future internet

Article

Real-Time Text-to-Cypher Query Generation with Large Language Models for Graph Databases

Markus Hornsteiner , Michael Kreussel , Christoph Steindl , Fabian Ebner , Philip Empl and Stefan Schönig *

Faculty of Informatics and Data Science, University of Regensburg, 93053 Regensburg, Germany; markus.hornsteiner@ur.de (M.H.); philip.empl@ur.de (P.E.)
* Correspondence: stefan.schoenig@ur.de

Abstract: Based on their ability to efficiently and intuitively represent real-world relationships and structures, graph databases are gaining increasing popularity. In this context, this paper proposes an innovative integration of a Large Language Model into NoSQL databases and Knowledge Graphs to bridge the gap in field of Text-to-Cypher queries, focusing on Neo4j. Using the Design Science Research Methodology, we developed a Natural Language Interface which can receive user queries in real time, convert them into Cypher Query Language (CQL), and perform targeted queries, allowing users to choose from different graph databases. In addition, the user interaction is expanded by an additional chat function based on the chat history, as well as an error correction module, which elevates the precision of the generated Cypher statements. Our findings show that the chatbot is able to accurately and efficiently solve the tasks of database selection, chat history referencing, and CQL query generation. The developed system therefore makes an important contribution to enhanced interaction with graph databases, and provides a basis for the integration of further and multiple database technologies and LLMs, due to its modular pipeline architecture.

Keywords: chatbot; ChatGPT; cypher language; graph database; knowledge graphs; LLM; natural language interface; Neo4j; question answering

check for updates

Citation: Hornsteiner, M.; Kreussel, M.; Steindl, C.; Ebner, F.; Empl, P.; Schönig, S. Real-Time Text-to-Cypher Query Generation with Large Language Models for Graph Databases. *Future Internet* **2024**, *16*, 438. https://doi.org/10.3390/fi16120438

Academic Editors: Filipe Portela and Athanasios D. Panagopoulos

Received: 2 September 2024
Revised: 15 November 2024
Accepted: 19 November 2024
Published: 22 November 2024

1. Introduction

Large Language Models (LLMs) have emerged as powerful tools in the realm of natural language processing, possessing the unique capability of understanding and generating human language. This capability positions LLMs as transformative assets across a wide array of applications. In recent years, the deployment of LLMs has expanded rapidly, with diverse use cases emerging across multiple sectors. The introduction of generative AI models such as ChatGPT [1] has brought LLMs into the public spotlight, significantly elevating the recognition and adoption of AI technologies. This surge in interest has opened new avenues for developers and users alike, leading to the integration of generative AI into various aspects of daily life.

One of the key strengths of LLMs is their ability to process and analyze vast amounts of data in a relatively short time frame. As the world becomes increasingly interconnected, enormous volumes of related data are transmitted across the internet, necessitating structured approaches for processing and utilization. To manage these data, specialized database management systems have been developed, particularly for handling graph data. Among these, Neo4j stands out as the most popular graph database management system [2].

In this context, data models like NoSQL databases, when integrated with LLMs, offer a flexible and efficient way to process and represent diverse data types and complex relationships. Traditional relational database models, which require join operations for every edge usage, are inefficient in comparison to NoSQL models within the context of knowledge graphs [3]. Knowledge graphs are sophisticated data structures that represent entities as nodes and their relationships as edges, with both entities and nodes containing various attributes [4]. This structure provides a comprehensive model of a given domain, making it an ideal foundation for training LLMs. Neo4j, with its inherent compatibility with knowledge graph structures, is well-suited for managing such data. The semantic richness of knowledge graphs aligns seamlessly with the capabilities of LLMs, enabling a deeper understanding of data and their interconnections [5].

This paper aims to explore the potential synergies between NoSQL databases, knowledge graphs, and LLMs, specifically focusing on data processing and contextual analysis. To achieve this objective, we have developed a chatbot that leverages multiple databases, including knowledge graphs, to answer user queries about the data contained within these graphs. The application is designed with a modular architecture, allowing researchers to customize the chatbot to meet their specific needs. For this work, we utilized the OpenAI GPT-4 Turbo API [6]. Furthermore, we tested the application to assess its real-world applicability and to evaluate the accuracy of the system. Additionally, we address potential limitations and challenges associated with generating Cypher Query Language (CQL) statements.

To this end, this paper presents four key contributions to advance research in complex database query generation: (1) an automated approach for CQL generation tailored to meet the demands of complex query requirements, (2) a fully automated process that removes the dependence on template-based insertion methods prevalent in prior research, (3) integrated error correction mechanisms designed to enhance query accuracy, and (4) a robust database selection framework that optimally aligns generated queries with suitable databases. Together, these contributions aim to establish a more reliable and versatile framework for CQL generation in database applications.

The remainder of this paper is structured as follows: In Section 2 the core principals that are necessary to understand the topic are explained. First, Large Language Models in general are examined, followed by Prompt Engineering and Sampling. After that, NoSQL is described, combined with Knowledge Graphs and CQL. In Section 3 we investigate existing literature that focuses on creating query statements from natural language. In Section 4 the research approach of this paper will be dissected. Following this, the technical realization and implementation of the developed chatbots architecture is then shown in Section 5, and a brief demonstration of the application is given. Then, an evaluation of the developed prototype is carried out in Section 6, based on several criteria and the results are presented. Afterwards, the findings, challenges, and limitations of the study will be discussed in Section 7. The paper concludes with an outlook on opportunities for improvement and further development in Section 8.

2. Theoretical Foundations

In this section, the theoretical foundations to understand the core principals of the proposed integration are explained.

2.1. Large Language Models

Large Language Models (LLMs) are artificial intelligences that are trained with enormous amounts of words. In this process, they utilize neural networks to deduce complex relationships between words in test-based training data. Most times, LLMs use the transformer architecture as a foundation for language processing tasks. The transformer architecture is based on the concept of self-attention, in which several parts of the input sequence are put into relation to enable a differentiated view of the sequence [7]. Through training on the data, LLMs learn the connection between words and how they can be used in a language. This offers the possibility to perform certain processing tasks with the human language [8]. Generally, LLMs can be divided into their four core principals: pre-training, adaptation, utilization, and evaluation. All of these are essential for the success of an LLM [9].

Pre-training: In the pre-training phase, the foundation of the LLMs skills is being created. Through training on vast amounts of data, the LLM learns foundational Natural Language Understanding and Natural Language Generation abilities [9,10].

Adaptation: After the pre-training phase, LLMs have the capability to perform several tasks concerning human language. Depending on the overall goal of the specific LLM, in the adaption phase, the skills are expanded or the behavior is adjusted to the demands of the developer [9].

Utilization: In the utilization phase, different prompting strategies to tackle the processing of the tasks are designed. For example, the prompting strategy of in-context learning demands the user to frame the task in natural language and input it into a text box. The model then analyses and implements the requested task [9,11].

Evaluation: To evaluate the effectiveness and the quality of an LLM, different benchmarking tasks must be performed to empirically analyze the abilities of the LLM [9].

2.2. Prompt Engineering and Prompt Tuning

Prompt Engineering is the process of creating particular prompts that are input into the Large Language Model to generate the requested output from the LLM [12]. Engineering refers, in this context, to the formulation of the prompt, which is adjusted so that the model creates the desired outcome. Prompt Engineering tasks can be performed in two different ways. On the one hand, the prompts can be generated by hand. The results are then compared afterwards in order to identify the best result. Another approach to perform Prompt Engineering is the use of backpropagation in prompt embedding spaces. Backpropagation performs significantly better in Prompt Engineering cases, but requires the user to have access to the specific model [13]. In both cases, the goal of the Prompt Engineering process is to identify the best possible input to create the desired output. Prompt Tuning is related to Prompt Engineering, but they use different concepts in the context of LLMs. While prompt engineering only focuses on the adaptation of the input prompts, Prompt Tuning utilizes soft prompts to influence the output. Soft prompts are prompts that are engineered by Artificial Intelligence to achieve the best possible outcome for the specific purpose. Since soft prompts consist of embeddings that distill knowledge from the model itself, soft prompts can be task-specific and act as a substitute for additional training data. They are very effective at guiding the model towards the desired output [14,15]. Few-shot prompting is a Prompt Engineering task that enables the LLM to solve a given task by providing it with only a limited amount of examples to learn from. For this the prompt has to be very precise and follow a structured formulation to result in satisfactory results [1,16]. All of these concepts are foundational to improve the quality and specificity of a given language processing task.

2.3. Sampling

Text decoding methods that solely focus on high probability outputs lead to very generic and repetitive outputs. That is why models like the GPT model utilize sampling to generate more human-like outputs [1]. Sampling is a method of text generation in which the probability of the next token is being calculated. The token with the highest probability in the probability distribution is chosen as the next word in the output [17]. There are two different sampling methods that can be utilized for the decoding process. These two methods are the deterministic method and the stochastic method.

Deterministic sampling: In deterministic sampling, also known as greedy sampling, the token with the highest probability to be the next one, based on the probability distribution, is generally chosen. This leads to a very predictable output, which can cause repetitive and generic outputs by virtue of the deterministic approach [18,19].

Stochastic sampling: In stochastic sampling, tokens are generally chosen based on the probability distribution. However, in contrast to deterministic sampling, the next token is chosen arbitrarily from a vocabulary subset. The problem with this approach can be the loss of semantic meaning caused by the randomness of the approach [19]. Stochastic sampling methods like nucleus sampling [17] try to avoid this problem by generating the text based on a dynamic nucleus, which is a small subset of the whole vocabulary containing a varying amount of potential words. This method stands in contrast to other stochastic sampling methods that rely on a fixed candidate pool.

2.4. NoSQL

NoSQL is a generic term, and stands for "Not Only SQL". It represents a category of database management systems. It refers to any data storage method used for storing and querying data that does not utilize the traditional relational database management system model [20]. They are especially useful in cases where the strict complexity of relational databases is unneeded, and a significantly higher data throughput is desired. They also offer other benefits such as the ability to be horizontally scalable and, therefore, will not cause the same operational efforts as relational database management systems [21]. In general, there are four main categories of NoSQL database models, including key-value-stores, document-based stores, graph-based stores, and column-oriented stores. In this paper, we mainly focus on graph-based stores. Graph-based NoSQL databases uses relationships and nodes to represent the stored data. With this characteristic, they are especially fitting for Knowledge Graphs, since they are able to represent the natural structure of Knowledge Graphs efficiently [20,22]. With the efficient way of storing and querying data, NoSQL databases are valuable in an LLM context. With the higher flexibility, they are also more fitting at handling the complex data structures that are used by LLMs than other databases.

