AskGraph: A Dependency-Aware Code Assistant Powered by Code Graphs and LLM-Generated Cypher Queries

Nan Yang
TNO-ESI
Eindhoven, The Netherlands
nan.yang@tno.nl

Joseph Reynolds
TNO-ESI
Eindhoven, The Netherlands
joe.reynolds@tno.nl

Laurens Prast
TNO-ESI
Eindhoven, The Netherlands
laurens.prast@tno.nl

Rosilde Corvino
TNO-ESI
Eindhoven, The Netherlands
rosilde.corvino@tno.nl

***Abstract*—Large Language Models (LLMs) have transformed code assistants by enabling personalization, interactivity, and higher abstraction. However, these assistants often struggle with a common limitation; they generate responses based on a limited set of relevant code snippets retrieved from the codebase using semantic similarity search. This mechanism prevents them from viewing the code structure holistically, making it difficult to give accurate and complete answers to questions on code dependencies and structure. This paper introduces a dependency-aware code assistant that answers structural questions developers cannot easily pose to general-purpose assistants like GitHub Copilot. We achieve this by enriching the LLM with dependency facts obtained from a code graph generated by a static-analysis pipeline customized specifically for industry-scale codebases. The dependency information is queried from a Neo4j database, which stores the code graph, via Text-to-Cypher translation powered by LLMs. Cypher is a query language, designed specifically for querying graph-structured data.**

We evaluated our solution at Philips Healthcare. Specifically, we performed a benchmark with 420 collected questions and a user study with seven industrial software engineers. By analyzing the results, we identified common mistakes made by GPT-4o in the Text-to-Cypher translation to query code graphs, including syntax, schema and semantic errors. This work lays the foundation for advancing Cypher query generation on industry-scale code graphs and for augmenting graph-based code analysis with LLMs.

***Index Terms*—Large language models, Text-to-Cypher, code graph, static analysis, code assistants**

I. Introduction

The emergence of Large Language Models (LLMs) has significantly advanced the integration of artificial intelligence into various software development activities, including, but not limited to, code completion [15], code generation [11], summarization [20], test generation [7], refactoring [1], and language translation [40]. Tools such as GitHub Copilot [9] and Tabnine [37] have been developed to offer these capabilities, providing valuable assistance to software engineers. Most code assistants are built on similar concepts, with the Retrieval Augmented Generation (RAG) framework [8] playing a key role. In these RAG-based code assistant, the codebase is converted into vector embeddings and stored in a vector database during the indexing phase. In response to user queries, these code assistants retrieve relevant code snippets and information from the embedded codebase, helping engineers by offering suggestions based on their own code. Researchers have empirically shown the effectiveness of these tools for various software engineering activities [6, 23].

However, these code assistants face a common challenge. Since they generate responses based on a limited number of relevant code snippets retrieved from the codebase, they cannot view the code structure holistically and fail to provide precise answers to questions related to code structure and dependency. This limitation arises because the retrieved snippets may not capture the full context of the required structure and dependencies, leading to incorrect, inaccurate, or incomplete suggestions. This type of structure and dependency information is crucial when performing large-scale code analysis for comprehension and refactoring [16, 18].

Before the advent of LLMs, traditional static analysis tools were developed and used to support dependency analysis. One such tool is Renaissance, which has proven effective in identifying complex dependency relationships in large codebases and supporting large-scale refactoring with various industry case studies [5, 4]. By parsing the code and modeling it as a code graph stored in a Neo4j graph database [28], the precision and determinism of dependency analysis can be guaranteed. However, achieving wide adoption for code analysis remains challenging due to the complexity of interacting with the code graph using Cypher queries. Cypher [30] is a query language designed for Neo4j graph databases. Previous hands-on sessions with developers suggested that writing Cypher queries is often challenging due to the complexity of the Cypher language, the underlying graph schema, and the analyzed codebase. To make such a code graph more accessible for analysis and enhance LLM-based code assistants to answer dependency questions, we envision that a hybrid solution that augments LLMs with dependency information automatically retrieved from the code graph is promising.

In this paper, we present a GraphRAG solution (as shown in Figure 1) that enables LLMs to access dependency information from a code graph that is extracted from a codebase using Renaissance. When receiving a user question in natural language, the code assistant converts the question into a Cypher query (Text-to-Cypher) that is executed on a Neo4j database that stores the code graph. The retrieved dependency information is then used to generate an answer. This solution differs from other code assistants by generating answers based on dependency information retrieved from a code graph via query generation, rather than based on code snippets retrieved from a vector database via semantic similarity search.

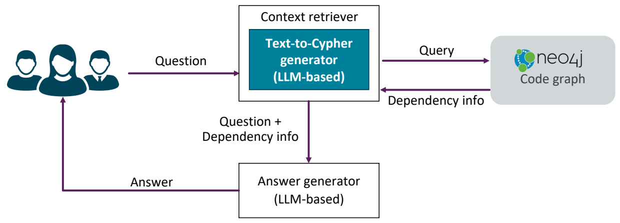

Context retriever
Text-to-Cypher generator (LLM-based)
Question
Query
Dependency info
neo4j
Code graph
Question + Dependency info
Answer
Answer generator (LLM-based)

Fig. 1: GraphRag-based code assistant

We validated our solution at Philips Healthcare which is a global company specializing in advanced imaging, patient monitoring, and healthcare informatics solutions. Firstly, we evaluated the ability of LLMs in translating questions in natural language into Cypher queries for the code graph, using a dataset of 420 questions from industrial engineers. Secondly, we performed a user study with seven industrial engineers. Although the current accuracy is not perfect, the 44% correct response rate in benchmark demonstrates the potential of LLM in automating complex query generation, especially considering the complexity of the code graph schema. This performance represents a meaningful step toward scalable and intelligent code analysis in industrial settings. Our user study further suggests that even when queries are not completely correct, they can serve as useful starting points for engineers.

Our main contribution is threefold: (1) a Text-to-Cypher dataset tailored for code graph analysis, constructed in an industrial setting; (2) an empirical analysis of the ability of LLMs to perform Text-to-Cypher translation for code graph analysis, including a characterization of the errors they commonly make; and (3) a set of suggestions to address common syntax, schema, and semantic errors in the translation. Our work provides practical insights for researchers and practitioners aiming to leverage LLMs in graph-based code analysis, and lay the groundwork for more accurate and scalable query generation for large-scale code graphs.

II. Related work

A. graph-based code assistants

Code assistants have been extensively studied since the rise of LLMs [9, 37, 6, 23]. Most of them rely on semantic similarity search to retrieve relevant contextual information [22]. Recently, researchers have explored searching code graphs to retrieve precise dependency information [24, 31, 25]. Table I compares CodexGraph [24], RepoUnderstander [25] and RepoGraph [31] with our work in several key aspects.

