universität wien

MASTERARBEIT | MASTER'S THESIS

Titel | Title

Enhancing Cheminformatics Documentation and Code Understanding: A Knowledge Graph Approach to Retrieval Augmented Generation

verfasst von | submitted by

Selina Vanessa Schöndorfer BSc MSc

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 066 910
Studienrichtung lt. Studienblatt | Degree programme as it appears on the student record sheet:	Masterstudium Computational Science
Betreut von | Supervisor:	Univ.-Prof. Mag. Dr. Thierry Langer

Acknowledgements

I want to thank my supervisors Oliver Wieder and Thierry Langer for giving me the opportunity to work on this topic and supporting me during this thesis. Further, I want to thank the Cheminformatics Lab for welcoming me into their group and creating a warm and positive work environment. Finally, a big thank you goes out to my friends and family, as this would not have been possible without them.

Abstract

Large Language Models (LLMs) have brought impressive advancements to fields like natural language and code generation. Despite their abilities, however, they still have some disadvantages: Most LLMs suffer from hallucinations and are inherently limited by the knowledge in their training corpus. A promising approach that has been shown to alleviate these issues is Retrieval Augmented Generation (RAG), which enhances the parametric memory of LLMs with an external knowledge base. Graph RAG uses Knowledge Graphs as the knowledge base, enabling the representation of complex relationships and hierarchies, which is an important aspect in code generation and documentation. In this thesis, we implemented a question answering and code generating Graph RAG system for the CDPKit, a cheminformatics toolkit, solely based on its Python API documentation. To do so, we first parsed its documentation into a Neo4j Knowledge Graph. The Graph RAG system then retrieved relevant context from the Knowledge Graph via a Cypher query and answered user queries based on this context. To make the Graph RAG available to users, we built a dashboard with a chat interface for it. The system was evaluated on a set of test questions by a human evaluator and an LLM. We calculated the metrics Accuracy, Recall, Precision and F1 score. We found that our Graph RAG system was able to answer specific questions about the CDPKit and provide basic code snippets, but its performance dropped for general questions. We therefore think that this work provides a good base for future development, and we provide ideas on how to improve the Graph RAG system further.

Zusammenfassung

Large Language Modelle (LLMs) haben in den letzten Jahren beeindruckende Fortschritte in Bereichen wie der Erzeugung natürlicher Sprache und Code herbeigeführt. Trotz ihrer Fähigkeiten haben sie auch einige Nachteile: Die meisten LLMs leiden an Halluzinationen und sind von Natur aus durch das Wissen ihres Trainingskorpus begrenzt. Ein vielversprechender Ansatz, der erwiesenermaßen bei diesen Problemen helfen kann, ist Retrieval Augmented Generation (RAG), das das parametrische Wissen der LLMs mit einer externen Wissensbasis erweitert. Graph RAG benutzt Knowledge Graphen als externe Wissensbasis und ermöglicht damit die Darstellung von komplexen Beziehungen und Hierarchien, was ein wichtiger Aspekt der Codegenerierung und Dokumentation ist. In dieser Arbeit haben wir ein Graph RAG System zur Fragenbeantwortung und Codegenerierung für das CDPKit, ein Toolkit aus der Chemieinformatik, implementiert, allein basierend auf dessen Python API Dokumentation. Dafür übersetzten wir die Dokumentation in einen Neo4j Knowledge Graphen. Dann bezog das Graph RAG System den relevanten Kontext vom Knowledge Graphen mit Hilfe einer Cypher Abfrage und beantwortete Nutzeranfragen basierend auf diesem Kontext. Um das Graph RAG System für Nutzer zugänglich zu machen, bauten wir ein Dashboard mit einer Chat Schnittstelle. Das System wurde von einem Menschen und einem LLM anhand eines Sets von Testfragen evaluiert. Wir berechneten die Metriken Accuracy, Recall, Precision und F1 Wert. Unsere Ergebnisse zeigen, dass das Graph RAG System spezifische Fragen zum CDPKit beantworten und einfache Code Teile generieren kann. Bei generellen Fragen sank die Performance. Wir sind der Meinung, dass diese Arbeit eine gute Grundlage für zukünftige Entwicklungen bietet, und wir schlagen Ideen vor, wie das Graph RAG System weiter verbessert werden kann.

Content

1 Introduction

In recent years, the field of natural language processing (NLP) has undergone significant advancements driven largely by the emergence of Large Language Models (LLMs) and conversational systems like ChatGPT [1]. NLP focuses on enabling computers to understand, process and generate human language by combining linguistics with computer science and artificial intelligence. Key applications of NLP include document retrieval (i.e. retrieving relevant documents for a query), information extraction (i.e. extracting relevant metadata from a text) and natural language generation [2], [3]. To address these tasks, language models can be used, which use probabilistic techniques to analyse and generate text [4].

Historically, language models were task-specific, requiring large amounts of annotated data for supervised learning. However, in addition to being time consuming, annotated data is not readily available for many domains. This limited the adaptability of language models to specialized fields [5]. To address these limitations, pre-trained language models (PLMs) make use of unsupervised training on large unlabelled text corpora, enabling more generalized representations and improved performance across tasks. As PLMs evolved, it became apparent that increasing the number of model parameters also increased the performance of the model. This led to the emergence of LLMs, models with billions of model parameters trained on extensive datasets.

Large Language Models (LLMs) rely on several foundational concepts to process and generate text effectively. A key preprocessing step is tokenization, which breaks down the input text into basic units (i.e. tokens), such as multiple words, words, subwords or characters [6], [7]. From the training corpus, LLMs construct a vocabulary of tokens and learn to predict the next token in a sequence based on context [5].

After tokenization, the tokens are converted into a form that the model can process, i.e. vectors [8]. Word embedding techniques create high-dimensional vector representations where similar tokens are clustered together [9]. Contextual embeddings, on the other hand, generate different vector representations for the same token depending on its surrounding. Therefore, they are better suited for

capturing the overall meaning of an input sequence [8].

Most modern LLMs are built upon a transformer architecture [10], [11], replacing the inherently sequential recurrent neural networks like GRU in language modelling [10], [12]. This reduces computational complexity and allows for parallelization. Transformers rely on self-attention mechanisms, which identify the more relevant tokens in an input sequence by assigning different weights to them. This is done by mapping a query, key and value vector for the input sequences. The attention weights are computed by multiplying the query vector of each token with the key vectors of all the other tokens and applying a softmax function. These weights are then applied to the value vectors to create the contextual representation [10]. Transformer architectures can have both an encoder and decoder or only one of the two, and can use different variants of attention [13]. An encoder embeds the input sequence tokens into a sequence of high-dimensional representations. The decoder takes these representations and produces an output sequence one element at a time.

Another important aspect of the Transformer architecture are positional encodings, which allow the model to keep track of the order of the sequence when processing it in parallel [10].

LLMs are typically trained in two stages. The first is unsupervised pre-training, where the model is trained to predict tokens on a large universal corpus of unlabeled data, learning general language patterns. This provides a strong foundation for adapting the model to more specific tasks via a supervised training with a smaller, annotated dataset in the fine-tuning stage [5]. While fine-tuning helps adapt LLMs to more specialized areas, pre-trained LLMs can be used effectively without fine-tuning for a lot of downstream tasks (e.g. GPT-3, T5, LLama) [13].

Another way to improve the performance of LLMs is prompt engineering, which refers to the process of adapting the input prompt to direct the LLMs output. Examples of prompt engineering include adding instructions on how to process the user query or one or multiple examples of a required task (i.e. one-shot and few-shot prompting) [14].

Since their appearance, LLMs have revolutionized NLP fields like translation, summarization and conversational interactions. Moreover, LLMs exhibit emergent abilities like reasoning, planning and decision making, despite not being explicitly trained for them. These advancements have led to the application of LLMs in many

domains like question answering, autonomous conversational agents and code generation [13], [15].

However, LLMs also face several challenges. For some tasks, fine-tuning is necessary, which requires a lot of memory and computational resources. Further, LLMs are prone to produce inaccurate information (i.e. hallucinations) that can be difficult to prevent and detect. Additionally, the knowledge of a LLM is constrained by its training data, which may be outdated or inaccurate. Updating the model through retraining or fine-tuning is often expensive and complex [16].

One approach to address these issues is Retrieval Augmented Generation (RAG), which extends the parametric memory of LLMs with non-parametric memory in the form of external knowledge bases. In RAG, the input query is encoded and used to retrieve relevant context from the knowledge base, which is then combined with the input query to generate a target output [17].This allows LLMs to access up-to-date, domain-specific knowledge without fine-tuning, reducing hallucinations and increasing accuracy [18], [19].

RAG systems typically consist of three components: Indexing, retrieval and generation. During indexing, information from external sources (e.g., PDFs, HTML) is processed into chunks and encoded as vectors using an embedding model. These vectors are then stored in a vector database. When the user enters a query, it is encoded into a vector with the same embedding model used in indexing. During retrieval, the input query vector is compared to the vector database and a similarity search retrieves the top-k relevant chunks from the database. During generation, the input query and the retrieved context are passed to the generating LLM. The model will generate an answer to the original input query based on the retrieved context and, if desirable, its own inherent parametric knowledge [18].

To evaluate RAG systems, three components need to be considered: retrieval, generation and the whole system. Due to the diverse nature of potential knowledge bases, evaluating the retrieval can be challenging. Further, the generation should accurately reflect the retrieved context while considering relevancy to the input query. Additionally, open-ended question answering can be difficult to evaluate as there is often more than one correct answer. Holistic evaluation is essential, as failures in retrieval or generation can result in incorrect outputs [20].

However, when evaluating question-answering systems with provided reference answers, there are several classical machine learning metrics that can be easily calculated and that can help gain insight into the performance of the system: Accuracy, Precision, Recall and F1 score [21].

The success of a RAG system greatly depends on the quality and structure of the external knowledge base [22]. Knowledge sources can range from unstructured text to structured formats like Knowledge Graphs [18]. Knowledge Graphs are useful for representing hierarchical and complex relationships between entities. Each node is an object with properties and the directed edges between them are the different relationship types. Knowledge Graphs represent knowledge as triples: head (node), relation (edge) and tail (node) [23]. To embed the graph into a vector space, several methods can be applied that differ in how the head, tail and relation are represented. These methods can be broadly categorized into geometric approaches, semantic matching models, and more recent neural network-based approaches [24], [25]. One classic example is TransE, a geometric approach which aims for an embedding in a low-dimensional vector space by representing relationships as translations in the embedding space. That way, entities that are connected by a relationship are closer to each other [26]. While triplet-based embeddings provide a foundational representation, they often lack sufficient context for LLMs. Adding textual descriptions or supplementary information to graph nodes enhances their utility and facilitates better integration with downstream tasks [22]. When RAG systems use a Knowledge Graph as the external memory base, they are called Graph RAG systems [27].

RAG presents a promising approach in software domains like code generation and documentation, where access to current, domain-specific information is critical [28]. For example, the REDCODER framework used RAG to retrieve relevant code snippets and summaries from a database as context for code generation, improving model performance [29]. It has been shown that retrieving code manuals and documentation as context enhanced the performance of pre-trained LLMs [30]. A benchmark study could show that RAG led to a higher accuracy of code generation across programming domains [28]. However, they also identified the importance of retrieving only the most relevant and factually correct data.

Because of their ability to represent hierarchical relationships and long range dependencies between documents, Knowledge Graphs are particularly interesting for domains like code generation and documentation. For instance, Graph4Code models

Python code as a knowledge graph with nodes representing classes, functions, and methods, and edges capturing usage and documentation relationships. This approach has been successfully applied to analyze Python repositories on GitHub [31].
Compared to traditional RAG systems, Graph RAG has the benefit of accurately encoding relationships not only between entities, but also between multiple documents [22].

1.1 Related Work

The implementations and applications of Graph RAG are diverse. For instance, SubgraphRAG [32] uses a lightweight multilayer perceptron to retrieve relevant subgraphs. KnowledGPT [33] first creates search queries for the Knowledge Graph, executes these and then uses the context to generate the response. Further, Knowledge Graphs have been used to represent source code, with a re-ranker mechanism limiting the amount of irrelevant data retrieved from the graphs [34].

To our knowledge, however, there is no research on Graph RAG systems that are built to both answer questions and generate code based on a specific library, source code or API documentation.

In this thesis, we developed a question answering and code generation Graph RAG system for a cheminformatics toolkit, the CDPKit [35]. CDPKit is an open-source software implemented in C++, which combines software tools with a programming library. In addition to the C++ source code, CDPKit also provides a Python interfacing layer with extensive documentation. Our aim was to build a Graph RAG system that supports both question answering and code generation tasks using CDPKit's domain-specific knowledge.

The first step in our framework was to construct a Neo4j Knowledge Graph [36] from the CDPKit’s documentation, encoding the hierarchical relationships and properties of the classes and functions. Instead of embedding the knowledge graph into vector representations for the retrieval, we used a LLM to translate the user query into a Cypher query. The Cypher query was then executed to retrieve the

relevant nodes and relationships. This retrieved context, combined with the user query, was passed to a second LLM to generate the final output.

To evaluate the performance of our Graph RAG system, we created a benchmark of test questions with reference contexts and answers specific to the CDPKit. For these questions, we calculated the metrics Accuracy, Precision, Recall and F1 Score using human evaluations and assessment by a third LLM.

Finally, we built a user-friendly dashboard to enable interaction with our Graph RAG system.

2 Graph RAG Design

2.1 Objectives

The goal was to build a Graph RAG system capable of answering general questions about the CDPKit and its structure, as well as providing Python code snippets and suggestions. This was achieved using pre-trained local LLMs without retraining or fine-tuning. The system was implemented in Python 3.11, with the Neo4j Knowledge Graph built from the CDPKit's Python interface documentation.

2.2 CDPKit Documentation Features

CDPKit is a C++ cheminformatics toolkit with a well-documented Python API available on GitHub [37]. The documentation is organized into several directories that contain domain-related doc files for classes and functions, including comments on their functionality. The Python API provides information about the definition, types, names, methods, input parameters, inheritance, default values, return types and decorators, but not on the internal content of classes or functions.

2.3 Workflow Overview

We started by parsing the CDPKit Python API documentation into a dictionary, extracting the relevant features and hierarchies of the classes and functions, as well as any associated comments. The dictionary was then used to create a Knowledge Graph in the Neo4j Aura Graph Database. The graph could be queried using Cypher, a declarative graph query language by Neo4j [38]. This enabled the retrieval of relevant context information, bypassing the need for often inadequate embedding of the Knowledge Graph into a vector database [22].

User queries were processed by the retriever, which used a LLM to translate the input into Cypher queries. These queries were then executed on the Knowledge Graph to retrieve relevant context. Finally the retrieved context, along with the

original query, was then passed to the generator LLM, which produced an answer based on the provided context.
Finally, the Graph RAG was evaluated on a set of test questions and a dashboard was built to make it accessible to users (see Fig. 1)

Figure 1: Schematic overview of the general workflow of this thesis.

2.4 Parsing the documentation

To extract the necessary information for the Knowledge Graph from the documentation, we ensured that all features and functionalities were parsed while keeping track of the code hierarchies. For this purpose, we implemented the DocParser class. This class takes the path to a directory, parses its contents, and updates an internal dictionary with details of all classes and functions found within the files it has processed so far.

The process begins by reading each file as unstructured text and extracting the comments with the Python module tokenize [39], which scans for comments and extracts them. Next, using the Python module ast [40], the content of the file is converted into an Abstract Syntax Tree (AST). An AST is a tree representation of source code,where nodes represent data elements and edges denote their hierarchical relationships [41]. Next to converting the code into an AST, ast also provides functions to traverse the tree and retrieve attributes such as line numbers, objects, types, and more.

The AST and extracted comments are then passed to the ClassAndFunctionVisitor class, which inherits from ast.NodeVisitor. This visitor class traverses the AST from root to leaf nodes, visiting each class and function node. For each class node, it parses the following attributes: name, base, decorator, methods, attributes (with name, value and comment), nested classes and comments. For each function node, it parses the following attributes: name, input parameters (with name,

type, default and comment), decorators, the return type (type and comment) and comments.

The parsed information is stored internally and can be accessed as a dictionary, providing the foundation for constructing the Knowledge Graph.

2.5 Generating the Knowledge Graph

A key requirement for the Knowledge Graph was to accurately represent the hierarchies and relationships of the individual entities. We based the naming and organization of the graph on the implementation of Strazh, a tool to build a Knowledge Graph from a C# Codebase [42].

The graph can be logically divided into two levels: project structure and implementation. The project structure level includes the node types Project, Folder and File, connected by edges of the type INCLUDED_IN and DECLARED_AT, indicating the relationships between these nodes and the nodes of the implementational level. The implementation level included the node types Class, Function, Parameter and Decorator, with edges of the type INHERITS_FROM, HAS, OF_TYPE, indicating the dependencies and relationships between them. To ensure uniqueness and capture all available information, we added additional properties to the nodes (associated comments, attributes, parameters, defaults, types, return types and decorators) (see Fig. 2).

We used Neo4j AuraDB, a free and fully managed graph database accessible via Python, for building the Knowledge Graph [43].

