

Synergizing LLMs and Knowledge Graphs: A Novel Approach to Software Repository-Related Question Answering

SAMUEL ABEDU, Concordia University, Canada

SAYEDHASSAN KHATOONABADI, Concordia University, Canada

EMAD SHIHAB, Concordia University, Canada

Software repositories contain valuable information for understanding the development process. However, extracting insights from repository data is time-consuming and requires technical expertise. While software engineering chatbots support natural language interactions with repositories, chatbots struggle to understand questions beyond their trained intents and to accurately retrieve the relevant data. This study aims to improve the accuracy of LLM-based chatbots in answering repository-related questions by augmenting them with knowledge graphs. We use a two-step approach: constructing a knowledge graph from repository data, and synergizing the knowledge graph with an LLM to handle natural language questions and answers. We curated 150 questions of varying complexity and evaluated the approach on five popular open-source projects. Our initial results revealed the limitations of the approach, with most errors due to the reasoning ability of the LLM. We therefore applied few-shot chain-of-thought prompting, which improved accuracy to 84%. We also compared against baselines (MSRBot and GPT-4o-search-preview), and our approach performed significantly better. In a task-based user study with 20 participants, users completed more tasks correctly and in less time with our approach, and they reported that it was useful. Our findings demonstrate that LLMs and knowledge graphs are a viable solution for making repository data accessible.

CCS Concepts: • **Software and its engineering** → **Software libraries and repositories; Software notations and tools.**

Additional Key Words and Phrases: Mining Software Repositories, Software Engineering Chatbots, Software Development Assistants, Empirical Software Engineering

ACM Reference Format:

Samuel Abedu, SayedHassan Khatoonabadi, and Emad Shihab. 2026. Synergizing LLMs and Knowledge Graphs: A Novel Approach to Software Repository-Related Question Answering. 1, 1 (February 2026), 42 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

1 INTRODUCTION

Software repositories are rich sources of information essential to the software development process. This includes data on source code, documentation, issue tracking data, and commit histories [66]. Analyzing this data can provide valuable insights about a project, such as developer activities and project evolution [28]. For instance, Begel and Zimmermann [10] and Sharma et al. [60] presented

Authors' addresses: Samuel Abedu, samuel.abedu@mail.concordia.ca, Data-driven Analysis of Software (DAS) Lab, Department of Computer Science & Software Engineering, Concordia University, Montreal, QC, Canada; SayedHassan Khatoonabadi, sayedhassan.khatoonabadi@concordia.ca, Data-driven Analysis of Software (DAS) Lab, Department of Computer Science & Software Engineering, Concordia University, Montreal, QC, Canada; Emad Shihab, emad.shihab@concordia.ca, Data-driven Analysis of Software (DAS) Lab, Department of Computer Science & Software Engineering, Concordia University, Montreal, QC, Canada.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2026/2-ART

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

questions that software practitioners are interested in answering regarding their projects. Question answering is a core part of day-to-day development: prior studies show that developers continuously seek answers about project state, processes, and code to make progress, underscoring the need for tools that surface precise, timely answers [10, 40]. Answering some of these questions requires mining and analyzing repository data. However, accessing and extracting meaningful insights from repositories is time-consuming and requires technical expertise [2, 9]. For example, in a StackOverflow post [35], a user seeking to calculate the number of lines changed since the last commit in a Git repository found that the solution required using specific Git commands like `git diff --shortstat`, which can be challenging for non-technical stakeholders. The technical knowledge and the time spent on such a task can be a barrier to software practitioners.

Prior studies have attempted to address this challenge by developing software engineering chatbots that provide intuitive, natural language interfaces to software repositories [2, 4]. However, a key challenge in software engineering chatbot development lies in natural language understanding (NLU), as the chatbot should accurately interpret user questions and map them to appropriate data retrieval actions [3]. Additionally, the NLU approach to chatbot development fails when the NLU model is not trained on the intent of the user's question. Each intent is typically mapped to a predefined action within the chatbot's framework [7, 56]. However, it is often impractical to define actions for every possible intent, especially as user requirements evolve, limiting the chatbot's functionality and adaptability. Large language models (LLMs) have demonstrated remarkable capabilities in understanding natural language and identifying the intents of input texts [59]. Nonetheless, leveraging LLMs to build chatbots for repository question answering using the naive retrieval-augmented generation (RAG) approach has proved challenging. Abedu et al. [4] reported that LLM-based RAG chatbots failed to retrieve accurate data to answer repository-related questions 83.3% of the time.

Knowledge graphs have the potential to enhance LLMs with external data to generate contextually relevant responses [6, 54]. Knowledge graphs are structured representations that model entities and their relationships, enabling enhanced semantic and structural understanding and reasoning [15, 36, 73]. Due to the structured nature of software repositories, prior studies have modelled software repositories as knowledge graphs to solve various problems in the software engineering domain [46, 48, 77]. Motivated by this, we aim to improve the accuracy of LLM-based chatbots in answering repository-related questions by synergizing LLMs with knowledge graphs. In this study, similar to prior studies, we limit the knowledge graph modelling to Git metadata [48] and focus on answering repository-related questions that are limited to Git metadata, similar to [2, 4].

Our approach is comprised of two key steps: (1) data ingestion and (2) interaction. The data ingestion step has one component, the Knowledge Graph Constructor, which collects and models repository data to construct a knowledge graph. The interaction step consists of three components: the Query Generator, which translates natural language questions into graph queries using an LLM; the Query Executor, which extracts and runs the graph queries against the knowledge graph to retrieve relevant information; and the Response Generator, which generates an answer to the user's question based on the retrieved data using an LLM.

Our approach relies on the ability of LLMs to generate correct graph queries from natural language. Therefore, we conducted an exploratory analysis to assess and identify the most efficient state-of-the-art LLM for graph query generation from natural language input. We evaluated several LLMs, including GPT-4o [53], Llama3 [50], and Claude3.5 [8], on their ability to generate accurate graph queries which we operationalize using Cypher [23] from natural language text and found GPT-4o the most accurate in our context. We integrated GPT-4o with the knowledge graph-based approach and evaluated it on five popular open-source repositories (AutoGPT, Bootstrap, Ohmyzsh, React, Vue) using 150 questions curated from Abdellatif et al. [2]. We first selected a subset of 20

questions spanning the intents and difficulty levels to run the initial evaluation and identify failure modes; we then ran the best configuration on the full set of 150 questions. Questions were grouped into three difficulty levels based on the number of relationships in the knowledge graph needed to retrieve the data. Each question was executed five times with a majority vote for correctness. We compared against MSRBot and GPT-4o-search-preview as baselines and conducted a task-based user study with 20 participants. Specifically, our evaluation aims to answer the following research questions:

RQ1: How effective is our approach in answering software repository-related questions?

We find that synergizing LLMs with knowledge graphs correctly answered repository-related questions 65% of the time. The approach performs well when answering simple questions but struggles with complex questions that require two or more relationships from the knowledge graph.

RQ2: What are the limitations of our approach in accurately answering software repository-related questions?

We find the reasoning of LLM during query generation is the most prevalent limitation affecting LLMs when enhanced with knowledge graphs. It hinders the ability of the LLM to accurately interpret and utilize the right nodes and relationships within the knowledge graph, leading to incorrect relationship modeling, faulty arithmetic logic, misapplied attribute filtering, and misapplied date formatting. Other limitations include the LLM making wrong assumptions and hallucination.

RQ3: Can chain-of-thought prompting improve the effectiveness of our approach in answering software repository-related questions?

We find that the chain-of-thought prompting approach answered the repository-related question 84% of the time, up from an initial 65% without chain-of-thought. Specifically, the accuracy of the complex questions requiring two or more relationships increased from 50% to 90%. This implies that chain-of-thought can help answer complex questions requiring multiple relationships.

RQ4: How do users perceive the usefulness of our approach in assisting them to answer repository-related questions?

We find that the study participants completed more tasks correctly and in less time with the chatbot than with their usual methods, and they perceived the chatbot as useful and time-saving.

Our findings demonstrate that the synergy of LLMs, knowledge graphs, and chain-of-thought prompting can be effective in answering repository-related questions. In summary, we make the following contributions in this paper:

- We provide empirical evidence demonstrating the capabilities of LLMs in generating Cypher queries for querying software repository data stored/represented in knowledge graphs.
- We discuss the limitations of augmenting LLMs with software repository data stored/represented in knowledge graphs.
- To the best of our knowledge, this is the first software repository question-answering approach based on knowledge graphs.
- We share the dataset and scripts for reproducibility and advancing the field at [5].

Paper organization. The rest of the paper is structured as follows: We begin by explaining the concepts in this paper with the related works in Section 2. We present our approach in Section 3 and the evaluation set-up in Section 4. We present the results of our research questions in Section 5 and discuss the findings in Section 6. We outline the threats that can affect the validity of our results in Section 7 and conclude the paper in Section 8.

2 BACKGROUND & RELATED WORKS

In this section, we provide an overview of the key concepts that form the foundation of our study. We discuss software repositories and their significance, software engineering chatbots, and knowledge graphs.

2.1 Software Repositories

Software repositories contain data that track the development process of a project [28]. Platforms like GitHub and Jira provide version control systems that facilitate collaboration among developers, track changes over time, and support issue tracking and project management. Software repositories contain a wealth of information, including details about commits, pull requests, issues, and developer activities.

Prior studies have analyzed repository data to investigate and understand various development processes. For instance, Dilhara et al. [19] conducted a large-scale analysis of commit data on GitHub to understand the evolution of machine learning library usage in open-source projects. Hata et al. [29] conducted a mixed-methods study to understand how developers use GitHub's Discussions feature by analyzing early adopters. Khatoonabadi et al. [38] utilized pull request data from 20 open-source projects from GitHub to develop a machine learning approach to predict the first response latency of both maintainers and contributors during the pull request review.

Accessing and interpreting this information is crucial for various stakeholders, including developers, project managers, and non-technical team members. However, analyzing and extracting meaningful insights from these data can be challenging without specialized knowledge and also time-consuming [2, 9].

2.2 Software Engineering Chatbots

Chatbots are conversational assistants designed to assist with specific tasks by interacting with users through natural language [56]. They aim to facilitate access to information, automate routine tasks, and support collaboration among team members [1]. Evidence from recent studies underscores that question-answering has become central to developer workflows: controlled and field studies report sizable productivity gains from conversational assistants, while surveys show widespread but cautious adoption of AI assistants for code search, explanation, and guidance [39, 55, 64]. Chatbots are increasingly becoming popular in the software engineering domain to accomplish specific software engineering tasks. For instance, Abdellatif et al. [2] proposed MSRBot, using a bot layered on top of software repositories to automate and simplify the extraction of useful information from the repository. Bradley et al. [12] proposed Devy, a Conversational Developer Assistant that enables developers to focus on high-level tasks by reducing the need for manual low-level commands across various tools. Dominic et al. [20] proposed a conversational bot to support newcomers in onboarding to open-source projects by recommending suitable projects, resources, and mentors. Okanović et al. [51] proposed PerformoBot, a chatbot that guides developers through configuring and executing load tests via natural language conversations. Also, Abedu et al. [4] developed an LLM-based chatbot to answer questions related to software repositories. Their LLM-based chatbot, which used the RAG approach, failed to retrieve the relevant data needed to answer questions in their evaluation questions most of the time. Beyond these task-specific bots, recent code-aware assistants such as GitHub Copilot Chat, CodeT5+, and StarCoder bring conversational capabilities into the IDE, supporting code generation, explanation, and navigation; however, they primarily target source-level reasoning rather than repository metadata Q&A across issues and commits [70, 79].

Recent developments have introduced agent-based systems for repository interaction. OpenDevin [69] is an open-source framework designed to replicate the capabilities of autonomous AI software engineers like Devin. It creates agents capable of executing complex software engineering tasks, including bug fixes and feature implementations, by integrating multiple tools such as a bash shell, code editor, and web browser. The framework is evaluated on benchmarks like SWE-bench, which measures the ability to resolve real GitHub issues by generating and testing code patches, focusing on code modification capabilities rather than repository metadata analysis. DeepWiki-open¹ is an open-source implementation of DeepWiki, a tool for automatically generating documentation and interactive wikis for repositories. It clones target repositories from platforms like GitHub, GitLab, or BitBucket and analyzes their code structure to create comprehensive wikis. DeepWiki-open features two key capabilities: “Ask” a RAG-powered chat interface that allows developers to query code semantics, and “DeepResearch” a multi-turn reasoning process that investigates complex technical topics within a repository. The system creates semantic embeddings of code files to support retrieval and is evaluated on the CodeWiki benchmark [31], which assesses documentation generation quality and code understanding. While these tools demonstrate the growing interest in AI-assisted repository interaction, they focus on different problem domains. For instance, OpenDevin targets autonomous coding tasks, while DeepWiki focuses on documentation and semantic code search. Our work complements these efforts by enabling structured analytical queries over repository metadata like commits, issues and users.

The increasing application of chatbots in software engineering and LLMs in chatbots like ChatGPT and BARD motivates our work to improve the accuracy of LLMs in software engineering chatbots. To the best of our knowledge, this is the first software repository question-answering approach based on knowledge graphs and LLMs. Abdellatif et al. [2] and Abedu et al. [4] are closest to our work. Like MSRBot [2], we follow a similar interaction pattern, where a user asks a question; the approach queries a repository’s data and then generates an answer for the user. Also, we evaluate our approach using the evaluation dataset from Abdellatif et al. [2]. However, unlike Abdellatif et al. [2]’s fixed intent/entity pipeline, our work uses an LLM generator with a repository knowledge graph, enabling support for more user questions with variable intents. Abedu et al. [4] showed that RAG-based LLM chatbots often fail because they retrieve irrelevant context; we address this by grounding the LLM in a structured knowledge graph that supplies precise repository facts rather than unstructured snippets, reducing retrieval mismatch and improving answer reliability [6, 54]. While agent-based systems like OpenDevin and DeepWiki enable code modification and documentation tasks, and Copilot-style systems allow users to interact with their repository at the code level, our focus is on allowing users to interact at the repository-metadata level.