2.5. Knowledge Graphs

Knowledge Graphs are structured representations of information. They comprise the three main components: entities, relationships, and semantic descriptions. Entities are the primary objects within the graph. They represent different items, such as real-world objects, e.g., people. Relationships represent the connections between the different entities. They define the relations and, therefore, provide the structure of the Knowledge Graph. The semantic description contains the properties and characteristics of the different entities, e.g., the name and gender of a person [23]. Knowledge Graphs are generally used in scenarios, where very accurate and explicit knowledge is required and the relationship between the entities is crucial [24]. In an LLM context, they can enhance the understanding of the data, since they provide knowledge concerning inference and interpretability. The representation of relationships provides a reasonable and accessible understanding for humans and, therefore, provide a lot of value. Neo4j [25] is a NoSQL, open-source graph database that uses Knowledge Graphs for data representation. Graph databases are databases that utilize the concept of knowledge graphs in their data representation and storage [26]. The advantage of Neo4j is the ability to represent the relationships between the entities bidirectional.

2.6. Cypher

Cypher is a versatile query language that is used for property graphs. It is main capabilities are querying and modifying property graph databases. It is mainly used for its flexibility and a query interface that is similar to SQL. Cypher queries transform property graphs into tables that represent patterns in the graph. Cypher queries follow a linear structure, so that each clause functions as a step in the progression of the query. The core principals of CQL are pattern matching and graph navigation. In the pattern matching operation, patterns are expressed visually as "ASCII art", and represent a limited form of regular path queries [27]. The patterns are expressed syntactically, and encode the nodes and edges by using arrows between them. It is the evaluation of graph patterns concerning graph databases. While the graph patterns provide the querying of the graph databases, it is important to be able to navigate the topology of the data more flexibly in the graph navigation operation [28]. Cypher is utilized within Neo4j for the intuitive way of working with graph data. The pattern matching operation allows for efficient querying and aligns with the graph-based databases.

3. Related Work

In recent years, the field of Natural Language Interfaces (NLIs) for databases has emerged as a prominent area of interest, drawing substantial interest from both academic research and various industry sectors. Such NLIs allow users to convert natural language into, for example, structured queries, which can then be used to retrieve the desired information from databases and reduce the complexity posed by conventional query languages [29]. Therefore, an overview of the existing literature on the tasks "Text to SQL" and "Text to NoSQL" will be given below.

3.1. Text to SQL

The area of text-to-SQL systems is one of the most extensively researched areas in the field of natural language interfaces for databases. SQLNet [30] and TypeSQL [31] each use bidirectional Long Short Term Memory Models (LSTM) to process and encode input in plain text. Both methods follow a so-called sketch-based approach. In this concept, the information extracted from the encoded input is integrated into predefined SQL templates with the help of decoders. To decode and predict the specific content of individual SQL clauses, such as SELECT or WHERE, both methods use multiple LSTM or a pointer network. Yu et al. extend this idea with their SyntaxSQLNet [32] model. In the encoding step, they not only take into account the user query and the column information, but also integrate the current decoding history of the SQL query. To generate the query, they introduce a syntax tree decoder specifically designed for SQL which recursively determines which SQL clause needs to be predicted next, and activates the correspondingly assigned decoder. This method enables a more accurate prediction of SQL clauses, and is able to solve much more complex queries. Besides using LSTM, there are also different techniques to extract information from the input. Montgomery et al. [33] developed an approach in which a graph is created that represents the schema of the SQL database, which is used to simplify the mapping of user queries to SQL queries. In parallel, they use NLP techniques to extract relevant information from the natural language query. ER-SQL [34] uses the pre-trained BERT-Large model [35] as an input encoder for the questions, the table schema, and the table content. Both of these methods also follow the sketch-based concept, in which the extracted information is integrated into SQL templates for building the query. Recent papers have also presented models, like SEQ2SQL [36] or X-SQL [37], which do not require the use of tools such as SQL templates in the text-to-SQL area, and can directly convert natural language input into SQL queries. In addition, more and more new approaches have recently emerged that use advanced LLMs such as Codex [38] or OpenAI's GPT-3 [1], experimenting with various prompt strategies. These models completely take over the encoding and decoding of the input, enabling a direct and efficient conversion of natural language into structured SQL queries. Such approaches are characterized by high flexibility and adaptability to different environments, as they do not require specific templates or database schemas [39]. Beyond that, various test data sets have been developed for the text-to-SQL task, of which WikiSQL [36] and Spider [40] are among the most prominent. WikiSQL has established itself as the main dataset for researching text-to-SQL applications on a single table. Spider was developed to better represent the complexity of actual use cases. Unlike WikiSQL, where queries are limited to single tables, Spider allows the formulation of more sophisticated and multi-layered queries across multiple tables.

3.2. Text to NoSQL

In contrast to traditional SQL-based systems, the variety of database schemas and less strict query structures in NoSQL query languages pose a particular challenge for the development of NLI, as they require flexible and dynamic processing and interpretation of the natural language. Therefore, various approaches have been developed in the literature that use classical and modern techniques to deal with this challenge. Mondal et al. [41] developed a query–response model that can handle various queries, like assertive, interrogative, imperative, compound, and complex forms. Their system is essentially based on the extraction of entities and attributes from natural language using NLP techniques, from which the query can then be formed after semantic analysis. Pradeep et al. [42] and Hossen et al. [43] both propose deep learning-based approaches for generating queries. In the former, after the identification of the question type by a question detection module, the natural question is converted into the query by an encoder–decoder architecture. For each different question type, such a deep learning module was created. In the latter, the question is first pre-processed using NLP techniques. The information obtained from this is used in the next step to extract collections and attributes using the Levenshtein distance algorithm. A BERT-based module is then used to extract the operation from the natural question. Finally, all extracted information is used to map it with a syntax tree, which is then used to generate the query. All these approaches have been developed and tested to generate queries for the database MongoDB. In contrast, Zhang et al. [44] developed an NLI for querying elasticsearch databases. In their system, the user's question is first converted into a template question using a pre-trained transformer-based sequence-to-sequence model into a template question. This is then used by an encoder to generate the condition fields and values, which can then be inserted for the elasticsearch query templates.

3.3. Graph-Based Query Languages

Graph databases pose a particular challenge for the automatic generation of query statements in the area of NoSQL databases. The reason for this lies in the complex nature of graph-based query structures, where not only the entities and their attributes, but also the relationships between them, play a crucial role. In this area, a large majority of the scientific literature focuses on the SPARQL Protocol And RDF Query Language, SPARQL [45] for short, a query language specifically for querying and manipulating data in Resource Description Framework (RDF) formats, which is often used in graph-based databases. Agahei et al. [46] proposed an NLI, which follows a sketch-based approach. In their model, the natural question is first processed using NLP techniques and label encoding, and then the question is classified into its corresponding query pattern, which is used as a template. Subsequently, the corresponding information is extracted from the question using entity linking and relation extraction, which can then be inserted into the selected SPARQL template. Liang et al. [47] created a model, which first classifies the user's query for syntactic information to determine the type of query. The information obtained from this is used to build the SELECT clause of a SPARQL query. Subsequently, the phrase mapping approach uses several models in the form of an ensemble method to extract the resources, properties, and classes of the RDF schema from the question. Finally, all possible queries are created with this information, and the most plausible are determined using a tree-LSTM for ranking. Some papers propose the use of transformer-based [48] or seq2seq-based [49] architectures for SPARQL query generation. These methods do not require any pre-processing with NLP techniques or auxiliary structures such as query templates. Due to the advance of LLMs, novel approaches analogous to the text to SQL task were recently introduced, which test OpenAI's language models GPT-3 [1], GPT-3.5-turbo, and GPT-4 [50] with different prompt strategies to generate SPARQL queries [51,52]. In addition, a large number of test data sets have already been designed for the text to SPARQL task, of which the LC-QuAD [53] and QALD [54] series are the most popular [55].

In contrast, the literature for NLIs to generate Cipher Query Language (CQL) statements for the Neo4j database is rather sparse, and it is not much researched yet. Several papers use NLP techniques to process the question in natural language. Hains et al. [56] presented a system that uses NLP to process the user's question, then maps it with question type dependent patterns and extracts labels and variables to generate the CQL statement. However, they only presented the concept of how this could be performed, but not an implementation of the system. Kobeissi et al. [57] created a NLI for querying process execution data with Cypher. Their system consists of two main components, namely Intent Detection/Entity Extraction and Query Construction. In the former, generic intent patterns are used to determine the type of question from which the form of the MATCH and RETURN clauses of the Cypher query. In addition, named entity recognition is used to derive entity-tag sets from the question. This information is then used to construct the CQL statement in the query construction module, starting with the MATCH clause and proceeding algorithmically. Zhou et al. [58] pursued a sketch-based approach to create Cypher queries for SCD files. Here, the user question is supplemented with additional information using the Knowledge Graph and synonyms are replaced. Semantic triplets are then generated from the question using a Bi-LSTM, which are then converted into Cypher code segments and used to select the predefined assembly template. Finally, the finished CQL statement is constructed from the code snippets and template. Chatterjee et al. [59] introduce an NLI for querying wind turbine maintenance data in Neo4j graph databases. In their work, they create both a seq2seq model extended with an attention mechanism and a transformer-based model to create Cypher queries. The advantage of these models is that the natural language input does not have to be pre-processed and can be converted directly into CQL statements without further auxiliary structures. They showed that the transformer based model is slightly more accurate in generating Cypher queries, while being almost 10 times more computationally efficient. With regard to the use of LLMs to generate Cypher queries, Feng et al. [60] are the only paper found in this literature search to have taken such an approach. In their system, a named entity recognition is first performed on the user question in which the entities from the question are mapped to nodes in the graph. The results of this and the original question are then incorporated into several prompts which are used by ChatGPT to generate the CQL statement. In addition, they also tested different prompt strategies, as well as an improvement of the query by ChatGPT in case of error/empty result. However, the ChatGPT version used was not mentioned in the paper.

4. Concept

As the existing literature attests, text-to-SQL systems have been thoroughly researched in the context of Natural Language Interfaces (NLIs) for databases. Shifting the focus to NoSQL databases, though, reveals more significant challenges due to their complexity. Query generation in graph databases is particularly challenging. Most of the literature focuses on SPARQL as a query language. Conversely, the Cypher Query Language (CQL), which is used to query graph databases in Neo4j, has been relatively less explored, both in terms of breadth and depth. To fill this research gap and develop an appropriate artifact for this topic, the Design Science Research Methodology (DSRM) as outlined by March and Smith [61] and further developed by Peffers et al. [62], was applied. This approach is particularly appropriate, as it allows a specific organizational problem to be addressed through the development of a meaningful IT artifact [63]. By applying DSRM, the research focuses on creating a solution that not only addresses the technical aspects of querying graph databases using CQL, but also considers the usability and practicality of the solution in organizational settings. The development of such an artifact, adapted to the complexities of NoSQL databases and Neo4j in particular, aims to make a significant contribution to the field by improving the capabilities and ease of use for practitioners and researchers alike.

4.1. Problem Identification and Motivation

In accordance with this methodology, the first step was to identify the problems and motivation for this research contribution. These were primarily outlined in the Introduction and Related Work sections. It is apparent that there is a substantial need for research in the area of Cypher Query Language for querying graph databases in Neo4j, whereas the most significant research gap can be identified in generating these CQL queries with LLMs. This need is critical not only for driving research in this area, but also for making the query capabilities of different databases more accessible.

