColors: [String]

"""A planet that this species originates from."""
homeworld: Planet

"""The ID of an object"""
id: ID!

"""The language commonly spoken by this species."""
language: String

"""The name of this species."""
name: String
personConnection(after: String, before: String, first: Int, last: Int):
SpeciesPeopleConnection

"""
Common skin colors for this species, null if this species does not typically
have skin.
"""
skinColors: [String]
}

"""A connection to a list of items."""
type SpeciesConnection {
  """A list of edges."""
  edges: [SpeciesEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
```

```
"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
species: [Species]

"""
A count of the total number of objects in this connection, ignoring pagination.
"""
totalCount: Int
}

"""An edge in a connection."""
type SpeciesEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Species
}

"""A connection to a list of items."""
type SpeciesFilmsConnection {
  """A list of edges."""
  edges: [SpeciesFilmsEdge]

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
  the edge to enable efficient pagination, this shortcut cannot be used, and the
  full "{ edges { node } }" version should be used instead.
  """
  films: [Film]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A count of the total number of objects in this connection, ignoring pagination.
  """
  totalCount: Int
}

"""An edge in a connection."""
type SpeciesFilmsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Film
}

"""A connection to a list of items."""
type SpeciesPeopleConnection {
```

```
""A list of edges.""
edges: [SpeciesPeopleEdge]

""Information to aid in pagination.""
pageInfo: PageInfo!

""
A list of all of the objects returned in the connection. This is a convenience
field provided for quickly exploring the API; rather than querying for
"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
""
people: [Person]

""
A count of the total number of objects in this connection, ignoring pagination.
""
totalCount: Int
}

""An edge in a connection.""
type SpeciesPeopleEdge {
  ""A cursor for use in pagination""
  cursor: String!

  ""The item at the end of the edge""
  node: Person
}

""A single transport craft that has hyperdrive capability.""
type Starship implements Node {
  ""
  The Maximum number of Megalights this starship can travel in a standard hour.
  A "Megalight" is a standard unit of distance and has never been defined before
  within the Star Wars universe. This figure is only really useful for measuring
  the difference in speed of starships. We can assume it is similar to AU, the
  distance between our Sun (Sol) and Earth.
  ""
  MGLT: Int

  ""The maximum number of kilograms that this starship can transport.""
  cargoCapacity: Float

  ""
  The maximum length of time that this starship can provide consumables for its
  entire crew without having to resupply.
  ""
  consumables: String

  ""The cost of this starship new, in galactic credits.""
  costInCredits: Float

  ""The ISO 8601 date format of the time that this resource was created.""
  created: String
}
```

```
""The number of personnel needed to run or pilot this starship.""
crew: String

""The ISO 8601 date format of the time that this resource was edited.""
edited: String
filmConnection(after: String, before: String, first: Int, last: Int):
StarshipFilmsConnection

""The class of this starships hyperdrive.""
hyperdriveRating: Float

""The ID of an object""
id: ID!

""The length of this starship in meters.""
length: Float

""The manufacturers of this starship.""
manufacturers: [String]

""
The maximum speed of this starship in atmosphere. null if this starship is
incapable of atmosphering flight.
""
maxAtmospheringSpeed: Int

""
The model or official name of this starship. Such as "T-65 X-wing" or "DS-1
Orbital Battle Station".
""
model: String

""The name of this starship. The common name, such as "Death Star".""
name: String

""The number of non-essential people this starship can transport.""
passengers: String
pilotConnection(after: String, before: String, first: Int, last: Int):
StarshipPilotsConnection

""
The class of this starship, such as "Starfighter" or "Deep Space Mobile
Battlestation"
""
starshipClass: String
}

""A connection to a list of items.""
type StarshipFilmsConnection {
  ""A list of edges.""
  edges: [StarshipFilmsEdge]

  ""
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
```

```
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
films: [Film]

"""Information to aid in pagination."""
pageInfo: PageInfo!

"""
A count of the total number of objects in this connection, ignoring pagination.
"""
totalCount: Int
}

"""An edge in a connection."""
type StarshipFilmsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Film
}

"""A connection to a list of items."""
type StarshipPilotsConnection {
  """A list of edges."""
  edges: [StarshipPilotsEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
A list of all of the objects returned in the connection. This is a convenience
field provided for quickly exploring the API; rather than querying for
"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
pilots: [Person]

"""
A count of the total number of objects in this connection, ignoring pagination.
"""
totalCount: Int
}

"""An edge in a connection."""
type StarshipPilotsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Person
}