Firstly, previous work has mainly focused on designing code assistants for Python, whereas our work targets C++, a more complex language widely used in embedded systems development. Since we utilize a static analysis tool designed for industry-scale codebases, our graph schema extends beyond conventional function, class, and module relationships. Unlike previous approaches, which typically model dependencies at a more localized level, our schema captures not only local dependencies but also architectural dependencies across modules, projects, and Visual Studio solutions, making it more effective in real-world software development scenarios. Furthermore, these assistants also differ in methods used to retrieve information from graphs. RepoUnderstander applies Monte Carlo Tree Search algorithm which expands the search by selecting child nodes based on exploration-exploitation trade-offs. RepoGraph uses K-hop ego-graph search algorithm [14] which expands the subgraph by retrieving all connected nodes within K hops from the ego node. Both search algorithms grow their search space iteratively, refining which nodes to explore further. However, both methods inherently approximate results rather than guaranteeing an exact match on the graph. Furthermore, these methods are designed for local search and thus cannot answer global questions that require aggregation (e.g., which function has the highest fan-in?). CodexGraph and our work both utilize LLMs for Text-to-Cypher translation in information retrieval. However, the CodexGraph study focuses on different use cases and does not evaluate the accuracy of LLMs in performing this translation.

Secondly, we adopted a distinct evaluation approach, providing new insights into the applicability of graph-based code assistants. Rather than relying solely on open-source benchmarks like SWE-Bench [17], we employed a mixed-method approach that combines a benchmark with an evaluation session with industry engineers. This mixed-method evaluation ensures both objective performance measurement and practical insights from developers, enhancing our understanding of how graph-based code assistants perform in real-world software development scenarios.

B. *Text-to-Query generation*

Our research contributes to the body of knowledge on Text-to-Query generation and evaluates the LLM-based Text-to-Query translator for software engineering applications in an industry setting. One of the query languages studied in Text-to-Query research is SQL (Structured Query Language) [36, 42] which is the standard language for managing and querying relational databases, since it is widely used across industries and many examples are publicly available. In contrast, Text-to-Cypher translation has received relatively limited attention despite growing demand. Nonetheless, several researchers and practitioners have open-sourced both datasets and LLM-based solutions to support this task [26, 12, 38]. The published datasets are designed for applications different from ours, including those in the film industry and specialized medical domains. An open-source repository offers several synthetic datasets for benchmarking LLMs in Text-to-Cypher tasks [21]. Zhong et al. [41] have constructed a synthetic dataset for two medical databases. Ozsoy et al. combined 16 publicly

TABLE I: Comparison with existing studies

	CodexGraph [24]	**RepoUnderstander** [25]	**RepoGraph** [31]	**Our work**
Target Language	Python	Python	Python	C++
Schema	Function, class, and module relationships	Function relationships	Function and class relationships	Relationships among code entities, file system components and Visual Studio project elements
Retrieval Method	Text-to-Cypher translation	Monte Carlo Tree Search algorithm	K-hop ego-graph search algorithm	Text-to-Cypher translation
Use Case	Code completion, issue fixing, and code generation	Issue fixing	Code completion and issue fixing	Q&A about software structure and dependency
Evaluation Method	CrossCodeEval, SWE-Bench, and EvoCodeBench	SWE-bench	CrossCodeEval and SWE-bench	Industry-based benchmark and user study

available datasets into a dataset comprising 44,387 instances. The benchmark study shows that fine-tuned models could achieve better performance than pre-trained models.

Our work made several distinct contributions. Firstly, unlike other studies, we evaluated the generation of Text-to-Cypher for code graphs in industrial contexts, providing valuable insights into its feasibility on software engineering applications. Secondly, we conducted an in-depth analysis to identify the mistakes and limitations of the models in this translation task, going beyond quantitative insights to offer a more comprehensive understanding of potential areas for improvement.

III. LLMs for code dependency analysis

In this section, we describe the architecture of our graph-based code assistant, which is augmented with dependency information represented in a code graph. Next, we introduce the static analysis tool used in this study for code graph extraction. Finally, we present examples of answers given to users by our code assistant.

A. Code assistant architecture

Figure 1 shows the components of our graph-based code assistant. A code graph stored in a Neo4j database is provided as the source of dependency information for the analyzed codebase. When receiving a dependency question in natural language, the component called *Text-to-Cypher* is triggered to translate this question into a Cypher query using an LLM. This LLM is provided with instructions, the user question, the schema of the code graph, and few-shot examples. The generated Cypher query is then executed in the Neo4j database to retrieve the relevant dependency information. The relevant dependency information is then used by an LLM to generate an answer to the question.

B. Code graph

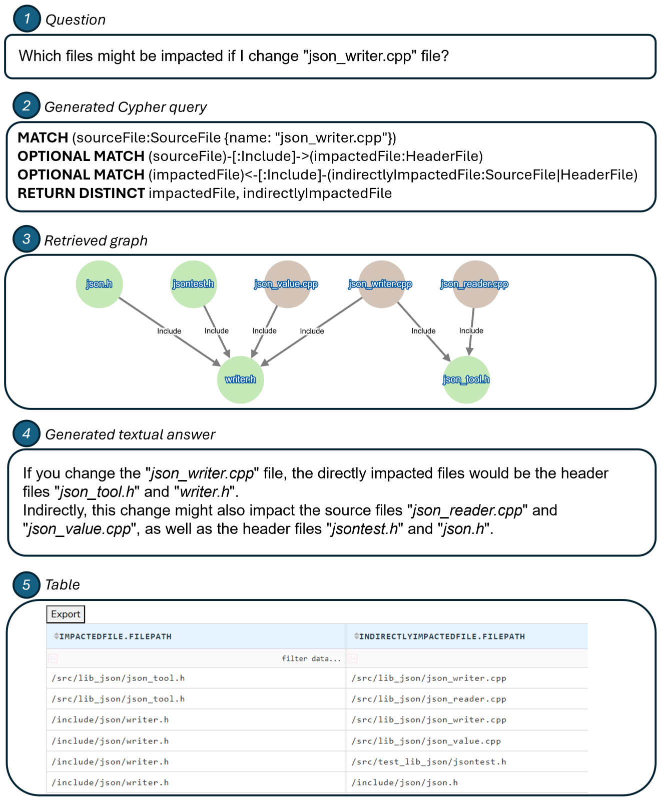
We employed a static analysis tool called Renaissance to extract code graphs from C++ codebases. Renaissance is a specially developed tool for large-scale code analysis and refactoring [5, 4]. The extracted code graph follows a rich schema with 11 node types and 18 relationship types, organized into three main categories to reflect different layers of software development. Code elements (e.g., function definitions, declarations, and classes) capture the core logical structure of the source code. File system components (e.g., files and folders) represent how the code is physically organized on disk, which is essential for understanding dependencies and file-level structure. Visual Studio elements (e.g., projects and solutions) model the development environment and build configuration, providing context on how the code is grouped and compiled. The edges capture relationships across these layers, such as function calls, class inheritance, file inclusions, folder-file hierarchies, and project containment.

Previous case studies in industry [5, 4] have shown that this code graph is effective and useful to analyze architectural dependencies in large-scale codebases. For example, a case study at Philips Healthcare focused on decoupling two components using the code graph not only to identify shared code but also to propose separation strategies, gather insights into architectural dependencies, and track progress throughout the refactoring process. As the users suggested, these insights could not be obtained without such a code graph, given the size of the codebase. However, during the same case study the users noted that interacting with the code graph is non-trivial. To formulate their questions as Cypher queries, they need a good understanding of the Cypher language, the graph schema and the codebase.