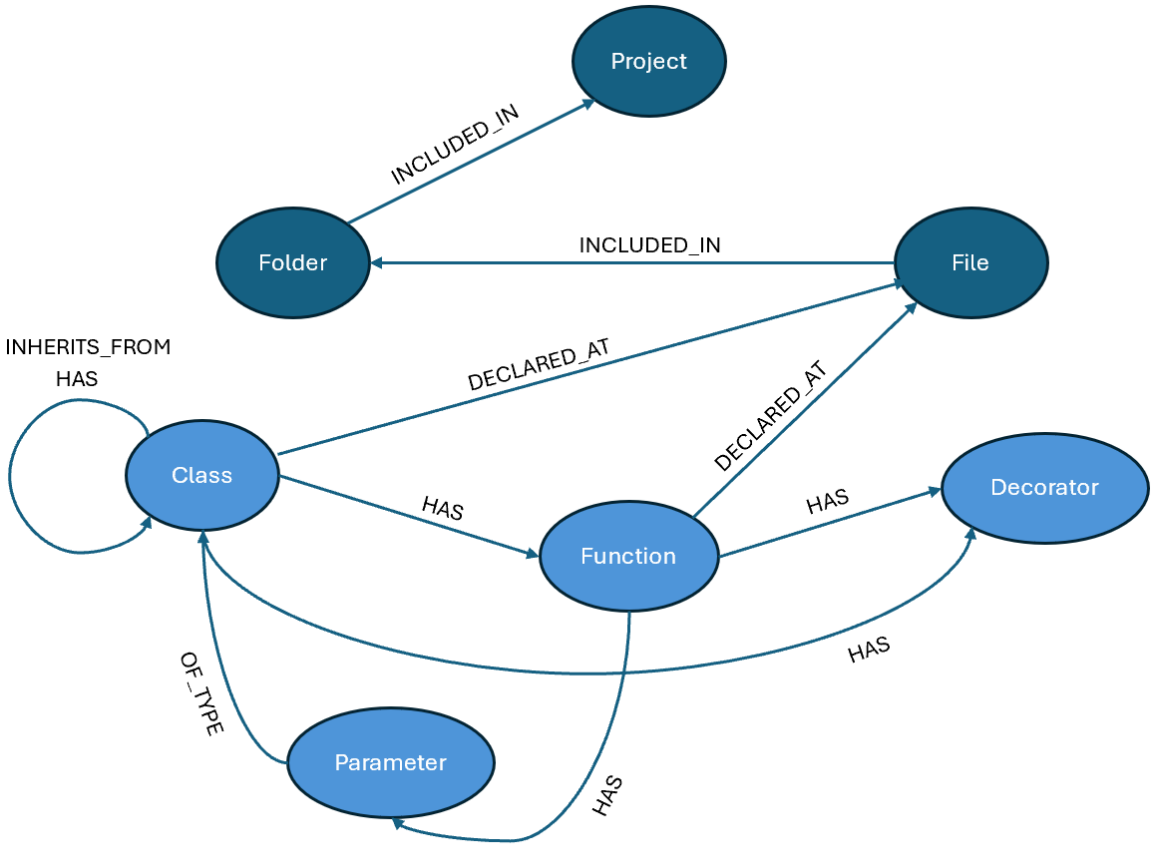


Figure 2: Nodes and Relationships of the Knowledge Graph. Dark blue nodes belong to the project structure, while light blue nodes belong to the implementational level.

To create the Knowledge Graph, we implemented manager classes for each node type (ProjectManager, FolderManager, FileManager, ClassManager, FunctionManager, ParameterManager, TypeManager and DecoratorManager). They were responsible for traversing the parsed documentation and executing Cypher queries to create the nodes and edges in the Neo4j database. The KnowledgeGraphManager class was responsible for coordinating the generation process via the manager classes. This approach ensured the Knowledge Graph accurately captured relationships and interactions among classes and functions across multiple files.

2.6 Graph RAG System

The Graph RAG system was implemented using two LLMs: one involved in the retrieval process and one for generating the final answer. Both were free, pre-trained LLMs accessed via text generation pipelines from Huggingface's transformer library [44]. These pipelines offer a straightforward API for using pre-trained LLMs in inference tasks such as question answering.

When an input query was received, it was first processed by the retrieval LLM, which translated the query into a Cypher query (see Fig. 3). The model we used for this was CodeLlama-13b-Instruct-hf, a model designed for general-purpose code synthesis and comprehension [45]. We based our implementation of the Cypher query generation on [46]. To improve the accuracy of the translation into Cypher, we applied prompt engineering, including clear instructions, the schema of the knowledge graph (node and edge types, as well as their directions), and two positive examples (see Fig. 4). The generated Cypher query was then executed on the Neo4j AuraDB to retrieve the relevant information.

Subsequently, the input query, its Cypher translation, and the retrieved context were passed to the generating LLM, Qwen2.5-7B-Instruct, which is capable of coding and understanding structured data and has a large context window [47]. Using prompt engineering again, we instructed the model to generate an answer strictly based on the provided context. If applicable, it was also tasked to include a concise Python example. The prompt included clear guidelines and one positive example to further refine the output (see Fig. 5).

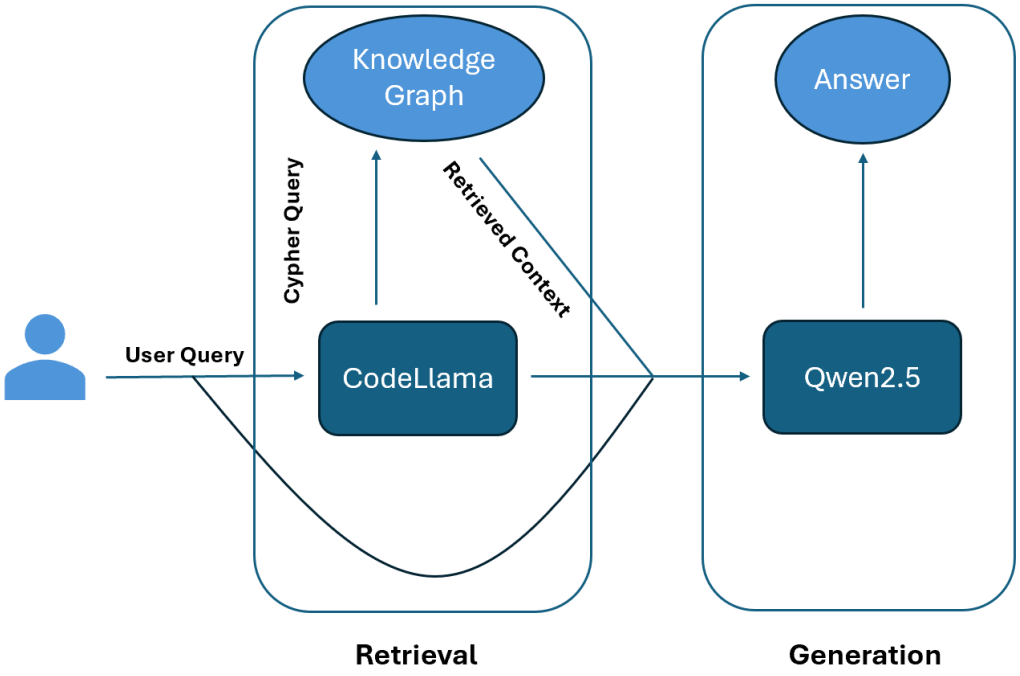


Figure 3: Schematic representation of the Graph RAG System.

```
You are experienced with translating questions into cypher queries. Provide answers in Cypher query
language, *strictly* based on the following graph Neo4j schema. Respect the possible direction of
relationships and the possible naming of nodes, properties and relationships. DO NOT CREATE NEW NODES OR
RELATIONSHIPS. Only answer with the query and nothing else.


### The Schema
Node properties are the following:
    [ {"labels": "Project", "properties": ["name"]},
        {"labels": "Folder", "properties": ["name"]},
        {"labels": "File", "properties": ["name"]},
        {"labels": "Class", "properties": ["name", "comment", "attributes"]},
        {"labels": "Function", "properties": ["name", "comment", "parameter", "decorators", "returns"]},
        {"labels": "Parameter", "properties": ["name", "comment", "default", "type"]},
        {"labels": "Decorator", "properties": ["name"]}]

Relationships point from source to target nodes:
    [{"relationship": "INCLUDED_IN", "source": "Folder", "target": "Project"},
        {"relationship": "INCLUDED_IN", "source": "File", "target": "Folder"},
        {"relationship": "INHERITS_FROM", "source": "Class", "target": "Class"},
        {"relationship": "HAS", "source": "Class", "target": ["Function", "Class", "Decorator"]},
        {"relationship": "DECLARED_AT", "source": "Class", "target": "File"},
        {"relationship": "HAS", "source": "Function", "target": "Decorator"},
        {"relationship": "HAS", "source": "Function", "target": "Parameter"}
        {"relationship": "DECLARED_AT", "source": "Function", "target": "File"},
        {"relationship": "OF_TYPE", "source": "Parameter", "target": "Class"}
```

```
### Examples

Q: What methods does AromaticSubstructure have?

A: MATCH (c:Class {{name: 'AromaticSubstructure'}})-[:HAS]->(f:Function) RETURN f.name, f.comment


Q: What does the class AtomPredicate do?

A: MATCH (c:Class {{name: "AtomPredicate"}}) RETURN c.comment
```

Figure 4: System prompt that was used to instruct Code Llama to translate the user query into a Cypher query.

```
You are a highly intelligent assistant. Your job is to answer user questions using *only* the information
from the retrieved context provided from a Neo4j knowledge graph database. If appropriate, add a small
python example for the retrieved context, but no cypher queries. Only add this Python code if it is
appropriate for the question. The retrieved context is the result of a cypher query. Ensure that your
response strictly relies on the retrieved context, and do not add any information from other sources.

Cypher query:  {cypher_query}
Retrieved Context: {retrieved_context}

### Example 1:

Q: What methods does AromaticSubstructure have?

Cypher query: MATCH (c:Class {{name: 'AromaticSubstructure'}})-[:HAS]->(f:Function) RETURN f.name,
f.comment

Retrieved Context: [['f.name', 'f.comment'], ['__init__', 'Constructs an empty
<tt>AromaticSubstructure</tt> instance.'], ['__init__', 'Construct a <tt>AromaticSubstructure</tt>
instance that consists of the aromatic atoms and bonds of the molecular graph <em>molgraph</em>.'],
['perceive', 'Replaces the currently stored atoms and bonds by the set of aromatic atoms and bonds of the
molecular graph <em>molgraph</em>.’]]

A: AromaticSubstructure has the following methods:

    - __init__: Constructs an empty <tt>AromaticSubstructure</tt> instance.

    - __init__: Construct a <tt>AromaticSubstructure</tt> instance that consists of the aromatic atoms and
                bonds of the molecular graph <em>molgraph</em>.

    - perceive: Replaces the currently stored atoms and bonds by the set of aromatic atoms and bonds of
                the molecular graph <em>molgraph</em>.
```

Figure 5: System prompt that was used to instruct Qwen2.5 to provide an answer to the input query based on a retrieved context.

2.7 Evaluation

To evaluate our Graph RAG system, we created a set of 32 test questions about the CDPKit. 17 of these questions were focused on specific structural aspects or classes and methods (see Tab. 1), while the other 15 were worded more general and open, sometimes not even naming classes or functions (see Tab. 2).

Table 1: Specific test set questions.

Specific Questions
What are the folders included in the CDPKit Project?
What files are included in the folder Base?
What class does the class AromaticRingSet inherit from?
What methods does the class AtomBondMapping have?
What parameters does calcGeometricalDiameter take?
What is the method calculate of the class MolecularComplexityCalculator good for?
Does the parameter atom of the function hasUFFType have a default value?
What does the function getMMFF94TypeIndex return?
What can you tell me about the class InteractionType?
What's the meaning of the attribute R of class AtomConfiguration?
What type is parameter feature of function perceiveExtendedType?
What does the parameter feature of function hasHydrophobicity stand for?
In what functions does the parameter feature appear?
What functions are in the file Feature_Functions.doc.py?
What type does getFeatures return?
How many parameters does norm1 take?
Can the method __add__ of the class ConstLMatrixTranspose take an integer for the parameter e?

Table 2: General test set questions.

General Questions
How does the method assign of the class MassComposition work?
How can I use the class ElementHistogram?
How can I initialize the class AromaticSubstructure?
How can I assign a BemisMurckoAnalyzer?
How to read in molecules?
How to read in SDF Molecules?
How to read in SMILES?
How to generate Conformations?
Can you generate Confromations with the CDPKit? If yes, how?
How to iterate over Atoms of a Molecule?
How can I use the CDPKit to iterate over Atoms?
How can I assign an instance of the class BemisMurckoAnalyzer?
How to read in molecules with the class MoleculeReader?
How to read in SDF Molecules with the class SDFMoleculeReader?
How to read in SMILES with the class SMILESMoleculeReader?
How to generate Conformations with the class ConformerGenerator?
How to iterate over Atoms of the class Molecule?

For each question, we provided a reference context and answer to serve as a baseline for comparison. However, these references did not represent the only possible correct solutions to the questions. The test set was processed by the Graph RAG system and executed 100 times.

We evaluated the performance of the system by scoring the retrieved context, the generated answer based on the context, and the final answer in relation to the input query. Results were categorized into true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). A result was classified as:

- TP if it matched the reference or provided an equally valid alternative.
- FP if it included incorrect information not present in the reference.

- FN if it failed to generate an answer despite the reference providing one.
- TN if neither the system nor the reference provided an answer.

From these classifications, we calculated the standard evaluation metrics:

- Accuracy: $\frac{TP + TN}{TP + TN + FP + FN}$
- Precision: $\frac{TP}{TP + FP}$
- Recall: $\frac{TP}{TP + FN}$
- F1 Score: $\frac{2 \times Precision \times Recall}{Precision + Recall}$

These metrics were computed for each of the 32 questions over the 100 repetitions, and we determined the mean and standard deviation of the results.

Since the system was also instructed to provide Python code snippets when relevant, we evaluated these snippets for syntactical correctness and from that calculated the Accuracy.

The scoring of the test set was performed by an LLM (Falcon3-10B-Instruct, *Tiiuae/Falcon3-10B-Instruct · Hugging Face*, 2024), which we provided with detailed scoring instructions. These included positive examples of TP, FP, TN and FN. Additionally, 20 of the 100 repetitions of the testset were evaluated manually by a human reviewer.

2.8 Dashboard

To make the final Graph RAG available to users, we built a chat-bot like dashboard with the dash library in Python [49]
The dashboard included a chat window (see Fig. 6), as well as example questions and an “About” page with an introduction to the system as well as the link to the CDPKit documentation (see Fig. 7).

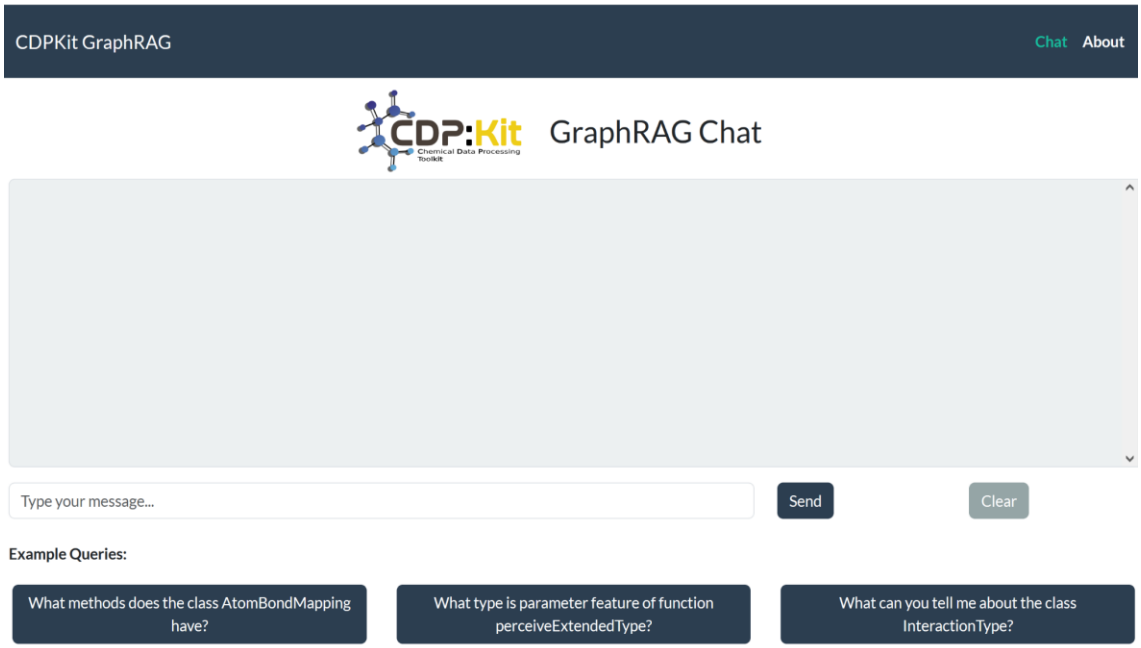


Figure 6: Chat window of the dashboard for the Graph RAG System.