"""A connection to a list of items."""
```

```
type StarshipsConnection {
  """A list of edges."""
  edges: [StarshipsEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
  the edge to enable efficient pagination, this shortcut cannot be used, and the
  full "{ edges { node } }" version should be used instead.
  """
  starships: [Starship]

  """
  A count of the total number of objects in this connection, ignoring pagination.
  """
  totalCount: Int
}

"""An edge in a connection."""
type StarshipsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Starship
}

"""A single transport craft that does not have hyperdrive capability"""
type Vehicle implements Node {
  """The maximum number of kilograms that this vehicle can transport."""
  cargoCapacity: Float

  """
  The maximum length of time that this vehicle can provide consumables for its
  entire crew without having to resupply.
  """
  consumables: String

  """The cost of this vehicle new, in Galactic Credits."""
  costInCredits: Float

  """The ISO 8601 date format of the time that this resource was created."""
  created: String

  """The number of personnel needed to run or pilot this vehicle."""
  crew: String

  """The ISO 8601 date format of the time that this resource was edited."""
  edited: String
  filmConnection(after: String, before: String, first: Int, last: Int):
  VehicleFilmsConnection
}
```

```
"""The ID of an object"""
id: ID!

"""The length of this vehicle in meters."""
length: Float

"""The manufacturers of this vehicle."""
manufacturers: [String]

"""The maximum speed of this vehicle in atmosphere."""
maxAtmospheringSpeed: Int

"""
The model or official name of this vehicle. Such as "All-Terrain Attack
Transport".
"""
model: String

"""
The name of this vehicle. The common name, such as "Sand Crawler" or "Speeder
bike".
"""
name: String

"""The number of non-essential people this vehicle can transport."""
passengers: String
pilotConnection(after: String, before: String, first: Int, last: Int):
VehiclePilotsConnection

"""The class of this vehicle, such as "Wheeled" or "Repulsorcraft"."""
vehicleClass: String
}

"""A connection to a list of items."""
type VehicleFilmsConnection {
  """A list of edges."""
  edges: [VehicleFilmsEdge]

  """
A list of all of the objects returned in the connection. This is a convenience
field provided for quickly exploring the API; rather than querying for
"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
  films: [Film]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
A count of the total number of objects in this connection, ignoring pagination.
"""
  totalCount: Int
}
```



```
"""An edge in a connection."""
type VehicleFilmsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Film
}

"""A connection to a list of items."""
type VehiclePilotsConnection {
  """A list of edges."""
  edges: [VehiclePilotsEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A list of all of the objects returned in the connection. This is a convenience
  field provided for quickly exploring the API; rather than querying for
  "{ edges { node } }" when no edge data is needed, this field can be used
  instead. Note that when clients like Relay need to fetch the "cursor" field on
  the edge to enable efficient pagination, this shortcut cannot be used, and the
  full "{ edges { node } }" version should be used instead.
  """
  pilots: [Person]

  """
  A count of the total number of objects in this connection, ignoring pagination.
  """
  totalCount: Int
}

"""An edge in a connection."""
type VehiclePilotsEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Person
}

"""A connection to a list of items."""
type VehiclesConnection {
  """A list of edges."""
  edges: [VehiclesEdge]

  """Information to aid in pagination."""
  pageInfo: PageInfo!

  """
  A count of the total number of objects in this connection, ignoring pagination.
  """
  totalCount: Int

  """
  A list of all of the objects returned in the connection. This is a convenience
```

```
field provided for quickly exploring the API; rather than querying for
"{ edges { node } }" when no edge data is needed, this field can be used
instead. Note that when clients like Relay need to fetch the "cursor" field on
the edge to enable efficient pagination, this shortcut cannot be used, and the
full "{ edges { node } }" version should be used instead.
"""
vehicles: [Vehicle]
}

"""An edge in a connection."""
type VehiclesEdge {
  """A cursor for use in pagination"""
  cursor: String!

  """The item at the end of the edge"""
  node: Vehicle
}
```

Swapi-HotChocolate Schema

Listing A.9: Swapi-HotChocolate Schema

```
# The `@defer` directive may be provided for fragment spreads and inline fragments
# to inform the executor to delay the execution of the current fragment to
# indicate deprioritization of the current fragment. A query with `@defer`
# directive will cause the request to potentially return multiple responses, where
# non-deferred data is delivered in the initial response and data deferred is
# delivered in a subsequent response. `@include` and `@skip` take precedence over
`@defer`.
directive @defer(
  # If this argument label has a value other than null, it will be passed on to
  # the result of this defer directive. This label is intended to give client
  # applications a way to identify to which fragment a deferred result belongs to.
  label: String
  # Deferred when true.
  if: Boolean
) on FRAGMENT_SPREAD | INLINE_FRAGMENT
# The `@stream` directive may be provided for a field of `List` type so that the
# backend can leverage technology such as asynchronous iterators to provide a
# partial list in the initial response, and additional list items in subsequent
# responses. `@include` and `@skip` take precedence over `@stream`.
directive @stream(
  # If this argument label has a value other than null, it will be passed on to
  # the result of this stream directive. This label is intended to give client
  # applications a way to identify to which fragment a streamed result belongs to.
  label: String
  # The initial elements that shall be send down to the consumer.
  initialCount: Int!
  # Streamed when true.
  if: Boolean!
) on FIELD
# A character in the Star Wars universe.
interface Character {
  # The unique identifier for the character.
  id: Int!
  # The name of the character.
```