By using LLMs to translate user questions into Cypher queries, we would like not only to provide LLMs with dependency information, overcoming the limitations of existing code assistants, but also to make such code graphs more accessible.

C. Answer presentation

Figure 2 shows an example of how our code assistant presents answers to users. It can be seen that the code assistant provides not only the final textual answer generated by the LLM but also the generated queries, retrieved subgraphs, and the table presenting the retrieved information. Engineers can inspect the generated queries and modify them if necessary to regenerate the answers. By examining these intermediate results, engineers can evaluate the quality of the generated answers, increasing transparency and trustworthiness. Furthermore, engineers have the flexibility to choose which representations of the answers to inspect, depending on the type of questions asked. For example, for questions about software metrics, the generated queries and the final textual responses might be sufficient. In contrast, for questions about dependencies between classes and functions, the retrieved subgraph could be more insightful. Additionally, the table is



Which files might be impacted if I change "json_writer.cpp" file?

```
MATCH (sourceFile:SourceFile {name: "json_writer.cpp"})
OPTIONAL MATCH (sourceFile)-[:Include]->(impactedFile:HeaderFile)
OPTIONAL MATCH (impactedFile)<-[:Include]-(indirectlyImpactedFile:SourceFile|HeaderFile)
RETURN DISTINCT impactedFile, indirectlyImpactedFile
```

If you change the "*json_writer.cpp*" file, the directly impacted files would be the header files "*json_tool.h*" and "*writer.h*".
Indirectly, this change might also impact the source files "*json_reader.cpp*" and "*json_value.cpp*", as well as the header files "*jsontest.h*" and "*json.h*".

IMPACTEDFILE.FILEPATH	INDIRECTLYIMPACTEDFILE.FILEPATH
/src/lib_json/json_tool.h	/src/lib_json/json_writer.cpp
/src/lib_json/json_tool.h	/src/lib_json/json_reader.cpp
/include/json/writer.h	/src/lib_json/json_writer.cpp
/include/json/writer.h	/src/lib_json/json_value.cpp
/include/json/writer.h	/src/test_lib_json/jsontest.h
/include/json/writer.h	/include/json/json.h

Fig. 2: Answer presentation

particularly useful when the retrieved subgraph is large and requires further processing, like filtering.

IV. Benchmark

In this section, we describe our evaluation of the ability of LLMs to translate users' inquiries about the structure of their codebase in natural language into Cypher queries.

A. Research questions

The following research questions are formulated to guide our evaluation.

RQ1: *To what extent can pre-trained LLMs accurately translate dependency questions into Cypher queries for retrieving dependency information from the code graph?*

Since LLMs can generate different Cypher queries that retrieve the same information, we use the execution result of queries as a proxy to assess the accuracy of the generated queries, which leads us to the following sub-questions:

RQ1.1: *Are the Cypher queries generated by pre-trained LLMs executable?*

RQ1.2: *How accurate is the dependency information retrieved by the Cypher queries generated by pre-trained LLMs?*

RQ2: *What are the main challenges pre-trained LLMs face in Text-to-Cypher translation?*

We constructed a dataset to quantitatively evaluate the ability of LLMs in Text-to-Cypher translation (**RQ1**). This evaluation addresses two sub-questions. **RQ1.1** studies the ability of LLMs in generating executable queries. **RQ1.2** takes this one step further to study whether LLMs can generate queries that retrieve correct information. We used the execution result of queries as a proxy for measuring query accuracy, as LLMs can generate different queries that still retrieve the necessary dependency information. Next, we conducted an in-depth analysis of cases where LLMs underperform, with four authors analyzing and discussing these instances to gain deeper insights (**RQ2**).

B. Data collection

We constructed a dataset for a code graph comprising 420 data points [1]. Each data point includes a user question in English, an expected Cypher query, the execution result of the expected Cypher query, and a difficulty level label.

a) Code graph: To build a dataset for Text-to-Cypher evaluation, we extracted a code graph using Renaissance from an open-source C++ codebase called jsoncpp [2]. We chose this repository for the following reasons. Firstly, it is written in C++ 14 which can be parsed by Renaissance. Secondly, this repository has been actively maintained by more than 100 contributors. This repository is by no means representative of the size of the repositories that Renaissance can analyze. Our primary objective is to assess whether LLMs can understand the semantic intent of user questions and accurately map them to the corresponding entities, relationships, and attributes defined in the code graph.

The extracted code graph from this repository contains 2751 nodes and 7722 relationships. In order to mimic an evaluation with a closed-source codebase, we obfuscated the code graph by renaming all entities in the graph. For example, the node representing *writer.h* was renamed *headerfile_1.h*. This obfuscation ensures that LLMs cannot generate queries based on their pre-trained knowledge of the codebase. This obfuscated code graph is stored in a Neo4j database.

b) Cypher query and difficulty level: We collected a set of real-life queries adapted to the obfuscated code graph. This involves gathering queries from Renaissance users at Philips Healthcare, modifying these queries to be compatible with the obfuscated code graph, and compiling the adapted queries into a dataset with 420 data points.

Specifically, we performed the following steps. Queries were collected from the users of Renaissance who have a shared database of questions and queries based on their analysis activities performed in practice. For example, one activity involved analyzing the code graph to identify complex dependency networks, which required restructuring and migrating tests to a new library. By collecting them, we would like to investigate whether LLMs can make the code graph more accessible by generating these queries for users. 84 queries were collected and categorized into simple, intermediate, advanced, and complex based on their difficulty. Simple queries only

[1]https://figshare.com/s/2d137b3b5b27b25658f4

[2]https://github.com/open-source-parsers/jsoncpp/tree/master Accessed on May 10, 2024.

search for one type of nodes or one type of relationship with a MATCH clause. Intermediate queries search for multiple types of node and relationship with some simple filtering logic (e.g., using String Comparison Operators such as STARTS WITH). Advanced queries feature multi-step paths, conditional logic including WHERE clauses with multiple conditions, and/or utilize aggregation functions. Complex queries involve complicated pattern matching with complicated conditional logic. Next, these queries were adapted to the obfuscated code graph. For example, if an original query was intended to search for dependency information of a specific class in their proprietary codebase, we adapted it to search for the dependency information of a randomly selected class in the obfuscated code graph.

c) Question and expected result: We collected the original natural language questions corresponding to these queries and prompted Google Gemini 1.5 Pro [10] to generate four additional variants of each question. Each generated question is reviewed and collaboratively refined by authors to ensure clarity and readability. We selected Google Gemini 1.5 Pro as it provides a distinct LLM from the models we evaluate, and it has demonstrated strong performance on various benchmarks, making it a suitable choice for generating initial questions and reducing manual effort. In the final step, we obtained the expected execution result by executing the 420 queries on the Neo4j database that contains the obfuscated code graph. Table II shows the distribution of different difficulty levels in our dataset and an example question for each difficulty level.