4.2. Objectives of a Solution

In line with this idea, the goal is to develop an application that takes a modular approach to integrating and querying different databases in natural language, with a particular focus on querying using CQL in Neo4j. On a more granular level, the following objectives of the solution are of particular importance. These are derived from the existing literature and our own assessment. The primary objective of this research is to enhance the natural language interface. The aim is to develop a system that can seamlessly translate natural language queries into CQL without any intermediate substructures. This advancement will allow users to interact with Neo4j graph databases using conversational language, making the process of accessing data both intuitive and user-friendly. In further stages of this approach, this will be a key aspect to make graph database technology more accessible to a wider audience, including those without technical expertise in database querying. Another important goal is to improve the accuracy of these CQL queries. It is important that the queries generated from natural language are correct. If this is not the case at the first attempt, the solution includes mechanisms to correct the query and proceed with optimizing the Cypher queries. In addition, there is a focus on facilitating complex data retrieval. The system will be designed to handle more complex queries, allowing users to access and retrieve data relationships stored in graph databases. Ensuring scalability and performance is also a key consideration. The system should be able to manage different types of datasets and graph schemas while maintaining high performance in data processing and query execution. In addition, the research aims to promote integration and compatibility. The proposed solution will be designed to easily integrate with existing systems and be compatible with various databases and large language models, ensuring its adaptability and long-term utility. Lastly, a user-friendly interface is another crucial aspect of our project. The success of the model depends on its ease of use, underlining the need for an interface that simplifies interactions and enhances the overall user experience.

4.3. Design Principles

Each of the aforementioned goals has been developed to bridge the gap between advanced database technology and user-friendly interfaces to enhance the field of natural language processing for database queries. Derived from these goals, the following design principles can now be derived, which were adopted in the development of the technical artifact.

Consistent approach: The design of our artifact is driven by a consistent approach, ensuring uniformity in functionality across different modules and aspects of the application. This consistency is essential for providing a seamless user experience, making the transition between different types of queries and databases intuitive and straightforward. By maintaining a consistent approach, users can expect predictable outcomes and interactions, which is crucial for building user trust and proficiency with the application.

Accurate query generation: Another fundamental aspect of our solution is its capability to generate correct queries from natural language inputs. This involves advanced processing models that can accurately interpret the user's intent and translate it into valid CQL queries. The system is designed to understand different verbal constructs and convert them into corresponding database operations, ensuring that the results match the user's expectations.

Robustness: This marks another core principle guiding our design. The artifact is constructed to handle a wide range of queries reliably, maintaining performance and accuracy even under varying conditions. This robustness includes the ability to manage complex queries, interpret nuances in natural language, and provide results consistently. Additionally, the system is designed to be error tolerant, offering clear feedback to users to correct issues or refine their queries.

Appropriate database selection and correct reference to chat history: The artifact intelligently selects the appropriate database based on the query context and user requirements. It includes a mechanism to understand the context within which a query is made, including references to chat history. This feature allows the system to provide more accurate and contextually relevant responses by understanding the user's interactions and the nature of their queries. The ability to reference and utilize chat history improves the system's ability to handle repetitive, multi-step interactions and thus improve performance.

4.4. Workflow Design

Derived from the objectives and design principles developed above, Figure 1 was created. The workflow was created to explain the sequential operations from user input to final response delivery. This section examines each step of the Business Process Model and Notation (BPMN) model, providing a detailed understanding of the decision making processes and interaction with the databases. The process is initiated when a user enters a question. This is the primary interaction between the human and system, and triggers the chatbot's response mechanism. The chatbot internally generates a prompt that encapsulates the user's query, preparing the system for subsequent analysis and response. The system evaluates the chat history to determine if a similar query has been previously addressed, which optimizes the response time by avoiding redundant database queries. If the system finds a relevant instance in the chat history, the chatbot constructs an answer in natural text form, ready to be presented to the user. If the chat history does not provide a satisfactory response, the system moves on to assess the feasibility of a database query with available databases. This decision is crucial, since it involves choosing the correct database for executing queries. If the underlying databases are not sufficient to answer the questions, an "Unsuccessful Response" message is generated. However, if it is possible to answer with the underlying database, the respective database schema for constructing an accurate and efficient query is retrieved. Using the user's prompt and the retrieved database schema, the chatbot proceeds to generate a structured query. This query is tailored to retrieve the relevant information from the database in response to the user's initial question. If the query execution fails, the chatbot generates an error prompt and reproduces improved queries to retry with optimized Cypher queries. Upon a successful query execution, the chatbot generates an answer from the query result. This answer is then formatted into a natural text response that can be easily understood by the user. In case the chatbot cannot generate a successful response from either the chat history or the database query, it generates an unsuccessful response. The final step in the chatbot's operational process is presenting the response to the user. Whether the response is a direct answer, an error message, or a notification of an unsuccessful query, the system communicates the outcome clearly and effectively to the user, maintaining transparency in the interaction.

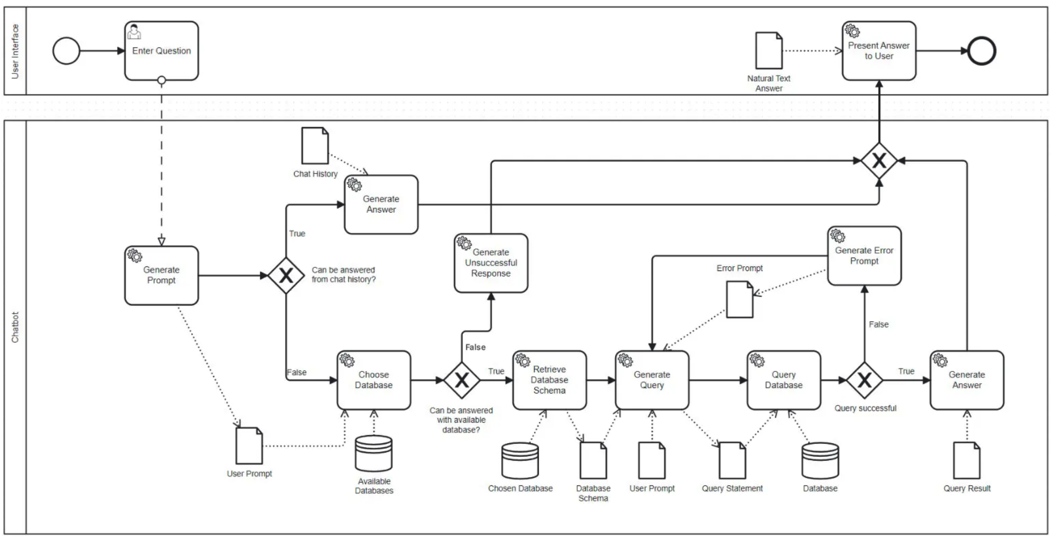


Figure 1. Artifact: workflow of the chatbot.

4.5. Technical Realization and Implementation

The following section marks a shift from the conceptual framework to the technical realization and implementation of the chatbot. It details the architecture, programming languages, and database connections that are integral to the construction of the chatbot, focusing on the design choices made to solve the identified problems.

4.6. Evaluation

Following the technical exposition, an evaluation phase will assess the chatbot's performance against pre-defined objectives. Metrics such as execution accuracy, response time and syntax errors will be central to this analysis.

4.7. Discussion and Limitations

The discussion will then place these findings in wider context of existing research, highlighting the implications of the study. This examination will serve to review new insights and limitations of the solution created.

4.8. Conclusion and Future Work

In the concluding section, the research will present a synthesis of the findings and suggest areas for future research. The conclusion will summarize the significance of the findings while acknowledging the scope and limitations of the study. The proposed future research directions will build on the reflective findings and suggest modifications and improvements for the next possible iterations of the chatbot.

5. Implementation

The NLI built as part of this project was designed on the basis of modular architecture. This approach allows for easier testability, maintainability of the individual components and, above all, expandability for additional databases or language models. Figure 2 shows the schematic architecture of the chatbot. The entire chatbot was implemented in the Python programming language. The whole system is hosted in docker [64] containers, and consists of three main pipelines, namely the *Agent*, *Chat_from_History*, and *QA*, which are responsible for processing the user's request and generating a suitable response. In the following, the implementation of the individual modules and the processing of user questions in them will be described.

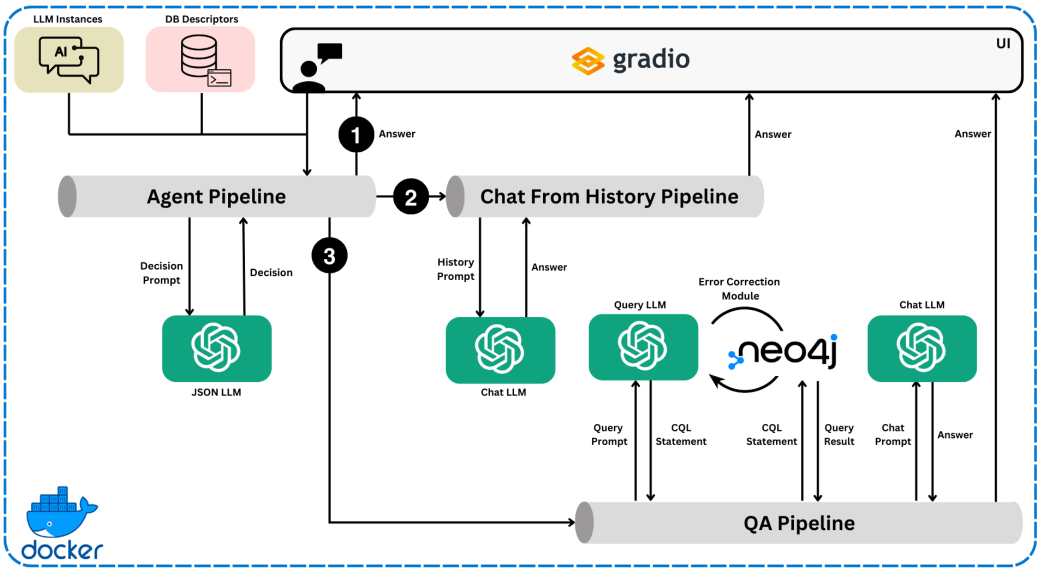


Figure 2. Schematic architecture of the chatbot.

At the start of the chatbot, the user interface, database descriptors, and language models are initialized first. For the creation of the user interface (UI), the open source python library gradio [65] was used, which provides a variety of components to build a UI for a chatbot quickly and easily.

The database descriptors Listing 1 contain the name of the available databases, as well as a brief description of the information they contain. This is necessary in order to be able to select the appropriate database for the user's question later in the agent pipeline. The database adapter contains the connection information and a client to the respective database. The required large language models are also initialized at the same time. Three independent LLMs, i.e., *Json LLM*, *Query LLM* and *Chat LLM*, are required for the chatbot. Due to hardware limitations, it was decided to use the latest version of Open AI's GPT-4-Turbo for each of the models. Table 1 shows an overview of the models and their parameters. Regarding the temperature parameter, it is particularly important for the first two models to be set to 0.0 so that the output is as focused and deterministic as possible.