2.3 Knowledge Graphs and Large Language Models

Knowledge graphs are structured representations of information that model entities (nodes) and the relationships (edges) between them [32]. They effectively organize and represent knowledge as triple facts (*head entity, relationship, tail entity*), allowing it to be efficiently utilized in advanced applications [15, 73]. Popularized by Google’s introduction in 2012 [62], knowledge graphs have been widely used in domains such as the semantic web, natural language processing, and recommendation systems [36].

In the software engineering domain, prior studies have represented software repositories as knowledge graphs. For instance, Zhao et al. [77] proposed GitGraph, a prototype tool that automatically constructs knowledge graphs from Git repositories to help developers and project managers comprehend software projects. Malik et al. [48] introduced a method for representing software

¹<https://github.com/deepwiki-open/deepwiki>

repositories as graphs to preserve the context between different features during anonymization for data sharing in software analytics. Additionally, Ma et al. [46] developed RepoUnderstander, a method that condenses critical information from entire software repositories into a repository knowledge graph to guide agents in comprehensively understanding the repositories.

By structuring repository data into a knowledge graph, it becomes possible to perform complex queries and infer new knowledge through graph traversal and pattern matching. Query languages like Cypher, used with graph databases such as Neo4j, Redis graphs, and MemGraph, enable querying of knowledge graphs using declarative language [23].

LLMs are effective at natural language but can hallucinate, lack provenance, and provide out-of-date information. Knowledge graphs provide explicit and verifiable facts with typed relations and support multi-hop querying. Integrating the two technologies improves LLM outputs and accuracy. Prior studies on this combination outline these benefits. For instance, Li et al. [45] converts tables into graphs and guides step-by-step reasoning, which filters noise and yields higher QA accuracy than text-based baselines. Xu et al. [74] retrieve entities, relations, and subgraphs and align them before prompting, which improves answer accuracy and logical form over text retrieval. Lavrinovics et al. [43] review knowledge graph-based strategies that reduce hallucinations during pretraining, inference, and post-generation, and they call for stronger evaluation. Hogan et al. [33] show that systems that delegate multi-hop facts to knowledge graphs, freshness to search, and fluency to LLMs deliver more reliable answers. Sequeda et al. [58] argue that knowledge graphs enable trust and provenance in enterprise QA, and they report better auditability and governance when answers are checked against knowledge graph facts.

For repository question answering, combining LLMs with a knowledge graph is a viable approach because a repository knowledge graph provides verifiable facts that ground the LLMs' responses and improve answer accuracy. To the best of our knowledge, this is the first study to address repository question answering using an LLM enhanced with a knowledge graph.

2.4 Prompt Engineering for LLMs in Software Engineering

Large language models are widely used for software engineering tasks such as code generation, summarization, and commenting. Prompt engineering guides these models with task-specific instructions without changing model weights. A recent systematic review by Hou et al. [34] lists the common prompt engineering techniques used in recent software engineering studies. These include zero-shot, few-shot, chain-of-thought (CoT), automatic prompt engineering (APE), prompt-based continuous prompting (PromptCS), Chain-of-Code (CoC), modular-of-thought (MoT), and structured chain-of-thought (SCoT), and reports the broad use of few-shot and CoT prompting [34].

Geng et al. [24] investigate multi-intent comment generation and report that providing ten or more in-context examples in the prompt (few-shot learning) enables diverse comments and outperforms a supervised baseline. Xu et al. [75] propose UniLog for automatic logging and show that five demonstration examples in the prompt (few-shot learning), with example selection and ordering, improve insertion quality without model tuning. Wu et al. [72] study security repair and use prompts that mark buggy and fixed regions (e.g., "BUG:"/"FIXED:"), finding that success is limited and mostly on simple cases.

Shin et al. [61] compares GPT-4 under basic, in-context, and task-specific prompts to fine-tuned models across summarization, generation, and translation, and reports that prompt-engineered GPT-4 is on par with fine-tuned models and offers ease of use, especially with a natural language interface. These studies show that prompt engineering can improve results of LLM applications on SE tasks, especially with few-shot and CoT prompting [34, 61].

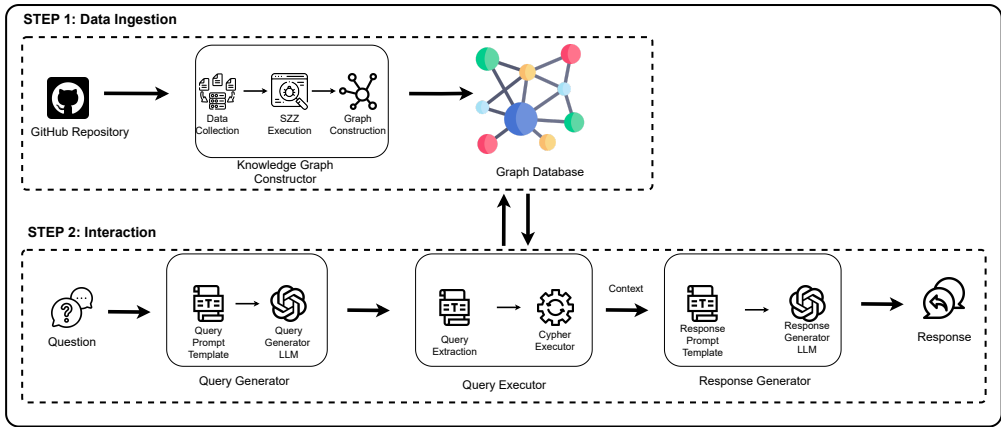


Fig. 1. Overview of our approach in answering software repository-related questions by synergizing LLMs and knowledge graphs.

3 APPROACH

Figure 1 provides an overview of our approach to answering repository-related questions. Our approach consists of four key components organized into two steps: (1) data ingestion and (2) interaction. In the data ingestion step, the **Knowledge Graph Constructor** component collects repository data and models it as a knowledge graph. During the interaction step, the **Query Generator** component takes the user’s natural language question as input and generates a graph query using an LLM to retrieve the relevant data required to answer the question. The **Query Executor** component then takes the generated query from the Query Generator component and executes it. It returns the results of the query, which are used by the **Response Generator** component as context to generate a natural language response to the user’s question using an LLM. In this section, we describe each component of our approach in detail, using the question “How many people have contributed to the code?” as our running example.

3.1 Knowledge Graph Constructor

The Knowledge Graph Constructor component aims to connect the entities in the software repository to form the repository knowledge graph. Using a knowledge graph allows us to model the complex relationships between the repository entities, facilitating analysis and inference of the repository data. Given the size of the official GitHub schema [27], we restrict the graph to four entities (*Users*, *Commits*, *Issues*, and *Files*) and their relationships. These four entities capture the core workflows we evaluate (developer activity, commit history, and defect lifecycle), enabling multi-hop analysis while keeping the schema manageable for verification and traceability of responses. The knowledge graph constructor collects the following types of data:

- **Commits:** Information about each commit to track code changes, authorship, and contributions over time.
- **Issues:** Details of issues (bugs) to track reported problems and identify their introducing and fixing commits.
- **Files:** File structures and changes over time to track modifications and identify files impacted by bugs.
- **Users:** Contributor information to analyze developer activities and contributions.

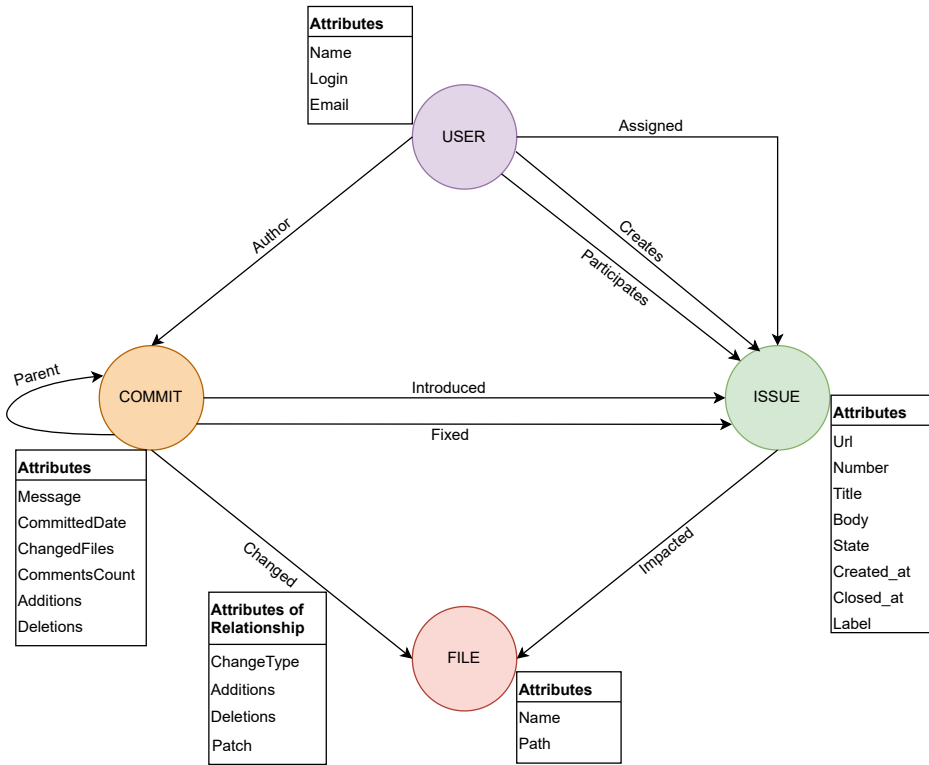


Fig. 2. Overview of the schema of the knowledge graph used in this study. The circles represent the entities (Nodes), the directed arrows represent the relationships (Edges), and the boxes show the attributes.

The knowledge graph constructor also identifies the bug-fixing and bug-introducing commits. Similar to prior studies [17], our approach identifies the bug-fixing commits by searching for the bug ID in the change logs of the commits. Then it identifies the buggy changes by employing the Davies et al. [18] variation of the SZZ algorithm referenced as R-SZZ [17]. The SZZ algorithm [63] is a widely used method in software engineering for detecting bug-introducing changes. The R-SZZ variation uses textual and dependence-related changes to improve on the original SZZ algorithm [18].

After the data collection and SZZ execution, it constructs the knowledge graph. Similar to prior study [48], we define the schema of the knowledge graph by establishing the relationship between the entities in the GitHub repository. Figure 2 shows an overview of the entities and relationships in the schema of our knowledge graph. A description of the relationships between the entities is presented in Table 1.

For entities that continuously change during the lifespan of the repository, such as Files, we assign their evolving attributes to the relationships rather than to the nodes. For instance, if a commit changes a file, the change type (added, deleted, renamed) is assigned as an attribute to the changed relationship between the commit and the file, not as an attribute of the file itself. After the construction of the knowledge graph, we store the knowledge graph in a graph database to allow for querying.

Table 1. Description of relationships in our knowledge graph. The relationship between two entities is represented as (*head entity*, relationship, *tail entity*)

Relationship	Description
(<i>User</i> , Author, <i>Commit</i>)	Indicates a <i>User</i> who authored a <i>Commit</i> .
(<i>User</i> , Assigned, <i>Issue</i>)	Indicates a <i>User</i> who is assigned to an <i>Issue</i> .
(<i>User</i> , Create, <i>Issue</i>)	Indicates a <i>User</i> who created an <i>Issue</i> .
(<i>User</i> , Participates, <i>Issue</i>)	Indicates a <i>User</i> who participated in the <i>Issue</i> discussion.
(<i>Commit</i> , Parentof, <i>Commit</i>)	Indicates that a <i>Commit</i> is the parent of another <i>Commit</i> .
(<i>Commit</i> , Introduced, <i>Issue</i>)	Indicates a <i>Commit</i> that introduced or caused an <i>Issue</i> .
(<i>Commit</i> , Fixed, <i>Issue</i>)	Indicates a <i>Commit</i> that fixed an <i>Issue</i> .
(<i>Commit</i> , Changed, <i>File</i>)	Indicates a <i>Commit</i> that modified (added, deleted, renamed, modified) a <i>File</i> . This relationship has properties indicating the type of change, the number of lines added to the file (additions), the number of lines deleted (deletions), and the changes (patch).
(<i>Issue</i> , Impacted, <i>File</i>)	Indicates an <i>Issue</i> is related to or impacted the changes in a <i>File</i> .

3.2 Query Generator

An essential step in our approach is retrieving the relevant information to answer a user’s question. The Query Generator component aims to generate graph queries that correspond to the user’s questions. In this study, we operationalize the graph query using Cypher, an evolving query language for graph databases that is supported by Neo4j, Redis Graph, and Memgraph [23].

The Query Generator uses an LLM to generate the Cypher query. The LLM uses the entities and relationships in the schema of the knowledge graph to generate the Cypher query using the prompt template shown in Figure 3. The prompt follows guidelines and best practices for prompt engineering [52] and accepts three main parameters to generate the Cypher query: (1) the current date and time, (2) the schema of the knowledge graph, and (3) the user’s natural language question. The current date and time were added to inform the LLM in answering questions requiring relative dates, such as “*How many commits from last month*”. The schema of the knowledge graph informs the LLM of the types of entities and relationships in the knowledge graph.

In our running example “*How many people have contributed to the code*”, the Query Generator component uses the schema of the knowledge and the question to generate the text containing the MATCH (u:User)-[:author]->(c:Commit) RETURN COUNT(u) AS contributors to get the number of contributors in the project.

3.3 Query Executor

To generate the response to the user’s question, we have to retrieve and pass the relevant information from the knowledge graph to the Response Generator. We achieve this through the Query Executor, which takes the generated output of the Query Generator and executes it. Although the Query Generator is prompted to only return the Cypher query, there are instances where it returns additional texts to the Cypher query, which can result in a syntax error when executed. As a result, the Query Executor component extracts the Cypher statement from the output of the Query Generator as a means of quality control using regular expression matching.

The Query Executor then executes the extracted Cypher query, returning the results from the knowledge graph database. The result is passed to the Response Generator component to generate a natural language response for the user. In the running example, the Query Executor extracts the

You are an AI assistant, your task is to generate a Cypher statement to query a Neo4j graph database by following the instructions below.

Instructions:
Use only the provided relationship types and properties in the schema.