TABLE II: Example question for each difficulty level

Difficulty level	Count	Example question
Simple	72	Identify the header files that are dependent on “headerfile_12”.
Intermediate	88	Identify the source files and their associated macro definitions where the macro file path includes “folder_255”.
Advanced	84	Identify C++ declarations that are declared in one source file but used in another, within a maximum of five indirect usages.
Complex	92	List all projects and their associated classes from the solution file located in “folder_188/folder_616”. Include details on the classes, their functions, the header files where these classes are defined, and the inheritance structure of these classes up to five levels deep.

C. Setups

a) Model: We experimented with three state-of-the-art pre-trained LLMs—GPT-3.5-turbo-0125, LLAMA3-70B, and GPT-4o-2024-08-06. They are selected for their strong performance in benchmarks like HumanEval [3].

b) Query execution timeout: Our experimental script processes a total of 420 questions. To prevent resource blocking, we enforced a 10-second timeout for each query executed on the Neo4j database. This threshold was chosen based on empirical testing, balancing performance and responsiveness; most well-formed queries complete well within this time. Queries exceeding 10 seconds often indicate excessive complexity or suboptimal execution plans. Such queries are logged and skipped, allowing the script to continue.

c) Neo4j transaction memory limit: The transaction memory limit in Neo4j defines the maximum memory that a single transaction can utilize during execution. This safeguard prevents excessively large transactions from dominating system memory. Additionally, transactions exceeding this limit typically involve graphs too large to be meaningfully interpreted by humans. In this experiment, we used Neo4j 5.18[3] and its default transaction memory limit of 512 MB.

D. Metric and statistical test

a) Metric: As mentioned in Section IV-A, we assess the accuracy of the retrieved dependency information rather than the accuracy of the generated queries to address **RQ1.2**. The existing metrics in the literature are mostly designed to measure the ability of LLMs in code generation and text generation [13, 32], which are not directly applicable to our case. Since the retrieved dependency information can be represented with subgraphs, we apply a graph metric to compare the expected subgraph (i.e., the execution result of the expected Cypher query) and the retrieved subgraph (i.e., the execution result of the generated Cypher query). Specifically, we use the metric Jaccard graph similarity [19] to quantify the overlap between these two subgraphs G_1 and G_2. The metric formula is shown below:

$$J(G_1, G_2) = \frac{|G_1 \cap G_2|}{|G_1 \cup G_2|}$$

Value 1 indicates that two subgraphs are identical, while Value 0 indicates that there is no overlap between the subgraphs. Values between 0 and 1 represent partial overlap, with higher values indicating greater similarity. We choose Jaccard similarity over precision and recall because it gives a single value that is easy to interpret and compare between models.

b) Statistical test: To compare model performance, we used the pairwise Mann-Whitney U test [27] to analyze the distribution of Jaccard similarity values across 420 questions, as it does not assume normality. The pairwise tests provides a p-value that helps us determine whether the observed differences in Jaccard similarity values between models are statistically significant. We used the standard 0.05 threshold, where p-values below this indicate statistically significant differences in model performance. Conducting multiple pairwise comparisons raises the risk of Type I errors. To address it, we adjusted the p-values using the Bonferroni correction [39].

E. Results

1) Quantitative analysis: We start with our results for **RQ1** that studies the ability of LLMs in accurately retrieving dependency information via Cypher queries.

Figure 3 shows the result of the execution of the Cypher queries generated with these three pre-trained models (**RQ1.1**).

[3]https://neo4j.com/developer/kb/neo4j-supported-versions/

It can be seen that four categories are identified from the execution results. *Executable* queries are runnable in the Neo4j database without returning errors. Queries with *Syntax Error* return error messages indicating syntactic problems. Queries with *Timeout Error* cannot be completed within 10 seconds. The last category includes queries with *Transaction Memory Limit Error*, which occurs when the retrieved subgraph exceeds the memory limit of the database.

It can be observed that, overall, GPT-4o produces more executable queries, with GPT-3.5-turbo coming in second. Promisingly, GPT-4o produces only nine queries with syntactic errors, suggesting that the model has a good knowledge of the current Cypher language grammar (version 9). LLAMA3-70B and GPT-3.5-turbo, however, generate a considerable number of queries with syntax errors. We observe that some of these errors occur when LLMs use legacy grammar constructs that are not present in the current Cypher language grammar. Another observation is that LLMs often introduce syntax errors when attempting to use advanced Cypher libraries (e.g., APOC library). These observations can be explained by the fact that Cypher is an evolving language, and LLMs might not be trained with sufficient examples for up-to-date Cypher grammar constructs and libraries.

Figure 3 shows that GPT-4o produces more timeout errors, particularly with complex questions. While GPT-3.5-turbo and LLAMA3-70B often produce queries with syntax errors in these cases, GPT-4o generates computationally complex queries that lead to timeouts. In this experiment, timeout and memory limit errors account for only 3% of cases but could become more prominent with larger code graphs, indicating a potential scalability concern.

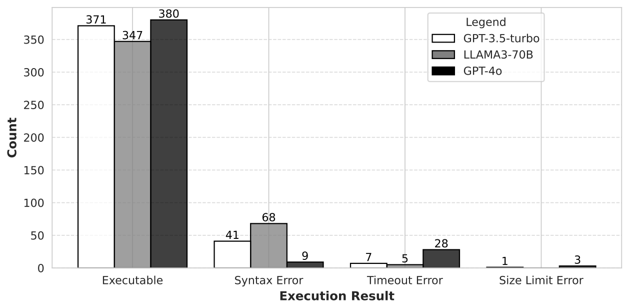


Fig. 3: Execution result of generated queries

Next, let us discuss the accuracy of the retrieved subgraph (**RQ1.2**). When an execution error occurs, the retrieved subgraph is considered to contain an empty set of nodes and edges, resulting in no overlap with the expected subgraph. Consequently, the similarity value is 0. Table III presents the result of the pairwise test while Figure 4 shows the distribution of similarity values across six bins: 0, 0.01–0.25, 0.26–0.50, 0.51–0.75, 0.76–0.99, and 1. Since all the adjusted p-values shown in Table III are below 0.05, we can conclude that the distributions of the similarity values for all the compared pairs are statistically different. It can be seen from Figure 4 that GPT-4o significantly outperforms GPT-3.5-turbo and LLAMA3-70B, resulting in perfect accuracy for 44% of questions. Additionally, LLAMA3-70B significantly outperforms GPT-3.5-turbo. It should be noted that queries resulting in imperfect accuracy (similarity value between 0 and 1) could still be very useful as a starting point, given they capture relevant information. Modifying them with knowledge of the graph schema can often lead to a better result.