```
name: String!
# The ids of the character's friends.
friends(
  first: Int
  after: String
  last: Int
  before: String
): CharacterConnection
# The episodes the character appears in.
appearsIn: [Episode!]!
# The height of the character.
height(unit: Unit = METERS): Float!
}

# A connection to a list of items.
type CharacterConnection {
  # Information to aid in pagination.
  pageInfo: PageInfo!
  # A list of edges.
  edges: [CharacterEdge!]
  # A flattened list of the nodes.
  nodes: [Character]
}

# An edge in a connection.
type CharacterEdge {
  # A cursor for use in pagination.
  cursor: String!
  # The item at the end of the edge.
  node: Character
}

input ComparableDoubleOperationFilterInput {
  eq: Float
  neq: Float
  in: [Float!]
  nin: [Float!]
  gt: Float
  ngt: Float
  gte: Float
  ngte: Float
  lt: Float
  nlt: Float
  lte: Float
  nlte: Float
}

input ComparableGuidOperationFilterInput {
  eq: Uuid
  neq: Uuid
  in: [Uuid!]
  nin: [Uuid!]
  gt: Uuid
  ngt: Uuid
  gte: Uuid
  ngte: Uuid
  lt: Uuid
```

```
nlt: Uuid
lte: Uuid
nlte: Uuid
}

input ComparableInt32OperationFilterInput {
  eq: Int
  neq: Int
  in: [Int!]
  nin: [Int!]
  gt: Int
  ngt: Int
  gte: Int
  ngte: Int
  lt: Int
  nlt: Int
  lte: Int
  nlte: Int
}

# This input represents the data needed to create a review.
input CreateReviewInput {
  # The review for which to create the review.
  episode: Episode!
  # The number of stars given for this review.
  stars: Int!
  # An explanation for the rating.
  commentary: String!
}

# This payload allows us to query the created review object.
type CreateReviewPayload {
  # The episode for which a review was created.
  episode: Episode!
  # The review that was being created.
  review: Review!
}

# This resolver class extends all object types implementing ICharacter.
type Droid implements Character {
  # The unique identifier for the character.
  id: Int!
  # The name of the character.
  name: String!
  # The episodes the character appears in.
  appearsIn: [Episode!]!
  # The droid's primary function.
  primaryFunction: String!
  # The height of the character.
  height(unit: Unit = METERS): Float!
  friends(
    first: Int
    after: String
    last: Int
    before: String
  ): CharacterConnection
}
```

```
# The Star Wars episodes.
enum Episode {
  # Star Wars Episode IV: A New Hope
  NEW_HOPE
  # Star Wars Episode V: Empire Strikes Back
  EMPIRE
  # Star Wars Episode VI: Return of the Jedi
  JEDI
}

input EpisodeOperationFilterInput {
  eq: Episode
  neq: Episode
  in: [Episode!]
  nin: [Episode!]
}

# This resolver class extends all object types implementing ICharacter.
type Human implements Character {
  # The unique identifier for the character.
  id: Int!
  # The name of the character.
  name: String!
  # The episodes the character appears in.
  appearsIn: [Episode!]!
  # The planet the character is originally from.
  homePlanet: String
  # The height of the character.
  height(unit: Unit = METERS): Float!
  friends(
    first: Int
    after: String
    last: Int
    before: String
  ): CharacterConnection
}

# A character in the Star Wars universe.
input ICharacterFilterInput {
  and: [ICharacterFilterInput!]
  or: [ICharacterFilterInput!]
  # The unique identifier for the character.
  id: ComparableInt32OperationFilterInput
  # The name of the character.
  name: StringOperationFilterInput
  # The ids of the character's friends.
  friends: ListComparableInt32OperationFilterInput
  # The episodes the character appears in.
  appearsIn: ListEpisodeOperationFilterInput
  # The height of the character.
  height: ComparableDoubleOperationFilterInput
}