The current date is {current_date}.

If the user query contains a date or datetime, format it in the iso format like "YYYY-MM-DDTHH:MM:SSZ" and if the datetime is without the timestamp, use the regex for the missing part.

Do not include any text except the generated Cypher statement.

Schema:
{schema}

The question is:
{question}

Fig. 3. Prompt template used by the Query Generator LLM. The prompt includes the current date and time, the schema of the knowledge graph, and the user's question.

Cypher MATCH (u:User)-[:author]->(c:Commit) RETURN COUNT(u) AS contributors from the generated text. It then executes the query and returns the result [contributors: 40], assuming there are 40 contributors.

3.4 Response Generator

The goal of the Response Generator Component is to generate a natural language response to the user's question based on the results returned by the Query Executor. The Response Generator prompts the Response Generator LLM to generate the natural language response using the prompt template shown in Figure 4. The prompt template accepts four parameters: (1) the schema of the knowledge graph, (2) the generated Cypher query serving as additional context for interpreting the results and generating an appropriate answer to the question, (3) the context, which is the results from the Query Executor, and (4) the user's question. To prevent hallucination, we instructed the LLM to respond "I don't know", if it is not sure of the answer and not make up a response. In the running example, the Response Generator takes the results of the query, the question, the schema, and the Cypher query and returns the natural language response "A total of 40 people have contributed to the code".

4 EVALUATION SETUP

The main goal of this study is to improve the accuracy of LLM-based chatbots in answering repository-related questions. In this section, we present the evaluation setup for our approach in detail. We begin by explaining the criteria for selecting the projects for the evaluation of our approach. Finally, we discuss the questions used for the evaluation and the implementation of our approach.

4.1 Selected Project

For this study, we selected software projects from GitHub based on a set of criteria for our evaluation. We selected the projects based on their popularity on GitHub. We used the number of stars of a project as a proxy for identifying the most popular projects on GitHub [11]. However, some of

You are an AI assistant for generating answers to software repository question answering task.

Instructions:
 You will be provided with a schema of a software repository represented as a knowledge graph, the graph query for the question, and the context (results of the graph) for answering the question.

Your task is only to generate a natural language response to the question using the context.

If you are not sure of the answer, then say that you don't know, can I help with anything else, don't try to make up an answer.

Schema: {schema}

Graph_query: {graph_query}

Context: {context}

Question: {question}

Answer:

Fig. 4. Prompt template used by the Response Generator LLM. The prompt includes the schema of the knowledge graph, the generated Cypher query for the question, the results returned from executing the Cypher query, and the question.

Table 2. Overview of the selected projects for evaluating our approach

Project	Domain	Stars	Language	Commits	Issues	Contributors
React	Web framework	225,068	JavaScript	19,008	13,090	416
Vue	Web framework	207,371	TypeScript	3,592	12,545	358
Ohmyzsh	Systems utility	170,736	Shell	7,295	12,034	392
Bootstrap	Web framework	168,051	JavaScript	22,833	37,634	361
AutoGPT	AI framework	163,726	Python	5,373	6,169	440

the most popular projects on GitHub are not software projects, such as a collection of awesome projects or educational projects. Therefore, we excluded projects that are not software projects, for example, the free-programming-books project [21]. In addition, we required that the projects have their code and issue tracking data on GitHub. This requirement ensures that all relevant development activities for the construction of the knowledge graph discussed in Section 3.1 are accessible through a unified platform, facilitating comprehensive data collection. For example, the Linux [65] project is one of the most popular projects on GitHub, but it was excluded because the issue tracking is not on GitHub. Also, we required that for commits that are fixing or closing issues in the project, the commit log should reference the issue id, for example, “fixes issue #123”. This linkage is for accurately mapping issues to their fixing changes when constructing the knowledge graph and serves as a start to progressively identify the bug introducing commits [18].

Based on these criteria, we selected five popular open-source projects shown in Table 2. The selected projects cover various domains and programming languages and have a median number of 170,736 stars, 7,295 commits, 12,545 issues, and 392 contributors. The data were collected on August 19, 2024.

4.2 Evaluation Questions

In evaluating the approach, we curated the questions by Abdellatif et al. [2], which they collected from 12 users interacting with software repositories to access various information for the completion of tasks assigned to them. These tasks include finding answers to questions that are commonly asked by developers and non-technical stakeholders, for instance, finding the commit that introduced a bug or the developer that fixed the most bugs [10, 60]. The users asked 165 questions representing 10 distinct intents, where each intent refers to the mapping between the user's question and a predefined action to be taken to complete the task [7].

Out of the 165 questions in the dataset, we exclude 15 questions that are not relevant to the evaluation. For example, *"What's your name?"*. We use the remaining questions as templates for forming the evaluation questions for each project. This involves inserting project-specific parameters into the question template to form the question. For instance, in the original dataset, for the question *"Who fixed the most bugs in the file HibernateEntityManager?"*, we replace the file `HibernateEntityManager` with a file specific to the project we are evaluating. Executing all 150 questions for each project would be expensive; therefore, we create a subset of the dataset by selecting two questions per intent, resulting in an evaluation set of 20 question templates. The two questions were selected for each intent based on the clarity of their phrasing. This approach reduces the cost of executions while ensuring variety by covering each of the 10 intents with two variations of questions.

Table 3 presents the classification of the selected questions into three difficulty levels, allowing for a more fine-grained evaluation of our approach. We define difficulty as the number of relationships in the knowledge graph required to answer the question. The level one questions include questions that only require a single entity and not a relationship to answer. For example, *"What is the latest commit"* only requires the `Commit` entity. Level two questions require a single relationship to answer. For example, *"Which commit fixed the bug X"* requires the `Commit` entity and the `Issue` entity linked by the `fixed` relationship. Lastly, the level three questions require two or more relationships to answer. For example, *"Determine the percentage of fixing commits that introduced bugs in June 2018"* requires the `Commit` entity, the `Issue` entity, the `fixed` relationship and `introduced` relationship. The 20 questions used for the evaluation with their corresponding intent and difficulty level can be found in Table 18 (Appendix A).

To establish the ground truth for our evaluation, the first author manually wrote Cypher queries corresponding to all 20 questions for each of the selected repositories. To ensure the correctness of these queries and eliminate potential bias, the authors collaboratively reviewed and discussed the logic employed in each query, adding an additional layer of scrutiny. The Cypher queries were then executed against the knowledge graphs, and the resulting outputs were used as the ground truth for comparison in our study. This process was repeated for the remaining 130 questions to establish the ground truth for the complete 150 dataset.

4.3 Implementation

We implement the approach discussed in Section 3 using Python and the Langchain framework. The Knowledge Graph Constructor begins the process by collecting data from the software repository using the GitHub GraphQL API [25]. We opted for the GraphQL API over the REST API [26] because GraphQL allows us to specify precisely the data we need in a single request, reducing the noise and the size of the document returned compared to the REST API and improving processing efficiency [13]. The collected data included information on users, commits, issues, and files (see Section 3.1). After collecting the data, we implemented the relationship between the entities following the schema defined in Figure 2. To store and manage the knowledge graph, we utilize the

Table 3. Definition of difficulty levels along with example questions and corresponding Cypher queries

Level	Definition	#	Example	Cypher Query
1	Questions requiring only a single entity, no relationship needed	4	What is the latest commit?	<pre>MATCH (c:Commit) RETURN c ORDER BY c.committedDate DESC LIMIT 1</pre>
2	Questions requiring one relationship	12	Determine the developers that had the most unfixed bugs?	<pre>MATCH (u:User)-[:assigned]->(i:Issue) WHERE i.state = "open" RETURN u, COUNT(i) AS openBugs ORDER BY openBugs DESC</pre>
3	Questions requiring two or more relationships	4	Determine the developers that fixed the most bugs in ReactDOMInput.js?	<pre>MATCH (u:User)-[:author]->(c:Commit) -[:fixed]->(i:Issue) -[:impacted]->(f:File {name: "ReactDOMInput.js"}) RETURN u, COUNT(i) AS fixedBugs ORDER BY fixedBugs DESC</pre>

Neo4j database, which is known for its robustness and maturity in handling graph data structures and widely adopted in prior studies [48]. Its compatibility with the Cypher query language enables efficient querying and manipulation of the graph data [23].

In the Query Generator and Response Generator components, we use OpenAI’s GPT-4o model through OpenAI’s API to translate the user’s natural language questions into Cypher queries and also generate user-friendly responses. The selection of the GPT-4o model is based on its performance in our exploratory question (discussed in Section 5.1). For the implementation of our approach, we use the default setting of the model except in the Query Generator component, where we set the temperature of the model to 0. We use temperature 0 to reduce the randomness in the generated Cypher queries and main consistency [44].

After the Cypher query is generated, it is executed using the Neo4j Python library, which provides a straightforward interface for communicating with the Neo4j database. This library enables the approach to run the query and retrieve the relevant results from the knowledge graph efficiently for response generation.

4.4 Baseline Set Up

We benchmark our approach against two baseline approaches, the MSRBot framework [2] and the GPT-4o model with web searching capability. The MSRBot framework is an intent-based chatbot framework that maps user questions to predefined intents and also extracts the relevant entities from the question to retrieve the relevant repository data. In the implementation of MSRBot, Google Dialogflow was used for natural language understanding. However, the authors explained that other NLU platforms can be used in the framework and further evaluated different NLU platforms [1]. In the study, Abdellatif et al. [1] found that Rasa NLU produces stable confidence scores (median > 0.91) and generally outperforms Dialogflow in confidence reliability. As a result, in this study we implement the MSRBot framework with the Rasa NLU.

The second baseline approach we consider is an open-domain LLM-based chatbot that can handle repository-related questions by augmenting a large language model with retrieval. A common method for this is RAG. The RAG approach integrates an LLM with a knowledge base: user queries trigger a search in an indexed repository dataset, and the retrieved documents are then used by the LLM to generate an answer. In principle, RAG allows the bot to provide up-to-date, factual answers even about dynamic repository data. We initially considered a RAG-based baseline, similar to the one explored by Abedu et al. [4] for mining software repositories. However, we do not include the

RAG baseline in our evaluation. The reason is that the RAG chatbot showed poor accuracy, failing to answer most of the questions in the MSRBot dataset. The authors explained the low accuracy as a result of retrieving irrelevant data or failing to generate correct answers. This outcome highlights that a naive RAG implementation is ineffective for this evaluation, as reported by Abedu et al. [4]. Given this finding, a RAG baseline would not be a competitive baseline. Instead, we replace the RAG approach with an LLM with web-search capability baseline: GPT-4o with web search capabilities (the GPT-4o-search-preview model). This baseline uses OpenAI's GPT-4o, which is a state-of-the-art LLM, augmented with the ability to perform live web searches. The GPT-4 model can thus retrieve current information from online sources (including public GitHub repositories) and incorporate that knowledge into its responses. This serves a similar purpose to the RAG approach, providing the language model with access to relevant up-to-date information.

4.5 Survey Design

To evaluate the usefulness of our approach in assisting users with repository-related questions, we designed a task-based user study. This study follows a methodology similar to that of Abdellatif et al. [2]. Participants are asked to perform a set of software repository-related tasks: first, using conventional tools (baseline), and then using our chatbot-based approach. By comparing the performance of the tasks in these two settings, we assess the usefulness of our approach for task accuracy and efficiency.

We adopted the 10 tasks performed by participants in Abdellatif et al. [2]. These tasks cover a range of questions developers ask about their repositories. To avoid overburdening participants, we divided the ten tasks into two sets of five. We randomly selected two projects (Vue and Ohmyzsh) from our study projects discussed in Section 4.1 as the subjects for these tasks. Participants were then randomly split into two groups (Group A and Group B). Group A was assigned the first set of five tasks on the Vue project, and Group B was assigned the second set of five tasks on the Ohmyzsh project. The groupings helped ensure that no participant had to perform all ten tasks and that each participant only dealt with one project's context, reducing fatigue and learning effects.

We developed and deployed a web-based chatbot application² implementing our approach, which participants used to interact with the projects. The study was administered via an online survey (Qualtrics), structured into four sections. The first section collects demographic information about the participants. This offers us insight into the professional background, the number of years of experience the participants have using Git tools, and how often they use the Git tools.

In the second section, participants were presented with five tasks and instructed to complete each task using any tools or methods of their choice (except our chatbot). They could use Git command-line queries, repository browsing, writing scripts, web searches, or AI assistants, as desired. In order for participants not to spend all their effort on a single task and keep within the allotted time of 60 minutes for the survey, we asked the participants to spend not more than 6 minutes per task (i.e., approximately 30 minutes total for the five tasks in this section). After completing each task, the participants provided their answer (the outcome of the task) and a brief description of how they completed the task. The description allowed us to understand the effort and techniques adopted by the participants to complete the tasks. We did not impose strict tool restrictions in order to emulate a realistic scenario where developers can leverage all resources at their disposal. This baseline phase established a point of comparison for task difficulty and time taken without the assistance of our tool.

In the third section, participants performed the same set of five tasks, but this time using our chatbot. We asked the participants to phrase each task as a question in their own words to the

²<https://repopchattool.streamlit.app/>

chatbot. For each task, the survey asked them to copy the exact question they posed to the chatbot and the answer the chatbot returned. After receiving the chatbot’s answer, participants were shown the expected correct answer (ground truth) for that task and asked to evaluate the chatbot’s response. They indicated whether the bot’s answer was Correct, Partially Correct, or Incorrect, and could optionally provide comments explaining their choice. We captured the time taken to complete each task with the chatbot as well. This section allowed us to assess the chatbot’s effectiveness in answering the questions directly compared to the baseline.

In the final section, participants answered a brief questionnaire about their overall experience with the chatbot. They rated, on a 5-point Likert scale, the chatbot’s accuracy, usefulness, time-saving capability, and their overall satisfaction with the chatbot (1 = Strongly Disagree to 5 = Strongly Agree for each statement). They were also given an open-ended text box to provide any additional feedback, comments, or suggestions about the chatbot. This feedback provides insights into the user-perceived benefits or issues with our approach.