TABLE III: Pairwise Mann-Whitney U test results (with Bonferroni correction)

model 1	model 2	p-value	adjusted p-value
GPT-3.5-turbo	LLAMA3-70B	1.09e-02	3.28e-02
GPT-3.5-turbo	GPT-4o	4.42e-15	1.32e-14
LLAMA3-70B	GPT-4o	9.11e-08	2.73e-07

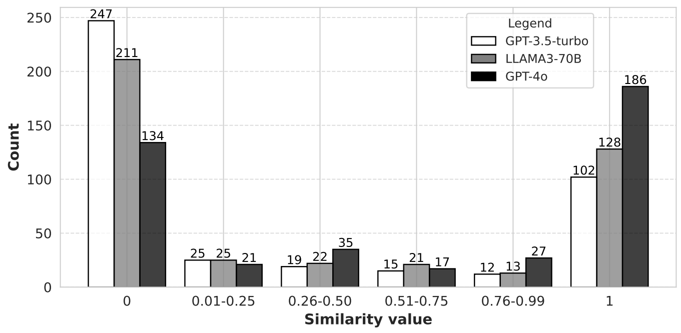


Fig. 4: Accuracy of retrieved subgraphs

Figure 5 focuses on the best-performing model, GPT-4o, and analyzes its performance across different question difficulty levels. It can be observed from this box-plot that most simple questions are translated into queries accurately (with median similarity value 1). However, accuracy decreases as difficulty level increases. This observation suggests that further work is required to answer complex questions effectively.

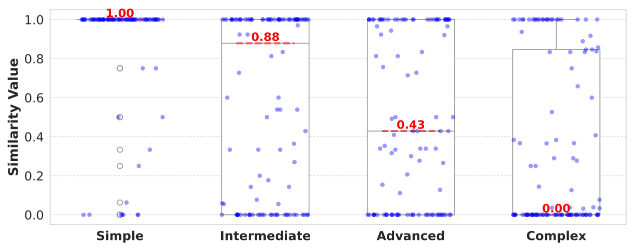


Fig. 5: Box-plot for distribution of similarity values with GPT-4o by difficulty level

2) Source of errors: As mentioned in Section IV-E1, we observe that LLMs could generate queries that are not executable due to syntax errors. We further examine, analyze, and discuss cases where GPT-4o generates executable queries, but the execution results have a similarity score below 1. We identify two main challenges that GPT-4o is still facing (**RQ2**).

a) Violation of graph schema: We observed that 17 queries generated by GPT-4o, while syntactically valid and executable, violate the graph schema. Due to mismatches in node and relationship types, these queries fail to retrieve the intended information.

An observation is that LLMs can invert the direction of the relationship in the generated Cypher statement. This problem has been identified by practitioners. An algorithm that corrects the direction has been proposed and integrated into LangChain[4]. However, our experiments with this algorithm shows that it often fails to fulfill its intended purpose. One key limitation is its reliance on regular expressions to extract node and relationship types from Cypher statements, rather than using pattern matching on the Cypher syntax tree. Consequently, this heuristic approach can misinterpret complex relationships and nested structures, especially in queries with uncommon patterns. An observed limitation is that spaces before or after an arrow symbol ($\rightarrow$ or $\leftarrow$) prevent proper correction. We identified this issue because LLMs often add spaces in generated Cypher statements, which remain syntactically correct. The sensitivity of this heuristic approach to formatting issues reduces its reliability, as minor and valid variations in syntax can affect its effectiveness.

Another observation is that LLMs can hallucinate a relationship type that is not present between two node types. For example, one incorrect query searches for all class declarations (*CppDeclaration* nodes in our schema) that are declared within a function definition (*CppFunctionDefinition* nodes in our schema). However, it uses relationship type *CppCalls* instead of *CppUses*. This violates our schema, as relationship type *CppCalls* is only present between functions, not between classes and functions. These observations suggest that LLMs can generate queries that violate the schema of the code graph. We discuss possible improvements in Section VI-0d.

b) Intent-concept mapping challenge: The main source of error arises when GPT-4o encounters questions involving high-level user intents. In such cases, the model often lacks clarity on how to map these abstract intents to specific graph concepts, such as node types or relationship types. For example, this issue is evident in questions related to class interactions. When asked, "*How does class A interact with other classes?*" without a clear definition of "interaction", the assistant generates a query that searches for direct relationships between classes, namely inheritance and composition (as shown in Query 1). This suggests that GPT-4o maps "class interaction" to inheritance and composition relationships in the graph schema, which might not be the complete and precise definition of class interaction intended by users.

Query 2 is generated by the assistant when asked: "*How does class A interact with other classes through inheritance, composition, and function calls?*". It can be seen that the generated query consists of three matching steps where each intended relationship is searched.

```
MATCH (classA:CppDeclaration {class: "A"})-[:CppInherits|
      CppContains]->(relatedClass:CppDeclaration)
RETURN relatedClass
```

Cypher Query 1: Generated query for question "How does class A interact with other classes?"

[4]https://python.langchain.com/api_reference/community/chains/langchain_community.chains.graph_qa.cypher_utils.CypherQueryCorrector.html

```
MATCH (classA:CppDeclaration {name: "A"})-[:CppInherits]->(
      superclass:CppDeclaration)
RETURN "Inheritance" AS interaction, superclass.name AS
      relatedClass
UNION MATCH (classA:CppDeclaration {name: "A"})-[:CppContains
      ]->(member:CppDeclaration)
RETURN "Composition" AS interaction, member.name AS relatedClass
UNION MATCH (classA:CppDeclaration {name: "A"})-[:CppContains
      ]->(func:CppFunctionDefinition)-[:CppCalls]->(calledFunc:
      CppFunctionDefinition)
RETURN "Function_Call" AS interaction, calledFunc.class AS
      relatedClass
```

Cypher Query 2: Generated query for question "How does class A interact with other classes through inheritance, composition, and function calls?"

In particular, the third part of the query identifies all instances of indirect relationships between classes through function calls, which was not captured by the query shown in Query 1. This example shows that by posing questions in a more detailed manner, it becomes possible to generate a query that more accurately captures user intents. This observation suggests the need for a mechanism to guide users to refine and clarify their intents, helping LLMs map the intentions to specific concepts within the graph schema.

V. User study

Next, we conducted a complementary evaluation with Philips Healthcare engineers to gather user feedback and identify areas for improvement.

A. Methodology

We had evaluation sessions with seven participants from IGT (Image-guided therapy) department of Philips Healthcare. Table IV shows the demographics of these participants. They work on a codebase with over one million lines of code, developed over more than a decade. Since they often need to perform dependency and structure analysis in their daily work, they are considered to be the target users of our code assistant. The extracted code graph for this codebase contains 500K+ nodes and 2M+ edges.

Each evaluation session consisted of three sections. In the first section, we collected the background information of the participant, including their current role, years of experience, and familiarity with C++ and the analyzed codebase. Each session also featured a section wherein the participant was given hands-on experience with our code assistant. Firstly, participants were instructed to experiment with example questions, in preparation for open-ended interaction where they could ask their own questions. This allowed us to observe the behaviour of an engineer and collect a new set of questions for the future validation of our system. In the final section, we gathered their feedback on potential improvements to the tool. Each session was conducted in person and lasted approximately $1.5 \sim 2$ hours. In these sessions, execution logs captured all user queries and the answers of the tool. We encouraged participants to share feedback as they interacted with it. We recorded whether users considered the answers correct and captured their responses. After the sessions, GPT-4o was used to extract common themes and generate labels, which were then validated through manual analysis to determine frequency and identify additional patterns.