Listing 1. Database Descriptors.

```
DATA_BASE_DESCRIPTORS = [
    DataBaseDescriptor(
        ''MovieDatabase'', ''A neo4j database containing information about
     movies, actors and directors'', MOVIE_DB_ADAPTER
    ),
]
```

Table 1. Used Large Language Models.

Parameters	Models		
	Json LLM	**Query LLM**	**Chat LLM**
model	gpt-4-1106-preview		
temperature	0.0	0.0	0.2
max_tokens	50	100	200
response_format	type: json_object	n.s.	n.s.

The reason behind this is that when making decisions or generating CQL statements, it is fundamentally important that the model only uses the nodes and parameters that are given to it in the prompt, and does not invent its own values here, as these would then lead to an incorrect decision/cipher query. For the *Chat LLM*, this parameter was set slightly higher in order to prevent the model to include its internal knowledge into the answer. In the latest GPT-4-Turbo models, a so-called response_format can also be defined. This allows the JSON mode [66] to be activated for a model, which is used in the *Json LLM*. Its functionality will be explained in more detail in the Agent Pipeline section. For the other two models, the default parameter of the response_format were applied.

5.1. Agent Pipeline

When the user enters a question, it is sent to the *Agent pipeline* together with the database descriptors, LLM instances and the current chat history. The task of this pipeline is first to decide whether the question posed by the user can be answered with the available resources. For this purpose, the special decision prompt, Listing 2, is created for the user question. This contains the available databases with their description from the database descriptors, the user question, the current chat history and an answer schema. In this, the response options for the expected answer can be defined in advance. In the first field, "database", the names of the available databases and the option "None" are given to the model as a selection option. It is important to note here that the model can only make one selection in the current implementation.

Listing 2. Decision Prompt.

```
''Decide if you can answer the question only with the information of the
    chat history and if not which Database should be used to answer the
    question.
Important: You must answer in JSON format!
You have the follwing Databases available:
{available_db_prompt}
The question is:{question}
The current chat history is:
{history.format_for_model()}
Follow this example for the output:
{{
  database: Literal[{available_db_list}],
  can_answer_from_history: bool,
}}''
```

In the second field, the model should use a boolean value to indicate whether the question can only be answered with information from the specified chat history. Now, using the JSON mode of Open AI, the language model will follow exactly this schema in its response, and does not add any additional explanations to the answer, which makes it possible to parse it as a JSON object. Depending on the model's answer, three different paths can now be initiated for further processing of the user question: First, if the *Json LLM* has decided that the question cannot be answered neither from the previous chat history or from a available database, then the user will be shown an apology which is streamed into the UI an ends the process (1). Secondly, if the question can be answered from the previous chat history, it is sent together with the current history and the *Chat LLM* instance to the *Chat_from_History* pipeline (2). The third and last possible case is when the question cannot be answered by the chat history but by a query to an available database (3). Here, a information message about the selected database is streamed back to the UI, and subsequently, the question is sent together with the *Query LLM* instance and the selected database descriptor to the *QA pipeline*, where it is further processed. In the following, the last two options will be explained in more detail below.

5.2. *Chat from History Pipeline*

The purpose of this pipeline is to generate an answer to the question based only on the current chat history. A chat history contains three different types of messages: SYSTEM, USER and ASSISTANT. The system message is added at the beginning of each conversation and assigns a role to the language model. All questions created by the user are flagged as USER messages. Full text responses from the *Chat LLM*, database query results and also auxiliary information, such as which database was selected, are all categorized as ASSISTANT messages. To prevent irrelevant information such as the auxiliary information from being taken into account when answering on the basis of the history and thus possibly distorting the answer, a process parameter is added to each message. This is a boolean value which specifies whether a message should be included in such tasks or not. In order to be able to generate a full text answer with the *Chat LLM*, another customized prompt Listing 3 is used, which contains the current formatted chat history, as well as the user question.

Particularly important in this prompt is the request to the model that the information provided is authoritative. This ensures that it neither adds internal knowledge to its response that does not originate from the chat history, nor attempts to correct potentially incorrect statements. When the *Chat LLM* has finished generating the answer, it is streamed back to the UI and presented to the user and the process is complete.

Listing 3. Chat from History Prompt.

```
''You are an assistant that helps to form nice and human understandable
    answers.
The information part contains the current chat history that you must use to
    answer the user.
The provided information is authoritative, you must never doubt it or try to
     use your internal knowledge to correct it.
Make the answer sound as a response to the question. Do not mention that you
     based the result on the given information.
If the provided information is empty, say that you don't know the answer.
Information:
{formatted_chat_history}
Question: {question}
Helpful Answer:''
```

5.3. Query Answering (QA) Pipeline

The task of this pipeline is to answer the user question from the corresponding database. The first step in this process is to generate a suitable CQL statement to query the underlying Neo4j database. For this purpose, an additional customized prompt, Listing 4, was created, which will be used for the *Query LLM*. In addition to the user question, this contains the schema of the selected graph, which lists all nodes and relationships between them, as well as the properties of them both. This schema is dynamically retrieved from the graph database for prompt creation, using the connection details from the selected DB descriptor. Furthermore, example queries can also be added for few shot prompting, which the model can use as a guide for query generation. In addition to the temperature parameter of the *Query LLM* the prompt also includes requests to strictly adhere to the provided schema. This is to prevent the model from creating new nodes/relationships or properties that do not exist in the graph and thus causing errors in the database during execution. It is also required not to explain the generated cipher statement to ensure that the query can be parsed for the database without problems. After the CQL statement has been generated by the *Query LLM*, it is first streamed back to the UI and presented to the user before being executed via the database client of the database descriptor. Depending on the result of the request, two different processing paths are now taken. If the query could be executed on the database without errors, the raw query result is first streamed back to the UI in JSON format. This allows the user to check it and recognize possible errors in the plain text response. The query result is then transferred to the *Chat LLM*. A customized prompt was also created for this similar to the chat from history prompt, which contains the raw JSON and user question.

Listing 4. Generate Query Prompt.

```
''Task:Generate Cypher statement to query a graph database.
Instructions:
Use only the provided relationship types and properties in the schema.
Do not use any other relationship types or properties that are not provided.
Schema:{schema}
Hint: You can use the following queries as examples:
{self.few_shots}
Note: Do not include any explanations or apologies in your responses.
Do not include any text except the generated Cypher statement.
The question is:{question}''
```

The model is also asked to consider the result as authoritative and absolute and not to try to correct or explain it. This is to ensure that the generated full-text answers are concise, brief, and accurate to the initial question, and only contain information which can be found in the database. When the model has finished generating the answer text, it is streamed to the UI and presented to the user. This concludes the processing procedure. However, if the generated query has caused an error such as a syntax or semantic error during execution on the database, the processing path through the error correction module is taken. Due to the fact that the cipher language of Neo4j can be very error-prone, due to the extensive syntax, and a smooth query generation can therefore never be guaranteed, this module was created to automate the error correction in CQL queries as best as possible, and thus improve the user experience. The task of the module is to improve the initially generated query depending on the user question and the error message. In the event of an error, first an error correction prompt Listing 5 is created. Analogous to the generated query prompt, it contains the user query, the schema of the graph, and few shot examples.

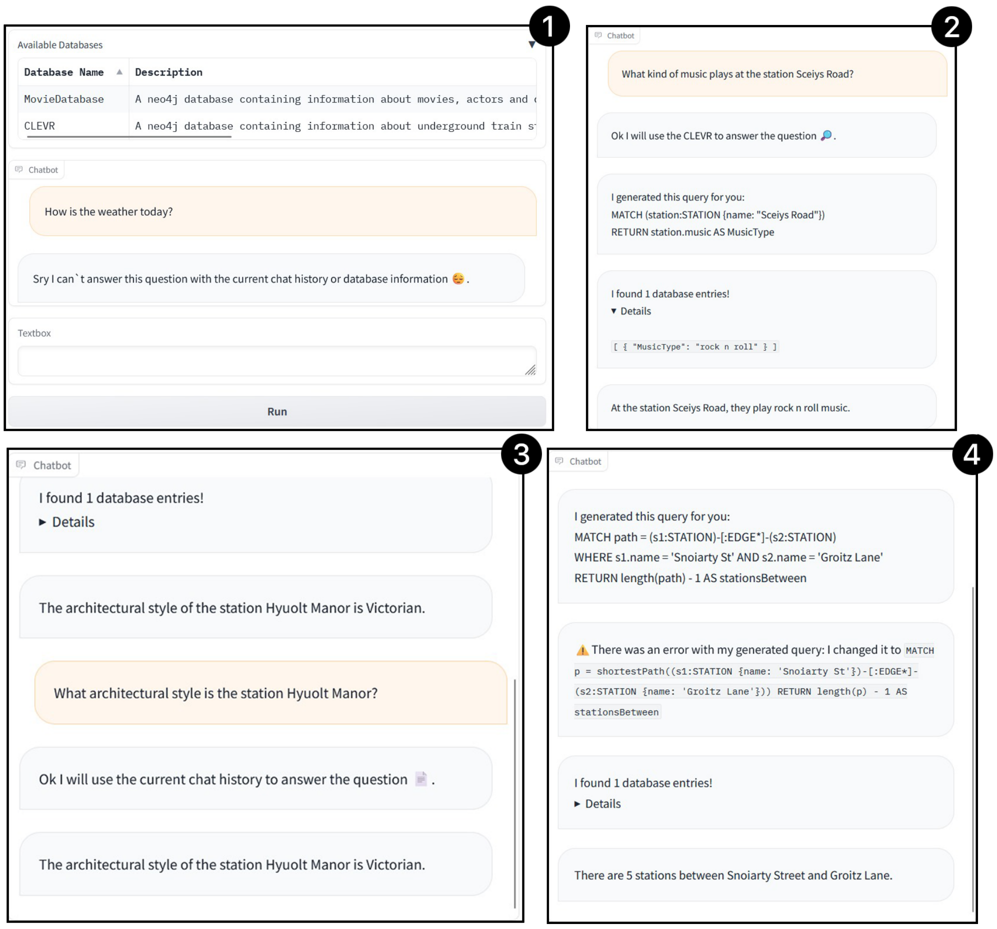
In addition, the incorrect CQL statement and the error that the Neo4j database has returned are also included. This is then sent to the *Query LLM*, which generates an improved query which is then executed a second time on the database. If the fixed CQL statement again causes an error, a new error correction prompt is created and the process starts again. As can be seen, the correction of the faulty query happens in a loop. To prevent this from potentially running for an infinitely long time, the maximum number of attempts was set to three in the implementation via a hardcoded parameter. When the maximum number of retries is reached, the module aborts the cycle and streams an error message back to the UI, asking the user to rephrase and resubmit their question. However, if the error could be resolved, or the revised cipher query no longer caused an exception, an information message is streamed back to the user in the UI. Following this, identical to error-free query generation, the database result is streamed back into the UI and passed to the *Chat LLM* to generate the full-text response. After this has also been streamed back to the UI, the processing of the request is finished. A major advantage in this approach is that the error correction module is error agnostic. This means that it is not necessary to define the possible error types and the corresponding reaction to them beforehand, but that the *Query LLM* decides independently how to deal with the error. Furthermore, this also ensures that each troubleshooting is customized to the query instead of using generic approaches. At the beginning of this section, it was emphasized that the chatbot was built in such a way that potentially several similar, but also different, databases can be used. In the current implementation, this is achieved by adapting the chatbot to a specific database type using only prompts, more precisely the query generation and error correction prompt. It must therefore be possible to exchange these two prompts flexibly depending on the database selected by the *Json LLM*. This is achieved by storing these prompts in the database client from the database descriptor. If, for example, a new SQL database is to be stored in the chatbot, its client must first be created programmatically, and thus also the two prompts. Then, depending on the selection of the database, the respective prompts and client connector are passed on to the *QA pipeline*, which can then be used to create suitable queries in the correct language. In this way, it is possible to use only one pipeline for the query generation of many different languages, which significantly improves maintainability and minimizes code overhead. In addition, the modular design of the pipelines makes it possible to easily add new pipelines or change the processing sequence.