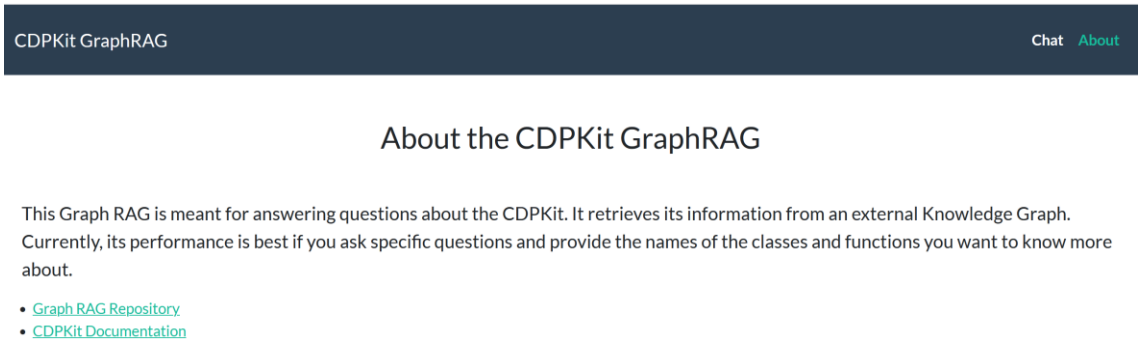


Figure 7: About page of the dashboard for the Graph RAG System.

3 Results

The evaluation of the Graph RAG system was divided into two categories: specific and general questions. The performance was scored across four dimensions: the retrieved context, the answer based on the retrieved context, the code snippet, and the overall answer quality in relation to the input question. Evaluations were conducted both manually and with an LLM.

3.1 Manual Evaluation

For specific questions, the manual evaluation revealed the following results (see Tab. 3):

- **Accuracy:** The retrieved context showed an accuracy of 0.58 (± 0.06), slightly above chance. This negatively impacted the overall answer accuracy, which was 0.59 (± 0.06). However, the answer based on the retrieved context and the code snippets achieved high accuracies of 0.84 (± 0.08) and 0.98 (± 0.03), respectively.
- **Precision:** The retrieved context showed a high precision of 0.9 (± 0.09), even outperforming the answer based on the context with 0.8 (± 0.96). The precision of the overall answer was 0.72 (± 0.09).
- **Recall:** The retrieved context achieved a recall of 0.62 (± 0.06), while the answer based on the context showed perfect recall (1 ± 0). The overall recall was 0.76 (± 0.08).
- **F1 Score:** The F1 Score for the retrieved context was 0.73 (± 0.05), while the context-based answer achieved an even higher score of 0.89 (± 0.05). The F1 score for the overall answer was 0.73 (± 0.06).

For general questions, the manual evaluation led to the following results (see Tab. 3):

- **Accuracy:** The retrieved context accuracy was low at 0.16 (± 0.07), which also affected the overall answer accuracy (0.16 ± 0.07). The answer based on

the retrieved context and the code snippets had high accuracies of 0.74 (± 0.08) and 0.98 (± 0.04).

- **Precision:** The retrieved context achieved good precision of 0.74 (± 0.24), outperforming the context-based answer (0.44 ± 0.11). The precision of the overall answer was 0.35 (± 0.16).
- **Recall:** The retrieved context recall was low at 0.17 (± 0.07), while the context based answer achieved perfect recall (1 ± 0). The recall of the overall answer was 0.22 (± 0.09).
- **F1 Score:** The F1 scores were 0.27 (± 0.11) for the retrieved context, 0.60 (± 0.11) for the context-based answer, and 0.27 (± 0.11) overall.

Comparing the results of the answers to the specific questions to those of the answers to the general questions, it becomes clear that the system performed better with specific questions. Metrics such as the accuracy and recall of the retrieved context, as well as the precision of the context-based answer, were much higher for specific questions. The accuracy of the code snippets, however, remained consistently high regardless of the question type.

Table 3: Result of the manual evaluation of 20 repetitions of the 32 testset questions.The metrics represent mean values with the standard deviation.

Type	Domain	Accuracy	Precision	Recall	F1
Specific	Context	0.58 ± 0.06	0.9 ± 0.09	0.62 ± 0.06	0.73 ± 0.05
	Answer	0.84 ± 0.08	0.8 ± 0.09	1 ± 0	0.89 ± 0.05
	Code	0.98 ± 0.03	/	/	/
	Overall	**0.59 ± 0.06**	**0.72 ± 0.09**	**0.76 ± 0.08**	**0.73 ± 0.06**
General	Context	0.16 ± 0.07	0.74 ± 0.24	0.17 ± 0.07	0.27 ± 0.11
	Answer	0.74 ± 0.08	0.44 ± 0.11	1 ± 0	0.6 ± 0.11
	Code	0.98 ± 0.04	/	/	/
	Overall	**0.16 ± 0.07**	**0.35 ± 0.16**	**0.22 ± 0.09**	**0.27 ± 0.11**

3.2 LLM-Based Evaluation

The LLM-based evaluation showed similar results to that of the manual evaluation, with similar trends across specific and general questions (see Tab. 4). For the specific questions, the accuracy of the retrieved context and the overall answer were close to chance, while the answer based on the context and the code showed a high accuracy. Precision of the context and context-based answer both were high, while the precision of the overall answer was above chance. Recall of the retrieved context was near chance, while the answer based on the context had a high recall. The recall of the overall answer was good. The F1 score of the context-based answer was very high, while the retrieved context and overall answer had a good F1 score.

The results within the general questions showed low accuracy, recall and F1 score for the retrieved context. The answer based on the context had good accuracy, precision and F1 score, with a slightly lower recall. The accuracy, recall and F1 score of the overall answer were low, while precision was slightly above chance. The code showed perfect mean accuracy (see Tab. 4).

In general it can be said that the specific questions lead to a much better performance regarding accuracy, recall and F1 score of the context and overall answer, while the accuracy, recall and F1 score of the answer based on the context were also higher in the specific questions, but not that extreme.

Table 4: Result of the evaluation of 100 repetitions of the 32 testset questions by the LLM Falcon3. The metrics represent mean values with the standard deviation.

Type	Domain	Accuracy	Precision	Recall	F1
Specific	Context	0.5 ± 0.08	1 ± 0.02	0.52 ± 0.08	0.68 ± 0.07
	Answer	0.88 ± 0.08	0.93 ± 0.06	0.91 ± 0.08	0.92 ± 0.05
	Code	0.99 ± 0.03	/	/	/
	Overall	**0.59 ± 0.09**	**0.64 ± 0.1**	**0.77 ± 0.1**	**0.67 ± 0.08**
General	Context	0.13 ± 0.06	1 ± 0	0.13 ± 0.06	0.23 ± 0.09
	Answer	0.73 ± 0.1	0.84 ± 0.14	0.64 ± 0.13	0.72 ± 0.12
	Code	1 ± 0.01	/	/	/
	Overall	**0.2 ± 0.08**	**0.53 ± 0.2**	**0.23 ± 0.07**	**0.3 ± 0.1**

3.3 Comparison Manual and LLM

In order to directly compare the results of the manual and automated evaluation by the LLM, we also calculated the metrics of the LLM evaluation solely based on the questions where a manual evaluation was available.

In general, manual and LLM evaluation showed a good agreement. For the specific questions, manual and LLM mean accuracy were similar for all domains, while the precision of the LLM results was slightly higher for the retrieved context and answer based on the context, but slightly lower for the overall answer. Recall and F1 score were also slightly lower for the LLM evaluation than for the manual one. When it comes to the general questions, the LLM evaluation yielded in higher accuracy and precision than the manual one, while recall and F1 score were very similar (see Tab. 3, Tab. 5)

Tab. 5: Result of the evaluation of the manually evaluated 20 repetitions of the 32 testset questions by the LLM Falcon3. The metrics represent mean values with the standard deviation.

Type	Domain	Accuracy	Precision	Recall	F1
Specific	Context	0.51 ± 0.07	0.98 ± 0.04	0.52 ± 0.06	0.68 ± 0.06
	Answer	0.87 ± 0.07	0.94 ± 0.05	0.9 ± 0.08	0.92 ± 0.04
	Code	1 ± 0.03	/	/	/
	Overall	**0.58 ± 0.09**	**0.65 ± 0.09**	**0.75 ± 0.09**	**0.69 ± 0.08**
General	Context	0.13 ± 0.07	1 ± 0	0.13 ± 0.07	0.22 ± 0.1
	Answer	0.69 ± 0.12	0.81 ± 0.14	0.61 ± 0.15	0.69 ± 0.13
	Code	1 ± 0	/	/	/
	Overall	**0.21 ± 0.07**	**0.51 ± 0.23**	**0.23 ± 0.07**	**0.3 ± 0.1**

4 Discussion

In this study we aimed to develop a Graph RAG system for the CDPKit, enabling question answering about its structure and usage, as well as generating code snippets in Python. The system was evaluated on a set of specific and general test questions using both a human evaluator and an LLM. We found that the performance of the system greatly depended on the type of questions, with specific questions leading to better results than general ones. The code snippets provided by the generator showed a consistently high syntactical accuracy.

In detail, we found that the accuracy for specific questions was around ten percent above chance for the retrieved context and the overall answer, while the code and answer based on the context showed a high accuracy. Further, the retrieved context showed the lowest recall within the evaluated components. This indicates that the performance of the overall answer of the system is limited by the retrieved context. To interpret this, it is important to remember that we translated the user query into a Cypher query, directly querying the Neo4j Knowledge Graph instead of embedding it into a vector index. Therefore, the accuracy and recall of the retrieved context were entirely dependent on the accuracy of the Cypher query. The difference to a classical similarity search here is that the direction and naming of the relationships and nodes is crucial for a correct result, and slight inaccuracies in the Cypher query will lead to completely incorrect results. The high accuracy, recall and precision of the answer strictly based on the context showed that the system is likely to perform well if the provided reference context is correct.

This also becomes apparent in the general questions, where we saw a similar pattern, but with overall poorer performance. Here accuracy and recall of the retrieved context are low, resulting in a low overall accuracy and recall. This, again, can be attributed to a poor performance of the translation of the user query into Cypher. When no specific class or function names were provided, CodeLlama struggled to formulate accurate Cypher queries, often trying to reference nonexistent classes or functions. Interestingly, in the manual evaluation, the precision of the answer based on the context also drops with the general questions. This seems to be

due to the fact that the generated answer could be influenced by an incorrect Cypher query, referencing non-existent objects that were mentioned in the Cypher query.

When comparing the manual evaluation with that of the LLM directly, it appears that they agreed on the accuracy, recall and F1 score. For precision, the LLM slightly underestimated the results of the specific questions and slightly overestimated that of the general ones. Therefore, we conclude that in this case, evaluation by the LLM led to reliable results, making it a valuable alternative to costly evaluation by human reviewers.

For the code example snippets, the syntactical accuracy is quite high. When looking at the provided code snippets, however, they were generally superficial, only showcasing basic use of the retrieved context. Often, they were not immediately executable, as they included placeholders or lacked the correct import statements, therefore only providing basic support to a programmer.

To contextualize the system's performance, we can compare its F1 scores to that of other RAG systems. SubgraphRAG, for example, reports F1 scores from 66% to 78%, which is similar to the scores of our system [32]. KnowledGPT on the other hand reports F1 scores of up to 93%, clearly outperforming our system [33]. However, comparisons of the performance of different RAG systems are of limited meaning when not executed on the same benchmark questions.

We think that this Graph RAG system is proficient at answering specific questions about the CDPKit and providing basic code snippets. However, it needs further development to be able to answer general open questions reliably and provide extensive code generation. One idea to improve the system would be to try out classical knowledge graph embedding and similarity search, which could mitigate the issues of the incorrect Cypher queries. To increase the complexity of the provided code, we suggest adding code samples to the Knowledge graph node properties. Another possible future development of the graph RAG system might involve enabling more complex retrieving techniques and enabling the system to decide whether the input user query needs to be translated into a Cypher query or should be sent directly to the generator.

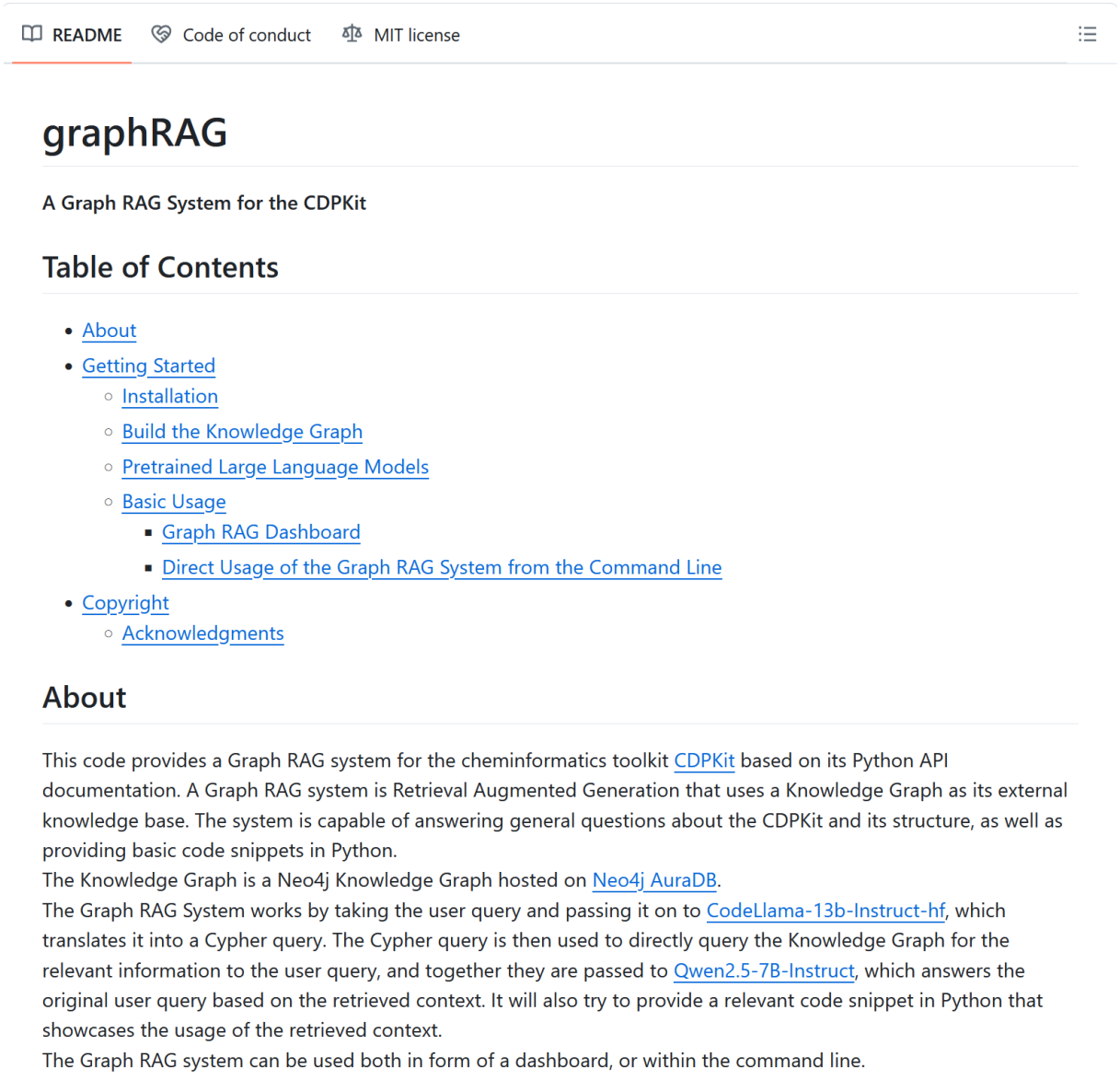
In summary, this study provides a Graph RAG system that enables specific question answering and basic code generation for the CDPKit. It has been made

accessible to users via a chat-bot like dashboard and provides opportunities for further development and expansion of its functionalities.

5 Code

5.1 Accessibility

The code for this thesis is available publicly on GitHub under the following link: https://github.com/molinfo-vienna/graphRAG. It was written by the author of this thesis, Selina Schöndorfer. The GitHub repository includes a README.md, which introduces the Graph RAG system and explains how to install the necessary dependencies, create the knowledge graph and run the system as a dashboard or within the command line.



README Code of conduct MIT license

graphRAG

A Graph RAG System for the CDPKit

Table of Contents

About

This code provides a Graph RAG system for the cheminformatics toolkit CDPKit based on its Python API documentation. A Graph RAG system is Retrieval Augmented Generation that uses a Knowledge Graph as its external knowledge base. The system is capable of answering general questions about the CDPKit and its structure, as well as providing basic code snippets in Python.
The Knowledge Graph is a Neo4j Knowledge Graph hosted on Neo4j AuraDB.
The Graph RAG System works by taking the user query and passing it on to CodeLlama-13b-Instruct-hf, which translates it into a Cypher query. The Cypher query is then used to directly query the Knowledge Graph for the relevant information to the user query, and together they are passed to Qwen2.5-7B-Instruct, which answers the original user query based on the retrieved context. It will also try to provide a relevant code snippet in Python that showcases the usage of the retrieved context.
The Graph RAG system can be used both in form of a dashboard, or within the command line.

Getting started

This short guide will help you get started with the Graph RAG system.