TABLE IV: Demographics of Participants

Participant ID	1	2	3	4	5	6	7
Professional Experience (#year)	12	3	2	3	3	11	1
Current Role [1]	PO	D	A	D	D	A	PO
C++ Ability [2]	4	4	4	4	4	5	4

1. D: Developer, A: Architect, PO: Product Owner
2. Participants were asked to rate their C++ proficiency on a 1–5 scale

B. Analysis of evaluation sessions

Across the evaluation sessions, participants asked our tool a total of 91 unique questions. Figure 6 illustrates the distribution of query outcomes based on user interactions. The results are categorized into four distinct response types. We illustrate these categories with examples and anonymize entities in these examples to preserve confidentiality.

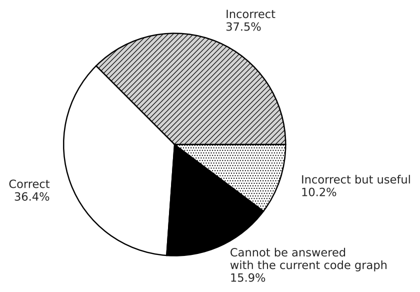


Fig. 6: Pie chart depicting the the success rate of our tool

1. Correct: These occurred when the tool generated queries that provided satisfactory outputs in text, tabular, or graph format. For instance, Participant 6 inquired *"What functions are called 'function1' and what are their namespaces?"*. It is common to have functions with the same name defined in different namespaces, modules, or projects in large-scale codebases. The question is asked to identify these functions invoked in different contexts. The tool generated a query that returned the desired results:

```
MATCH (f:CppFunctionDeclaration|CppFunctionDefinition)
WHERE f.name = "function1"
RETURN f.name, f.namespace
```

2. Incorrect: These cases involved responses that were either irrelevant or failed to assist in problem-solving. For example, Participant 2 asked *"What projects are included by both solution s1.sln and s2.sln that are not test projects?"* This question aimed to identify shared production projects and exclude test projects in a multi-solution Visual Studio setup, which cannot be answered with Visual Studio built-in features. The assistant generated an overly restrictive query that filters project names using an exact match for the string *test*, resulting in an empty subgraph:

```
MATCH (solution1:Solution {name: "solution1.sln"})-[:SlnContains
      ]->(project:ProjectCOrCpp),(solution2:Solution {name: "
      solution2.sln"})-[:SlnContains]->(project)
WHERE NOT project.name = "test"
RETURN DISTINCT project
```

Replacing the exact match with a "contains" condition yields more meaningful results. This indicates that incorporating domain knowledge, such as common naming patterns or project conventions, could guide the generation of more appropriate query conditions and improve result relevance.

3. Incorrect but useful: These responses provided partial solutions or valuable starting points for users. For example, when Participant 7 asked *"If I change 'sourceFile1', what are the impacted functionality?"* the tool produced the query:

```
MATCH (file:SourceFile {name: "soureFile1.cpp"})
OPTIONAL MATCH (file)-[:CppUses]-(func:CppFunctionDeclaration|
      CppFunctionDefinition|CppDeclaration)
RETURN DISTINCT func
```

The user had expected a list of executables but received a list of the impacted functions instead. This acted as a helpful starting point and the Cypher query was successfully edited to return the desired information about the parent files of these functions and hence the expected answer.

4. Cannot be answered with the code graph: These questions required information not captured within the current graph. For example participant 3 asked *"What project has on average the largest number of lines of code per source file?"* Information about the number of lines of code in a source file is missing from the graph, therefore our tool, in its current state, is not be able to answer these questions.

The observed alignment between user session outcomes and the quantitative benchmark provides empirical evidence for the potential effectiveness of the assistant in real-world settings. Both studies used GPT-4o and had success rates in a comparable range: the benchmark showed a correct response rate of 44. 2%, compared to 36. 4% in user evaluation sessions.

C. User feedback

The key areas for improvement and the frequency with which they were identified are as follows:

User Support & Responsiveness *5 mentions*
Users requested more informative responses beyond simple *"I do not know"* statements when the tool cannot provide a helpful answer. Suggestions include providing explanations for query failures, requesting clarification when needed, and offering interactive query building with advice.

Text-to-Cypher Improvements *4 mentions*
Users suggested to focus on Text-to-Cypher capabilities, specifically improvements in complexity handling, consistency, accuracy, and response speed.

Visualizations & Diagrams *4 mentions*
Users expressed interest in automatic diagram generation for class, sequence, and state diagrams based on specified nodes. More general requests include enhanced aesthetics, filtering options, and export functionality for the current graph display.

Graph schema & Structure *3 mentions*
Users emphasized the need for greater transparency in the underlying graph schema. Suggestions include displaying the graph schema alongside the main interface, providing more comprehensive information about queries and data structure,

and implementing type hints for available operations.

Error Handling & UI	*2 mentions*
Users desired improved error handling with clear error messages, alongside general improvements to the user interface.	

In this user study, we confirm the need to improve the Text-to-Cypher translation, as previously identified in our benchmark study. Furthermore, we identified the issues that can only be observed through user interactions with our tool, such as user support and responsiveness.

VI. Possible improvements for Text-to-Cypher

As demonstrated in our benchmark study and user study, LLM-based Text-to-Cypher translation shows promising results. However, further research is needed to enhance its feasibility for industrial applications. In this section, we discuss potential improvements to achieve more accurate information retrieval from large-scale code graphs.

a) Providing LLMs with up-to-date grammar and library knowledge: As observed, LLMs can generate queries with syntax errors. Given that Cypher is an evolving query language with updates to its grammar and libraries, it is essential to provide LLMs with the latest knowledge of Cypher syntax and features. Existing research from the field of code generation for general-purpose languages (GPL) and domain-specific languages (DSL) can be borrowed to address this problem.

One way to address this challenge is by using a RAG approach that incorporates relevant grammar and library information with question-query examples. This can be done by retrieving similar and correct question-query pairs from a database to dynamically construct few-shot prompts tailored to the input question. Another approach is to embed grammar knowledge by fine-tuning LLMs specifically for the current version of Cypher. The comparison between RAG and fine-tuning solutions has been experimented by a research study that aims to generate code in DSLs [2]. Interestingly, the RAG solution delivers quality comparable to the fine-tuned model for the DSL. This suggests that RAG can be an alternative to fine-tuning. It could also simplify the maintenance of LLM-based assistants by removing the need for retraining as DSLs evolve. However, both methods require a dataset with examples that demonstrate the correct use of up-to-date grammar, and neither guarantees syntax-error-free generation.