The survey was configured to automatically record the time spent on each task (using Qualtrics’ timing features), giving us precise measurements for task completion times in both the baseline and chatbot settings. We piloted the survey with two participants. We used the feedback from these participants to improve and refine the survey for clarity before administering it to the other participants.

4.6 Survey Participants

We recruited participants through a combination of purposive and snowball sampling, following the approach of Latendresse et al. [42]. We initially contacted software practitioners from our professional networks in both academia and industry who had experience using Git. We then employed snowball sampling, asking initial participants to share the survey with other professionals in their networks who regularly use Git tools.

Table 4 shows the breakdown of the demographics of the 20 study participants recruited. Participants comprised 65.0% students and 35.0% industry practitioners, with the majority (55.0%) having over five years of Git experience. Most participants use Git daily (55.0%). Although the majority of our participants are students, this does not influence our findings for these reasons: first, the student participants demonstrated substantial expertise: 7 of the 13 students (53.8%) had over five years of Git experience, 6 students (46.2%) used Git daily, and an additional 7 students (53.8%) used Git weekly. Second, prior studies have shown that students with experience are a good proxy for industry experts, especially when the study is focused on emerging technology [57].

Table 4. Demographics of Participants in the Survey

Category	Experience	Frequency	%
Background	Student	13	65
	Industry Practitioner	7	35
Years of Experience with Git Tools	5+ years	11	55
	3–5 years	4	20
	1–2 years	5	25
Frequency of Using Git Tools	Daily	11	55
	Weekly	9	45
Total Participants		20	

5 RESULTS

In this section, we first present the results of the exploratory analysis, and then present the results of the four main research questions. For each research question, we present the motivation, the approach, and the results.

5.1 RQ0: How good are LLMs in generating Cypher queries for knowledge graphs?

Motivation. Before answering the research questions in this study, we first evaluate the capability of different LLMs to accurately generate Cypher queries from natural language text. This RQ aims to empirically assess the context of this study (i.e., the ability of LLMs to generate accurate Cypher queries from natural language text for retrieving data from a knowledge graph). Secondly, this RQ aims to empirically identify the most efficient LLM in generating Cypher queries, which will serve as the LLM model in our implementation to answer the remaining RQs.

Approach. To evaluate the capability of LLMs to generate valid Cypher queries from natural language text, we selected three state-of-the-art models considering both open-source and closed-source options: GPT-4o [53], Llama3-8B [50], and Claude3.5 [8]. These models have been widely used in software engineering literature [47, 78]

Similar to the strategy by Li et al. [44], we evaluate the models under zero-shot settings to assess their generalization ability to generate Cypher queries using their pre-existing knowledge without prior exposure or clues. We evaluate the models on the 20 questions described in Section 4.2 using the prompt template shown in Figure 3 to generate the Cypher query. Due to the stochastic nature of LLMs, we run the experiments five times, each time on a different repository [22].

As our evaluation metric, we use the Execution Accuracy (EX) [44] because it measures the correctness of the generated Cypher queries in terms of their execution results, which is essential for applications that require accurate retrieval of data. EX is defined in Equation 1 as the proportion of the evaluation set in which the executed results of the generated queries are similar to the ground truth, relative to the examples in the evaluation set and formalized as:

$$EX = \frac{\sum_{n=1}^N \mathbb{1}(V_n, \hat{V}_n)}{N} \quad (1)$$

and $\mathbb{1}(\cdot)$ is a function represented as:

$$\mathbb{1}(V_n, \hat{V}_n) = \begin{cases} 1, & \text{if } V_n = \hat{V}_n \\ 0, & \text{if } V_n \neq \hat{V}_n \end{cases}$$

where N is the total number of executions, V_n is the result set from executing the ground-truth Cypher queries, and \hat{V}_n is the result set from executing the generated Cypher queries.

Results. Table 5 compares the execution accuracy of the selected models in generating Cypher queries. We find that GPT-4o is the most efficient model in translating natural language questions into accurate Cypher queries within the given zero-shot setting, achieving an EX score of 0.65. The superior performance of GPT-4o can be attributed to the advanced language understanding capabilities of the GPT-4 family of LLMs in capturing the semantic details required for precise Cypher query generation as demonstrated in prior studies [44]. The performance difference between GPT-4o, Claude3.5, and Llama3 highlights the variability in capability among different LLMs when applied to the task of generating Cypher queries from natural language text. This finding informs our decision to utilize GPT-4o for the subsequent research questions, as it offers the most reliable performance for synergizing LLMs with knowledge graph data.

Table 5. Comparison of three state-of-the-art LLMs in terms of execution accuracy (EX) in generating valid Cypher queries from natural language text

Model	Questions	Executions	Correct	EX
GPT-4o	20	100	65	0.65
Claude3-5	20	100	61	0.61
Llama3	20	100	20	0.20

RQ0 Summary: LLMs have the capability to capture semantic details for Cypher query generation from natural language text. Specifically, GPT-4o demonstrated to be the most efficient in the task of generating Cypher queries from natural language text.

5.2 RQ1: How effective is our approach in answering software repository-related questions?

Motivation. The synergy between knowledge graphs and large language models (LLMs) has the potential to enhance the ability of LLMs to provide accurate and contextually relevant answers [54]. Knowledge graphs encapsulate structured information about entities and their relationships, which can be crucial for understanding complex queries and providing precise answers. In this research question, we investigate the effectiveness of generating an accurate response to a user question by adding a layer of semantic understanding. This enables the LLM to generate a Cypher query and retrieve the relevant information to answer the question.

Approach. To evaluate the performance of our approach in answering software repository-related questions, we conducted an end-to-end evaluation of the approach from the moment the Query Generator receives the natural language query to when the Response Generator outputs the final response (see step 2 in Figure 1). The end-to-end evaluation measures the practical performance of our approach in generating accurate answers [22].

For this purpose, we evaluated the 20 questions described in Section 4.2 for each of the selected projects. For each question, we also executed the process five times to account for the stochastic nature of LLMs in the generation process. We compared the final responses generated by our approach to the oracle answers (predetermined correct answers based on the data in the knowledge graph). A question was considered correctly answered if our approach provided the correct response in at least 3 out of 5 executions. If it failed in three or more executions, the question was marked as incorrect. In instances where the approach returned “*I don’t know*” or “*I don’t have this information*”, we marked the question as unanswered.

Results. Table 6 compares the accuracy of our approach across the selected projects. We observed that the accuracy of our approach varies between 60% and 75% across the projects. The highest accuracy was achieved for the AutoGPT repository at 75%, while both Ohmyzsh and Vue had the lowest accuracy at 60%. The average accuracy across all repositories was 65%, indicating that our approach has a moderate overall effectiveness in answering questions related to the software repositories.

Table 7 also compares the accuracy of our approach based on the difficulty level of the questions. The results show a correlation between the difficulty level and the accuracy of our approach. Our approach achieved an 80% accuracy on level 1 questions, and the accuracy decreased to 65% for level 2 questions. The accuracy further dropped to 50% for level 3 questions. This trend suggests

Table 6. Comparison of the accuracy of our approach across the selected projects

Project	Questions	Answered	Correct	Accuracy
AutoGPT	20	20	15	0.75
Bootstrap	20	17	13	0.65
Ohmyzsh	20	14	12	0.60
React	20	18	13	0.65
Vue	20	17	12	0.60
Overall	100	86	65	0.65

Table 7. Comparison of the accuracy of our approach based on the difficulty level of the questions

Level	Questions	Answered	Correct	Accuracy
1	20	16	16	0.80
2	60	51	39	0.65
3	20	19	10	0.50
Total	100	86	65	0.65

that while our approach is effective at handling straightforward queries, its effectiveness decreases as the questions get more complex. We present all the questions answered correctly or incorrectly in this research question in Appendix B (Table 19).

Consistency of responses. To better understand the reliability of our approach, we analyzed the consistency of the responses across multiple runs for each of the 100 questions. As shown in Figure 5, the chatbot shows high consistency by answering the majority (55%) of the questions correctly across all 5 executions. An additional 5% (5 out of 100 questions) were correct in 4/5 runs, and 5% (5 out of 100 questions) were correct in 3/5 runs. Under the majority vote criterion, showing an execution accuracy of 65%.

RQ1 Summary: Synergizing LLMs with knowledge graphs correctly answered 65% of the repository-related questions. The approach performs well when answering simple questions but struggles with complex questions that require two or more relationships from the knowledge graph.

5.3 RQ2: What are the limitations of our approach in accurately answering software repository-related questions?

Motivation. While our approach achieved better performance compared to previous LLM-based approaches [4], the task is still challenging for our approach, with an accuracy of 65%. To be able to improve our approach to achieve a higher accuracy, we need to understand the reasons for which our approach fails. Therefore, in this research question, our goal is to identify the limitations of our approach by manually analyzing the incorrectly generated responses of our approach.

Approach. To understand the limitations of our approach, we selected all the incorrectly answered questions from the 500 executions in RQ1, that is, 20 questions each executed five times for five

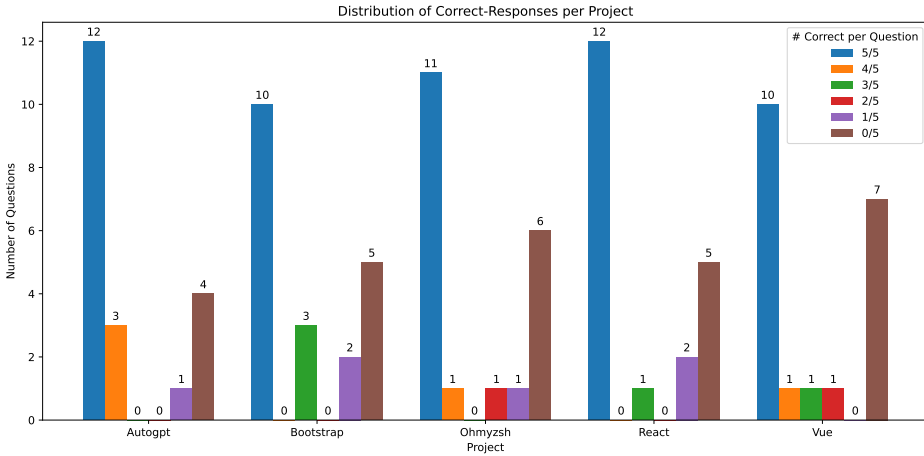


Fig. 5. Distribution of correct responses across multiple runs of the 20 questions for each project in this. The blue bar represents when all 5 answers are correct, the orange bar when 4 of the 5 answers are correct, the green bar when 3 of 5 are correct, the red bar when 2 of the 5 are correct, the purple bar when 1 of the 5 is correct and the brown bar when all 5 answers are incorrect.

repositories. We identified and manually analyzed a total of 164 executions that returned incorrect answers. To identify the limitations, we adopted an open-card sorting approach as used in prior studies [42]. The executions were sorted based on the final response of our approach, the generated Cypher query, and the results from executing the Cypher query. The main author read through all 164 executions to identify recurring themes and patterns that may have led to incorrect responses to come up with labels. To ensure the labels are less biased and go through a level of scrutiny, the authors discussed each question and the preliminary labels. This step ensured that the labels had clarity and were relevant. Based on this step, some of the labels were merged, split, or modified to provide more clarity.

Results. Table 8 summarizes the definitions, frequencies, and percentages of the six main limitations we identified from manually analyzing the instances where our approach generated incorrect answers: Incorrect relationship modeling, Faulty arithmetic logic, Misapplied attribute filtering, Invalid assumptions, Misapplied date formatting, and Hallucination. There were cases where multiple limitations were identified within a single instance. Thus the frequency reported reflects the total number of limitations rather than the 164 instances analyzed.

The most prevalent limitation was the **incorrect relationship modeling**, occurring in 123 out of 164 incorrect responses (75.0%). This limitation occurs when the logic used by the LLM to generate the query deviates from the intention of the question. This limitation is due to the interpretation the LLM places on the question; if the interpretation is incorrect, the LLM will proceed to generate a logically incorrect query. For instance, as shown in Table 9, consider the question: *“Determine the percentage of the fixing commits that introduced bugs on July 2023”*. The correct Cypher query first matches all commits that fixed an issue in July 2023 to find the total fixing commits. It then matches the commits that both fixed an issue and introduced another issue in July 2023 and calculates the percentage accordingly. In contrast, the LLM generated a syntactically correct but logically flawed Cypher query. It incorrectly matched commits that fixed an issue as the fixing commits and separately matched commits that introduced an issue as the introducing

commits without linking the two actions within the same commits. This misinterpretation of the relationships leads to an incorrect result.

Faulty arithmetic logic was observed in 19 instances, signifying 11.6% of the total instances analyzed. This limitation deals with instances where the LLM is required to perform an arithmetic operation to return a final response to the user but fails to perform the correct arithmetic operation. For instance, in the example in Table 9, the fixingCommits should be the denominator when calculating a percentage of fixing commits. However, in the generated query by the LLM, the introducing commit was used as the denominator in the context, which is incorrect.

Also, in 16 instances (9.8%), we found the LLM **misapplying the attributes when filtering** the matched data. In these instances, the LLM used the wrong attribute to filter the data based on the constraints specified in the question. The example in Table 10 shows the LLM correctly matching commits that introduced bugs. However, when filtering to the specified date in the question, the LLM misapplied the filter to the issue creation date instead of the commit date. This error by the LLM led to generating an incorrect final response.

Making Incorrect Assumptions was also observed in 15 instances (9.1%) of the incorrectly answered executions. In these cases, the LLM generated a syntactically correct query but made incorrect assumptions that led to an incorrect answer. For example, as shown in Table 11, when asked “Return a commit message on July 31?”, the LLM assumed a specific year “2024-07-31” in the query, whereas the correct approach would be to use a wildcard for the year and match any date that includes “-07-31” to retrieve commits on July 31 of any year. This incorrect assumption limits the query to a specific date, potentially excluding relevant results. If there’s no record for the assumed date, then LLM will respond incorrectly.

We also identified **misapplied date formatting** in 10 instances (6.1%). This refers to cases when the format of the data in the query does not reflect the format accepted by the graph database or the formatting incorrectly filters out data that are not to be filtered out. For instance, in the example in Table 12, the query generated by the LLM unintentionally filters out data due to the formatting of the date. These filtered data will be returned if the LLM uses a wildcard.