Installation

Run the following command to copy the repository to your local system and to install the necessary dependencies:

```
git clone https://github.com/molinfo-vienna/graphRAG.git
cd graphRAG
pip install -r requirements.txt
```

Build the Knowledge Graph

This step can be skipped if you already have your Neo4j AuraDB instance with the Knowledge Graph.
Otherwise, create an account and AuraDB instance here.
Once this is done, set your Neo4j uri, user and password as environment variables. To do so once for the current shell and all processes started from it, execute

```
export NEO4j_URI="yourvalue"
export NEO4j_USER="yourvalue"
export NEO4j_PASSWORD="yourvalue"
```

If you want to add this to your environment permanently for all future bash sessions as well (recommended), go to your $HOME directory and add these lines to your .bashrc file. Your Neo4j credentials will also be need to be set as an environment variable for later use of the Graph RAG.
Next, you need to copy the CDPKit Python API documentation to a local directory and again set the path to the documentation as an environment variable:

```
export CDPKit_PATH="path/to/your/directory/CDPKit/Doc/Doxygen/Python-API/Source/CDPL/"
```

To create the Knowledge Graph in your AuraDB instance, make sure to set all necessary environment variables and then run:

```
python knowledgeGraph/KnowledgeGraphManager.py
```

Pretrained Large Language Models

The Graph RAG system works by using pretrained LLMs from huggingface. Set

```
export MODEL_LOCATION="/path/to/local/directory"
```

in your .basrc file as the location where the pretrained models should be downloaded and stored. This will take around **40GB** of space on your disk. When running the Graph RAG system for the first time, it will download the models to your specified location once.
Further, to ensure optimal performance, it is recommended to run the Graph RAG system on a machine that is equipped with one or more GPUs.

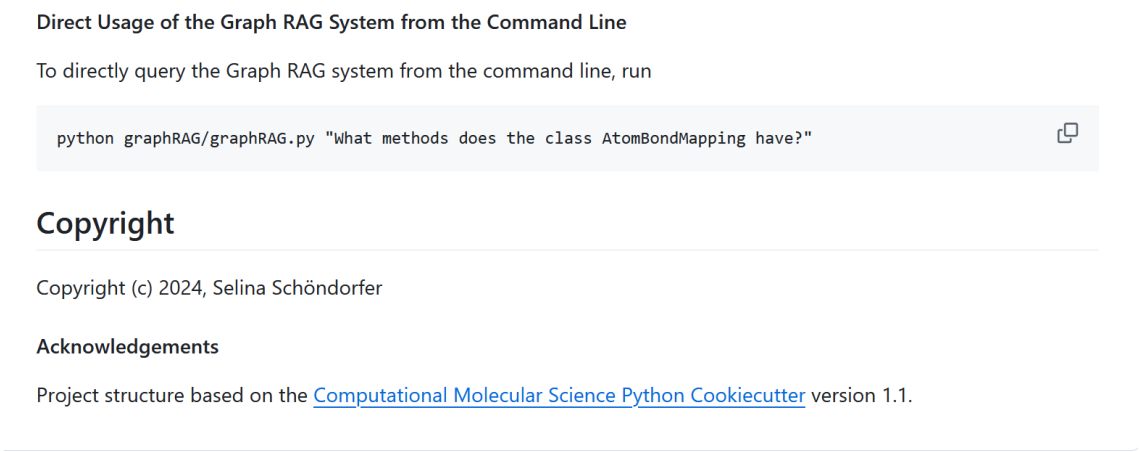
Basic Usage

Graph RAG Dashboard

Once the Knowledge Graph is created and all variables have been set accordingly, this repository provides access to user friendly Dash dashboard that can be hosted by running

```
python graphRAG/graphRag_dashboard.py
```

It has a chat-like interface with example questions and can be used to directly question the system about the CDPKit.



Direct Usage of the Graph RAG System from the Command Line

To directly query the Graph RAG system from the command line, run

```
python graphRAG/graphRAG.py "What methods does the class AtomBondMapping have?"
```

Copyright

Copyright (c) 2024, Selina Schöndorfer

Acknowledgements

Project structure based on the Computational Molecular Science Python Cookiecutter version 1.1.

Figure 8: README.md file of the Graph RAG systems GitHub repository.

5.2 Implementation

Parsing the Documentation

To parse the CDPKit Python API documentation, we implemented the `DocParser` class (see Fig. A1). This class requires the path to the target directory as an input parameter upon initialization and stores the parsed info into a dictionary. Its key methods include:

- `parse_dir`: Extracts .doc files and the folder name from the specified directory into a list, then calls `parse_files`.
- `parse_files`: Reads the content of each .doc file into a string, extracts comments via `extract_comments`, and parses the file into an Abstract Syntax Tree (AST). It then uses `ClassAndFunctionVisitor` to process the parsed comments and visit the AST, storing the returned information a dictionary.
- `extract_comments`: Extracts comments from a string.

Additionally, `DocParser` includes helper methods to correct syntax errors in the .doc files.

The other important class to parse the documentation was the `ClassAndFunctionVisitor` class (see Fig. A2), which inherits from the class `NodeVisitor` from the ast package [40]. Its key methods include:

- `visit_ClassDef`: Processes class nodes using `parse_class`, visits the child nodes and tracks nested class hierarchies.
- `visit_FunctionDef`: Processes function nodes that are not methods of a class by using `parse_function` and visits the child nodes.
- `parse_class`: Collects the class information (name, bases, decorators, methods, attributes, nested classes and associated comments) by using `parse_comments` and `traverse_body` to gather the details.
- `traverse_body`: Parses a class node's child nodes depending on their type by calling relevant methods like `parse_function` for methods, `parse_attributes` for class attributes and `parse_class` for nested classes.
- `parse_comments`: Extracts comments based on tags that indicate their corresponding object type and on line numbers.
- `parse_function`: Parses function information (name, parameters, decorators, return type, comments).
- `parse_attributes`: Parses class attribute information (name, value, associated comments).
- `parse_parameters`: Parses parameter information (name, type, default value, comments).

Knowledge Graph

The Neo4j knowledge graph was built using several manager classes, each responsible for handling specific types of nodes and relationships (see Fig. 2). The central class, `KnowledgeGraphManager` (see Fig. A3), coordinates the submanagers and provides them with parsed information from the `DocParser`. Its key function is:

- `create_graph`: Invokes the create methods of `ProjectManager`, `FolderManager`, `FileManager`, `ClassManager` and `FunctionManager`.

The submanager classes and their key functions are:

- `ProjectManager` (see Fig. A4):
  - `_create_project_node`: Creates project node using its name.
- `FolderManager` (see Fig. A5):
  - `_create_folder_nodes`: Creates folder node using its name.
  - `_create_folder_relationships`: Creates an INCLUDED_IN relationship between a project and its folder.
- `FileManager` (see Fig. A6):
  - `_create_file_nodes`: Creates the file node based on name.
  - `_create_file_relationships`: Creates an INCLUDED_IN relationship between a folder and its file.
- `ClassManager` (see Fig. A7):
  - `_create_class_node`: Creates the class node based on name.
  - `_create_class_inheritance_relationship`: Creates an INHERITS_FROM relationship between classes.
  - `_create_class_file_relationship`: Creates a DECLARED_AT relationship between a file and its classes.
  - `_create_nested_class_relationship`: Creates a HAS relationship between a nested class and its parent.
  - `_create_class_relationship`: Manages the creation of class nodes and all associated relationships, including nodes and relationships for methods.
- `FunctionManager` (see Fig. A8):
  - `_create_function_node`: Creates a function node based on its name, comment, input parameters, decorators and return types to differentiate between functions with the same name.
  - `_create_function_file_relationship`: Creates a DECLARED_AT relationship between a file and a function.

    - `_create_function_class_relationship`: Creates a HAS relationship between a class and its method.
    - `_create_function_inputs`: Creates the nodes and relationships of the function input parameters and decorators by invoking the corresponding methods of their classes.
- `ParameterManager` (see Fig. A9):
    - `_create_parameter_node`: Creates a parameter node based on its name, comment, type and default value.
    - `_create_parameter_function_relationship`: Creates a HAS relationship between a function and its parameters.
- `TypeManager` (see Fig. A10):
    - `_create_type_relationship`: Creates an OF_TYPE relationship between a parameter and a class.
- `DecoratorManager` (see Fig. A11):
    - `_create_decorator_node`: Creates a decorator node based on its name.
    - `_create_decorator_class_relationship`: Creates a HAS relationship between a class and its decorator.
    - `_create_decorator_function_relationship`: Creates a HAS relationship between a function and its decorator.

Graph RAG System

The Graph RAG system was designed to be runnable from the command line by executing the graphRag.py file with a user query as a command line argument. This file initializes the pipelines for both LLMs and passes the user query along with the pipelines to the key function (see Fig. A12):

- `question_rag`: Processes the user query and pipelines by calling `retrieve_context` to get the translated Cypher query and the retrieved context. It then uses `generate_rag_prompt` to create the system prompt, which is passed to `generate_answer_qwen` for generating the final answer.

The function returns the Cypher query, the retrieved context and the generated answer to the user query.

To retrieve the relevant context for the user query, the following key functions were used (see Fig. A13):

- `retrieve_context`: Passes the user query to `get_cypher_query`, then runs the resulting Cypher query using `run_query`.
- `get_cypher_query`: Uses `generate_cypher_query_prompt` and passes the generated prompt to `generate_answer_code_llama` to generate the Cypher query.
- `generate_cypher_query_prompt`: Creates an instructional prompt for CodeLlama to translate the user query into a Cypher query.
- `generate_answer_code_llama`: Combines the user query and system prompt, formats them for CodeLlama, and parses the output to return only the translated Cypher query.

To generate the final answer to the user query, the following key functions were implemented (see Fig. A14):

- `generate_rag_prompt`: Creates a prompt for Qwen by combining the Cypher query, retrieved context, and positive examples to instruct it to generate an answer based strictly on the context.
- `generate_answer_qwen`: Combines the user query and system prompt, passes them to the Qwen pipeline and returns the generated answer.

The Graph RAG system also includes several utility functions that were reused across the main functions (see Fig. A15). To host the Graph RAG dashboard, the file graphRag_dashboard.py can be executed (see Fig. A20).

Evaluation

The function `benchmark_rag` was used to run the Graph RAG system on the test set questions (see Fig. A12):

- `benchmark_rag`: Parses the test set questions and passes them to `question_rag` a hundred times. For each iteration, it saves the user

prompt, retrieved context, final answer, Cypher query, a model generated context, model answer, and placeholders for scores into dictionaries. These results are stored as JSON files.

To score the answers to the test set questions with an LLM, the key functions used were (see Fig. A16):

- `testset_scoring_llm`: Iterates through the test set results files, gets the necessary prompts and passes them to `generate_answer_falcon`. The scores are appended to the result files and saved in the corresponding JSON file.
- `generate_answer_falcon`: Passes the appropriate prompt to the pipeline and returns the generated score.

Utility functions also supported the scoring process (see Fig. A17).

To evaluate the scores of the test set results, utility functions (see Fig. A18) supported the following main functions (see Fig. A19):

- `testset_evaluation`: Computes metrics such as Accuracy, Precision, Recall, and F1 Score for various evaluation categories (e.g., retrieved context, generated answer, code) based on manual or automated scoring.
- `compare_manual_automated`: Compares the results of the automated scoring with that of the manual ones for each category.

References

[1] OpenAI *et al.*, “GPT-4 Technical Report,” Mar. 04, 2024, *arXiv*: arXiv:2303.08774. doi: 10.48550/arXiv.2303.08774.

[2] P. Jackson and I. Moulinier, *Natural Language Processing for Online Applications : Text Retrieval, Extraction, and Categorization*, no. v. 5. in Natural Language Processing. Amsterdam: John Benjamins Publishing Co, 2002.

[3] C. Dong *et al.*, “A Survey of Natural Language Generation,” *ACM Comput Surv*, vol. 55, no. 8, p. 173:1-173:38, Dezember 2022, doi: 10.1145/3554727.

[4] J. Hirschberg and C. D. Manning, “Advances in natural language processing,” *Science*, vol. 349, no. 6245, pp. 261–266, Jul. 2015, doi: 10.1126/science.aaa8685.

[5] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving Language Understanding by Generative Pre-Training,” 2018.

[6] J. J. Webster and C. Kit, “Tokenization as the initial phase in NLP,” in *Proceedings of the 14th conference on Computational linguistics*, Nantes, France: Association for Computational Linguistics, 1992, p. 1106. doi: 10.3115/992424.992434.

[7] S. J. Mielke *et al.*, “Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP,” Dec. 20, 2021, *arXiv*: arXiv:2112.10508. doi: 10.48550/arXiv.2112.10508.

[8] Q. Liu, M. J. Kusner, and P. Blunsom, “A Survey on Contextual Embeddings,” Apr. 13, 2020, *arXiv*: arXiv:2003.07278. doi: 10.48550/arXiv.2003.07278.

[9] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient Estimation of Word Representations in Vector Space”.

[10] A. Vaswani *et al.*, “Attention is All you Need,” *Adv. Neural - Form. Process. Syst. 30 Annu. Conf. Neural Inf. Process. Syst.*, 2017.

[11] W. X. Zhao *et al.*, “A Survey of Large Language Models,” Oct. 13, 2024, *arXiv*: arXiv:2303.18223. doi: 10.48550/arXiv.2303.18223.

[12] F. M. Shiri, T. Perumal, N. Mustapha, and R. Mohamed, “A Comprehensive Overview and Comparative Analysis on Deep Learning Models: CNN, RNN, LSTM, GRU,” Oct. 24, 2024, *arXiv*: arXiv:2305.17473. doi: 10.48550/arXiv.2305.17473.

[13] H. Naveed *et al.*, “A Comprehensive Overview of Large Language Models,” Oct.

17, 2024, *arXiv*: arXiv:2307.06435. doi: 10.48550/arXiv.2307.06435.

[14] B. Chen, Z. Zhang, N. Langrené, and S. Zhu, “Unleashing the potential of prompt engineering in Large Language Models: a comprehensive review,” Sep. 05, 2024, *arXiv*: arXiv:2310.14735. doi: 10.48550/arXiv.2310.14735.

[15] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models,” in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, in CHI EA ’22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 1–7. doi: 10.1145/3491101.3519665.

[16] J. Kaddour, J. Harris, M. Mozes, H. Bradley, R. Raileanu, and R. McHardy, “Challenges and Applications of Large Language Models,” Jul. 19, 2023, *arXiv*: arXiv:2307.10169. doi: 10.48550/arXiv.2307.10169.

[17] P. Lewis *et al.*, “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2020, pp. 9459–9474. Accessed: Dec. 01, 2024. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/6b493230205f780e1bc26945df7481e5-Abstract.html

[18] Y. Gao *et al.*, “Retrieval-Augmented Generation for Large Language Models: A Survey,” Mar. 27, 2024, *arXiv*: arXiv:2312.10997. doi: 10.48550/arXiv.2312.10997.

[19] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, “Retrieval Augmentation Reduces Hallucination in Conversation,” Apr. 15, 2021, *arXiv*: arXiv:2104.07567. doi: 10.48550/arXiv.2104.07567.

[20] H. Yu, A. Gan, K. Zhang, S. Tong, Q. Liu, and Z. Liu, “Evaluation of Retrieval-Augmented Generation: A Survey,” Jul. 03, 2024, *arXiv*: arXiv:2405.07437. doi: 10.48550/arXiv.2405.07437.

[21] H. Bahak, F. Taheri, Z. Zojaji, and A. Kazemi, “Evaluating ChatGPT as a Question Answering System: A Comprehensive Analysis and Comparison with Existing Models,” Dec. 11, 2023, *arXiv*: arXiv:2312.07592. doi: 10.48550/arXiv.2312.07592.

[22] T. T. Procko and O. Ochoa, “Graph Retrieval-Augmented Generation for Large Language Models: A Survey,” in *2024 Conference on AI, Science, Engineering, and Technology (AIxSET)*, Sep. 2024, pp. 166–169. doi:

10.1109/AIxSET62544.2024.00030.

[23] S. Ji, S. Pan, E. Cambria, P. Marttinen, and P. S. Yu, “A Survey on Knowledge Graphs: Representation, Acquisition, and Applications,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 2, pp. 494–514, Feb. 2022, doi: 10.1109/TNNLS.2021.3070843.

[24] Z. Li, H. Liu, Z. Zhang, T. Liu, and N. N. Xiong, “Learning Knowledge Graph Embedding With Heterogeneous Relation Attention Networks,” *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 33, no. 8, pp. 3961–3973, Aug. 2022, doi: 10.1109/TNNLS.2021.3055147.

[25] Q. Wang, Z. Mao, B. Wang, and L. Guo, “Knowledge Graph Embedding: A Survey of Approaches and Applications,” *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 12, pp. 2724–2743, Dec. 2017, doi: 10.1109/TKDE.2017.2754499.

[26] A. Bordes, N. Usunier, A. Garcia-Duran, J. Weston, and O. Yakhnenko, “Translating Embeddings for Modeling Multi-relational Data,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2013. Accessed: Dec. 14, 2024. [Online]. Available: https://proceedings.neurips.cc/paper/2013/hash/1cecc7a77928ca8133fa24680a88d2f9-Abstract.html

[27] “NebulaGraph Launches Industry-First Graph RAG: Retrieval-Augmented Generation with LLM Based on Knowledge Graphs.” Accessed: Dec. 14, 2024. [Online]. Available: https://www.nebula-graph.io/posts/graph-RAG

[28] Z. Z. Wang *et al.*, “CodeRAG-Bench: Can Retrieval Augment Code Generation?,” Jun. 20, 2024, *arXiv*: arXiv:2406.14497. doi: 10.48550/arXiv.2406.14497.

[29] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, “Retrieval Augmented Code Generation and Summarization,” Sep. 10, 2021, *arXiv*: arXiv:2108.11601. doi: 10.48550/arXiv.2108.11601.

[30] S. Zhou, U. Alon, F. F. Xu, Z. Wang, Z. Jiang, and G. Neubig, “DocPrompting: Generating Code by Retrieving the Docs,” Feb. 18, 2023, *arXiv*: arXiv:2207.05987. doi: 10.48550/arXiv.2207.05987.