Listing 5. Error Correction Prompt.

```
‘‘Task:Fix this Cypher statement to query a graph database.
Instructions:
Use only the provided relationship types and properties in the schema.
Do not use any other relationship types or properties that are not provided.
Schema: {schema}
Hint: You can use the following queries as examples:
{self.few_shots}
Note: Do not include any explanations or apologies in your responses.
Do not include any text except the generated Cypher statement.
The question that should be answered is:{question}
The generated Cypher statement is: ‘{query}‘
The error message is: ‘{error_message}‘
’’
```

5.4. Demonstration

This section provides a brief overview of the graphical user interface and application of the chatbot. Figure 3 shows the Gradio UI of the bot and the four possible communication scenarios. As the interface in 1 shows, the UI consists of three main components. In the center of the screen is the chat box for any located interaction with the bot. All user questions and any answers or help information from the bot are streamed into this. Above this, the available database descriptors are listed in a table with the data abbreviation and their short description. This should help the user to receive a quick overview of the available databases and their content. Below the chat is the input box, in which the user can ask and send questions to the bot. The first example shows the scenario in which the user asks a question that the chatbot cannot answer either from the chat history or with the help of a database. In the second exemplary chat process, an answer to the user's question is provided by a query to the database. Here, it can be observed as to how the chatbot first tells the user which database it uses to answer the question. It then presents the generated query and, after it has been executed, the number of results it has found. In this chat message, a dropdown is also added, which contains the raw JSON result of the database query. The last message of the bot displays the full text answer to the question. The third scenario shows the answering of a user request with the help of the chat history. For better visualization, in this simplified example, the same question was asked twice in a row, but in reality it would also be possible for the chatbot to refer to messages that are longer in the past. In this conversation, you can clearly see how the bot tells the user that it has decided to answer their question using history. It can also be seen here, that due to the rather low value of the temperature parameter of the *Chat LLM*, and the requirement in the prompt to adhere strictly to the history, the bot tries to deviate his answers as little as possible from the previous answers. The last scenario shows the error correction module in action. The user question asked in this example was "How many stations are between Snoiarty St and Groitz Lane ?" Analogous to regular CQL generation, the user is again presented with the initially created cipher statement. However, as a timeout error occurs on the database when this query is executed, it is fixed within the error correction module, and the user is informed of the error and presented with the revised version. In the event that the error in the query could not be resolved in the first attempt, this message is presented to the user for each new version, as long as the loop is running. When the error has been fixed, the raw database results in the dropdown and the full text answer will be displayed again, identical to the second scenario.



Database Name	Description
MovieDatabase	A neo4j database containing information about movies, actors and
CLEVR	A neo4j database containing information about underground train s

How is the weather today?

Sry I can`t answer this question with the current chat history or database information 😔.

What kind of music plays at the station Sceiys Road?

Ok I will use the CLEVR to answer the question 🔎.

I generated this query for you:
MATCH (station:STATION {name: "Sceiys Road"})
RETURN station.music AS MusicType

I found 1 database entries!
▾ Details

```
[ { "MusicType": "rock n roll" } ]
```

At the station Sceiys Road, they play rock n roll music.

I found 1 database entries!
▸ Details

The architectural style of the station Hyuolt Manor is Victorian.

What architectural style is the station Hyuolt Manor?

Ok I will use the current chat history to answer the question 📄.

The architectural style of the station Hyuolt Manor is Victorian.

I generated this query for you:
MATCH path = (s1:STATION)-[:EDGE*]-(s2:STATION)
WHERE s1.name = 'Snoiarty St' AND s2.name = 'Groitz Lane'
RETURN length(path) - 1 AS stationsBetween

⚠️ There was an error with my generated query: I changed it to `MATCH p = shortestPath((s1:STATION {name: 'Snoiarty St'})-[:EDGE*]-(s2:STATION {name: 'Groitz Lane'})) RETURN length(p) - 1 AS stationsBetween`

I found 1 database entries!
▸ Details

There are 5 stations between Snoiarty Street and Groitz Lane.

Figure 3. Communication scenarios in the chatbot.

6. Evaluation

For the evaluation of the chatbot, a suitable test data set for the text to CQL task was first required. In Section 3 of this work, it was already shown that, with Spider [40] and WikiSQL [36], established and famous data sets for text to SQL exist. In addition, there are also widely used data sets for the text to SPARQL task in the area of graph databases with LC-QuAD [53] and QALD [54] series. Considering this, a suitable test data set for the generation of cipher queries was also searched for in order to evaluate the translation capabilities of the chatbot. However, after an extensive search, it was discovered that only a few exist or have been made publicly available. A total of four data sets were found: Guo et al. [67] created a Chinese dataset with 10,000 native language cipher pairs as part of their work. While these were published on GitHub [68], the underlying knowledge graph needed for query generation is only available for download on the Chinese website Baidu, which can only be accessed from selected countries which currently does not include Germany. Chatterjee et al. [59] and Kobeissi et al. [57] created test data sets for their respective use cases in the area of maintenance information on wind turbines [69] and process execution data [70], respectively. Although the question–CQL pairs and the available graph database are publicly available, they could not be used for evaluation. The reason for this is that the schemas of the graphs are rather complex and, therefore, very extensive, which would result in large query prompts. Since each of these contains the schema of the graph and properties of the nodes/connections, the resulting costs for the Open AI API would significantly exceed the cost constraints of this work. The fourth test data set found was CLEVR [71]. The underlying graph behind CLEVR simulates an artificial subway network inspired by the London tube and train network. The included nodes and relationships have been expanded to include a variety of properties such as cleanliness or music played, which can be queried. The repository contains scripts for generating a random graph and the corresponding test dataset. Due to the relatively lightweight nature of the graph, which nonetheless allows complex queries, it was decided to use this dataset to evaluate the text to CQL capabilities of the chatbot. In addition to the generation of Cypher statements, other capabilities of the chatbot, such as database selection and response from history, were also evaluated. Since there are no test data sets for such tasks, these were self-constructed on the basis of the available graphs. In total, three different Knowledge Graphs were used across all parts of the evaluation, the characteristics of which are briefly presented in Table 2. The Movie Graph contains information about movies and people who were in certain relationships to them, for example actors or directors. Northwind represents a traditional retail system with products, orders, customers, suppliers and employees.

Table 2. Graphs used for evaluation.

Details	Graphs		
	Movie	**Northwind**	**CLEVR**
# Nodes	171	1.035	287
# Relationships	253	3.139	337
# Distinct Relation. types	6	4	1
# Distinct Properties	8	47	22

Both of these graphs are part of sample datasets provided directly by NEO4J [72]. In the following, the evaluation methods used are explained and the results presented.

For the evaluation of all experiments, the Exact Set Match Accuracy (EM) metric was utilized to ensure a consistent measurement of performance across different evaluation targets [73].

$$Score_{LF}(\hat{Y}, Y) = \begin{cases} 1, \hat{Y} = Y \\ 0, \hat{Y} \neq Y \end{cases}$$

The EM metric is calculated by comparing each predicted answer, denoted as Yhat, with the ground truth answer Y. This comparison is performed for all N instances within the respective dataset. If the predicted answer exactly matches the ground truth, it is considered correct. The final accuracy is then derived by taking the ratio of correctly matched instances to the total instances in the dataset. Given that each experiment in this study targeted different aspects of the chatbots capabilities, tailored question schemas were necessary to align with specific evaluation goals. These schemas were designed to accommodate the unique requirements of each experiment, ensuring that the evaluations accurately captured the performance of the chatbot in generating CQL, selecting the appropriate database, and providing responses based on conversational context. Descriptions of these question schemas, as well as further analysis of the EM metric results, are presented in the subsequent sections.

6.1. Database Decision Evaluation

The aim of this evaluation was to find out how accurate the chatbot is at selecting the right database to answer the user's question. For this, 282 questions and correct database selection pairs were created by hand, covering all databases from Table 2. In order to be able to evaluate whether the bot is also able to correctly recognize that it cannot answer a question with the available databases, the test data set also contains questions that are not related to one of the databases. Consequently, the correct choice for these questions was expected to be "None". Only the *Json LLM* was used for the evaluation, as only this part of the chatbot is responsible for the database selection.

The schema for database selection was structured as follows: "Given the [Database Descriptions] and [question], determine which database is best suited for the task." In the evaluation dataset, the ground truth answers were the names of the respective databases, serving as the benchmark for accuracy in database selection.

The options available for selection were "MovieDatabase, Northwind, CLEVR, None". For each of the databases, a short and concise description of their content was given in the corresponding descriptor. The selection accuracy is defined by the number of correctly selected databases divided by the sum of all questions. As can be seen in Table 3, the chatbot, or rather the *Json LLM*, is very good at selecting the correct database for the user's question, reaching a overall selection accuracy of 96.45%. The data set for CLEVR alone is a little less accurate in comparison. After closer examination, the reason behind this is that some of the questions in this test data set do not contain any keywords from the CLEVR database description, which prevents the model from finding an assignment to a descriptor. In addition, due to the fact that CLEVR was generated synthetically, none of the station names or subway lines have an equivalent in the real world that the LLM could reference from its internal knowledge. This is different in the movie dataset. Here, the model can conclude from its training data that, for example, the person "Keanu Reeves" is an actor and that a question containing this person can be answered with the help of the movie graph.

Table 3. Database decision evaluation result.