# A character in the Star Wars universe.
input ICharacterSortInput {
  # The unique identifier for the character.
```

```
    id: SortEnumType
    # The name of the character.
    name: SortEnumType
    # The height of the character.
    height: SortEnumType
  }

  input ListComparableInt32OperationFilterInput {
    all: ComparableInt32OperationFilterInput
    none: ComparableInt32OperationFilterInput
    some: ComparableInt32OperationFilterInput
    any: Boolean
  }

  input ListEpisodeOperationFilterInput {
    all: EpisodeOperationFilterInput
    none: EpisodeOperationFilterInput
    some: EpisodeOperationFilterInput
    any: Boolean
  }

  # The mutations related to reviews.
  type Mutation {
    # Creates a review for a given Star Wars episode.
    createReview(input: CreateReviewInput!): CreateReviewPayload!
  }

  # Information about pagination in a connection.
  type PageInfo {
    # Indicates whether more edges exist following the set defined by the clients
    arguments.
    hasNextPage: Boolean!
    # Indicates whether more edges exist prior the set defined by the clients
    arguments.
    hasPreviousPage: Boolean!
    # When paginating backwards, the cursor to continue.
    startCursor: String
    # When paginating forwards, the cursor to continue.
    endCursor: String
  }

  # The queries related to reviews.
  type Query {
    # Retrieve a hero by a particular Star Wars episode.
    #
    #
    # **Returns:**
    # The hero character.
    hero(
      # The episode to retrieve the hero.
      episode: Episode!
    ): Character!
    # Gets all character.
    #
    #
    # **Returns:**
    # The character.
  }
}
```

```
characters(  
  first: Int  
  after: String  
  last: Int  
  before: String  
  where: ICharacterFilterInput  
  order: [ICharacterSortInput!]  
): CharacterConnection  
# Gets a character by it`s id.  
#  
#  
# **Returns:**  
# The character.  
character(  
  # The ids of the human to retrieve.  
  ids: [Int!]!  
): [Character!]!  
# Search the repository for objects that contain the text.  
#  
#  
# **Returns:**  
# Returns the union type ISearchResult.  
search(  
  # The text we are searching for.  
  text: String!  
): [SearchResult!]!  
reviews(  
  first: Int  
  after: String  
  last: Int  
  before: String  
  episode: Episode!  
  where: ReviewFilterInput  
  order: [ReviewSortInput!]  
): ReviewConnection  
}  
  
# A review of a particular movie.  
type Review {  
  # The ID of the review.  
  id: Uuid!  
  # The number of stars given for this review.  
  stars: Int!  
  # An explanation for the rating.  
  commentary: String!  
}  
  
# A connection to a list of items.  
type ReviewConnection {  
  # Information to aid in pagination.  
  pageInfo: PageInfo!  
  # A list of edges.  
  edges: [ReviewEdge!]  
  # A flattened list of the nodes.  
  nodes: [Review!]  
}
```

```
# An edge in a connection.
type ReviewEdge {
  # A cursor for use in pagination.
  cursor: String!
  # The item at the end of the edge.
  node: Review!
}

# A review of a particular movie.
input ReviewFilterInput {
  and: [ReviewFilterInput!]
  or: [ReviewFilterInput!]
  # The ID of the review.
  id: ComparableGuidIdOperationFilterInput
  # The number of stars given for this review.
  stars: ComparableInt32OperationFilterInput
  # An explanation for the rating.
  commentary: StringOperationFilterInput
}

# A review of a particular movie.
input ReviewSortInput {
  # The ID of the review.
  id: SortEnumType
  # The number of stars given for this review.
  stars: SortEnumType
  # An explanation for the rating.
  commentary: SortEnumType
}

union SearchResult = Human | Droid | Starship
enum SortEnumType {
  ASC
  DESC
}

# A starship in the Star Wars universe.
type Starship {
  # The Id of the starship.
  id: Int!
  # The name of the starship.
  name: String!
  # The length of the starship.
  length(unit: Unit = METERS): Float!
}

input StringOperationFilterInput {
  and: [StringOperationFilterInput!]
  or: [StringOperationFilterInput!]
  eq: String
  neq: String
  contains: String
  ncontains: String
  in: [String]
  nin: [String]
  startsWith: String
}
```



```
nstartsWith: String
endsWith: String
nendsWith: String
}

# The subscriptions related to reviews.
type Subscription {
  # The OnReview event is invoked whenever a new review is being created.
  #
  #
  # **Returns:**
  #
  # The review that was created.
  onReview(
    # The episode to which you want to subscribe to.
    episode: Episode!
  ): Review!
}

# Different units of measurement.
enum Unit {
  FOOT
  METERS
}

scalar Uuid
```