[31] I. Abdelaziz, J. Dolby, J. McCusker, and K. Srinivas, “A Toolkit for Generating Code Knowledge Graphs,” 2020, *arXiv*. doi: 10.48550/ARXIV.2002.09440.

[32] M. Li, S. Miao, and P. Li, “Simple is Effective: The Roles of Graphs and Large Language Models in Knowledge-Graph-Based Retrieval-Augmented

Generation," Nov. 11, 2024, *arXiv*: arXiv:2410.20724. doi: 10.48550/arXiv.2410.20724.

[33] X. Wang *et al.*, "KnowledGPT: Enhancing Large Language Models with Retrieval and Storage Access on Knowledge Bases," Aug. 17, 2023, *arXiv*: arXiv:2308.11761. doi: 10.48550/arXiv.2308.11761.

[34] I. Saberi and F. Fard, "Context-Augmented Code Generation Using Programming Knowledge Graphs," Oct. 09, 2024, *arXiv*: arXiv:2410.18251. doi: 10.48550/arXiv.2410.18251.

[35] T. Seidel and O. Wieder, "Chemical Data Processing Toolkit documentation pages." Accessed: Jan. 06, 2025. [Online]. Available: https://cdpkit.org

[36] "Knowledge Graph," Graph Database & Analytics. Accessed: Jan. 06, 2025. [Online]. Available: https://neo4j.com/use-cases/knowledge-graph/

[37] Seidel, "Chemical Data Processing Toolkit source code repository." Accessed: Jan. 06, 2025. [Online]. Available: https://github.com/molinfo-vienna/CDPKit

[38] "What is Cypher - Getting Started," Neo4j Graph Data Platform. Accessed: Jan. 06, 2025. [Online]. Available: https://neo4j.com/docs/getting-started/cypher/

[39] "tokenize — Tokenizer for Python source," Python documentation. Accessed: Dec. 15, 2024. [Online]. Available: https://docs.python.org/3/library/tokenize.html

[40] "ast — Abstract Syntax Trees," Python documentation. Accessed: Jan. 06, 2025. [Online]. Available: https://docs.python.org/3/library/ast.html

[41] J. L. Espejel, "Basic understanding of Abstract Syntax Tree (AST)," Medium. Accessed: Dec. 15, 2024. [Online]. Available: https://medium.com/@jessica_lopez/basic-understanding-of-abstract-syntax-tree-ast-d40ff911c3bf

[42] V. Batushkov, "Codebase Knowledge Graph," Neo4j Developer Blog. Accessed: Jan. 06, 2025. [Online]. Available: https://medium.com/neo4j/codebase-knowledge-graph-204f32b58813

[43] "Fully Managed Graph Database Service | Neo4j AuraDB." Accessed: Jan. 06, 2025. [Online]. Available: https://neo4j.com/product/auradb/

[44] "Pipelines." Accessed: Jan. 06, 2025. [Online]. Available: https://huggingface.co/docs/transformers/main_classes/pipelines

[45] "codellama/CodeLlama-13b-Instruct-hf · Hugging Face." Accessed: Jan. 06, 2025. [Online]. Available: https://huggingface.co/codellama/CodeLlama-13b-Instruct-hf

[46] S. Onofrei, “Code Llama’s ‘Knowledge’ of Neo4j’s Cypher Query Language,” Medium. Accessed: Jan. 06, 2025. [Online]. Available: https://medium.com/@silviaonofrei/code-llamas-knowledge-of-neo4j-s-cypher-query-language-54783d2ad421

[47] “Qwen/Qwen2.5-7B-Instruct · Hugging Face.” Accessed: Jan. 06, 2025. [Online]. Available: https://huggingface.co/Qwen/Qwen2.5-7B-Instruct

[48] “tiiuae/Falcon3-10B-Instruct · Hugging Face.” Accessed: Jan. 11, 2025. [Online]. Available: https://huggingface.co/tiiuae/Falcon3-10B-Instruct

[49] “Dash Documentation & User Guide | Plotly.” Accessed: Jan. 13, 2025. [Online]. Available: https://dash.plotly.com/

List of Tables

List of Figures

Abbreviations

API	Application Programming Interface
AST	Abstract Syntax Tree
LLM	Large Language Model
NLP	Natural Language Processing
PLM	Pre-trained Language Model
RAG	Retrieval Augmented Generation

Appendix

```
import os
import glob
import ast
import re
import tokenize
from io import StringIO
from ClassAndFunctionVisitor import ClassAndFunctionVisitor

class DocParser():

    def __init__(self, dir_path: str, all_files_info: dict = {}) -> None:
        self.dir_path = dir_path    # path to directory that should be parsed
        self.all_files_info = all_files_info    # dict with parsed information

    def parse_dir(self) -> dict:
        # parses files within the dir_path into the all_files_info_dict
        file_pattern = os.path.join(self.dir_path, '*.doc.py')  # only parse .doc.py files
        file_list = glob.glob(file_pattern)
        folder = self.extract_cdpl_substring(self.dir_path)
        self.all_files_info[folder] = {}    # the folder names are the keys
        self.parse_files(file_list, folder)
        return self.all_files_info

    def parse_files(self, file_list: list, folder: str) -> None:
        # parse classes and functions from files
        unreadable_files = 0    # Some files contain syntax errors
        for file_path in file_list: # Loop through each file in the list
            with open(file_path, 'r') as f: # Open and read the text file
                content = f.read()
                comments = self.extract_comments(content)
            try:
                tree = ast.parse(content) # get an AST tree from the content of the doc.py file
            except SyntaxError as e:
                try:
                    content = self.clean_unreadable_text(content)   # there are syntax errors in the documentation
                    tree = ast.parse(content)
                except SyntaxError as e:
                    print(f'Syntax error in file {file_path}: {e}')
                    unreadable_files = unreadable_files + 1
                    continue

            visitor = ClassAndFunctionVisitor(comments) # Create an instance of the ClassAndFunctionVisitor and visit the AST
            visitor.visit(tree)
            file_name = self.extract_doc_py_filename(file_path) # get the name of the file
            self.all_files_info[folder][file_name] = {"classes": visitor.classes, "functions": visitor.functions }
        print(f"Amount of files that could not be parsed: {unreadable_files}")


    def extract_comments(self, text: str) -> list:
        # Identifies the comments in the text and returns them
        comments = []
        tokens = tokenize.generate_tokens(StringIO(text).readline)
        for tok_type, tok_string, start, _, _ in tokens:
            if tok_type == tokenize.COMMENT:
                comments.append((start[0], tok_string.strip()))
        return comments
```

```python
    def clean_unreadable_text(self, text: str) -> str:
        # trys out all the encountered syntax errors and attempts to solve them
        text = self.replace_naming_clash(text)
        text = self.remove_empty_colon_parameters(text)
        text = self.insert_default_strings(text)
        return text

    def extract_cdpl_substring(self, path: str) -> str | None:
        # Regular expression to find the substring after 'CDPL/' until the next '/'
        match = re.search(r'CDPL/([^/]+)', path)
        if match:
            return match.group(1)
        return None

    def extract_doc_py_filename(self, path: str) -> str | None:
        # Regular expression to find filenames that end with '.doc.py'
        match = re.search(r'([^/]+\.doc\.py)$', path)
        if match:
            return match.group(1)
        return None

    def replace_naming_clash(self, text: str) -> str:
        # Regular expression to find 'is' within parentheses and replace it with 'stream'
        return re.sub(r'\(([^)]*?)\bis\b([^)]*?)\)', lambda match: f'({match.group(1)}stream{match.group(2)})', text)

    def remove_empty_colon_parameters(self, text: str) -> str:
        # Remove function parameters that are just a colon
        return re.sub(r'\(\s*:\s*\)', '()', text)
```

```python
    def insert_default_strings(self, text: str) -> str:
        # Insert an = where there are default parameters with '' directly after the parameter name
        def replacer(match):
            before = match.group(1)
            default_str = match.group(2)
            after = match.group(3)
            return f"({before}={default_str}{after})"
        return re.sub(r"\(\s*([^)]*?mime_type)('[^=)]*?')([^)]*?)\)", replacer, text)
```

Figure A1: `DocParser` *class.*

```python
import ast


# Visitor class that will collect class and function information from the AST
class ClassAndFunctionVisitor(ast.NodeVisitor):
    def __init__(self, comments: list) -> None:
        self.classes = []
        self.functions = []
        self.comments = comments
        self.class_stack = [] # stack to keep track of nested classes
        super().__init__() # inherits the base functionalities of ast.NodeVisitor

    def get_name(self, node) -> str:
        # parses the name of the node depending on the node type
        if isinstance(node, ast.Name):
            name = node.id
        elif isinstance(node, ast.Constant):
            name =str(node.value)
        elif isinstance(node, ast.Attribute):
            name = self.get_name(node.value) + "." + node.attr
        elif isinstance(node, ast.Call):
            name = {"callable": self.get_name(node.func), "arguments": [self.get_name(arg) for arg in node.args]}
        else:
            name = "No value"
        return name
```

```
def parse_parameters(self, node, comments: list) -> list:
    # parses the input parameters of a function
    default_values = [self.get_name(d) for d in node.args.defaults]
    params_with_defaults = ["No default"] * (len(node.args.args) - len(default_values)) + default_values    # add default values
    params = []
    for i, arg in enumerate(node.args.args):
        param_info = {
            "name": arg.arg,
            "type": self.get_name(arg.annotation),
            "default": params_with_defaults[i],
            "comment": comments.get(arg.arg, "")    # if there is no comment, an empty string is added
        }
        params.append(param_info)
    return params


def parse_function(self, node) -> dict:
    parsed_comments = self.parse_comments(node.lineno)
    return_type = self.get_name(node.returns)
    params = self.parse_parameters(node, parsed_comments["param"])
    return {
        'name': node.name,
        'params': params,
        "decorators": [self.get_name(d) for d in node.decorator_list],
        "return_type": {"type": return_type, "comment": parsed_comments.get("return", "")},
        "comment": parsed_comments.get("brief", "")
    }

def parse_attribute(self, node) -> dict:
    parsed_comments = self.parse_comments(node.lineno)
    return {
        "name": node.targets[0].id,
        "value": self.get_name(node.value),
        "comment": parsed_comments.get("brief", "")
    }
```

```
def traverse_body(self, info: dict, node):
    # traverses the body of a node and parses the child nodes
    for elem in node.body:
        if isinstance(elem, ast.FunctionDef):
            info['methods'].append(self.parse_function(elem))
        elif isinstance(elem, ast.Assign):
            info['class_attributes'].append(self.parse_attribute(elem))
        elif isinstance(elem, ast.ClassDef):
            nested_class_info = self.parse_class(elem)
            info['nested_classes'].append(nested_class_info)

def get_associated_comments(self, lineno: int) -> list:
    # gets the comments that belong to a specific node within the AST
    associated_comments = []
    for line, comment in reversed(self.comments):   # start from the bottom
        if line < lineno:   # look at the comments directly above the class/function etc.
            if comment.startswith("##"): # marks the end of the comment belonging to this node
                break
            if comment == "#":
                continue
            associated_comments.append(comment.strip("#").strip())  # remove the unnecessary whitespace and #
    associated_comments.reverse()   # need to reverse as we started from the bottom
    return associated_comments
```

```python
def parse_class(self, node) -> dict:
    parsed_comments = self.parse_comments(node.lineno)  # get associated comments
    class_info = {
        'name': node.name,
        "bases": [self.get_name(b) for b in node.bases],    # base it inherits from
        "decorators": [self.get_name(d) for d in node.decorator_list],
        'methods': [],
        "class_attributes": [],
        "nested_classes": [],
        "comment": parsed_comments.get("brief", "")
    }
    self.traverse_body(class_info, node)
    return class_info

def visit_ClassDef(self, node) -> None:
    # needed to overwrite the default function from ast
    self.class_stack.append(node.name)  # handles nested classes
    class_info = self.parse_class(node)
    if len(self.class_stack) == 1:
        self.classes.append(class_info)
    self.generic_visit(node)
    self.class_stack.pop()

def visit_FunctionDef(self, node) -> None:
    # needed to overwrite the default function from ast
    if not self.class_stack:
        self.functions.append(self.parse_function(node))
    self.generic_visit(node)

def parse_comments(self, lineno: int) -> dict:
    associated_comments = self.get_associated_comments(lineno=lineno)
    parsed_comments = {"param": {}}
    current_tag = None
    current_param = None

    for comment in associated_comments:
        # the comments start with certain tags that indicate what they will describe
        if comment.startswith("\\brief"):
            current_tag = "brief"
            parsed_comments["brief"] = comment.strip("\\brief").strip()

        elif comment.startswith("\\param"):
            current_tag = "param"
            if len(comment.split()) > 1:
                current_param = comment.split()[1]
            else:
                continue
            parsed_comments["param"][current_param] = comment.strip(f"\\param {current_param}").strip()

        elif comment.startswith("\\return"):
            current_tag = "return"
            parsed_comments["return"] = comment.strip("\\return").strip()

        else:
            # comments will go over multiple lines
            if current_tag == "brief":
                parsed_comments["brief"] += " " + comment.strip()
            elif current_tag == "param" and current_param:
                parsed_comments["param"][current_param] += " " + comment.strip()
            elif current_tag == "return":
                parsed_comments["return"] += " " + comment.strip()

    return parsed_comments
```

Figure A2: `ClassAndFunctionVisitor` ***class.***

```python
from neo4j import GraphDatabase, Transaction
from DocParser import DocParser
from ProjectManager import ProjectManager
from FolderManager import FolderManager
from FileManager import FileManager
from ClassManager import ClassManager
from FunctionManager import FunctionManager
import os


class KnowledgeGraphManager():
    # responsible for calling the submanagers
    def __init__(self, uri: str, user: str, password: str, project_name: str, info_dict: dict):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))
        self.project_manager = ProjectManager(self.driver, project_name)    # handles project nodes and edges
        self.folder_manager = FolderManager(self.driver, info_dict, project_name)   # handles folder nodes and edges
        self.file_manager = FileManager(self.driver, info_dict )    # handles file nodes and edges
        self.class_manager = ClassManager(self.driver, info_dict)   # handles class nodes and edges
        self.function_manager = FunctionManager(self.driver, info_dict) # handles function nodes and edges
        self.info_dict = info_dict  # dict from DocParser

    def close(self) -> None:
        self.driver.close()

    def create_graph(self) -> None:
        # workflow to create entire knowledge graph
        self.project_manager.create_project()
        self.folder_manager.create_folders()
        self.file_manager.create_files()
        self.class_manager.create_classes()
        self.function_manager.create_functions()

    def clean_database(self) -> None:
        # removes all nodes and edges from the graph database
        with self.driver.session() as session:
            session.execute_write(self._clean_database)

    @staticmethod
    def _clean_database(tx: Transaction) -> None:
        # tx is the transaction object passed from the database session
        query = """
        MATCH (n)
        DETACH DELETE n
        """
        tx.run(query)


if __name__ == "__main__":
    uri = os.getenv("NEO4j_URI")
    username = os.getenv("NEO4j_USER")
    password = os.getenv("NEO4j_PASSWORD")
    root_path = os.getenv("CDPKit_PATH")
    cdp_folders = [root_path + "Chem",
                   root_path + "Pharm",
                   root_path + "Base",
                   root_path + "Biomol",
                   root_path + "ConfGen",
                   root_path + "Descr",
                   root_path + "ForceField",
                   root_path + "GRAIL",
                   root_path + "Grid",
                   root_path + "Math",
                   root_path + "MolProp",
                   root_path + "Shape",
                   root_path + "Util",
                   root_path + "Vis"]
    for folder in cdp_folders:
        all_files_info = DocParser(folder).parse_files()
        cdpkit_graph_manager = KnowledgeGraphManager(uri, username, password, project_name="CDPKit", info_dict=all_files_info)
        cdpkit_graph_manager.create_graph()
```

Figure A3: `KnowledgeGraphManager` class.

```python
from neo4j import Driver, Transaction

class ProjectManager():
    def __init__(self, driver: Driver, project_name: str) -> None:
        self.project_name = project_name
        self.driver = driver

    def create_project(self) -> None:
        with self.driver.session() as session:
            session.execute_write(self._create_project_node, self.project_name)

    def _create_project_node(self, tx: Transaction, project_name: str) -> None:
        # Creates the project node if it does not already exist
        query = """
        MERGE (p:Project {name: $project_name})
        RETURN p
        """
        tx.run(query, project_name=project_name)
```

Figure A4: `ProjectManager` class.

```python
from neo4j import Driver, Transaction

class FolderManager():
    def __init__(self, driver: Driver, info_dict: dict, project_name: str):
        self.driver = driver
        self.info_dict = info_dict
        self.project_name = project_name

    def create_folders(self) -> None:
        with self.driver.session() as session:
            session.execute_write(self._create_folder_nodes_and_relationships, self.info_dict, self.project_name)


    def _create_folder_nodes_and_relationships(self, tx: Transaction, info_dict: dict, project_name: str) -> None:
        for folder in info_dict.keys(): # loops through each folder and creates its node and relationships
            self._create_folder_nodes(tx, folder)
            self._create_folder_relationships(tx, project_name, folder)

    def _create_folder_nodes(self, tx: Transaction, folder: str) -> None:
        # creates the folder node if it does not already exist
        query = """
        MERGE (f:Folder {name: $folder_name})
        RETURN f
        """
        tx.run(query, folder_name=folder)

    def _create_folder_relationships(self, tx: Transaction, project_name: str, folder: str) -> None:
        # creates the edge between folder and the project it is included in (if it does not already exist)
        query = """
        MATCH (p:Project {name: $project_name}), (f:Folder {name: $folder_name})
        MERGE (f)-[:INCLUDED_IN]->(p)
        """
        tx.run(query, project_name=project_name, folder_name=folder)
```