Non-data-driven solutions to improve code generation have also been studied in the literature. For example, a technique called Syncode [2] incorporates formal language grammar directly into the code generation process. Using grammar rules, Syncode ensures that the generated code adheres strictly to the syntax requirements of the target languages. As shown in their experiment, Syncode offers a more reliable solution for generating syntax-error-free code in the target GPL compared to out-of-box LLMs. However, it remains unclear whether this grammar-based method is feasible and effective for database query languages. We suggest that researchers study these data-driven and non-data-driven solutions and conduct comparative experiments to evaluate their effectiveness.

b) Guiding query regeneration with graph schema checkers: Another potential solution to address syntax errors and graph schema violations is to formally validate whether generated queries conform to the Cypher syntax and graph schema. This can be achieved by parsing Cypher queries into syntax trees using a parser, such as one generated with ANTLR based on the OpenCypher grammar [29]. The syntax tree representation of queries facilitates detailed analysis of each query component, enabling precise detection and correction of syntax and schema inconsistencies before execution. Detected syntax and schema violations can be addressed in two ways. First, some errors can be deterministically resolved by rewriting the query using its syntax tree. For example, arrow direction errors, as discussed in Section IV-E2a, can be corrected this way. Alternatively, these violations can be used as feedback for LLMs, supporting self-reflection and enabling iterative refinement of the generated queries. Combining LLMs with deterministic checker of Cypher syntax and graph schema can greatly increase the quality of the generated queries.

c) Mapping high-level intents into concrete concepts in graph schema: As observed in our experiment, LLMs often struggle in associating words and terms from the user's question with the corresponding types of nodes and relationships in the graph schema. The challenge is that such a mapping is often not one-to-one. For example, the term interaction may correspond to multiple relationship types, depending on the question context and user interpretation. One way to address this could be creating a mapping between common words and terms into concrete types of relationships and nodes. This mapping can be formulated into rules and injected in the prompt. However, this approach does not scale well and lacks flexibility. Alternatively, one could develop an ontology or hierarchical representation of the schema, such as a multi-level graph that captures relationships and compound entities. By leveraging such structures, LLMs can more effectively align user intents with relevant schema elements.

We believe it is essential to keep the user in the loop to accurately capture their intent. When a question is ambiguous, the assistant should be allowed to initiate a lightweight clarification loop with the user. For example, if a user asks about class interaction without clearly defining it, the model could generate a response like, "*There are multiple relationships between classes, such as composition, inheritance and function calls. Which relationships are you interested in?*" This interactive loop allows users to refine their intent and provides the model with more context.

d) Decomposing complex questions into smaller manageable questions: Converting a complex question into a precise query is a common challenge in Text-to-Query research. As discussed in the area of Text-to-SQL, this challenge can be addressed by decomposing a complex question into smaller and manageable sub-questions. For example, Pourreza et al. [33] proposed a methodology to decompose user questions by designing several LLM-based modules to 1) link the question into concrete tables and columns in schemata, 2) classify and decompose the question, 3) generate SQL queries using different strategies based on the type of question, and 4) perform self-correction based on errors returned from execution. Experiments with three LLMs demonstrate that this approach consistently enhances their basic few-shot performance by approximately 10%. A follow-up study has explored fine-tuning LLMs specifically for these decomposition steps [34]. Results indicate that models trained on individual decomposition steps outperform those trained to handle the entire query generation task in a single step. It is interesting to explore if this multi-LLM solution for question decomposition could be useful for Text-to-Cypher generation.

VII. Threats to validity

Threats to **internal validity** relate to factors that may have affected the outcomes of the study. In our benchmark study, we constructed a semi-synthetic dataset by collecting queries and corresponding questions from engineers and inferring variants of questions using an LLM. There is a risk that the generated questions might not accurately reflect the language of the real world user. To mitigate this risk, we reviewed the generated questions to ensure that they accurately captured the queries. We rephrased questions that seemed unnatural or overly simplified to better align with realistic query phrasing. We conducted our benchmark using a code graph derived from an open-source codebase. This setup carries a risk that LLMs might generate responses based on prior knowledge of the codebase, since many open-source repositories have probably been included in the LLMs' pre-training datasets. To mitigate this risk, we obfuscated the code graph by altering the names of entities, thereby removing semantic information embedded in these names. This process preserved the graph structure while anonymizing identifiers, reducing the likelihood that LLMs could leverage prior knowledge specific to this codebase. Furthermore, we conducted a user study using a code graph extracted from a large-scale industry codebase, which confirms our findings and provides new insights into user experience with our assistant.

Threats to **external validity** questions whether our conclusions are valid in a more general context. We used a specific static analysis technique to extract the code graph. This raises the question of whether the schema of our code graph is generalizable to code graphs extracted using other techniques. The graph schema has been successfully applied and validated within several companies for representing relationships in large-scale codebases, demonstrating its usefulness in real-world environments. We acknowledge that variations in static analysis techniques can lead to differences in the resulting code graph schema, which may affect the applicability of our findings to graphs generated by alternative approaches. Additionally, while the user study involved only seven participants and may not be representative of the entire department or broader population, it was designed as an exploratory effort to identify key issues and inform future work. The primary goal of our study is to demonstrate both the potential and the challenges of leveraging LLMs to augment an existing code analysis solution used in industry.

Threats to **construct validity** arise when a study fails to accurately measure the intended concepts, leading to potential misinterpretations of the results. In this study, we used the graph Jaccard similarity metric to evaluate the execution results of Cypher queries, serving as a proxy for the accuracy of generated queries; however, this approach assumes that the overlap reliably reflects correctness, which may not fully capture semantic or logical accuracy. To mitigate this risk, we conducted a user study where engineers provided feedback on the correctness and relevance of the generated queries and their results. This qualitative evaluation triangulates our findings in the benchmark study, providing an additional layer of validation beyond purely quantitative metrics.

VIII. Conclusions and Future work

This paper presents a code assistant that answers questions about code dependencies and structure, complementing existing tools. We augment LLMs with dependency information from a code graph extracted using an industry-proven static analysis tool. Dependency information is retrieved from a Neo4j database via Cypher queries generated by the LLM. We conducted a quantitative analysis to assess LLM performance in Text-to-Cypher translation and a user study with seven engineers to better understand the associated challenges.

Our benchmark results indicate that GPT-4o achieves 44% accuracy in correctly translating questions into Cypher queries. Both the benchmark and user evaluation results reveal that the generated queries can contain syntax errors and violations of the graph schema. Furthermore, LLMs often struggle to link high-level user intents in questions to concrete entities in the graph schema. While the complexity of the code graph schema used in industry presents challenges and results are not yet perfect, this study demonstrates the potential of LLMs for Text-to-Cypher translation in extracting dependency information. It also identifies opportunities for improvement, paving the way for more scalable and automated code analysis in industry settings. As ongoing work, we are addressing several challenges. First, we aim to improve translation accuracy through fine-tuning techniques. Second, we are exploring interactive methods that allow LLMs to ask clarification questions and iteratively refine queries. Finally, we are developing an agent-based assistant with an orchestrator to combine search strategies such as graph traversal and semantic similarity search for handling diverse information needs [35].

IX. Acknowledgment

This work was conducted as part of the LLM4Legacy project, in collaboration with Philips, and funded by Holland High Tech — TKI HSTM through the Public-Private Partnership Innovation Scheme (PPP-I). We gratefully acknowledge the support of Daan van der Munnik and Lena Filatova from Philips, and Anca Lichiardopol from TNO-ESI.