Hallucination was another limitation, occurring in 10 instances (6.1%). This happens when the LLM makes use of nodes and relationships that are not present in the knowledge graph schema. As illustrated in Table 13, for the question “Determine the developers that fixed the most bugs in challenge.py?”, the LLM introduced a non-existent relationship [fixed] directly between the User and Issue nodes. In the correct schema, the [fixed] relationship exists between the Commit and Issue nodes, not between User and Issue. This hallucination leads to an invalid query and incorrect results.

The limitation on incorrectly modeling relationships, faulty arithmetic logic, misapplied attribute filtering, and date formatting highlights the limitation in the reasoning ability of the LLM. This hinders the LLM in accurately interpreting and utilizing the right nodes and relationships within the knowledge graph. This aligns with prior studies that highlight the challenges LLMs face in reasoning tasks [14, 67, 68]. The incorrect reasoning often leads to misconstructured queries that do not align with the intended question, resulting in inaccurate or irrelevant answers. Addressing these limitations is crucial for improving our approach. By refining the LLM’s understanding of the knowledge graph schema and enhancing its reasoning capabilities, we can reduce the incidence of incorrect responses.

Table 8. Summary of the limitations identified with their frequency and percentage (the sum of the frequencies is more than 164 because we identified multiple limitations in some instances)

Limitation	Definition	Frequency	Percentage
Incorrect relationship modelling	The LLM deviates from the intended logic of the question, leading to a query that incorrectly models relationships within the knowledge graph	123	75.0%
Faulty arithmetic logic	The LLM fails to perform correct arithmetic operations required to generate the final response	19	11.6%
Misapplied attribute filtering	The LLM uses incorrect attributes to filter data, not adhering to the constraints specified in the question	16	9.8%
Invalid assumptions	The LLM makes incorrect assumptions, such as assuming specific values not provided, leading to incorrect or limited results	15	9.1%
Misapplied date formatting	The LLM uses incorrect date formats, not matching the database format and improperly filtering out data	10	6.1%
Hallucination	The LLM introduces nodes or relationships not present in the knowledge graph schema, resulting in invalid queries	10	6.1%

Table 9. Example of incorrect relationship modeling and faulty arithmetic logic in the generated query

Correct Query	Generated Incorrect Query
<p>Q: Determine the percentage of the fixing commits that introduced bugs on July 2023?</p> <pre>MATCH (c:Commit)-[:fixed]->(i:Issue) WHERE c.committedDate =~ "2023-07.*" WITH COUNT(c) AS totalFixingCommits MATCH (i2:Issue)<-[:introduced]-(c:Commit) -[:fixed]->(i:Issue) WHERE c.committedDate =~ "2023-07.*" WITH totalFixingCommits, COUNT(c) AS fixingAndIntroducingBugs RETURN (toFloat(fixingAndIntroducingBugs) / totalFixingCommits) * 100 AS percentage</pre>	<p>Q: Determine the percentage of the fixing commits that introduced bugs on July 2023?</p> <pre>MATCH (c1:Commit)-[:fixed]->(:Issue)<-[: introduced]-(c2:Commit) WHERE c1.committedDate =~ '2023-07.*' AND c2 .committedDate =~ '2023-07.*' WITH count(c1) AS fixingCommits, count(c2) AS introducingCommits RETURN (toFloat(fixingCommits) / introducingCommits) * 100 AS percentage</pre>

Table 10. Example of misapplied attribute filtering in the generated query

Correct Query	Generated Incorrect Query
<p>Q: What commits were buggy on June 08, 2023?</p> <pre>MATCH (c:Commit)-[:introduced]->(i:Issue) WHERE c.committedDate =~ '2023-06-08T.*' RETURN c</pre>	<p>Q: What commits were buggy on June 08, 2023?</p> <pre>MATCH (c:Commit)-[:introduced]->(i:Issue) WHERE i.created_at =~ '2023-06-08T.*' RETURN c</pre>

Table 11. Example of invalid assumptions in the generated query

Correct Query	Generated Incorrect Query
<p>Q: Return a commit message on July 31?</p> <pre>MATCH (c:Commit) WHERE c.committedDate =~ ".*-07-31T.*" RETURN c.message</pre>	<p>Q: Return a commit message on July 31?</p> <pre>MATCH (c:Commit) WHERE c.committedDate =~ '2024-07-31T.*' RETURN c.message</pre>

Table 12. Example of misapplied date formatting in the generated query

Correct Query	Generated Incorrect Query
<p>Q: How many fixing commits caused bugs on July 2023?</p> <pre>MATCH (c:Commit)-[:fixed]->(Issue), (c)-[:introduced]->(Issue) WHERE c.committedDate =~ '2023-07-.*' RETURN count(c) AS fixingIntroducingCommits</pre>	<p>Q: How many fixing commits caused bugs on July 2023?</p> <pre>MATCH (c:Commit)-[:fixed]->(Issue) WHERE c.committedDate =~ '2023-07-..TZ' RETURN count(c) AS fixingCommits</pre>

Table 13. Example of hallucination in the generated query

Correct Query	Generated Incorrect Query
<p>Q: Determine the developers that fixed the most bugs in challenge.py?</p> <pre>MATCH (u:User)-[:author]->(c:Commit)-[:fixed]->(i:Issue)-[:impacted]->(f:File {name : "challenge.py"}) RETURN u, COUNT(i) AS fixedBugs ORDER BY fixedBugs DESC</pre>	<p>Q: Determine the developers that fixed the most bugs in challenge.py?</p> <pre>MATCH (u:User)-[:fixed]->(i:Issue)-[: impacted]->(f:File {name: "challenge.py "}) RETURN u.name AS developer, COUNT(i) AS bugs_fixed ORDER BY bugs_fixed DESC LIMIT 1</pre>

RQ2 Summary: The reasoning of LLM during query generation is the most prevalent limitation affecting LLMs when enhanced with knowledge graphs. It hinders the ability of the LLM to accurately interpret and utilize the right nodes and relationships within the knowledge graph, leading to incorrect relationship modeling, faulty arithmetic logic, misapplied attribute filtering, and misapplied date formatting. Other limitations include the LLM making wrong assumptions and hallucination.

5.4 RQ3: Can chain-of-thought prompting improve the effectiveness of our approach in answering software repository-related questions?

Motivation. In RQ2, the failure analysis showed that most of the limitations leading to an incorrect response are due to the *faulty reasoning of the LLM during query generation*. To address this limitation, we introduce *chain-of-thought (CoT) prompting* in the LLM underlying the **Query Generator**. Our goal for this research question is to improve the reasoning ability of the LLM in

the **Query Generator** to minimize faulty reasoning. We do this by introducing chain-of-thought (CoT) prompting into our approach, feeding the LLM step-by-step reasoning examples to guide it in generating a reasoning path to reach an answer. Prior studies have shown that using chain-of-thought (CoT) to generate a series of intermediate steps before the final answer can improve the reasoning ability of LLMs [41, 71]. This can be achieved under zero-shot settings (prompting the LLM to think step by step) [41] and few-shot settings (prompting the LLM with a few chain-of-thought examples).

Approach. To evaluate if chain-of-thought prompting can improve the performance of our approach in answering software repository questions, we first evaluated the approach by including a zero-shot chain-of-thought instruction, that is: “*Let’s think step by step*” [41] to the prompt template in Figure 3. This did not improve the results presented in RQ1 (See Table 21 and Table 22 in Appendix D for the results of the zero-shot chain-of-thought across the selected projects and difficulty levels). We experimented with few-shot chain-of-thought by incorporating examples into the prompt provided to the Query Generator. Similar to Wei et al. [71], we adopted the format that begins with the input question, the step-by-step reasoning process, and the final output in our few-shot chain-of-thought prompting. This format aims to guide the LLM in generating intermediate reasoning steps before arriving at the final answer. We constructed the chain-of-thought prompt as presented in Figure 6. The prompt consists of two examples, consisting of difficulty level 2 and 3 questions. By providing these examples, we intended to show the LLM how to generate reasoning paths that can help in constructing correct queries. It is important to note that the questions used in the chain-of-thought examples are not part of our evaluation questions.

For the evaluation, we first measure the gains in execution accuracy made by incorporating chain-of-thought and then benchmark against the baselines described in Section 4.4. To evaluate the gains, we use the same set of 20 questions described in Section 4 and used in RQ1, executing the experiments five times for each question to account for the stochastic nature of the LLM’s generation [22]. A question is considered correctly answered if the correct response is generated most of the time; otherwise, it is marked as incorrect.

For the baseline comparison, we use the 150 evaluation questions described in Section 4.2 and run each baseline five times, as done previously. The baselines include the intent-based MSRBot and the GPT-4o model with web search capabilities. MSRBot produces identical answers across runs due to its deterministic intent-matching process, while GPT-4o-search-preview is run with the default parameters through the OpenAI API across iterations to ensure a fair comparison.

Results. Table 14 compares the accuracy of the few-shot chain-of-thought approach across the selected project. The accuracy across the projects ranged between 80% and 90%, with an average of 84%. The results improved across all projects to the result in RQ1 (see Table 6). Table 15 also presents the performance based on the difficulty level of the questions. The accuracy improved across all levels with the application of few-shot chain-of-thought prompting compared to without the few-shot chain-of-thought prompt (see Table 7). For level 1 questions, the accuracy increased to 85% from the previous 80% in RQ1. For level 2 questions, the accuracy improved to 82% from 65% in RQ1. For the level 3 questions, which were previously challenging for our approach, the accuracy significantly increased to 90% from 50% in RQ1, indicating a substantial improvement in handling complex queries. Comparing the results with those from RQ1, the overall accuracy improved from 65% to 84%, signifying an improvement of 19% percentage points compared to the results in RQ1.

The improvement in the results, especially in the level 3 questions, suggests that few-shot chain-of-thought prompting effectively mitigated the faulty reasoning limitation identified in RQ2. By providing reasoning steps, our approach could better navigate the complex relationships within the knowledge graph. For example, when asked to “*Determine the percentage of the fixing*

You are an AI assistant, your task is to generate a Cypher statement to query a Neo4j graph database by following the instructions below.

Instructions:

Use only the provided relationship types and properties in the schema.

The current date is {current_date}.

If the user query contains a date or datetime, format it in the iso format like "YYYY-MM-DDTHH:MM:SSZ". If the datetime is without the timestamp, use a regex for the missing part.

Before you start, determine the intention of the question. If the question can be interpreted in multiple ways, list all the possible interpretations and select the most probable one.

Schema:

{schema}

The question is:

Q: Who is the most experienced developer?

A: To find the most experienced developer, we first need to list all possible interpretations of developer experience. An experienced developer can be the user who has fixed the most issues, the user who has opened the most issues, the user who has made the most commits, the user who has participated in the most issues, or the user who has been assigned the most issues. The most appropriate interpretation here is the user who has made the most commits. Therefore, to find the most experienced developer, we must find the user with the most commits. The relevant nodes are the User and Commit nodes. The relevant relationship between user and commit for this question is the author relationship. The metric to measure developer experience would be the number of commits.

Find the users that have authored commits:

```
MATCH (u:User)-[:author]->(c:Commit)
```

Aggregate the number of commits by each user:

```
RETURN u.login AS developer, COUNT(c) AS contributions
```

Sort the contributions in descending order to find the users with most contributions:

```
ORDER BY contributions DESC
```

Therefore the complete query is:

```
<query>
```

```
MATCH (u:User)-[:author]->(c:Commit)
```

```
RETURN u.login AS developer, COUNT(c) AS contributions
```

```
ORDER BY contributions DESC
```

```
</query>
```

Q: What files have AAAAA modified the most?

A: There is only one interpretation of the question, that is to find the files that user AAAAA has modified, and find the number of times user AAAAA has modified each file, and list the ones with the highest number of modifications.

First, we need to identify all the relevant nodes. AAAAA is a user, which corresponds to the User node. We also need to find the files, which is represented by the File Node. There is no direct relationship between User and File in the schema, therefore we need the Commit node as an intermediary. The relevant relationship between User and Commit for this question is the author relationship to find the commits authored by AAAAA. The relevant relationship between Commit and File is the changed relationship to find the files modified in the commit.

Therefore the query to find the files that AAAAA has modified the most is:

```
MATCH (u:User{name: 'AAAAA'})-[:author]->(c:Commit)-[:changed]->(f:File)
```

Next, we have to aggregate the number of modifications by each file:

```
RETURN f.name AS file, COUNT(c) AS modifications
```

Finally, we sort the files in descending order to find the files that AAAAA has modified the most:

```
ORDER BY modifications DESC
```

Since we want the most modified files, we limit the results to the top 10:

```
LIMIT 10
```

Therefore the complete query is:

```
<query>
```

```
MATCH (u:User {name: 'AAAAA'})-[:author]->(c:Commit)-[:changed]->(f:File)
```

```
RETURN f.name AS file, COUNT(c) AS modifications
```

```
ORDER BY modifications DESC
```

```
LIMIT 10
```

```
</query>
```

Q: {question}

A:

Fig. 6. Prompt template for the few-shot chain-of-thought. The prompt includes the current date and time, the schema of the knowledge graph, the chain-of-thought examples, and the user's question.

commits that introduced bugs on July 2023?”, the few-shot chain-of-thought prompting approach, correctly identified the required relationships between the commits that fixed an issue and, at the same time, introduced another issue to formulate an accurate Cypher query. However, some limitations persisted. Our approach still encountered difficulties with certain questions, leading to incorrect answers or unanswered questions. In level 1, for instance, the accuracy did not improve as substantially as in level 3, indicating that while chain-of-thought prompting aids in complex reasoning, it may not fully address all types of reasoning errors. We present all the questions answered correctly or incorrectly in this research question in Appendix C (Table 20).

The results with the few-shot chain-of-thought prompting align with prior studies that have shown the effectiveness of chain-of-thought prompting in improving the reasoning abilities of LLMs [41, 71]. By augmenting the LLM with structured reasoning examples, we facilitated better logical processing, leading to more accurate responses. Chain-of-thought prompting has a positive impact on our approach’s performance in answering the questions. It effectively reduces poor reasoning and improves the model’s ability to handle complex queries involving multiple relationships.