Figure A5: `FolderManager` class.

```python
from neo4j import Driver, Transaction

class FileManager():
    def __init__(self, driver: Driver, info_dict: dict) -> None:
        self.driver = driver
        self.info_dict = info_dict

    def create_files(self) -> None:
        with self.driver.session() as session:
            session.execute_write(self._create_file_nodes_and_relationships)

    def _create_file_nodes_and_relationships(self, tx: Transaction) -> None:
         for folder, nested_dict in self.info_dict.items(): # loops through all the files in a folder and creates their nodes and relationships
             for file_name in nested_dict.keys():
                self._create_file_nodes(tx, file_name)
                self._create_file_relationships(tx, folder, file_name)


    def _create_file_nodes(self, tx: Transaction, file_name: str) -> None:
        # creates file node if it does not already exist
        query = """
        MERGE (f:File {name: $file_name})
        RETURN f
        """
        tx.run(query, file_name=file_name)

    def _create_file_relationships(self, tx: Transaction, folder: str, file_name: str) -> None:
        # creates relationship from file to folder it is included in (if it does not already exist)
        query = """
        MATCH (fo:Folder {name: $folder_name}), (fi:File {name: $file_name})
        MERGE (fi)-[:INCLUDED_IN]->(fo)
        """
        tx.run(query, folder_name=folder, file_name=file_name)
```

Figure A6: `FileManager` class.

```python
from neo4j import Driver, Transaction
from FunctionManager import FunctionManager
from DecoratorManager import DecoratorManager
import json

class ClassManager():
    def __init__(self, driver: Driver|None = None, info_dict: dict|None = None):
        self.driver = driver
        self.info_dict = info_dict
        self.FunctionManager = FunctionManager()
        self.DecoratorManager = DecoratorManager()

    def create_classes(self) -> None:
        with self.driver.session() as session:
            session.execute_write(self._create_classes)

    def _create_classes(self, tx: Transaction) -> None:
        for nested_dict in self.info_dict.values():
            for file_name, class_and_function_dicts in nested_dict.items():
                # loops through all the classes in all files and creates their nodes and edges
                for c in class_and_function_dicts["classes"]:
                    ClassManager._create_class_node(tx, c["name"])
                    self._set_class_comment(tx, c["name"], c["comment"])    # this is necessary if the class already exists but without a comment
                    self._create_class_relationships(tx, file_name, c)

    @staticmethod
    def _create_class_node(tx: Transaction, class_name: str) -> None:
        # creates a class node if it does not already exist
        query = """
        MERGE (f:Class {name: $class_name})
        RETURN f
        """
        tx.run(query, class_name=class_name)
```

```python
def _create_class_inheritance_relationship(self, tx: Transaction, class_name: str, base_name: str) -> None:
    # creates the relationship between a class and the class it inherits from (if it not already exists)
    query = """
    MATCH (fo:Class {name: $class_name}), (fi:Class {name: $base_name})
    MERGE (fo)-[:INHERITS_FROM]->(fi)
    """
    tx.run(query, class_name=class_name, base_name=base_name)

def _create_class_file_relationship(self, tx: Transaction, file_name: str, class_name: str) -> None:
    # creates the relationship between a class and the file it is declared at (if it not already exists)
    query = """
    MATCH (fo:File {name: $file_name}), (fi:Class {name: $class_name})
    MERGE (fi)-[:DECLARED_AT]->(fo)
    """
    tx.run(query, file_name=file_name, class_name=class_name)

def _set_class_attributes(self, tx: Transaction, class_name: str, attributes: str) -> None:
    # sets attributes of the class node
    query = """
    MATCH (f:Class {name: $class_name})
    set f.attributes = $attributes
    RETURN f
    """
    tx.run(query, class_name=class_name, attributes=attributes)

def _set_class_comment(self, tx: Transaction, class_name: str, class_comment: str) -> None:
    # sets the comment of the class node
    query = """
    MATCH (f:Class {name: $class_name})
    set f.comment = $class_comment
    RETURN f
    """
    tx.run(query, class_name=class_name, class_comment=class_comment)


def _create_nested_class_relationship(self, tx: Transaction, class_name: str, nested_class_name: str) -> None:
    # creates the relationship between a class and its nested class (if it not already exists)
    query = """
    MATCH (fo:Class {name: $class_name}), (fi:Class {name: $nested_class_name})
    MERGE (fo)-[:HAS]->(fi)
    """
    tx.run(query, class_name=class_name, nested_class_name=nested_class_name)
```

```python
def _create_class_relationships(self, tx: Transaction, file_name: str, c: dict) -> None:
    # takes a class dict and creates all relationships and nodes related to that class
    class_name = c["name"]
    self._create_class_file_relationship(tx, file_name, class_name)

    for base in c["bases"]:
        base_name = base
        if "." in base:
            modules = base.split(".")
            base_name = modules[-1]
        ClassManager._create_class_node(tx, class_name=base_name)   # this is why comments and attributes can be also set later if a class node was already created through inheritance
        self._create_class_inheritance_relationship(tx, class_name=class_name, base_name=base_name)

    for d in c["decorators"]:
        self.DecoratorManager._create_decorator_node(tx, decorator_name=d)
        self.DecoratorManager._create_decorator_class_relationship(tx, class_name=class_name, decorator_name=d)

    for m in c["methods"]:
        # a method is treated the same way as a normal function definition, but it has a relationship to the class node
        method_name = m["name"]
        method_comment = m["comment"]
        parameters = json.dumps(self.FunctionManager.ParameterManager._parse_parameters(m["params"]))
        self.FunctionManager._create_function_node(tx, method_name, method_comment, parameters, json.dumps(m["decorators"]), return_type=json.dumps(m["return_type"]))
        self.FunctionManager._create_function_class_relationship(tx, method_name, method_comment, class_name, parameters, json.dumps(m["decorators"]), return_type=json.dumps(m["return_type"]))
        self.FunctionManager._create_function_inputs(tx, m)

    if c["class_attributes"]:
        self._set_class_attributes(tx, class_name, json.dumps(c["class_attributes"]))

    for nc in c["nested_classes"]:
        ClassManager._create_class_node(tx, class_name=nc["name"])
        self._set_class_comment(tx, nc["name"], nc["comment"])
        self._create_class_relationships(tx, file_name=file_name, c=nc)
        self._create_nested_class_relationship(tx, class_name=class_name, nested_class_name=nc["name"])
```

Figure A7: `ClassManager` class.

```python
from neo4j import Driver, Transaction
from DecoratorManager import DecoratorManager
from ParameterManager import ParameterManager
from TypeManager import TypeManager
from ClassManager import ClassManager
import json

class FunctionManager():
    def __init__(self, driver: Driver|None = None, info_dict: dict|None = None):
        self.driver = driver
        self.info_dict = info_dict
        self.DecoratorManager = DecoratorManager()
        self.ParameterManager = ParameterManager()
        self.TypeManager = TypeManager()

    def create_functions(self) -> None:
        with self.driver.session() as session:
            session.execute_write(self._create_functions)

    def _create_functions(self, tx: Transaction) -> None:
        for nested_dict in self.info_dict.values():
            for file_name, class_and_function_dicts in nested_dict.items():
                for f in class_and_function_dicts["functions"]:
                    # loops through all functions declared in the files and creates the nodes and relationships
                    function_name = f["name"]
                    function_comment = f["comment"]
                    parameter_dict = json.dumps(self.ParameterManager._parse_parameters(f["params"]))
                    function_decorators = json.dumps(f["decorators"])
                    function_return_type = json.dumps(f["return_type"])
                    self._create_function_node(tx, function_name, function_comment, parameter_dict, function_decorators, function_return_type)
                    self._create_function_file_relationship(tx, function_name, function_comment, parameter_dict, function_decorators, function_return_type, file_name)
                    self._create_function_inputs(tx, f)

    def _create_function_node(self, tx: Transaction, function_name: str, function_comment: str,
                              parameter_dict: str, decorator_list: str, return_type: str) -> None:
        # creates the node of the function (if it does not already exist)
        query = """
        MERGE (f:Function {name: $function_name, comment: $function_comment, parameter: $parameter_dict, decorators: $decorator_list, returns: $return_type})
        RETURN f
        """
        tx.run(query, function_name=function_name, function_comment=function_comment, parameter_dict=parameter_dict, decorator_list=decorator_list, return_type=return_type)

    def _create_function_file_relationship(self, tx: Transaction, function_name: str, function_comment: str,
                                           parameter_dict: str, decorator_list: str, return_type: str, file_name: str) -> None:
        # creates the relationship between a function and the file it is declared at (if it does not already exist)
        query = """
        MATCH (fo:File {name: $file_name}), (fi:Function {name: $function_name, comment: $function_comment, parameter: $parameter_dict, decorators: $decorator_list, returns: $return_type})
        MERGE (fi)-[:DECLARED_AT]->(fo)
        """
        tx.run(query, file_name=file_name, function_name=function_name, function_comment=function_comment, parameter_dict=parameter_dict, decorator_list=decorator_list, return_type=return_type)

    def _create_function_class_relationship(self, tx: Transaction, function_name: str, function_comment: str,
                                            class_name: str, parameter_dict: str, decorator_list: str, return_type: str) -> None:
        # if the function is a method in a class, it creates this relationship (if it does not already exist)
        query = """
        MATCH (fo:Class {name: $class_name}), (fi:Function {name: $function_name, comment: $function_comment, parameter: $parameter_dict, decorators: $decorator_list, returns: $return_type})
        MERGE (fo)-[:HAS]->(fi)
        """
        tx.run(query, class_name=class_name, function_name=function_name, function_comment=function_comment, parameter_dict=parameter_dict, decorator_list=decorator_list, return_type=return_type)

    def _create_function_class_relationship(self, tx: Transaction, function_name: str, function_comment: str,
                                            class_name: str, parameter_dict: str, decorator_list: str, return_type: str) -> None:
        # if the function is a method in a class, it creates this relationship (if it does not already exist)
        query = """
        MATCH (fo:Class {name: $class_name}), (fi:Function {name: $function_name, comment: $function_comment, parameter: $parameter_dict, decorators: $decorator_list, returns: $return_type})
        MERGE (fo)-[:HAS]->(fi)
        """
        tx.run(query, class_name=class_name, function_name=function_name, function_comment=function_comment, parameter_dict=parameter_dict, decorator_list=decorator_list, return_type=return_type)

    def _create_function_inputs(self, tx: Transaction, function_dict: dict) -> None:
        # creates the nodes and relationships for the function inputs
        function_name = function_dict["name"]
        function_comment = function_dict["comment"]
        parameter_dict = json.dumps(self.ParameterManager._parse_parameters(function_dict["params"]))
        function_decorators = json.dumps(function_dict["decorators"])
        function_return_type = json.dumps(function_dict["return_type"])
        for p in function_dict["params"]:
            parameter_name = p["name"]
            parameter_comment = p["comment"]
            parameter_type = json.dumps(p["type"])
            parameter_default = json.dumps(p["default"])
            self.ParameterManager._create_parameter_node(tx, parameter_name, parameter_comment, parameter_type, parameter_default)
            self.ParameterManager._create_parameter_function_relationship(tx,  function_name, function_comment, parameter_dict, function_decorators, function_return_type, parameter_name, parameter_comment,
            type_name = parameter_type
            if "." in parameter_type:
                modules = p["type"].split(".")
                type_name = modules[-1]
            ClassManager._create_class_node(tx, class_name=type_name)   # also creates class nodes for types
            self.TypeManager._create_type_relationship(tx, parameter_name, parameter_comment, parameter_type, parameter_default, type_name)

        for d in function_dict["decorators"]:
            self.DecoratorManager._create_decorator_node(tx, decorator_name=d)
            self.DecoratorManager._create_decorator_function_relationship(tx, d, function_name, function_comment, parameter_dict, function_decorators, function_return_type)
```

Figure A8: `FunctionManager` *class.*

```python
from neo4j import Transaction

class ParameterManager():
    def _create_parameter_node(self, tx: Transaction,
                               parameter_name: str, parameter_comment: str, parameter_type: str, parameter_default: str) -> None:
        # creates the parameter node (if it does not already exist)
        query = """
        MERGE (f:Parameter {name: $param_name, comment: $parameter_comment, type: $type_name, default: $default_value})
        RETURN f
        """
        tx.run(query, param_name=parameter_name, parameter_comment=parameter_comment, type_name=parameter_type, default_value=parameter_default)

    def _create_parameter_function_relationship(self, tx: Transaction, function_name: str, function_comment: str,
                                                parameter_dict: str, decorator_list: str, return_type: str,
                                                parameter_name: str, parameter_comment: str, parameter_type: str, parameter_default: str) -> None:
        # creates the relationship between an input parameter and its function (if it does not already exist)
        query = """
        MATCH (fo:Function {name: $function_name, comment: $function_comment, parameter: $parameter_dict, decorators: $decorator_list, returns: $return_type}), (fi:Parameter {name: $param_name, comment: $parameter
        MERGE (fo)-[:HAS]->(fi)
        """
        tx.run(query,function_name=function_name, function_comment=function_comment, parameter_dict=parameter_dict, decorator_list=decorator_list, return_type=return_type, param_name=parameter_name, parameter_com

    def _parse_parameters(self, parameters: list) -> list:
        parameters_list = []
        for p in parameters:
            parameters_list.append({key: value for key,value in p.items()})
        return parameters_list
```

Figure A9: `ParameterManager` *class.*

```python
from neo4j import Transaction

class TypeManager():
    def _create_type_relationship(self, tx: Transaction,
                                  parameter_name: str, parameter_comment: str, parameter_type: str, parameter_default: str,
                                  type_name: str) -> None:
        # creates the relationship between a parameter and its type (if it does not already exist)
        query = """
        MATCH (fo:Parameter {name: $param_name, comment: $parameter_comment, type: $parameter_type, default: $default_value}), (fi:Class {name: $type_name})
        MERGE (fo)-[:OF_TYPE]->(fi)
        """
        tx.run(query, param_name=parameter_name, parameter_comment=parameter_comment, parameter_type=parameter_type, type_name=type_name, default_value=parameter_default)
```

Figure A10: `TypeManager` *class.*

```python
from neo4j import Transaction


class DecoratorManager():
    def _create_decorator_node(self, tx: Transaction, decorator_name: str) -> None:
        # creates the decorator node (if it does not already exist)
        query = """
        MERGE (f:Decorator {name: $decorator_name})
        RETURN f
        """
        tx.run(query, decorator_name=decorator_name)

    def _create_decorator_class_relationship(self, tx: Transaction, class_name: str, decorator_name: str) -> None:
        # creates the relationship between a class and its decorator (if it does not already exist)
        query = """
            MATCH (fo:Class {name: $class_name}), (fi:Decorator {name: $decorator_name})
            MERGE (fo)-[:HAS]->(fi)
            """
        tx.run(query, class_name=class_name, decorator_name=decorator_name)

    def _create_decorator_function_relationship(self, tx: Transaction, decorator_name: str, function_name: str, function_comment: str,
                                                parameter_dict: str, decorator_list: str, return_type: str) -> None:
        # creates the relationship between a function and its decorator (if it does not already exist)
        query = """
        MATCH (fo:Function {name: $function_name, comment: $function_comment, parameter: $parameter_dict, decorators: $decorator_list, returns: $return_type}), (fi:Decorator {name: $decorator_name})
        MERGE (fo)-[:HAS]->(fi)
        """
        tx.run(query, decorator_name=decorator_name, function_name=function_name, function_comment=function_comment, parameter_dict=parameter_dict, decorator_list=decorator_list, return_type=return_type)
```