References

[1] Eman Abdullah AlOmar et al. "How to refactor this code? An exploratory study on developer-ChatGPT refactoring conversations". In: *Proceedings of the 21st International Conference on Mining Software Repositories*. 2024, pp. 202–206.

[2] Nastaran Bassamzadeh and Chhaya Methani. "A Comparative Study of DSL Code Generation: Fine-Tuning vs. Optimized Retrieval Augmentation". In: *arXiv preprint arXiv:2407.02742* (2024).

[3] Mark Chen et al. "Evaluating large language models trained on code". In: *arXiv preprint arXiv:2107.03374* (2021).

[4] Dennis Dams et al. "Developing and Applying Custom Static Analysis Tools for Industrial Multi-Language Code Bases." In: *BENEVOL*. 2021.

[5] Dennis Dams et al. "Model-based software restructuring: Lessons from cleaning up COM interfaces in industrial legacy code". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 552–556.

[6] Nicole Davila et al. "An Industry Case Study on Adoption of AI-based Programming Assistants". In: *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*. 2024, pp. 92–102.

[7] Amirhossein Deljouyi. "Understandable Test Generation Through Capture/Replay and LLMs". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 2024, pp. 261–263.

[8] Yunfan Gao et al. "Retrieval-augmented generation for large language models: A survey". In: *arXiv preprint arXiv:2312.10997* (2023).

[9] GitHub. *GitHub Copilot*. Accessed: 2024-11-03. 2023. URL: https://github.com/features/copilot.

[10] Google. *Google Gemini*. Accessed: 2024-11-03. 2023. URL: https://gemini.google.com/.

[11] Lianghong Guo et al. "When to stop? towards efficient code generation in llms with excess token prevention". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2024, pp. 1073–1085.

[12] Markus Hornsteiner et al. "Real-time text-to-cypher query generation with large language models for graph databases". In: *Future Internet* 16.12 (2024), p. 438.

[13] Xinyi Hou et al. "Large language models for software engineering: A systematic literature review". In: *ACM Transactions on Software Engineering and Methodology* (2023).

[14] Yuntong Hu et al. "GRAG: Graph Retrieval-Augmented Generation". In: *arXiv preprint arXiv:2405.16506* (2024).

[15] Rasha Ahmad Husein, Hala Aburajouh, and Cagatay Catal. "Large language models for code completion: A systematic literature review". In: *Computer Standards & Interfaces* (2024), p. 103917.

[16] James Ivers et al. "Industry experiences with large-scale refactoring". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 1544–1554.

[17] Carlos E Jimenez et al. "Swe-bench: Can language models resolve real-world github issues?" In: *arXiv preprint arXiv:2310.06770* (2023).

[18] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. "An empirical study of refactoringchallenges and benefits at microsoft". In: *IEEE Transactions on Software Engineering* 40.7 (2014), pp. 633–649.

[19] Peter M Kogge. "Jaccard coefficients as a potential graph benchmark". In: *2016 IEEE international parallel and distributed processing symposium workshops (IPDPSW)*. IEEE. 2016, pp. 921–928.

[20] Jahnavi Kumar and Sridhar Chimalakonda. "Code summarization without direct access to code-towards exploring federated llms for software engineering". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024, pp. 100–109.

[21] Neo4j Labs. *Text2Cypher Datasets*. https://github.com/neo4j-labs/text2cypher/tree/main/datasets. 2023.

[22] Yichen Li et al. "Enhancing llm-based coding tools through native integration of ide-derived static context". In: *Proceedings of the 1st International Workshop on Large Language Models for Code*. 2024, pp. 70–74.

[23] Jenny T Liang, Chenyang Yang, and Brad A Myers. "A large-scale survey on the usability of ai programming assistants: Successes and challenges". In: *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. 2024, pp. 1–13.

[24] Xiangyan Liu et al. "Codexgraph: Bridging large language models and code repositories via code graph databases". In: *arXiv preprint arXiv:2408.03910* (2024).

[25] Yingwei Ma et al. "How to Understand Whole Software Repository?" In: *arXiv preprint arXiv:2406.01422* (2024).

[26] Ioanna Mandilara et al. "Decoding the Mystery: How Can LLMs Turn Text Into Cypher in Complex Knowledge Graphs?" In: *IEEE Access* (2025).

[27] Patrick E McKnight and Julius Najab. "Mann-Whitney U Test". In: *The Corsini encyclopedia of psychology* (2010), pp. 1–1.

[28] Neo4j, Inc. *Neo4j*. Accessed: 2024-11-03. 2023. URL: https://neo4j.com/.

[29] opencypher. *openCypher Grammar*. https://github.com/opencypher/openCypher/tree/master/grammar. Accessed: 2024-11-10. 2023.

[30] OpenCypher Community. *OpenCypher Resources*. Accessed: 2024-11-21. 2024. URL: https://opencypher.org/resources/.

[31] Siru Ouyang et al. “RepoGraph: Enhancing AI Software Engineering with Repository-level Code Graph”. In: *arXiv preprint arXiv:2410.14684* (2024).

[32] Debalina Ghosh Paul, Hong Zhu, and Ian Bayley. “Benchmarks and Metrics for Evaluations of Code Generation: A Critical Review”. In: *arXiv preprint arXiv:2406.12655* (2024).

[33] Mohammadreza Pourreza and Davood Rafiei. “Din-sql: Decomposed in-context learning of text-to-sql with self-correction”. In: *Advances in Neural Information Processing Systems* 36 (2024).

[34] Mohammadreza Pourreza and Davood Rafiei. “Dts-sql: Decomposed text-to-sql with small large language models”. In: *arXiv preprint arXiv:2402.01117* (2024).

[35] Laurens Prast et al. *Orchestrating Graph and Semantic Searches for Code Analysis*. https://publications.tno.nl/publication/34644253/xS9zUaY0/TNO-2025-R10992.pdf. Accessed: 2025-07-24. Eindhoven, The Netherlands, 2025.

[36] Liang Shi, Zhengju Tang, and Zhi Yang. “A Survey on Employing Large Language Models for Text-to-SQL Tasks”. In: *arXiv preprint arXiv:2407.15186* (2024).

[37] Tabnine. *Tabnine*. Accessed: 2024-11-03. 2023. URL: https://www.tabnine.com/.

[38] Aman Tiwari et al. “Auto-Cypher: Improving LLMs on Cypher generation via LLM-supervised generation-verification framework”. In: *arXiv preprint arXiv:2412.12612* (2024).

[39] Eric W Weisstein. “Bonferroni correction”. In: *https://mathworld. wolfram. com/* (2004).

[40] Zhen Yang et al. “Exploring and unleashing the power of large language models in automated code translation”. In: *Proceedings of the ACM on Software Engineering* 1.FSE (2024), pp. 1585–1608.

[41] Ziije Zhong et al. “SyntheT2C: Generating Synthetic Data for Fine-Tuning Large Language Models on the Text2Cypher Task”. In: *arXiv preprint arXiv:2406.10710* (2024).

[42] Xiaohu Zhu et al. “Large Language Model Enhanced Text-to-SQL Generation: A Survey”. In: *arXiv preprint arXiv:2410.06011* (2024).