Comparison with baselines. We further compared our few-shot chain-of-thought approach with the intent-based MSRBot and the GPT-4o model with web search, which serve as baselines for this task (see Section 4.4). Table 16 summarises the accuracy of the three methods across the five projects. Our approach achieved an average accuracy of 0.82 across 750 evaluation questions, outperforming MSRBot (0.70) and GPT-4o-search-preview (0.19). The GPT-4o-search baseline struggled to answer the repository-related questions, corroborating prior observations that retrieval-augmented LLMs often fail, although this approach outperformed the prior study [4]. Table 17 reports the execution accuracy by question difficulty. Our approach consistently outperformed MSRBot on level-1 and level-3 questions, achieving 0.87 accuracy on the most complex level-3 questions compared to 0.60 for MSRBot. For level-2 questions, MSRBot slightly outperformed our approach. GPT-4o-search performed poorly across all levels, reinforcing that naïve web-enabled LLMs are not yet suitable for this task. Overall, few-shot chain-of-thought prompting yields higher accuracy than the baselines on most question types.

Consistency of responses. Beyond execution accuracy, we examined the *consistency* of responses across multiple runs of the same question for our approach. Specifically, we counted the number of times the correct answer was generated in the five runs for each question. The distribution in Figure 7 shows this for all five projects. Overall, 76.4% of questions were answered correctly each time the question was repeated (*5/5 correct runs*). This increases to 81.7% under our majority-vote criterion (i.e., *3/5, 4/5, or 5/5 correct runs*) in our evaluation. This shows that our approach is not only accurate on average but also *reliably* produces the correct answer most of the time. This stability is an important quality for practical deployment, as it reduces variability in responses for the user.

RQ3 Summary: With few-shot chain-of-thought, the execution accuracy increases from 0.65 to 0.84 overall, and, for complex questions requiring two or more relationships, from 0.50 to 0.90. This implies that chain-of-thought can help answer complex questions requiring multiple relationships. The approach also outperformed the baselines, MSRBot (0.70) and GPT-4o-search-preview (0.19).

Table 14. Comparison of the accuracy of the few-shot chain-of-thought approach across the selected projects

Project	Questions	Answered	Correct	Accuracy
AutoGPT	20	20	18	0.90
Bootstrap	20	20	16	0.80
Ohmyzsh	20	18	16	0.80
React	20	19	18	0.90
Vue	20	18	16	0.80
Overall	100	95	84	0.84

Table 15. Comparison of the accuracy of the few-shot chain-of-thought approach based on the difficulty level of the questions

Level	Questions	Answered	Correct	Accuracy
1	20	18	17	0.85
2	60	57	49	0.82
3	20	20	18	0.90
Overall	100	95	84	0.84

Table 16. Comparison with baselines across the selected projects

Project	Questions	Correct			Accuracy		
		Our Approach	MSRBot	GPT-4o-Search-Preview	Our Approach	MSRBot	GPT-4o-Search-Preview
AutoGPT	150	122	114	26	0.81	0.76	0.17
Bootstrap	150	123	90	36	0.82	0.60	0.24
Ohmyzsh	150	127	90	32	0.85	0.60	0.21
React	150	124	110	24	0.83	0.73	0.16
Vue	150	117	123	23	0.78	0.82	0.15
Overall	750	613	527	141	0.82	0.70	0.19

Table 17. Comparison with baselines based on the difficulty level of the questions

Level	Questions	Correct			Accuracy		
		Our Approach	MSRBot	GPT-4o-Search-Preview	Our Approach	MSRBot	GPT-4o-Search-Preview
1	200	174	123	101	0.87	0.62	0.51
2	395	304	311	30	0.77	0.79	0.08
3	155	135	93	10	0.87	0.60	0.06
Overall	750	613	527	141	0.82	0.70	0.19

5.5 RQ4: How do users perceive the usefulness of our approach in assisting them to answer repository-related questions?

Motivation. Building on the accuracy gains observed in RQ3, this research question examines whether these improvements translate into practical value for developers. Specifically, we examine

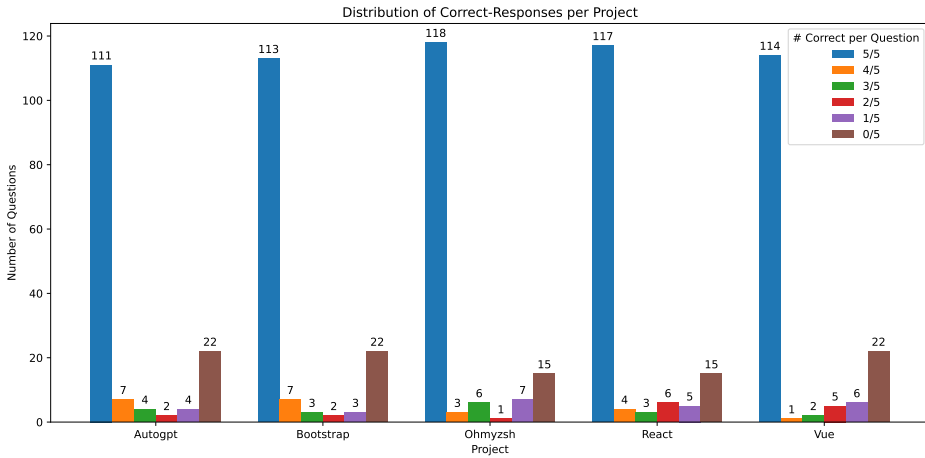


Fig. 7. Distribution of correct responses across multiple runs for each project. The blue bar represents when all 5 answers are correct, the orange bar when 4 of the 5 answers are correct, the green bar when 3 of 5 are correct, the red bar when 2 of the 5 are correct, the purple bar when 1 of the 5 is correct and the brown bar when all 5 answers are incorrect.

whether users find our approach helpful in answering repository-related questions. To this end, we analyze task accuracy, completion time, and qualitative feedback, as these jointly capture both the efficiency gains and the subjective value users derive from the chatbot. This evaluation is critical for validating the approach from the end-user perspective, complementing quantitative performance metrics with insights into perceived utility.

Approach. To evaluate the usefulness of our approach with actual users, we conducted an online user study with 20 participants presented in section 4.6 to assess the perceived usefulness of our approach. The study compared task performance when using our chatbot implementation of the approach against manual task execution. Participants first completed a set of repository-related tasks manually, then repeated similar tasks using the chatbot. We measured task accuracy and time taken, and collected qualitative feedback through a post-task questionnaire (see Section 4.5).

Results. Each of the 20 participants was tasked with completing 5 repository-related tasks manually and then performing them using the chatbot. The task completion rate and the correctness of the final answers for each participant are presented in Table 25 (in Appendix F). The baseline (manual) approach led to the participants completing 69 out of the 100 tasks and providing a final answer, whereas, when using the chatbot, participants completed 91 tasks. Only three participants completed all five tasks manually, but 15 participants did so when using the chatbot, indicating the chatbot helped users finish more of their work.

Among the tasks that participants completed when working manually, they provided correct answers for 36% of the tasks, compared with 84% when using the chatbot. On average, participants solved approximately two tasks correctly without the assistance of our chatbot and more than four tasks correctly using the chatbot. No participant performed worse with the chatbot, and many improved from zero or one correct answer to four or five. These results indicate substantial gains in correctly obtaining information from the repository when using our approach. **We observed that the low manual performance was largely driven by task incompleteness and the difficulty of multi-step manual investigation for several tasks.** In particular, participants frequently reported needing to clone the repository and rely on manual inspection, command-line executions

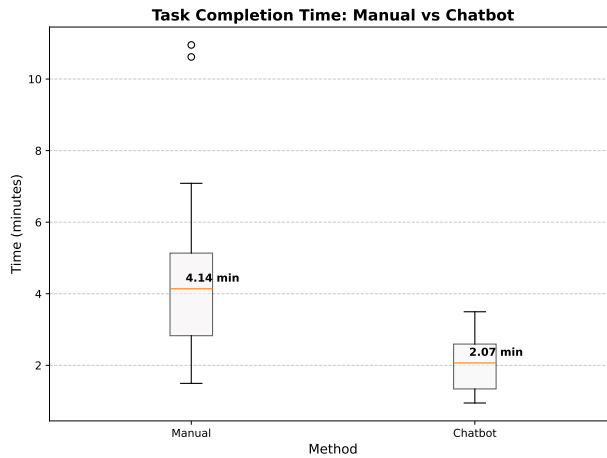


Fig. 8. Boxplot of task completion times: participants consistently completed tasks faster with the chatbot than with manual methods.

(for example, searching commit history with time/keyword filters), or ad-hoc scripts, and several still indicated they could not locate information to complete the task. This is also reflected in participants' self-reported approaches, e.g., "I tried several solutions, but was unable to solve this task without manual analysis", "Manually by filtering based on the dates", and "I cloned the repo, and it's taking too long to process". This shows that the manual baseline was limited by the practical effort and error-proneness of cross-referencing information across the repository entities under the imposed constraints.

Efficiency gains were also substantial. Across all five tasks, participants spent a median of 20.7 minutes when working manually, whereas using the chatbot required a median time of 10.3 minutes. Participants typically saved about 12 minutes per session, halving their effort. On a per-task basis, when using manual methods, participants took, on average, 4.5 minutes per answer, while using the chatbot reduced this to just over 2 minutes. Note that the 4.5-minute manual average reflects the 6-minute cap per task and is not an unconstrained effort time. Figure 8 shows the time participants spent on a task without using the chatbot and when using the chatbot.

To und

The post-survey responses further underscore the chatbot's perceived usefulness. Participants rated their agreement on a five-point scale and strongly agreed that the chatbot understood their questions (65% gave it the highest rating of 5), returned correct answers (50%), made the tasks easier (85%), and saved them significant time (80%). A majority also said they preferred using the chatbot over their usual methods (55% strongly agree) and would recommend it to others (60% strongly agree), with the remainder generally agreeing. The open-ended comments show a positive sentiment, with users calling the chatbot "amazing," "very helpful and easy to use," and noting that it solved problems they could not handle manually.

Overall, the results show that the chatbot not only boosts completion and accuracy but also substantially reduces effort. These benefits are clearly appreciated by the intended users.

RQ4 Summary: Users perceived the chatbot as both accurate and efficient for answering their repository-related questions; it doubled the number of correct answers and saved time compared to other methods. Overall, participants found it helpful and preferable to other methods.

6 DISCUSSIONS

In this study, we explored the synergy between large language models (LLMs) and knowledge graphs to enhance the accuracy of software engineering chatbots in answering software repository-related questions. Our findings demonstrate that synergizing LLMs and knowledge graphs can effectively improve the performance of LLM-based chatbots in answering software repositories-related questions. Beyond accuracy, our user study in RQ4 shows that these technical gains translate into practical value: participants completed more tasks, answered more correctly, did so in less time when using the chatbot, and reported high perceived usefulness.

We evaluated our approach on five popular open-source GitHub projects, with an accuracy range of 60%–75% in RQ1 and 80%–90% in RQ3. The difference in the accuracy of the approach on the projects is due to the non-deterministic nature of the LLM rather than the information in the projects. Our approach relies on the schema of the knowledge graph (see Figure 2), which is the same for all evaluated projects, to generate a query for retrieving the relevant data. Thus, if the LLM generated the same Cypher query for each question each time the question was executed, the accuracy would be constant across the repositories. This shows that the approach is generalizable across projects. Supplementing these findings, the baseline comparisons in RQ3 indicate that our method outperforms both MSRBot and the GPT-4o-search-preview model across projects and difficulty levels, reinforcing the benefit of combining knowledge and LLMs with CoT reasoning for repository Q&A.

We identified faulty reasoning as the prevalent challenge facing the approach, accounting for four of the limitations. This highlights the LLM’s challenges in accurately interpreting complex relationships and reasoning over the knowledge graph. In addressing the reasoning limitations, we introduced few-shot chain-of-thought prompting. This technique significantly improved the chatbot’s performance by 19% percentage points compared to the initial accuracy. In RQ4, this improvement aligned with observed usability gains: participants roughly halved their median task time with the chatbot while increasing correctness, suggesting that better reasoning support helps understand the user’s question and subsequently provide a correct answer.

In the evaluation, one aspect we examined was the handling of ambiguous user questions. In RQ3, we instructed the LLM to list all possible interpretations of a question if it detects any ambiguity in the question. The LLM effectively recognized ambiguous questions. For instance, the question “Which developer has the most number of bugs yet to be fixed?”, the LLM listed possible interpretations which included “Developers who have been assigned to these issues”, “Developers who have created these issues”, and “Developers who have participated in these issues” and then selected “Developers who have been assigned to these issues” which is the right interpretation. Nonetheless, there were instances where, after listing probable interpretations, it selected the wrong interpretation, leading to an incorrect answer.

Also, the LLM sometimes provided reasonable interpretations of some of the ambiguities, which may lead to correct responses in other contexts. For example, in the question “Determine the developers that fixed the most bugs in bootstrap-grid.scss?”. The LLM interpreted it as getting the users that authored commits that modified the file bootstrap-grid.scss? and also fixed bugs. This logic would have been true if the bug was in the file bootstrap-grid.scss. However, if

the bug fixing changes are not in the `bootstrap-grid.scss` file but in a different file, the logic becomes incorrect. The right logic for this question should be getting the users that authored commits which fixed issues that impacted the `bootstrap-grid.scss` file. In the evaluation in our RQ3, we considered these scenarios as incorrectly answered questions. However, if we had considered such scenarios in our evaluation as correct, the accuracy in RQ3 increased from an average of 84% to 94% as presented in Table 23 and Table 24 (See Appendix E for the comparison of the accuracy of our approach using chain-of-thought prompt across different projects and difficulty levels, when the logic of the LLM is deemed as sound due to ambiguity in the question).