Selection Options	**Metrics**	
	# Questions	**Selection Accuracy (%)**
Movie	118	100
CLEVR	100	92
Northwind	30	100
None	34	100
$\sum$	282	96.45

6.2. *Chat from History Evaluation*

Further evaluation aimed to assess the ability of the *Json LLM* to effectively reuse information from previous interactions in the chat history. The chatbot was tested on a series of questions within the Movie and CLEVR databases, categorized into zero-step, one-step, two-step, and three-step reasoning questions, as shown in Table 4. Throughout the process, the ability to recognize the question in the chat history is evaluated, not necessarily the correctness of the answer, meaning the results can be either true or false. For evaluating the chatbots capability to reference prior interactions, the following schema used was: "Given the current [chat history] and [question], determine if the question can be answered solely using the chat history." Here, the ground truth answers were boolean values, indicating whether the required information was indeed present in the conversation history.

The zero-step reasoning describes the posing of an initial question without any historical context for that question, implying for the model that it would correctly output "false" in such cases. In one-step reasoning, the question selected at the start is asked once again and then checked to see if it was recognized in the existing history, and the history would be used to answer it. In this case, the "true" result would be appropriate. The two-step reasoning now merges the first question with the second question in the chat history and asks a composite question. It then checks if the chatbot recognizes this combined question from the chat history and correctly handles it. Again, "true" would be the proper return value. With increasing complexity, an analogous approach was applied to three-step reasoning. For each reasoning step, 214 questions were asked, resulting in a total of 856 questions.

Table 4. Chat from history evaluation result.

Sel. Options	**Metrics**	
	# Questions	**Sel. Accuracy (%)**
0-Step Reasoning	214	100.00
1-Step Reasoning	214	82.24
2-Step Reasoning	214	79.44
3-Step Reasoning	214	87.38
$\sum$	856	87.27

The evaluation revealed varying patterns in response accuracy. The results revealed the model's ability to perform zero-step reasoning questions with 100% accuracy, demonstrating flawless decisions when there is no prior input. In contrast, one-step reasoning questions showed a decline in accuracy, with the model correctly using chat history 82.24% of the time. This trend continued with two-step reasoning questions, where the model's accuracy decreased to 79.44%. However, an interesting pattern appeared in the three-step reasoning category, where accuracy increased to 87.38%. This indicates that although the model struggles with intermediate complexity, it is better at handling more complex referencing tasks, which likely involve more robust integration of contextual information.

6.3. CQL Query Generation Evaluation

The evaluation further investigated the effectiveness of CQL statement generation of the model using the CLEVR dataset, as shown in Table 5. The schema for CQL generation was framed as: "Given the selected [graph schema] and [question], generate a Cypher query". This schema required the chatbot to formulate accurate CQL queries based on the graph's structure and the specified query intent. The test data set contains an English question, a so-called "gold query", which generates the correct query result and the raw query result. Thereby, two distinct setups were compared, zero-shot prompting, where the system generates queries without prior context, and few-shot prompting, where the system utilizes a small number of examples, respectively 4, to support its query generation. In order to evaluate the ability of the *Query LLM* to generate database queries from natural language input, both the raw query results of the generated query and the full text responses were considered. In this approach, we checked both whether the raw database results matched and whether the generated full-text response was correct. The reason for this is that some test questions from CLEVR are aimed at yes/no answers, so the result set of the generated statement may be empty. In this case, the answer would be wrong if our chatbot answered "it does not know the answer". The so-called execution accuracy, presented by Guo et al. [67], was used as a suitable evaluation metric. This describes how many of the generated CQL statements produce a correct query result in relation to all generated queries. A conscious decision was made not to use metrics such as logical accuracy, which check whether the generated CQL query is identical to the gold query, because the evaluation was about how well the system is able to generate statements that produce correct answers and not identical CQL statements. The evaluation was performed using 500 questions per prompt type and syntax and semantic errors, respectively, timeout errors, were also recorded as indicators of the system's proficiency.

Table 5. CQL query generation evaluation result.

Metrics	**Selection Option**	
	0-Shot Prompt.	**Few-Shot Prompt.**
Sel. Accuracy (%)	61.00	92.80
Syntax Errors	48	0
Semantic Errors	20	0
# Questions	500	500

The findings revealed a notable discrepancy in the agent's performance between the zero-shot and few-shot prompting. In the zero-shot scenario, where the chatbot was required to generate queries without prior examples, the execution accuracy stood at 61%. However, the model's accuracy improved significantly in the few-shot context, reaching 92.8%. This indicates the chatbot's ability to learn and adapt from examples, improving its query generation capabilities. In particular, the few-shot approach eliminated syntax and semantic errors, with zero instances detected, while the zero-shot approach produced 48 syntax errors and 20 semantic errors.

6.4. Performance Evaluation

To ensure practical application in real-world scenarios and to identify any potential bottlenecks in the system, the performance times during the execution of all evaluation tasks have been measured (see Table 6). To evaluate the performance of our approach, we use the average duration (in seconds) of all observations as the primary performance measure. Mathematically, the performance measure is defined as

$$\text{Performance Measure}(\bar{x}) = \frac{1}{n}\sum_{i=1}^{n} x_i$$

where x_i represents the duration (in seconds) for the i-th observation, and n is the total number of observations. This was gathered simultaneously during the previous evaluation techniques in the previous sections, starting with the database selection times for each database, including Movie, CLEVR, Northwind, and None if no applicable database is the correct reference for the model. Then, the chat history evaluation times were measured with increasing complexity from zero-step reasoning to three-step reasoning. Lastly, the times of zero-shot and few-shot were compared, differentiating between query generation time, query execution time, and answer generation time. Indeed, the evaluation provided differentiated insights into the performance of our system. As for the database selection task, the CLEVR dataset revealed slightly higher durations compared to the other datasets, which is consistent with the previous findings of slightly lower accuracy. When examining the reasoning tasks, we observed a trend where increased context led to slightly longer durations.

However, the model showed stable performance in scenarios with extended context lengths. The three-step reasoning trials showed that the model could maintain efficient processing times despite the added complexity. During zero-shot prompting, query generation was the most time-consuming process, confirming the general assumption, that the initial context setup requires a high amount of resources. In contrast, query execution was performed in near real-time, demonstrating the model's efficiency in translating queries into database actions. Notably, performance time improvements were observed, as expected, in the few-shot query scenarios. The model used previous examples to optimize query generation, resulting in a reduced duration for subsequent tasks.

Table 6. Performance evaluation metrics.

	$\bar{x}$ **Duration (in sec.)**	n
Database Decision		
Movie	1.79	118
CLEVR	2.21	100
Northwind	1.74	30
None	1.53	34
History Evaluation		
0-Step Reasoning	1.56	214
1-Step Reasoning	1.52	214
2-Step Reasoning	1.52	214
3-Step Reasoning	1.69	214
0-Shot Prompting		
Query Generation	3.29	500
Query Execution	0.02	500
Answer Generation	1.72	500
Few-shot Prompting		
Query Generation	3.01	500
Query Execution	0.01	500
Answer Generation	1.34	500

7. Discussion and Limitations

In this paper, an innovative approach for the creation and execution of Cypher queries for Neo4j by a chatbot was presented, which is characterized by selecting the appropriate one to answer a question from several predefined databases and recognizing when a question can only be answered from the current chat history. However, this approach involved both technical and conceptual challenges, which are discussed below. One of the primary challenges was the problem of hallucination, where the chatbot generates queries with incorrect syntax or properties. The results show that 24.62% of all incorrectly generated queries in a zero-shot setting are attributed to syntax errors. This underlines the need for continuous refinement of the model's understanding and generation capabilities, especially in the context of a specialized and extensive query language, such as CQL. Furthermore, this also emphasizes the importance of developing error checking mechanisms, as implemented in this work, to identify and correct syntax errors prior to query execution. Considering this, the evaluation results also show the compelling ability of GPT-4-turbo to learn from example queries. With few shot prompting, we were able to increase the execution accuracy from 61% to 92.8%, with a complete elimination of syntax and semantic errors. This improvement highlights the effectiveness of few-shot learning in increasing the precision and reliability of the model in generating Cypher queries. However, the current implementation of the chatbot also raises important privacy and security considerations. Given the potential of the chatbot to access all information contained in the graph, which may contain sensitive information, the current implementation should not be used to handle private data that cannot be disclosed to Open AI or in general. For this purpose, alternative Large Language Models, such as Mixtral 8x7B [74] or Code Llama [75], should be considered which are publicly available and can, therefore, be executed on proprietary hardware for which suitable data protection measures can be taken. Additionally, the chatbot's ability to generate not only selection, but also deletion and creation queries requires strict countermeasures to prevent unauthorized or unwanted database modifications. For this, different approaches can be taken depending on the database technology used. Neo4j, for example, allows to set the access rights of a database client read-only, which prevents any modifying transactions and generates a corresponding error. This solution was chosen in our implementation. If such setting options are not available, another possibility would be the definition of keywords according to which the generated queries are checked and, if present, rejected. At the beginning of this project, it was also considered to use the Python framework Langchain for implementation. With its Neo4j DB QA Chain [76], this already offers pre-built functionalities for generating and executing CQL on graph databases. Ultimately, however, the decision was made not to utilize this library, as there would have been problems particularly with the dynamic generation of prompts, on which our system is primarily based. The reason for this is that it would have been necessary to extend Langchain's pre-implemented prompts, for which they are not intended. Consequently, this would have led to a continuous attempt to adapt the program logic to Langchain, which would have required more effort than programming it ourselves. Finally, it is also necessary to note the limitations of the test dataset used for CQL generation. The current dataset, based on the CLEVR framework, currently supports 33 different question types, which are extended by permutation and substitution techniques. While this approach demonstrates the capabilities of the chatbot within a limited set of queries, it does not reach the complexity and diversity found in state-of-the-art datasets such as Spider [40], WikiSQL [36], or the LC-QuAD/QUALD series [53,54]. In order to fully validate and benchmark the performance of our system, due to the lack of state-of-the-art datasets in the text to Cypher domains, the development of a new, more comprehensive evaluation dataset that reflects the diversity and complexity of real-world database query scenarios is essential. At the time of writing, initiatives from the neo4j community are already underway to fill this gap [77].

8. Conclusions and Future Work

In this paper, we propose a way to integrate NoSQL and Knowledge Graphs into Large Language Models. The paper enriches the field of natural language interfaces by applying the design science research methodology to develop a chatbot that can answer user queries by generating CQL queries. It also provides an extension that allows the bot to select the appropriate database or chat history to answer the question. Through a comprehensive evaluation, we were able to show that the chatbot is reliable and accurate in generating Cypher statements, as well as making the right decision regarding the database and chat history. We also provided an overview about the necessary theoretical foundations and the related work concerning literature in the tasks "Text to SQL" and "Text to NoSQL". The further development of the chatbot was already considered during implementation. Although only graph databases were used as a knowledge base in this paper, the modular architecture of the system was developed in such a way that it is potentially possible to support both multiple and different storage technologies simultaneously, such as relational or document-oriented databases. Furthermore, it was also ensured that it is possible to exchange the Large Language Models in the back-end. Since the processing logic of the system was designed independently of the underlying database technology or LLM, it is possible to extend the chatbot by defining the appropriate adapters that are fed into the pipelines without having to change the logic in the rest of the system. In this way, the chatbot should form a basis for meeting diverse user requirements in terms of database technology and language models, with the aim of achieving a polyglot persistent system.