Figure A11: `DecoratorManager` *class.*

```python
import os
os.environ['HF_HOME'] = os.getenv("MODEL_LOCATION")
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"

from utils.rag_utils import initialize_neo4j, get_kg_schema, get_pipeline_from_model
from retriever import retrieve_context
from generator import generate_rag_prompt, generate_answer_qwen
import json
import re
from transformers.pipelines.text_generation import TextGenerationPipeline
import argparse

# Base idea from this article:
# https://medium.com/@silviaonofrei/code-llamas-knowledge-of-neo4j-s-cypher-query-language-54783d2ad421


def question_rag(user_prompt: str, pipe_cypher: TextGenerationPipeline, pipe_answer: TextGenerationPipeline) -> tuple[str, str]:
    # function to pass a user prompt to the Graph RAG system
    driver = initialize_neo4j() # initialize the neo4j driver to communicate with the Knowledge Graph
    schema = get_kg_schema() # get the KG schema
    try:
        query_result, cypher_query = retrieve_context(driver, user_prompt, pipe_cypher, schema)
    except Exception as e:
        print("Exception while retrieving context: ", e)
        query_result = "Context could not be retrieved" # if there is an exception, the query was not functional
        cypher_query = "None" # set it to None to flag for non-runnable queries during benchmarking

    system_prompt_rag = generate_rag_prompt(query_result, cypher_query) # get the final system prompt for the rag

    final_answer = generate_answer_qwen(user_prompt, system_prompt_rag, pipe_answer) # generate the final answer

    return cypher_query, query_result, final_answer

def benchmark_rag(pipe_cypher: TextGenerationPipeline, pipe_answer: TextGenerationPipeline) -> None:
    # function for running the benchmark questions
    with open("/data/shared/projects/graphRAG/graphRAG/graphRAG/benchmark_questions.txt", "r") as f:
        testset = f.read()

    pattern = r"Q:\s*(.+?)\nQuery:\s*(.+?)\nC:\s*(.+?)\nA:\s*(.+?)(?=\nQ:|\Z)"

    # Find all matches
    matches = re.findall(pattern, testset, re.DOTALL)

    # Parse into a list of dictionaries
    parsed_questions = [
        {"Question": match[0].strip(), "Query": match[1].strip(), "Context": match[2].strip(), "Answer": match[3].strip()}
        for match in matches
    ]

    for i in range(97, 100):
        benchmark = []
        for question in parsed_questions:
            cypher_query, query_result, final_answer = question_rag(question["Question"], pipe_cypher, pipe_answer)
            benchmark.append({"user_prompt": question["Question"], "cypher_query": cypher_query,
                              "retrieved_context": query_result,
                                "final_answer": final_answer,
                                "model_cypher": question["Query"], "model_answer": question["Answer"],
                                "model_context": question["Context"],
                                "score_cypher_automated": "",
                                "score_context_automated":"",
                                "score_answer_automated": "",
                                "score_code_automated":"",
                                "score_overall_automated": "",
                                "score_cypher_manual": "",
                                "score_context_manual":"",
                                "score_answer_manual": "",
                                "score_code_manual":"",
                                "score_overall_manual": ""
                                })

        with open(f"/data/shared/projects/graphRAG/graphRAG/graphRAG/benchmark_results/benchmark_results_{i+1}.json", "w") as file:
            json.dump(benchmark, file, indent=4)
```

```python
if __name__ == "__main__":
    model_cypher = "codellama/CodeLlama-13b-Instruct-hf"

    model_answer = "Qwen/Qwen2.5-7B-Instruct"

    pipe_cypher = get_pipeline_from_model(model_cypher)

    pipe_answer = get_pipeline_from_model(model_answer)

    parser = argparse.ArgumentParser(description="CDPKit Graph RAG: Ask a question via command line")

    parser.add_argument(
        "user_query",
        type=str,
        help="The question you want to ask the CDPKit Graph RAG system."
    )

    args = parser.parse_args()

    _, _, answer = question_rag(args.user_query, pipe_cypher, pipe_answer)

    print(answer)

    # benchmark_rag(pipe_cypher, pipe_answer)
```

Figure A12: graphRag.py

```python
from utils.rag_utils import run_query
from neo4j import Driver
from transformers.pipelines.text_generation import TextGenerationPipeline


def retrieve_context(driver: Driver, user_prompt: str, pipe: str, schema: str) -> tuple[str, str]:
    # retrieves the context from the KG
    cypher_query = get_cypher_query(user_prompt, pipe, schema)

    query_result = run_query(driver, cypher_query)

    return query_result, cypher_query

def get_cypher_query(user_prompt: str, pipe: TextGenerationPipeline, schema: str) -> str:
    # gets the cypher query
    system_prompt_query= generate_cypher_query_prompt(schema)

    cypher_query = generate_answer_code_llama(user_prompt, system_prompt_query, pipe)

    return cypher_query



def generate_cypher_query_prompt(schema: str) -> str:
    # generates the prompt for the generation of the cypher query
    # the prompt includes a clear instruction and positive examples
    return """
    You are experienced with translating questions into cypher queries. Provide answers in Cypher query language, *strictly* based on the following graph Neo4j schema.

    ### The Schema

    {schema}

    ### Examples
    Q: What methods does AromaticSubstructure have?
    A: MATCH (c:Class {{name: 'AromaticSubstructure'}})-[:HAS]->(f:Function) RETURN f.name, f.comment

    Q: What does the class AtomPredicate do?
    A: MATCH (c:Class {{name: "AtomPredicate"}}) RETURN c.comment
    """.format(schema = schema)
```

```python
def generate_answer_code_llama(user_prompt: str, system_prompt: str, pipe: TextGenerationPipeline, **kwargs: dict) -> str:
    # https://medium.com/@silviaonofrei/code-llamas-knowledge-of-neo4j-s-cypher-query-language-54783d2ad421
    # generates the answer from the code llama model
    full_prompt = "<s>[INST]<<SYS>>\n{system}\n<</SYS>>\n\n{user}[/INST]\n\n".format(system=f"{system_prompt}",
                                                                                     user=f'{user_prompt}') # format the full prompt in a way that is easily understood by Code LLama

    if "max_new_tokens" not in kwargs:
        kwargs["max_new_tokens"] = 512 # Set max new tokens if not provided

    kwargs.setdefault("temperature", 0.7) # controls randomness of sampling

    output = pipe(full_prompt,
                  do_sample=True, # enables sampling and a more varied generation
                  top_k=5, # take the top 5 most likely tokens at each generation step
                  top_p=0.9,
                  **kwargs)

    output = output[0]["generated_text"] # Extract the relevant part of the generated text
    return output.split('<</SYS>>', 1)[1].split("[/INST]")[1] # takes only the relevant part of the answer
```

Figure A13: retriever.py

```python
from transformers.pipelines.text_generation import TextGenerationPipeline

def generate_answer_qwen(user_prompt: str, system_prompt: str, pipe: TextGenerationPipeline, **kwargs: dict) -> str:
    # https://medium.com/@silviaonofrei/code-llamas-knowledge-of-neo4j-s-cypher-query-language-54783d2ad421
    full_prompt = f"System: {system_prompt}\nUser: {user_prompt}\nAssistant:" # combine the system and user prompt into a from that is easily understoof by Qwen

    if "max_new_tokens" not in kwargs:
        kwargs["max_new_tokens"] = 512 # Set a default max_new_tokens if not provided

    # generate the answer
    output = pipe(full_prompt,
                  do_sample=True, # enables sampling and a more varied generation
                  top_k=5, # take the top 5 most likely tokens at each generation step
                  top_p=0.9,
                  temperature = 0.7, # controls randomness of sampling
                  **kwargs
                  )

    output = output[0]["generated_text"] # Extract the relevant part of the generated text
    return output.split('Assistant:', 1)[1].strip() # takes only the portion of the text after the "Assistant: "


def generate_rag_prompt(retrieved_context: str, cypher_query: str) -> str:
    # takes the cypher query and retrieved context and creates a system prompt
    # positive examples are added as well as clear instructions
    return """

    You are a highly intelligent assistant. Your job is to answer user questions using *only* the information from the retrieved context provided from a Neo4j knowledge
    The retrieved context is the result of a cypher query.
    Ensure that your response strictly relies on the retrieved context, and do not add any information from other sources.

    Cypher query:
    {cypher_query}
    Retrieved Context:
    {retrieved_context}

    ### Example 1:
    Q: What methods does AromaticSubstructure have?
    Cypher query: MATCH (c:Class {{name: 'AromaticSubstructure'}})-[:HAS]->(f:Function) RETURN f.name, f.comment
    Retrieved Context: [['f.name', 'f.comment'], ['__init__', 'Constructs an empty <tt>AromaticSubstructure</tt> instance.'], ['__init__', 'Construct a <tt>AromaticSubs
    A: AromaticSubstructure has the following methods:
    - __init__: Constructs an empty <tt>AromaticSubstructure</tt> instance.
    - __init__: Construct a <tt>AromaticSubstructure</tt> instance that consists of the aromatic atoms and bonds of the molecular graph <em>molgraph</em>.
    - perceive: Replaces the currently stored atoms and bonds by the set of aromatic atoms and bonds of the molecular graph <em>molgraph</em>.
    """.format(retrieved_context = retrieved_context, cypher_query = cypher_query)
```

Figure A14: generator.py

```python
import os
os.environ['HF_HOME'] = os.getenv("MODEL_LOCATION")
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"

from huggingface_hub import hf_hub_download
from neo4j import GraphDatabase, Driver
from transformers import AutoTokenizer, pipeline
from transformers.pipelines.text_generation import TextGenerationPipeline
import torch



def get_pipeline_from_model(model: str) -> TextGenerationPipeline:
    # this will generate a TextGenerationPipeline for the given model
    tokenizer = AutoTokenizer.from_pretrained(model,
                                              padding_side = "left")  # loads the tokenizer for the specified model, padding is added to left side of the input s

    return pipeline(
        "text-generation", # the pipeline will be used for text generation
        model=model, # loads the specified model
        tokenizer=tokenizer,
        torch_dtype=torch.float16, # precision of the pipeline
        trust_remote_code=True, # if the model has any non standard behavior
        device_map="auto" # automatically maps the model to the hardware that is available (e.g. GPUs)
    )

def get_pipelines(model_cypher: str, model_answer: str) -> tuple[TextGenerationPipeline, TextGenerationPipeline] :
    # gets the two pipelines necessary
    pipe_cypher = get_pipeline_from_model(model_cypher)

    pipe_answer = get_pipeline_from_model(model_answer)

    return pipe_cypher, pipe_answer


def initialize_neo4j() -> Driver:
    # initializes the neo4j driver necessary to query the KG
    return GraphDatabase.driver(os.getenv("NEO4j_URI"), auth=(os.getenv("NEO4j_USER"), os.getenv("NEO4j_PASSWORD")))


def run_query(driver: Driver, query: str, params: dict|None =None) -> list:
    with driver.session() as session: # starts new session
            result = session.run(query, params) # executes the query
            output = [r.values() for r in result] # turns the query output into a list with only the values of the result dict
            output.insert(0, result.keys()) # inserts the column headers of the query result at the beginning of the output list
            return output

def get_kg_schema() -> str:
    # provides the schema of the KG, which is the nodes and relationships that exist
    return """
    Node properties are the following:
    [
        {"labels": "Project", "properties": ["name"]},
        {"labels": "Folder", "properties": ["name"]},
        {"labels": "File", "properties": ["name"]},
        {"labels": "Class", "properties": ["name", "comment", "attributes"]},
        {"labels": "Function", "properties": ["name", "comment", "parameter", "decorators", "returns"]},
        {"labels": "Parameter", "properties": ["name", "comment", "default", "type"]},
        {"labels": "Decorator", "properties": ["name"]}
    ]

    Relationships point from source to target nodes:
    [
        {"relationship": "INCLUDED_IN", "source": "Folder", "target": "Project"},
        {"relationship": "INCLUDED_IN", "source": "File", "target": "Folder"},
        {"relationship": "INHERITS_FROM", "source": "Class", "target": "Class"},
        {"relationship": "HAS", "source": "Class", "target": ["Function", "Class", "Decorator"]},
        {"relationship": "DECLARED_AT", "source": "Class", "target": "File"},
        {"relationship": "HAS", "source": "Function", "target": "Decorator"},
        {"relationship": "HAS", "source": "Function", "target": "Parameter"}
        {"relationship": "DECLARED_AT", "source": "Function", "target": "File"},
        {"relationship": "OF_TYPE", "source": "Parameter", "target": "Class"}
    ]
    """
```

Figure A15: rag_utils.py

```python
import os
os.environ['HF_HOME'] = os.getenv("MODEL_LOCATION")
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"

from huggingface_hub import hf_hub_download
import json
import re
from transformers.pipelines.text_generation import TextGenerationPipeline
from utils.rag_utils import get_pipeline_from_model
from utils.scoring_utils import generate_answer_eval_prompt, generate_code_eval_prompt, generate_context_eval_prompt, generate_cypher_eval_prompt, generate_overall_


def generate_answer_falcon(prompt: str, pipe: TextGenerationPipeline, **kwargs: dict) -> str:
    # passes the user prompt to Falcon and gets the output score
    full_prompt = [{"role": "user", "content": prompt}] # format that Falcon handles best

    if "max_new_tokens" not in kwargs:

        kwargs["max_new_tokens"] = 512 # Set a default max_new_tokens if not provided

    # generate the answer
    output = pipe(full_prompt,
                  do_sample=True, # enables sampling and a more varied generation
                  top_k=5, # take the top 5 most likely tokens at each generation step
                  top_p=0.9,
                  temperature = 0.7, # controls randomness of sampling
                  **kwargs
                  )

    output = output[0]["generated_text"][1]["content"] # Extract the relevant part of the generated text
    number_match = re.search(r"-?\d+", output) # Extract only the number in case the answer also had an explanation
    if number_match:
        return number_match.group(0)
    else:
        return ""


def testset_scoring_llm() -> None:
    model = "tiiuae/Falcon3-10B-Instruct" #scoring LLM
    directory = "/data/shared/projects/graphRAG/graphRAG/graphRAG/benchmark_results"
    pipe = get_pipeline_from_model(model)

    for ix, file in enumerate(sorted(os.listdir(directory), key=lambda x: int(x.split('_')[2].split('.')[0]))):
        if file.endswith('.json'):
            file_path = os.path.join(directory, file)

            with open(file_path, "r") as f:
                results = json.load(f)

            # go through all the benchmark files and score the answers
            for q in results:
                user_prompt = generate_cypher_eval_prompt(q["cypher_query"], q["model_cypher"])
                q["score_cypher_automated"] = generate_answer_falcon(user_prompt, pipe)
                user_prompt = generate_context_eval_prompt(q["retrieved_context"], q["model_context"])
                q["score_context_automated"] = generate_answer_falcon(user_prompt, pipe)
                user_prompt = generate_answer_eval_prompt(q["user_prompt"], q["final_answer"], q["retrieved_context"])
                q["score_answer_automated"] = generate_answer_falcon(user_prompt, pipe)
                user_prompt = generate_code_eval_prompt(q["final_answer"])
                q["score_code_automated"] = generate_answer_falcon(user_prompt, pipe)
                user_prompt = generate_overall_eval_prompt(q["final_answer"], q["model_answer"])
                q["score_overall_automated"] = generate_answer_falcon(user_prompt, pipe)

            with open(file_path, "w") as f:
                json.dump(results, f, indent=4)
```

Figure A16: llm_scoring.py

```
def generate_cypher_eval_prompt(cypher: str, model: str) -> str:
    # provides the prompt for auomatically evaluating the cypher query
    return """
    You are a highly intelligent assistant. Your job is to evaluate cypher queries against a model cypher query.
    Ignore any escape sequences but pay attention to the direction of the relationships.
    If the cypher query matches the model exactly, return 1.
    If the cypher query and the model are not the same, return -1.
    If the cypher query is None, return -1.

    IMPORTANT: Only answer with a single number (1, -1). Do not add any text, explanations, or comments.

    ### Example 1:
    Cypher: \n\nMATCH (p:Project {{name: "CDPKit"}})-[:INCLUDED_IN]->(f:Folder) RETURN f.name
    Model: \\n\\nMATCH (f:Folder)-[:INCLUDED_IN]->(p:Project {{name: 'CDPKit'}}) RETURN f.name
    Answer: -1

    ### Example 2:
    Cypher: \n\nMATCH (f:File)-[:INCLUDED_IN]->(fld:Folder {{name: "Base"}}) RETURN f.name
    Model: \\n\\nMATCH (f:File)-[:INCLUDED_IN]->(folder:Folder {{name: 'Base'}}) RETURN f.name
    Answer: 1

    Cypher: {cypher}
    Model: {model}
    """.format(cypher = f"{cypher}", model = f"{model}")

def generate_context_eval_prompt(context: str, model: str) -> str:
    # provides the prompt for auomatically evaluating the context
    return """
    You are a highly intelligent assistant. Your job is to evaluate a retrieved context against a model context.
    Ignore any formatting.
    If the context matches the model return 1.
    If the context is empty except for placeholders or it says that the context could not be retrieved return 0.
    If the context provides info that is not in the model, return -1.

    IMPORTANT: Only answer with a single number (1, -1, 0). Do not add any text, explanations, or comments.

    ### Example 1:
    Context: [['f.name']]
    Model: ['f.name'], ['Chem'], ['Pharm'], ['Base'], ['Biomol'], ['ConFGen'], ['ConfGen'], ['Descr'], ['ForceField'], ['GRAIL'], ['Grid'], ['Math'], ['MolProp'],
    Answer: 0

    ### Example 2:
    Context: [['p.name'],['FragmentList']]
    Model: ["p.name"],["FragmentList"]
    Answer: 1

    ### Example 3:
    Context: [["p.name","p.type"],["bond",""Chem.Bond""]]
    Model: ['f.returns'], ['{{type: "int", "comment": ""}}']
    Answer: -1


    Context: {context}
    Model: {model}
    """.format(context = f"{context}", model = f"{model}")
```

```
def generate_answer_eval_prompt(question: str, answer: str, context: str) -> str:
    # provides the prompt for auomatically evaluating the answer based on the context
    return """
    You are a highly intelligent assistant. Your job is to evaluate if an answer correclty answers a question based on the retrieved context. Ignore any
    Ignore any formatting.
    If the answer says it correctly cant answer based on the context return 2.
    If the answer says it wrongly cant answer based on the context even tho the answer would be in the context return 0.
    If the answer positively provides the questioned information based on the context return 1.
    If the answer provides information that is not present in the context, return -1.

    IMPORTANT: Only answer with a single number (1, -1, 0, 2). Do not add any text, explanations, or comments.

    ### Example 1:
    Question: What are the folders included in the CDPKit Project?
    Context: f.name
    Answer: The CDPKit Project includes the following folder:\n\n- [f.name] (The specific name of the folder is not provided in the retrieved context.)
    Evaluation: 2

    ### Example 2:
    Question: What class does the class AromaticRingSet inherit from?
    Context: [['p.name'],['FragmentList']]
    Answer: The class AromaticRingSet inherits from FragmentList.
    Score: 1

    ### Example 3:
    Question: What class does the class AromaticRingSet inherit from?
    Context: [['p.name'],['FragmentList']]
    Answer: I cannot answer this based on the provided context.
    Score: 0

    ### Example 4:
    Question: How to generate Conformations with the class ConformerGenerator??
    Context: f.name, f.comment
    Answer: To generate conformations using the `ConformerGenerator` class, you can use the method `generateConformations`.
    Score: -1
def generate_code_eval_prompt(answer: str) -> str:
    # provides the prompt for auomatically evaluating the code example
    return """
    You are a highly intelligent assistant. Your job is to evaluate if the python code provided in a text is syntactically correct.
    Ignore any formatting.
    If it is correct return 1.
    If the text does not contain a code example return 2.
    If the code example is syntactically incorrect or contains a cypher query return -1

    IMPORTANT: Only answer with a single number (1, -1, 2). Do not add any text, explanations, or comments.

    ### Example 1:
    Text: The class AromaticRingSet inherits from FragmentList. \n\nPython example:\n```python\n# This is how you might represent the inheritance relationship
    Score: 1

    Text: {answer}
    """.format(answer = f"{answer}")
```

```
def generate_overall_eval_prompt(answer: str, model: str) -> str:
    # provides the prompt for auomatically evaluating the overall answer of the RAG
    return """
    You are a highly intelligent assistant. Your job is to evaluate if an answer matches a model answer.
    Ignore any formatting or python code.
    If the answer and the model both say they cannot answer something, return 2.
    If the answer and the model match, return 1
    If the answer does not match the model but rather provides wrong information, return -1.
    If the answer says it cannot answer something, but the model can, return 0.

    IMPORTANT: Only answer with a single number (1, -1, 0, 2). Do not add any text, explanations, or comments.

    ### Example 1:
    Answer: The CDPKit Project includes the following folder:\n\n- [f.name] (The specific name of the folder is not provided in the retrieved context.) \n\nGiven the
    Model: The folders included in the CDPKit Project are:\\n- Chem\\n- Pharm\\n- Base\\n- Biomol\\n- ConfGen\\n- Descr\\n- ForceField\\n- GRAIL\\n- Grid\\n- Math\\n-
    Evaluation: 0

    ### Example 2:
    Answer: The class AromaticRingSet inherits from FragmentList. \n\n```python\n# Example of accessing the inherited class in Python\nclass AromaticRingSetInheritanc
    Model: The class AromaticRingSet inherits from FragmentList.
    Score: 1

    ### Example 3:
    Answer: The parameter `feature` of the function `hasHydrophobicity` stands for \"Feature\". There is no additional comment or default value specified for this par
    Model: I cannot tell what it stands for.
    Score: 2

    ### Example 4:
    Answer: To generate conformations using the `ConformerGenerator` class, you can use the method `generateConformations`.
    Model: To generate conformations using ConformerGenerator, you can use the `generate` method. Here's how you can do it: - `generate`: Generates conformations.
    Score: -1


    Answer: {answer}
    Model: {model}
    """.format(answer = f"{answer}", model = f"{model}")
```