Our approach achieved results beyond the evaluation questions in Section 4. Our approach was able to answer ad-hoc queries that were not part of the original test set. For example, using lines of code as a metric for productivity [16, 49], we asked the question “*Who added the most lines of code in December 2023*” on the Vue project, it responded correctly by going through the right reasoning (See Figure 9 in Appendix G). This highlights the promising potential of using LLMs with knowledge graphs to transform software engineering chatbots, making them more capable of handling a wide range of user queries. Together with the user study’s time savings and preference ratings, these observations suggest that the approach is not only accurate but also practically useful for everyday repository inquiries.

Another limitation of the generated Cypher query is it uses exact matching when querying the knowledge graph instead of pattern matching. For instance, “*Give me all the commits for vnode.js file?*” returns all the commits that modified the `vnode.js` file however, “*Give me all the commits for vnode file?*” the response is “*I don’t know*” because `vnode` does not match any filename in the files.

Also, our approach can complement emerging agent-based tools like OpenDevin and DeepWiki-open to provide developers with comprehensive repository assistance. While OpenDevin executes code modifications and DeepWiki enables semantic exploration, our knowledge graph approach delivers precise analytical insights over repository metadata, these capabilities can support developers in obtaining timely repository information and performing programming tasks by drawing on the strengths of each tool.

6.1 Implication for Practitioners.

The findings of our study have implications for software developers, project managers, and other stakeholders involved in software development. Augmenting chatbots with LLMs and knowledge graphs can significantly enhance accessibility to repository information, making it easier for non-technical team members to retrieve project information without requiring technical expertise. This accessibility can facilitate better collaboration, informed decision-making, and increased efficiency within development teams.

Practitioners like chatbot developers should consider implementing interactive features in chatbots, allowing the chatbot to ask follow-up questions that lead to better understanding and more accurate responses, ultimately improving user satisfaction. This will help improve the chatbot’s responses when there is ambiguity. The identified limitations, such as the exact matching instead of the pattern matching in the generated Cypher queries, highlight the need for chatbot developers to include robust error handling and validation in chatbot systems.

6.2 Implication for Researchers.

The findings of this study open avenues for further investigation. Using chain-of-thought improved the accuracy of the reasoning ability of the LLM in our approach. Nonetheless, there are other proposed approaches for enhancing the reasoning ability of LLMs [30, 37, 76]. Researchers can build upon this work by investigating other reasoning techniques, integrating symbolic reasoning

with neural networks, or exploring alternative prompting strategies to improve the reasoning in LLMs for software engineering chatbots.

The handling of ambiguous queries presents another area for research. Researchers should explore methods for quantifying and reducing ambiguity in user queries. They can focus on developing models that can manage ambiguity by generating multiple interpretations with corresponding confidence levels.

7 THREATS TO VALIDITY

In this section, we discuss threats to the validity of our study and the measures taken to mitigate them. We consider threats to construct validity, internal validity, and external validity.

7.1 Construct Validity

Construct validity pertains to the extent to which our evaluation measures accurately reflect the theoretical constructs they are intended to assess. A threat to construct validity is the dependency on the knowledge graph schema. The LLM's ability to generate a correct Cypher query is dependent on its understanding of the schema of the knowledge graph. Any discrepancies or ambiguities in the schema can lead to incorrect Cypher query generation. If the knowledge graph schema does not accurately represent the repository data, the LLM may produce queries that do not retrieve the intended information. We mitigated this by providing explanations of the meaning of the relationships between the entities in the knowledge graph.

7.2 Internal Validity

Internal validity refers to the extent to which the observed effects can be attributed to the variables under investigation rather than other factors. A threat to internal validity in our study is the stochastic nature of LLM outputs. Despite setting the temperature parameter to zero to reduce randomness, inherent variability in the LLM's responses could influence the results. Correct answers might occasionally occur by chance rather than due to the effectiveness of our approach.

To address this, we executed the experiment for each question five times and used a majority-vote criterion to determine correctness. A question was considered correctly answered if the chatbot provided the correct response at least 50% of the time (three out of the five attempts). This approach aimed to mitigate situations where a question was correctly answered by chance or otherwise.

Another threat to internal validity is related to the data dependencies in our approach. Specifically, if the bug ID or issue number is not specified in the commit log of the fixing commit, our approach will not identify it as such. The identification of bug-fixing commits in our approach relies on these references to link commits to the issues accurately. Missing or inconsistent references can affect the completeness of information in the knowledge graph. However, this approach of identifying the fixing commits for issues follows procedures presented in prior studies [17]

Another threat to internal validity is that survey participants lacking sufficient Git experience would be unable to perform the tasks assigned in the survey. Inexperienced participants may not fully understand the requirements of the tasks, resulting in unreliable responses. We mitigated this threat by recruiting experienced Git users: 75% of our participants have over three years of Git experience, and 55% use Git daily. While this ensures that participants understood the tasks, it introduces a potential selection bias. Our results represent the performance of expert users and may not generalize to novices.

7.3 External Validity

External validity concerns the generalizability of our findings beyond this study. A first threat is generalizing beyond the evaluation questions. We evaluated on 150 questions covering 10 intents

(Section 4). Although these questions cover different difficulty levels and intents, they may not encompass the full diversity of questions that users might pose in real-world scenarios. To mitigate this, we ran a task-based user study with 20 participants (Section 4.5) where users phrased tasks in their own words and worked both manually and with our chatbot. We also posed ad hoc questions outside the benchmark. These observations suggest some generalizability, but different question types could yield different results.

A second threat is the scope of our knowledge graph. The current schema models four entity types (Users, Commits, Issues, and Files), so questions that depend on non-modelled entities (e.g., pull requests, releases/tags, or CI/CD build events) cannot be answered. Although these four entities are critical to many repository-related questions, other entities could be added to the knowledge graph to answer more questions.

8 CONCLUSION

In this study, we investigated the synergy between large language models (LLMs) and knowledge graphs to improve the accuracy of software engineering chatbots in answering software repository-related questions. Our approach aimed to accurately answer natural language questions by generating Cypher queries to retrieve relevant repository data from the knowledge graph. Then use the retrieved information as context to generate a natural language response, making repository information accessible to both technical and non-technical stakeholders. We empirically evaluated our approach using five popular open-source GitHub repositories and a set of 20 questions curated from Abdellatif et al. [2] and categorized into three levels of difficulties. The findings demonstrated that LLMs, specifically the GPT-4o model, can answer repository-related questions by generating Cypher queries to retrieve accurate data from the knowledge graph. The initial accuracy of 65% achieved by our approach highlighted the potential limitation of synergizing LLMs with knowledge graphs. We manually investigated the instances where the approach failed to generate an accurate response and identified the faulty reasoning by the LLM as the predominant factor (80.5%) affecting the approach. We conducted further empirical evaluation if using few-shot chain-of-thought prompting can improve the accuracy. This technique significantly enhanced the reasoning ability of the LLM in our approach and improved the overall accuracy from 65% to 84%. There was a notable increase in the accuracy of the level 3 questions from 50% to 90%, signifying an improvement in the approach to handling complex queries. In addition to these gains, the baseline comparisons showed that combining LLMs with knowledge outperforms the intent-based approach, MSRBot, and GPT-4o-search-preview across projects and difficulty levels. Our findings highlight the integration of LLMs with knowledge graphs as a viable solution for making repository data accessible to both technical and non-technical stakeholders. The user study demonstrated that the technical improvements translate into practical benefits. In the study, the participants completed more tasks correctly and in less time with the chatbot than with their usual methods, and they reported high perceived usefulness. Also, our study highlights the importance of enhancing reasoning capabilities in LLMs. This opens avenues for further investigation in this direction. Our approach focuses on evaluating the baseline capability of LLMs to generate Cypher queries in a single pass. We identified instances of unanswered question which resulted from the Cypher queries returning empty results. Future work can focus on improving the answering rate and execution accuracy by introducing a feedback component into the framework. The feedback component will allow the model to analyze the output of the **Query Executor** and, if a query returns an empty result or an execution error, the model will then leverage the error message to reflect on its previous logic and refine the Cypher query. In this study, we focus on Git metadata (Users, Commits, Issues, Files) and do not answer questions that require program analysis. Future work can extend the repository knowledge graph

with code entities and program-analysis edges and align the code-level entities with the existing metadata to answer questions about code comprehension.

REFERENCES

- [1] Ahmad Abdellatif, Khaled Badran, Diego Elias Costa, and Emad Shihab. 2022. A Comparison of Natural Language Understanding Platforms for Chatbots in Software Engineering. *IEEE Transactions on Software Engineering* 48, 8 (Aug. 2022), 3087–3102. <https://doi.org/10.1109/TSE.2021.3078384>
- [2] Ahmad Abdellatif, Khaled Badran, and Emad Shihab. 2020. MSRBot: Using Bots to Answer Questions from Software Repositories. *Empirical Software Engineering* 25, 3 (May 2020), 1834–1863. <https://doi.org/10.1007/s10664-019-09788-5>
- [3] Ahmad Abdellatif, Diego Costa, Khaled Badran, Rabe Abdalkareem, and Emad Shihab. 2020. Challenges in Chatbot Development: A Study of Stack Overflow Posts. In *Proceedings of the 17th International Conference on Mining Software Repositories (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 174–185. <https://doi.org/10.1145/3379597.3387472>
- [4] Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. 2024. LLM-Based Chatbots for Mining Software Repositories: Challenges and Opportunities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering (EASE '24)*. Association for Computing Machinery, New York, NY, USA, 201–210. <https://doi.org/10.1145/3661167.3661218>
- [5] Samuel Abedu, SayedHassan Khatoonabadi, and Emad Shihab. 2024. Sabedu/Knowledge_graph_llm_synergy: Replication Package Release. Zenodo. <https://doi.org/10.5281/zenodo.14271490>
- [6] Hasan Abu-Rasheed, Christian Weber, and Madjid Fathi. 2024. Knowledge Graphs as Context Sources for LLM-Based Explanations of Learning Recommendations. <https://doi.org/10.48550/arXiv.2403.03008> arXiv:2403.03008
- [7] Eleni Adamopoulou and Lefteris Moussiades. 2020. An Overview of Chatbot Technology. In *Artificial Intelligence Applications and Innovations*, Ilias Maglogiannis, Lazaros Iliadis, and Elias Pimenidis (Eds.). Springer International Publishing, Cham, 373–383. https://doi.org/10.1007/978-3-030-49186-4_31
- [8] Anthropic. 2024. Introducing Claude 3.5 Sonnet \ Anthropic. <https://www.anthropic.com/news/claude-3-5-sonnet>.
- [9] Sean Banerjee and Bojan Cukic. 2015. On the Cost of Mining Very Large Open Source Repositories. In *2015 IEEE/ACM 1st International Workshop on Big Data Software Engineering*. IEEE Press, 37–43. <https://doi.org/10.1109/BIGDSE.2015.16>
- [10] Andrew Begel and Thomas Zimmermann. 2014. Analyze This! 145 Questions for Data Scientists in Software Engineering. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 12–23. <https://doi.org/10.1145/2568225.2568233>
- [11] Hudson Borges and Marco Tulio Valente. 2018. What’s in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (Dec. 2018), 112–129. <https://doi.org/10.1016/j.jss.2018.09.016>
- [12] Nick C. Bradley, Thomas Fritz, and Reid Holmes. 2018. Context-Aware Conversational Developer Assistants. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 993–1003. <https://doi.org/10.1145/3180155.3180238>
- [13] Gleison Brito, Thais Mombach, and Marco Tulio Valente. 2019. Migrating to GraphQL: A Practical Assessment. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE Computer Society, 140–150. <https://doi.org/10.1109/SANER.2019.8667986>
- [14] Meiqi Chen, Yubo Ma, Kaitao Song, Yixin Cao, Yan Zhang, and Dongsheng Li. 2024. Improving Large Language Models in Event Relation Logical Prediction. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 9451–9478. <https://doi.org/10.18653/v1/2024.acl-long.512>
- [15] Xiaojun Chen, Shengbin Jia, and Yang Xiang. 2020. A Review: Knowledge Reasoning over Knowledge Graph. *Expert Systems with Applications* 141 (March 2020), 112948. <https://doi.org/10.1016/j.eswa.2019.112948>
- [16] Lan Cheng, Emerson Murphy-Hill, Mark Canning, Ciera Jaspan, Collin Green, Andrea Knight, Nan Zhang, and Elizabeth Kammer. 2022. What Improves Developer Productivity at Google? Code Quality. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 1302–1313. <https://doi.org/10.1145/3540250.3558940>
- [17] Daniel Alencar da Costa, Shane McIntosh, Weiyl Shang, Uirá Kulesza, Roberta Coelho, and Ahmed E. Hassan. 2017. A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. *IEEE Transactions on Software Engineering* 43, 7 (July 2017), 641–657. <https://doi.org/10.1109/TSE.2016.2616306>
- [18] Steven Davies, Marc Roper, and Murray Wood. 2014. Comparing Text-Based and Dependence-Based Approaches for Determining the Origins of Bugs. *Journal of Software: Evolution and Process* 26, 1 (2014), 107–139. <https://doi.org/10.1002/smr.1619>