Future work could focus on several promising directions to extend the capabilities and flexibility of the chatbot. In this study, only closed-source foundational models were used to support the workflow. However, a valuable area for further research lies in fine-tuning open-source models on text-to-Cypher datasets, allowing a comparison of these models performance with the closed-source results achieved in this paper. Adaptive prompt tuning could also significantly enhance the chatbot's performance. By storing corrected Cypher queries or specific user questions in a vector database, the chatbot could dynamically reference these stored examples when faced with similar queries in the future, reducing the likelihood of repeated mistakes. The use of DSPy, an open-source framework that facilitates prompt optimization through programmatically defined modules, could further streamline this adaptive tuning process. By treating language model interactions as structured modules, DSPy enables a more systematic prompt optimization framework and reduces reliance on manually crafted prompts [78]. Another promising research direction involves expanding the chatbot's functionality to clarify ambiguous user questions. When a query lacks specificity, an LLM could engage users with clarifying questions to refine their input, ensuring responses that are accurate and contextually relevant. Frameworks like LangGraph could be used to structure these interactions, enabling iterative dialogues to better capture user intent. This capability would reduce misunderstandings and improve answer quality, creating a more responsive, user-centered experience and advancing the field of natural language interfaces.

Author Contributions: Conceptualization, M.H., M.K., C.S. and F.E.; Funding acquisition, S.S.; Methodology, M.H., M.K., C.S. and F.E.; Project administration, S.S.; Supervision, P.E. and S.S.; Writing—original draft, M.H., M.K., C.S. and F.E.; Writing—review and editing, M.H., P.E. and S.S. All authors have read and agreed to the published version of the manuscript.

Funding: This work is funded by the "Bavarian Ministry of Economic Affairs, Regional Development and Energy" within the project Security Iiot pRocEss Notation (SIREN).

Data Availability Statement: The original data presented in the study are openly available at: https://drive.google.com/file/d/1fJVcK5A3F8BIBVm9MNvfn-l-rJS6XsaS/view (accessed on 1 September 2024), Google Drive. The developed artifact is also openly available at: https://github.com/mogelkill/text2cypher (accessed on 1 September 2024), GitHub. The evaluation results are also openly available at: https://github.com/mogelkill/text2cypher/tree/main/evaluation_results (accessed on 1 September 2024), GitHub.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language Models are Few-Shot Learners. *arXiv* **2020**, arXiv:2005.14165.
2. DB-Engines. Graph DBMS Ranking. Available online: https://db-engines.com/en/ranking/graph+dbms (accessed on 21 November 2024).
3. Corbellini, A.; Mateos, C.; Zunino, A.; Godoy, D.; Schiaffino, S.N. Persisting big-data: The NoSQL landscape. *Inf. Syst.* **2017**, *63*, 1–23. [CrossRef]
4. Hogan, A.; Blomqvist, E.; Cochez, M.; D'amato, C.; Melo, G.D.; Gutierrez, C.; Kirrane, S.; Gayo, J.E.L.; Navigli, R.; Neumaier, S.; et al. Knowledge Graphs. *ACM Comput. Surv.* **2021**, *54*, 1–37. [CrossRef]
5. Becker, R.; Eick, S.; Wilks, A. Visualizing network data. *IEEE Trans. Vis. Comput. Graph.* **1995**, *1*, 16–28. [CrossRef]
6. OpenAI. GPT-4 and GPT-4 Turbo Documentation. Available online: https://platform.openai.com/docs/models/gpt-4-and-gpt-4-turbo (accessed on 21 November 2024).
7. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, L.; Polosukhin, I. Attention Is All You Need. In Proceedings of the 31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA, 4–9 December 2017.
8. Thirunavukarasu, A.J.; Ting, D.S.J.; Elangovan, K.; Gutierrez, L.; Tan, T.F.; Ting, D.S.W. Large Language Models in Medicine. *Nat. Med.* **2023**, *29*, 1930–1940. [CrossRef]
9. Zhao, W.X.; Zhou, K.; Li, J.; Tang, T.; Wang, X.; Hou, Y.; Min, Y.; Zhang, B.; Zhang, J.; Dong, Z.; et al. A Survey of Large Language Models. *arXiv* **2023**, arXiv:2303.18223.
10. Kulkarni, P.; Mahabaleshwarkar, A.; Kulkarni, M.; Sirsikar, N.; Gadgil, K. Conversational AI: An Overview of Methodologies, Applications & Future Scope. In Proceedings of the 2019 5th International Conference On Computing, Communication, Control And Automation (ICCUBEA), Pune, India, 19–21 September 2019; pp. 1–7. [CrossRef]
11. Dong, Q.; Li, L.; Dai, D.; Zheng, C.; Wu, Z.; Chang, B.; Sun, X.; Xu, J.; Li, L.; Sui, Z. A Survey on In-context Learning. *arXiv* **2022**, arXiv:2301.00234.
12. Amyeen, R. Prompt-Engineering and Transformer-based Question Generation and Evaluation. *arXiv* **2023**, arXiv:2310.18867.
13. Sorensen, T.; Robinson, J.; Rytting, C.M.; Shaw, A.G.; Rogers, K.J.; Delorey, A.P.; Khalil, M.; Fulda, N.; Wingate, D. An Information-theoretic Approach to Prompt Engineering Without Ground Truth Labels. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics, Dublin, Ireland, 22–27 May 2022; Volume 1, pp. 819–862. [CrossRef]
14. Zhu, W.; Tan, M. Improving Prompt Tuning with Learned Prompting Layers. *arXiv* **2023**, arXiv:2310.20127.
15. Lester, B.; Al-Rfou, R.; Constant, N. The Power of Scale for Parameter-Efficient Prompt Tuning. *arXiv* **2021**, arXiv:2104.08691.
16. Zhao, T.Z.; Wallace, E.; Feng, S.; Klein, D.; Singh, S. Calibrate Before Use: Improving Few-Shot Performance of Language Models. In *International Conference on Machine Learning*; PMLR: New York, NY, USA, 2021.

17. Holtzman, A.; Buys, J.; Du, L.; Forbes, M.; Choi, Y. The Curious Case of Neural Text Degeneration. *arXiv* **2019**, arXiv:1904.09751.
18. Fan, A.; Lewis, M.; Dauphin, Y. Hierarchical Neural Story Generation. *arXiv* **2018**, arXiv:1805.04833.
19. Su, Y.; Lan, T.; Wang, Y.; Yogatama, D.; Kong, L.; Collier, N. A Contrastive Framework for Neural Text Generation. *Adv. Neural Inf. Process. Syst.* **2022**, *35*, 21548–21561.
20. Razu Ahmed, M.; Arifa Khatun, M.; Asraf Ali, M.; Sundaraj, K. A literature review on NoSQL database for big data processing. *Int. J. Eng. Technol.* **2018**, *7*, 902. [CrossRef]
21. Pagán, J.E.; Cuadrado, J.S.; Molina, J.G. A repository for scalable model management. *Softw. Syst. Model.* **2015**, *14*, 219–239. [CrossRef]
22. Marino, A.; Palmonari, M.; Spahiu, B. Towards an Access Control Model for KnowledgeGraphs. In Proceedings of the SEBD 2021 Italian Symposium on Advanced Database Systems—Proceedings of the 29th Italian Symposium on Advanced Database Systems, Pizzo, Italy, 5–9 September 2021; Volume 2994.
23. Ji, S.; Pan, S.; Cambria, E.; Marttinen, P.; Yu, P.S. A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. *IEEE Trans. Neural Netw. Learn. Syst.* **2021**, *33*, 494–514. [CrossRef]
24. Pan, J.Z.; Razniewski, S.; Kalo, J.C.; Singhania, S.; Chen, J.; Dietze, S.; Jabeen, H.; Omeliyanenko, J.; Zhang, W.; Lissandrini, M.; et al. Large Language Models and Knowledge Graphs: Opportunities and Challenges. *arXiv* **2023**, arXiv:2308.06374.
25. Neo4j. The Leader in Graph Technology. Available online: https://neo4j.com/ (accessed on 21 November 2024).
26. Angles, R.; Gutierrez, C. Survey of graph database models. *ACM Comput. Surv.* **2008**, *40*, 1–39. [CrossRef]
27. Francis, N.; Green, A.; Guagliardo, P.; Libkin, L.; Lindaaker, T.; Marsault, V.; Plantikow, S.; Rydberg, M.; Selmer, P.; Taylor, A. Cypher: An Evolving Query Language for Property Graphs. In Proceedings of the 2018 International Conference on Management of Data. ACM, Houston, TX, USA, 10–15 June 2018; pp. 1433–1445. [CrossRef]
28. Angles, R.; Arenas, M.; Barceló, P.; Hogan, A.; Reutter, J.; Vrgoč, D. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* **2018**, *50*, 1–40. [CrossRef]
29. Hendrix, G.G.; Sacerdoti, E.D.; Sagalowicz, D.; Slocum, J. Developing a natural language interface to complex data. *ACM Trans. Database Syst.* **1978**, *3*, 105–147. [CrossRef]
30. Xu, X.; Liu, C.; Song, D. SQLNet: Generating Structured Queries From Natural Language Without Reinforcement Learning. *arXiv* **2017**, arXiv:1711.04436. [CrossRef]
31. Yu, T.; Li, Z.; Zhang, Z.; Zhang, R.; Radev, D. TypeSQL: Knowledge-based Type-Aware Neural Text-to-SQL Generation. *arXiv* **2018**, arXiv:1804.09769. [CrossRef]
32. Yu, T.; Yasunaga, M.; Yang, K.; Zhang, R.; Wang, D.; Li, Z.; Radev, D. SyntaxSQLNet: Syntax Tree Networks for Complex and Cross-Domain Text-to-SQL Task. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 31 October–4 November 2018; pp. 1653–1663. [CrossRef]
33. Montgomery, C.; Isah, H.; Zulkernine, F. Towards a Natural Language Query Processing System. In Proceedings of the 2020 1st International Conference on Big Data Analytics and Practices, Bangkok, Thailand, 25–26 September 2020. [CrossRef]
34. Guo, A.; Zhao, X.; Ma, W. ER-SQL: Learning enhanced representation for Text-to-SQL using table contents. *Neurocomputing* **2021**, *465*, 359–370. [CrossRef]
35. Devlin, J.; Chang, M.W.; Lee, K.; Toutanova, K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), Minneapolis, MN, USA, 2–7 June 2019; pp. 4171–4186. [CrossRef]
36. Zhong, V.; Xiong, C.; Socher, R. Seq2SQL: Generating Structured Queries from Natural Language using Reinforcement Learning. *arXiv* **2017**, arXiv:1709.00103.
37. He, P.; Mao, Y.; Chakrabarti, K.; Chen, W. X-SQL: Reinforce schema representation with context. *arXiv* **2019**, arXiv:1908.08113.
38. Chen, M.; Tworek, J.; Jun, H.; Yuan, Q.; Pinto, H.P.d.O.; Kaplan, J.; Edwards, H.; Burda, Y.; Joseph, N.; Brockman, G.; et al. Evaluating Large Language Models Trained on Code. *arXiv* **2021**, arXiv:2107.03374.
39. Arora, A.; Bhaisaheb, S.; Patwardhan, M.; Vig, L.; Shroff, G. A GENERIC PROMPT FOR AN LLM THAT ENABLES NL-TO-SQL ACROSS DOMAINS AND COMPOSITIONS. In Proceedings of the Eleventh International Conference on Learning Representations, Kigali, Rwanda, 1–5 May 2023; pp. 1–12.
40. Yu, T.; Zhang, R.; Yang, K.; Yasunaga, M.; Wang, D.; Li, Z.; Ma, J.; Li, I.; Yao, Q.; Roman, S.; et al. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, 31 October–4 November 2018; pp. 3911–3921. [CrossRef]
41. Mondal, S.; Mukherjee, P.; Chakraborty, B.; Bashar, R. Natural Language Query to NoSQL Generation Using Query-Response Model. In Proceedings of the 2019 International Conference on Machine Learning and Data Engineering (iCMLDE), Taipei, Taiwan, 2–4 December 2019; pp. 85–90. [CrossRef]
42. Pradeep, T.; Rafeeque, P.C.; Murali, R. Natural Language To NoSQL Query Conversion using Deep Learning. Available online: https://ssrn.com/abstract=3436631(accessed on 1 September 2024).
43. Hossen, K.; Uddin, M.; Arefin, M.; Uddin, M.A. BERT Model-based Natural Language to NoSQL Query Conversion using Deep Learning Approach. *Int. J. Adv. Comput. Sci. Appl.* **2023**, *14*, 810–821. [CrossRef]