Figure A17: scoring_utils.py

```
import os
import json
import copy
from utils.evaluation_utils import calculate_metrics, print_metrics, print_comparison_dict, compare_result


def testset_evaluation(mode: str, directory: str) -> None:
    # evaluates the different categories of the testset evaluation for either "manual" mode or "automated" for all the files in the directory
    if mode not in ["manual", "automated"]:
        raise ValueError("Mode must be one of the following: manual, automated.")
    categories = ["cypher", "context", "answer", "code", "overall"]
    metrics_template = {cat: {"accuracy": [], "precision": [], "recall": [], "f1": [], "counts": []} for cat in categories} # template to store the results in
    metrics_specific = copy.deepcopy(metrics_template)
    metrics_general = copy.deepcopy(metrics_template)

    files = sorted([f for f in os.listdir(directory) if f.endswith('.json')], key=lambda x: int(x.split('_')[2].split('.')[0])) # sort the result files by ending


    if mode == "manual":
        files = files[:20]
    for file in files:
        file_path = os.path.join(directory, file)
        with open(file_path, "r") as f:
            results = json.load(f)

        # calculate the metrics for the specific and the general questions
        calculate_metrics(results[:17], metrics_specific, categories, mode)
        calculate_metrics(results[17:], metrics_general, categories, mode)

    print_metrics("SPECIFIC", metrics_specific)
    print_metrics("GENERAL", metrics_general)
```

```python
def compare_manual_automated(directory: str) -> None:
    # compares the result of the automated evaluation with that of the manual one for each category
    categories = ["cypher", "context", "answer", "code", "overall"]
    # the manual evaluation is the 'ground truth' and the dict will show how often a score has been correctly classified/missclassified as something else
    comparison_template = {cat: {"tp": {"tp": 0, "tn": 0, "fp": 0, "fn": 0}, "tn": {"tp": 0, "tn": 0, "fp": 0, "fn": 0}, "fp": {"tp": 0, "tn": 0, "fp": 0, "fn": 0},
    comparison_specific = copy.deepcopy(comparison_template)
    comparison_general = copy.deepcopy(comparison_template)

    files = sorted([f for f in os.listdir(directory) if f.endswith('.json')], key=lambda x: int(x.split('_')[2].split('.')[0]))

    for file in files[:20]:
        file_path = os.path.join(directory, file)
        with open(file_path, "r") as f:
            results = json.load(f)

        for q in results[:17]:
            if "score_context_manual:" in q.keys():
                q["score_context_manual"] = q["score_context_manual:"] # correct a naming error
            for cat in categories:
                compare_result(cat, comparison_specific, q[f"score_{cat}_manual"], q[f"score_{cat}_automated"])
        for q in results[17: ]:
            if "score_context_manual:" in q.keys():
                q["score_context_manual"] = q["score_context_manual:"]
            for cat in categories:
                compare_result(cat, comparison_general, q[f"score_{cat}_manual"], q[f"score_{cat}_automated"])

    print("#### SPECIFIC:")
    print_comparison_dict(comparison_specific)
    print("#### GENERAL:")
    print_comparison_dict(comparison_general)
```

Figure A18: evaluation.py

```python
import copy
import numpy as np

def calculate_metrics(questions: dict, metrics: dict, categories: list, mode: str) -> None:
    # calculate all the metrics for the questions
    counts_template = {"tp": 0, "tn": 0, "fp": 0, "fn": 0}
    counts = {cat: copy.deepcopy(counts_template) for cat in categories}

    for q in questions:
        if "score_context_manual:" in q.keys():
            q["score_context_manual"] = q["score_context_manual:"] # correct naming error
        for cat in categories:
            parse_result(q[f"score_{cat}_{mode}"], counts[cat]) # get the score counts

    for cat in categories:
        # calculate and store the calculated metrics
        metrics[cat]["accuracy"].append(calculate_accuracy(counts[cat]))
        metrics[cat]["precision"].append(calculate_precision(counts[cat]))
        metrics[cat]["recall"].append(calculate_recall(counts[cat]))
        metrics[cat]["f1"].append(calculate_f1_score(counts[cat]))
        metrics[cat]["counts"].append(get_counts(counts[cat]))

def calculate_accuracy(counts: dict) -> float:
    # returns the accuracy
    total = counts["tp"] + counts["tn"] + counts["fp"] + counts["fn"]
    return (counts["tp"] + counts["tn"]) / total if total > 0 else 0

def calculate_precision(counts: dict) -> float:
    # returns precision
    total = counts["tp"] + counts["fp"]
    return counts["tp"] / total if total > 0 else 0

def calculate_recall(counts: dict) -> float:
    # returns recall
    total = counts["tp"] + counts["fn"]
    return counts["tp"] / total if total > 0 else 0
```

```python
def calculate_f1_score(counts: dict) -> float:
    # returns f1 score
    precision = calculate_precision(counts)
    recall = calculate_recall(counts)
    return (2 * precision * recall) / (precision + recall) if (precision + recall) > 0 else 0

def get_counts(counts: dict) -> tuple:
    # returns a couple with the number of tp, tn, fp, fn
    return (counts["tp"], counts["tn"], counts["fp"], counts["fn"])

def parse_result(result: str, counts: dict) -> None:
    # parses the result int into its meaning (tp, tn, fp, fn) and increases the count by one
    if result == "":
        return
    result_mapping = {1: "tp", 2: "tn", -1: "fp", 0: "fn", -2: "fp"}
    if (key := result_mapping.get(int(result))) is not None:
        counts[key] += 1

def print_metrics(label: str, metrics: dict) -> None:
    # prints the metrics result in an easily readable format
    print(f"### RESULTS {label} QUESTIONS ###\n")
    for metric in ["accuracy", "precision", "recall", "f1"]:
        print(f"## {metric.capitalize()}: ")
        for cat in metrics:
            # we take the mean of the metric and its sd
            print(f"{cat.capitalize()}: ", np.mean(metrics[cat][metric]), " +- ", np.std(metrics[cat][metric]))
        print("\n")

def print_comparison_dict(comparison_dict: dict) -> None:
    # prints the comparison results in an easily readable form
    for category, counts in comparison_dict.items():
        print(f"## {category}:")
        for score, values in counts.items():
            print(score, ": ", values)
        print("")

def compare_result(category: str, compare_dict: dict, result_manual: str, result_automated: str) -> None:
    # parses the results of the manual and automated evaluation, adds them to the comparison dict
    if result_automated == "":
        return
    result_mapping = {1: "tp", 2: "tn", -1: "fp", 0: "fn", -2: "fp"}
    if (key := result_mapping.get(int(result_manual))) is not None:
        if (value := result_mapping.get(int(result_automated))) is not None:
            compare_dict[category][key][value] += 1
```

Figure A19: evaluation_utils.py

```python
from dash import Dash, dcc, html, Input, Output, State, callback_context
import dash_bootstrap_components as dbc
from graphRAG import question_rag
from utils.rag_utils import get_pipeline_from_model

# Load models globally to ensure they are loaded only once
model_cypher = "codellama/CodeLlama-13b-Instruct-hf"
model_answer = "Qwen/Qwen2.5-7B-Instruct"

pipe_cypher = get_pipeline_from_model(model_cypher)
pipe_answer = get_pipeline_from_model(model_answer)


# Initialize app with suppressed callback exceptions
app = Dash(__name__, external_stylesheets=[dbc.themes.FLATLY], suppress_callback_exceptions=True)
server = app.server

# Navbar
navbar = dbc.NavbarSimple(
    children=[
        dbc.NavItem(dbc.NavLink("Chat", href="/", id="chat-link", active=True, style={"fontWeight": "bold"})),
        dbc.NavItem(dbc.NavLink("About", href="/about", id="about-link", active=False, style={"fontWeight": "bold"})),
    ],
    brand=html.Div(
        [
            html.Span("CDPKit GraphRAG", className="navbar-brand mb-0 h1"),
        ],
        style={"display": "flex", "alignItems": "center"}
    ),
    brand_href="/",
    color="primary",
    dark=True,
    fluid=True
)
```

```
# Chat layout
chat_layout = html.Div([
    dbc.Row(
        dbc.Col(
            html.Div(
                [
                    html.Img(
                        src="https://cdpkit.org/_static/logo.svg",
                        alt="CDPKit Icon",
                        style={"width": "175px", "height": "85px", "marginRight": "30px", "marginBottom": "15px",
                               "marginTop": "15px"}  # Image size and margin
                    ),
                    html.H1("GraphRAG Chat", className="d-inline align-middle", style={"marginBottom": "15px",
                                                                                        "marginRight": "30px",
                                                                                        "marginTop": "15px",
                                                                                        "fontSize": "2rem"})  # Title next to image
                ],
                style={"display": "flex", "alignItems": "center"}  # Flexbox to align horizontally
            ),
            style={"display": "flex", "justifyContent": "center", "height": "100px"}
        )
    ),

    dbc.Row(dbc.Col(
        dcc.Loading(
            id="loading-indicator",
            type="circle",
            children=[
                html.Div(id="chat-history", className="p-3 border rounded bg-light",
                         style={"height": "300px", "overflowY": "scroll"})
            ],
            fullscreen=False  # Set True to cover the entire screen while loading
        )
    )),

    # Input, send button, and clear button
    dbc.Row([
        dbc.Col(dcc.Input(id="query-input", placeholder="Type your message...", type="text",
                          className="form-control", style={"width": "100%"}), width=8),
        dbc.Col(dbc.Button("Send", id="send-button", n_clicks=0, color="primary",
                           className="btn-block"), width=2),
        dbc.Col(dbc.Button("Clear", id="clear-button", n_clicks=0, color="secondary",
                           className="btn-block"), width=2),
    ], className="mt-3"),

    # Example queries
    html.Div([
        html.P("Example Queries:", className="fw-bold mt-4"),
        dbc.Row([
            dbc.Col(dbc.Button("What methods does the class AtomBondMapping have?", id="example-1",
                               color="primary", className="m-1", n_clicks=0)),
            dbc.Col(dbc.Button("What type is parameter feature of function perceiveExtendedType?", id="example-2",
                               color="primary", className="m-1", n_clicks=0)),
            dbc.Col(dbc.Button("What can you tell me about the class InteractionType?", id="example-3",
                               color="primary", className="m-1", n_clicks=0)),
        ])
    ], style={"marginBottom": "20px"}),
], style={"height": "100vh", "display": "flex", "flexDirection": "column", "padding": "5px"})

# Documentation layout
docs_layout = html.Div([
    html.H2("About the CDPKit GraphRAG", className="text-center my-5", style={"fontSize": "2rem"}),
    html.P("This Graph RAG is meant for answering questions about the CDPKit. It retrieves its information from an external Knowledge Graph. Currently, its
           className="lead", style={"marginTop": "20px", "marginLeft": "20px", "marginRight": "20px"}),
    html.Ul([
        html.Li(html.A("Graph RAG Repository", href="https://github.com/molinfo-vienna/graphRAG?tab=readme-ov-file", target="_blank")),
        html.Li(html.A("CDPKit Documentation", href="https://cdpkit.org/", target="_blank")),
])
])
```

```python
# Main layout
app.layout = html.Div([
    dcc.Location(id="url", refresh=False),  # Tracks the current URL
    navbar,  # Navbar at the top
    html.Div(id="page-content")
])

# Callback to render content based on URL
@app.callback(
    Output("page-content", "children"),
    Input("url", "pathname")
)
def display_page(pathname):
    if pathname == "/about":
        return docs_layout
    # Default to Chat layout
    return chat_layout

# Callback to manage active navbar links
@app.callback(
    [Output("chat-link", "active"),
     Output("about-link", "active")],
    Input("url", "pathname")
)

def update_active_links(pathname):
    return pathname == "/", pathname == "/about"


@app.callback(
    [Output("chat-history", "children"), Output("query-input", "value")],
    [
        Input("send-button", "n_clicks"),
        Input("clear-button", "n_clicks"),
        Input("example-1", "n_clicks"),
        Input("example-2", "n_clicks"),
        Input("example-3", "n_clicks"),
        Input("query-input", "n_submit"),
    ],
    [State("query-input", "value"), State("chat-history", "children")],
    prevent_initial_call=True
)
def handle_chat_interactions(send_clicks, clear_clicks, ex1_clicks, ex2_clicks, ex3_clicks, n_submit, query, chat_history):
    ctx = callback_context
    if not ctx.triggered:
        return chat_history, ""

    triggered_id = ctx.triggered[0]["prop_id"].split(".")[0]

    if chat_history is None:
        chat_history = []

    if triggered_id == "send-button" or triggered_id == "query-input":
        if query:
            try:
                # Pass the pipelines to the question_rag function
                _, _,response = question_rag(query, pipe_cypher=pipe_cypher, pipe_answer=pipe_answer)
            except Exception as e:
                response = f"An error occurred while processing the query."

            # Format the new messages
            response = response.replace("\n", "  \n")
            new_message = html.Div([
                html.Div(f"{query}", className="text-end text-primary fw-bold mb-3"),
                html.Div(dcc.Markdown(response), className="text-start text-primary fw-normal mb-3"),
            ])
```

```
            return chat_history + [new_message], ""

    elif triggered_id == "clear-button":
        return [], ""

    elif triggered_id in ["example-1", "example-2", "example-3"]:
        example_queries = {
            "example-1": "What methods does the class AtomBondMapping have?",
            "example-2": "What type is parameter feature of function perceiveExtendedType?",
            "example-3": "What can you tell me about the class InteractionType?",
        }
        query = example_queries.get(triggered_id, "")
        try:
            # Pass the pipelines to the question_rag function for example queries
            _, _,response = question_rag(query, pipe_cypher=pipe_cypher, pipe_answer=pipe_answer)
        except Exception as e:
            response = f"An error occurred while processing the query."

        response = response.replace("\n", "  \n")
        new_message = html.Div([
            html.Div(f"{query}", className="text-end text-primary fw-bold mb-3"),
            html.Div(dcc.Markdown(response), className="text-start text-primary fw-normal mb-3"),
        ])
        return chat_history + [new_message], query

    return chat_history, ""

if __name__ == '__main__':
    # run this to run the dashboard

    app.run_server(debug=False, host='0.0.0.0', port=8050)
```

Figure A20: graphRag_dashboard.py