44. Zhang, W.; Zeng, K.; Yang, X.; Shi, T.; Wang, P. Text-to-ESQ: A Two-Stage Controllable Approach for Efficient Retrieval of Vaccine Adverse Events from NoSQL Database. In Proceedings of the 14th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics. Houston, TX, USA, 3–6 September 2023; pp. 1–10. [CrossRef]
45. W3C. SPARQL 1.1 Overview. Available online: https://www.w3.org/TR/sparql11-overview/ (accessed on 21 November 2024).
46. Aghaei, S.; Raad, E.; Fensel, A. Question Answering Over Knowledge Graphs: A Case Study in Tourism. *IEEE Access* **2022**, *10*, 69788–69801. [CrossRef]
47. Liang, S.; Stockinger, K.; de Farias, T.M.; Anisimova, M.; Gil, M. Querying knowledge graphs in natural language. *J. Big Data* **2021**, *8*, 3. [CrossRef]
48. Rony, M.R.A.H.; Kumar, U.; Teucher, R.; Kovriguina, L.; Lehmann, J. SGPT: A Generative Approach for SPARQL Query Generation From Natural Language Questions. *IEEE Access* **2022**, *10*, 70712–70723. [CrossRef]
49. Purkayastha, S.; Dana, S.; Garg, D.; Khandelwal, D.; Bhargav, G.S. A Deep Neural Approach to KGQA via SPARQL Silhouette Generation. In Proceedings of the 2022 International Joint Conference on Neural Networks (IJCNN), Padua, Italy, 18–23 July 2022; ISSN: 2161-4407. [CrossRef]
50. OpenAI. Models Overview Documentation. Available online: https://platform.openai.com/docs/models/overview (accessed on 21 November 2024).
51. An, Y.; Greenberg, J.; Kalinowski, A.; Zhao, X.; Hu, X.; Uribe-Romo, F.J.; Langlois, K.; Furst, J.; Gómez-Gualdrón, D.A. Knowledge Graph Question Answering for Materials Science (KGQA4MAT): Developing Natural Language Interface for Metal-Organic Frameworks Knowledge Graph (MOF-KG). *arXiv* **2023**, arXiv:2309.11361. https://doi.org/10.48550/arXiv.2309.11361.
52. Meyer, L.P.; Stadler, C.; Frey, J.; Radtke, N.; Junghanns, K.; Meissner, R.; Dziwis, G.; Bulert, K.; Martin, M. LLM-assisted Knowledge Graph Engineering: Experiments with ChatGPT. In *Working conference on Artificial Intelligence Development for a Resilient and Sustainable Tomorrow*; Springer: Cham, Switzerland, 2023. [CrossRef]
53. Dubey, M.; Banerjee, D.; Abdelkawi, A.; Lehmann, J. LC-QuAD 2.0: A Large Dataset for Complex Question Answering over Wikidata and DBpedia. In Proceedings of the Semantic Web—ISWC 2019, Auckland, New Zealand, 26–30 October 2019; Ghidini, C., Hartig, O., Maleshkova, M., Svátek, V., Cruz, I., Hogan, A., Song, J., Lefrançois, M., Gandon, F., Eds.; Springer International Publishing: Cham, Switzerland, 2019; pp. 69–78. [CrossRef]
54. Perevalov, A.; Diefenbach, D.; Usbeck, R.; Both, A. QALD-9-plus: A Multilingual Dataset for Question Answering over DBpedia and Wikidata Translated by Native Speakers. In Proceedings of the 2022 IEEE 16th International Conference on Semantic Computing (ICSC), Laguna Hills, CA, USA, 26–28 January 2022; pp. 229–234, ISSN: 2325-6516, [CrossRef]
55. Perevalov, A.; Yan, X.; Kovriguina, L.; Jiang, L.; Both, A.; Usbeck, R. Knowledge Graph Question Answering Leaderboard: A Community Resource to Prevent a Replication Crisis. *arXiv* **2022**, arXiv:2201.08174. https://doi.org/10.48550/arXiv.2201.08174.
56. Hains, G.J.D.R.; Khmelevsky, Y.; Tachon, T. From natural language to graph queries. In Proceedings of the 2019 IEEE Canadian Conference of Electrical and Computer Engineering (CCECE), Edmonton, AB, Canada, 5–8 May 2019; pp. 1–4. [CrossRef]
57. Kobeissi, M.; Assy, N.; Gaaloul, W.; Defude, B.; Haidar, B. An Intent-Based Natural Language Interface for Querying Process Execution Data. In Proceedings of the 2021 3rd International Conference on Process Mining (ICPM), Eindhoven, The Netherlands, 31 October–4 November 2021; pp. 152–159. [CrossRef]
58. Zhou, Q.; Wu, C.; Yang, J.; Han, L.; Wu, B. Natural Language Query for SCD File. In Proceedings of the 4th International Conference on Information Technologies and Electrical Engineering. Association for Computing Machinery, Changde, China, 29–31 October 2022; ICITEE '21, pp. 1–6. [CrossRef]
59. Chatterjee, J.; Dethlefs, N. Automated Question-Answering for Interactive Decision Support in Operations & Maintenance of Wind Turbines. *IEEE Access* **2022**, *10*, 84710–84737. [CrossRef]
60. Feng, G.; Zhu, G.; Shi, S.; Sun, Y.; Fan, Z.; Gao, S.; Hu, J. Robust NL-to-Cypher Translation for KBQA: Harnessing Large Language Model with Chain of Prompts. In *Knowledge Graph and Semantic Computing: Knowledge Graph Empowers Artificial General Intelligence*; Communications in Computer and Information Science; Wang, H., Han, X., Liu, M., Cheng, G., Liu, Y., Zhang, N., Eds.; Springer Nature: Singapore, 2023; pp. 317–326. [CrossRef]
61. March, S.; Smith, G. Design and Natural Science Research on Information Technology. *Decis. Support Syst.* **1995**, *15*, 251–266. [CrossRef]
62. Peffers, K.; Tuunanen, T.; Rothenberger, M.; Chatterjee, S. A design science research methodology for information systems research. *J. Manag. Inf. Syst.* **2007**, *24*, 45–77. [CrossRef]
63. Hevner, A.R.; March, S.T.; Park, J.; Ram, S. Design Science in Information Systems Research. *MIS Q.* **2004**, *28*, 75–105. [CrossRef]
64. Docker. Docker Documentation. Available online: https://docs.docker.com/ (accessed on 21 November 2024).
65. Gradio. Build and Share Delightful Machine Learning Apps. Available online: https://www.gradio.app/ (accessed on 21 November 2024).
66. OpenAI. Structured Outputs: JSON Mode. Available online: https://platform.openai.com/docs/guides/structured-outputs#json-mode (accessed on 21 November 2024).
67. Guo, A.; Li, X.; Xiao, G.; Tan, Z.; Zhao, X. SpCQL: A Semantic Parsing Dataset for Converting Natural Language into Cypher. In Proceedings of the 31st ACM International Conference on Information & Knowledge Management, Atlanta, GA, USA, 17–21 October 2022; pp. 3973–3977. [CrossRef]
68. Guo, A. Text-to-CQL: A Dataset for Converting Natural Language into Cypher. Available online: https://github.com/Guoaibo/Text-to-CQL (accessed on 21 November 2024).

69. Chatterjee, J.; Dethlefs, N. WindTurbine-QAKG: Automated Question-Answering Over Knowledge Graphs in O&M of Wind Turbines. Available online: https://github.com/joyjitchatterjee/WindTurbine-QAKG (accessed on 21 November 2024).
70. van Dongen, B. BPI Challenge 2017. Available online: https://data.4tu.nl/articles/dataset/BPI_Challenge_2017/12696884 (accessed on 21 November 2024).
71. Mack, D.; Jefferson, A. CLEVR Graph: A Dataset for Graph-Based Reasoning. Available online: https://github.com/Octavian-ai/clevr-graph (accessed on 21 November 2024).
72. Neo4j. Example Datasets - Getting Started. Available online: https://neo4j.com/docs/getting-started/appendix/example-data/ (accessed on 21 November 2024).
73. Tran, Q.B.H.; Waheed, A.A.; Chung, S.T. Robust Text-to-Cypher Using Combination of BERT, GraphSAGE, and Transformer (CoBGT) Model. *Appl. Sci.* **2024**, *14*, 7881. [CrossRef]
74. Mistral AI. Mixtral of Experts. Available online: https://mistral.ai/news/mixtral-of-experts/ (accessed on 21 November 2024).
75. Meta AI. Introducing Code Llama, a state-of-the-art large language model for coding. Available online: https://ai.meta.com/blog/code-llama-large-language-model-coding/ (accessed on 21 November 2024).
76. LangChain. Neo4j Cypher Integration Documentation. Available online: https://python.langchain.com/docs/integrations/graphs/neo4j_cypher/ (accessed on 21 November 2024).
77. Bratanic, T. Crowdsourcing Text2Cypher Dataset. Available online: https://bratanic-tomaz.medium.com/crowdsourcing-text2cypher-dataset-e65ba51916d4 (accessed on 21 November 2024).
78. Khattab, O.; Singhvi, A.; Maheshwari, P.; Zhang, Z.; Santhanam, K.; Vardhamanan, S.; Haq, S.; Sharma, A.; Joshi, T.T.; Moazam, H.; et al. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv* **2023**, arXiv:2310.03714.