- [19] Malinda Dilhara, Ameya Ketkar, and Danny Dig. 2021. Understanding Software-2.0: A Study of Machine Learning Library Usage and Evolution. *ACM Transactions on Software Engineering and Methodology* 30, 4 (July 2021), 1–42. <https://doi.org/10.1145/3453478>
- [20] James Dominic, Jada Houser, Igor Steinmacher, Charles Ritter, and Paige Rodeghero. 2020. Conversational Bot for Newcomers Onboarding to Open Source Projects. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20)*. Association for Computing Machinery, New York, NY, USA, 46–50. <https://doi.org/10.1145/3387940.3391534>
- [21] EbookFoundation. 2024. EbookFoundation/Free-Programming-Books. Free Ebook Foundation.
- [22] Darren Edge, Ha Trinh, Newman Cheng, Joshua Bradley, Alex Chao, Apurva Mody, Steven Truitt, and Jonathan Larson. 2024. From Local to Global: A Graph RAG Approach to Query-Focused Summarization. <https://doi.org/10.48550/arXiv.2404.16130> [cs]
- [23] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1433–1445. <https://doi.org/10.1145/3183713.3190657>
- [24] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2024. Large Language Models are Few-Shot Summarizers: Multi-Intent Comment Generation via In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering (New York, NY, USA, 2024-02-06) (ICSE '24)*. Association for Computing Machinery, 1–13. <https://doi.org/10.1145/3597503.3608134>
- [25] GitHub. 2024. GitHub GraphQL API Documentation. <https://docs.github.com/en/graphql>.
- [26] GitHub. 2024. GitHub REST API Documentation. <https://docs.github.com/en/rest>.
- [27] GitHub. 2024. Public Schema. <https://docs.github.com/en/graphql/overview/public-schema>.
- [28] Ahmed E. Hassan. 2008. The Road Ahead for Mining Software Repositories. In *2008 Frontiers of Software Maintenance. IEEE*, 48–57. <https://doi.org/10.1109/FOSM.2008.4659248>
- [29] Hideaki Hata, Nicole Novielli, Sebastian Baltes, Raula Gaikovina Kula, and Christoph Treude. 2021. GitHub Discussions: An Exploratory Study of Early Adoption. *Empirical Software Engineering* 27, 1 (Oct. 2021), 3. <https://doi.org/10.1007/s10664-021-10058-6>
- [30] Alex Havrilla, Sharath Rparathy, Christoforus Nalmpantis, Jane Dwivedi-Yu, Maksym Zhuravinskyi, Eric Hambro, and Roberta Raileanu. 2024. GLoRe: When, Where, and How to Improve LLM Reasoning via Global and Local Refinements. <https://doi.org/10.48550/arXiv.2402.10963> arXiv:2402.10963
- [31] Anh Nguyen Hoang, Minh Le-Anh, Bach Le, and Nghi D. Q. Bui. 2025. CodeWiki: Evaluating AI's Ability to Generate Holistic Documentation for Large-Scale Codebases. <https://doi.org/10.48550/arXiv.2510.24428> arXiv:2510.24428 [cs].
- [32] Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia D'amato, Gerard De Melo, Claudio Gutierrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan Sequeda, Steffen Staab, and Antoine Zimmermann. 2021. Knowledge Graphs. *ACM Comput. Surv.* 54, 4 (July 2021), 71:1–71:37. <https://doi.org/10.1145/344772>
- [33] Aidan Hogan, Xin Luna Dong, Denny Vrandečić, and Gerhard Weikum. 2025. Large Language Models, Knowledge Graphs and Search Engines: A Crossroads for Answering Users' Questions. <https://doi.org/10.48550/arXiv.2501.06699> arXiv:2501.06699 [cs]
- [34] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large Language Models for Software Engineering: A Systematic Literature Review. 33, 8 (2024), 220:1–220:79. <https://doi.org/10.1145/3695988>
- [35] izkeros. 2022. How Can I Calculate the Number of Lines Changed since Last Commit in Git?
- [36] Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S. Yu. 2022. A Survey on Knowledge Graphs: Representation, Acquisition, and Applications. *IEEE Transactions on Neural Networks and Learning Systems* 33, 2 (Feb. 2022), 494–514. <https://doi.org/10.1109/TNNLS.2021.3070843>
- [37] Aditya Kalyanpur, Kailash Karthik Saravanakumar, Victor Barres, Jennifer Chu-Carroll, David Melville, and David Ferrucci. 2024. LLM-ARC: Enhancing LLMs with an Automated Reasoning Critic. <https://doi.org/10.48550/arXiv.2406.17663> arXiv:2406.17663
- [38] SayedHassan Khatoonabadi, Ahmad Abdellatif, Diego Elias Costa, and Emad Shihab. 2024. Predicting the First Response Latency of Maintainers and Contributors in Pull Requests. *IEEE Transactions on Software Engineering* 50, 10 (Oct. 2024), 2529–2543. <https://doi.org/10.1109/TSE.2024.3443741>
- [39] Ranim Khojah, Mazen Mohamad, Philipp Leitner, and Francisco Gomes de Oliveira Neto. 2024. Beyond Code Generation: An Observational Study of ChatGPT Usage in Software Engineering Practice. 1 (2024), 81:1819–81:1840. Issue FSE. <https://doi.org/10.1145/3660788>
- [40] Amy J. Ko, Robert DeLine, and Gina Venolia. 2007. Information Needs in Collocated Software Development Teams. In *29th International Conference on Software Engineering (ICSE'07) (2007-05)*. 344–353. <https://doi.org/10.1109/ICSE.2007.45>

ISSN: 1558-1225.

- [41] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large Language Models Are Zero-Shot Reasoners. *Advances in Neural Information Processing Systems* 35 (Dec. 2022), 22199–22213.
- [42] Jasmine Latendresse, Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. 2024. An Exploratory Study on Machine Learning Model Management. *ACM Trans. Softw. Eng. Methodol.* (Aug. 2024). <https://doi.org/10.1145/3688841>
- [43] Ernests Lavrinovics, Russa Biswas, Johannes Bjerva, and Katja Hose. 2025. Knowledge Graphs, Large Language Models, and Hallucinations: An NLP Perspective. 85 (2025), 100844. <https://doi.org/10.1016/j.websem.2024.100844>
- [44] Jinyang Li, Binyuan Hui, Ge Qu, Jiayi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Ma Chenhao, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs. *Advances in Neural Information Processing Systems* 36 (Dec. 2023), 42330–42357.
- [45] Qianlong Li, Chen Huang, Shuai Li, Yuanxin Xiang, Deng Xiong, and Wenqiang Lei. 2024. GraphOTTER: Evolving LLM-based Graph Reasoning for Complex Table Question Answering. <https://doi.org/10.48550/arXiv.2412.01230> arXiv:2412.01230 [cs]
- [46] Yingwei Ma, Qingping Yang, Rongyu Cao, Binhua Li, Fei Huang, and Yongbin Li. 2024. How to Understand Whole Software Repository? <https://doi.org/10.48550/arXiv.2406.01422> arXiv:2406.01422 [cs]
- [47] Yacine Majdoub and Eya Ben Charrada. 2024. Debugging with Open-Source Large Language Models: An Evaluation. In *Proceedings of the 18th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '24)*. Association for Computing Machinery, New York, NY, USA, 510–516. <https://doi.org/10.1145/3674805.3690758>
- [48] Akshat Malik, Bram Adams, and Ahmed Hassan. 2024. Towards Graph-Anonymization of Software Analytics Data: Empirical Study on JIT Defect Prediction. *Empirical Software Engineering* 29, 4 (June 2024), 76. <https://doi.org/10.1007/s10664-024-10464-6>
- [49] K.D. Maxwell, L. Van Wassenhove, and S. Dutta. 1996. Software Development Productivity of European Space, Military, and Industrial Applications. *IEEE Transactions on Software Engineering* 22, 10 (Oct. 1996), 706–718. <https://doi.org/10.1109/32.544349>
- [50] Meta. 2024. Meta-Llama/Meta-Llama-3-8B · Hugging Face. <https://huggingface.co/meta-llama/Meta-Llama-3-8B>.
- [51] Dušan Okanović, Samuel Beck, Lasse Merz, Christoph Zorn, Leonel Merino, André van Hoorn, and Fabian Beck. 2020. Can a Chatbot Support Software Engineers with Load Testing? Approach and Experiences. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 120–129. <https://doi.org/10.1145/3358960.3375792>
- [52] OpenAI. 2024. Best Practices for Prompt Engineering with the OpenAI API. <https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api>.
- [53] OpenAI. 2024. Hello GPT-4o. <https://openai.com/index/hello-gpt-4o/>.
- [54] Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. 2024. Unifying Large Language Models and Knowledge Graphs: A Roadmap. *IEEE Transactions on Knowledge and Data Engineering* 36, 7 (July 2024), 3580–3599. <https://doi.org/10.1109/TKDE.2024.3352100>
- [55] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirel. 2023. The Impact of AI on Developer Productivity: Evidence from GitHub Copilot. <https://doi.org/10.48550/arXiv.2302.06590> arXiv:2302.06590 [cs]
- [56] Kiran Ramesh, Surya Ravishankaran, Abhishek Joshi, and K. Chandrasekaran. 2017. A Survey of Design Techniques for Conversational Agents. In *Information, Communication and Computing Technology*, Saroj Kaushik, Daya Gupta, Latika Kharb, and Deepak Chahal (Eds.). Springer, Singapore, 336–350. https://doi.org/10.1007/978-981-10-6544-6_31
- [57] Ilaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 666–676. <https://doi.org/10.1109/ICSE.2015.82> ISSN: 1558-1225.
- [58] Juan Sequeda, Dean Allemang, and Bryon Jacob. 2025. Knowledge Graphs as a source of trust for LLM-powered enterprise question answering. (2025), 100858. <https://doi.org/10.1016/j.websem.2024.100858>
- [59] Chirag Shah, Ryen W. White, Reid Andersen, Georg Buscher, Scott Counts, Sarkar Snigdha Sarathi Das, Ali Montazer, Sathish Manivannan, Jennifer Neville, Xiaochuan Ni, Nagu Rangan, Tara Safavi, Siddharth Suri, Mengting Wan, Leijie Wang, and Longqi Yang. 2024. Using Large Language Models to Generate, Validate, and Apply User Intent Taxonomies. <https://doi.org/10.48550/arXiv.2309.13063> arXiv:2309.13063 [cs]
- [60] Vibhu Saujanya Sharma, Rohit Mehra, and Vikrant Kaulgud. 2017. What Do Developers Want? An Advisor Approach for Developer Priorities. In *2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*. IEEE Press, 78–81. <https://doi.org/10.1109/CHASE.2017.14>
- [61] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayeibi, Song Wang, and Hadi Hemmati. 2025. Prompt Engineering or Fine-Tuning: An Empirical Assessment of LLMs for Code. In *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR) (2025-04)*. 490–502. <https://doi.org/10.1109/MSR66628.2025.00082> ISSN: 2574-3864.

- [62] Amit Singhal. 2012. Introducing the Knowledge Graph: Things, Not Strings. <https://blog.google/products/search/introducing-knowledge-graph-things-not/>.
- [63] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When Do Changes Induce Fixes? *ACM SIGSOFT Software Engineering Notes* 30, 4 (May 2005), 1–5. <https://doi.org/10.1145/1082983.1083147>
- [64] Stackoverflow. 2024. *AI | 2024 Stack Overflow Developer Survey*. <https://survey.stackoverflow.co/2024/ai>
- [65] Linus Torvalds. 2024. Torvalds/Linux.
- [66] M. Vidoni. 2022. A Systematic Process for Mining Software Repositories: Results from a Systematic Literature Review. *Information and Software Technology* 144 (April 2022), 106791. <https://doi.org/10.1016/j.infsof.2021.106791>
- [67] Peiyi Wang, Lei Li, Liang Chen, Zefan Cai, Dawei Zhu, Binghuai Lin, Yunbo Cao, Qi Liu, Tianyu Liu, and Zhifang Sui. 2023. Large Language Models Are Not Fair Evaluators. <https://doi.org/10.48550/arXiv.2305.17926> arXiv:2305.17926
- [68] Peiyi Wang, Lei Li, Liang Chen, Feifan Song, Binghuai Lin, Yunbo Cao, Tianyu Liu, and Zhifang Sui. 2023. Making Large Language Models Better Reasoners with Alignment. <https://doi.org/10.48550/arXiv.2309.02144> arXiv:2309.02144
- [69] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenDevin: An Open Platform for AI Software Developers as Generalist Agents. (2024).
- [70] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi D. Q. Bui, Junnan Li, and Steven C. H. Hoi. 2023. CodeT5+: Open Code Large Language Models for Code Understanding and Generation. <https://doi.org/10.48550/arXiv.2305.07922> arXiv:2305.07922 [cs]
- [71] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems* 35 (Dec. 2022), 24824–24837.
- [72] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (New York, NY, USA, 2023-07-13) (ISSTA 2023). Association for Computing Machinery, 1282–1294. <https://doi.org/10.1145/3597926.3598135>
- [73] Ruobing Xie, Zhiyuan Liu, Jia Jia, Huanbo Luan, and Maosong Sun. 2016. Representation Learning of Knowledge Graphs with Entity Descriptions. *Proceedings of the AAAI Conference on Artificial Intelligence* 30, 1 (March 2016). <https://doi.org/10.1609/aaai.v30i1.10329>
- [74] Derong Xu, Xinhang Li, Ziheng Zhang, Zhenxi Lin, Zhihong Zhu, Zhi Zheng, Xian Wu, Xiangyu Zhao, Tong Xu, and Enhong Chen. 2025. Harnessing Large Language Models for Knowledge Graph Question Answering via Adaptive Multi-Aspect Retrieval-Augmentation. <https://doi.org/10.48550/arXiv.2412.18537> arXiv:2412.18537 [cs]
- [75] Junjielong Xu, Ziang Cui, Yuan Zhao, Xu Zhang, Shilin He, Pinjia He, Liqun Li, Yu Kang, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. 2024. UniLog: Automatic Logging via LLM and In-Context Learning. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (New York, NY, USA, 2024-02-06) (ICSE '24). Association for Computing Machinery, 1–12. <https://doi.org/10.1145/3597503.3623326>
- [76] Yizhuo Zhang, Heng Wang, Shangbin Feng, Zhaoxuan Tan, Xiaochuang Han, Tianxing He, and Yulia Tsvetkov. 2024. Can LLM Graph Reasoning Generalize beyond Pattern Memorization? <https://doi.org/10.48550/arXiv.2406.15992> arXiv:2406.15992
- [77] Yanjie Zhao, Haoyu Wang, Lei Ma, Yuxin Liu, Li Li, and John Grundy. 2019. Knowledge Graphing Git Repositories: A Preliminary Study. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 599–603. <https://doi.org/10.1109/SANER.2019.8668034>
- [78] Ting Zhou, Yanjie Zhao, Xinyi Hou, Xiaoyu Sun, Kai Chen, and Haoyu Wang. 2024. Bridging Design and Development with Automated Declarative UI Code Generation. <https://doi.org/10.48550/arXiv.2409.11667> arXiv:2409.11667
- [79] Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang, Binyuan Hui, Niklas Muennighoff, David Lo, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro Von Werra. 2025. BigCodeBench: Benchmarking Code Generation with Diverse Function Calls and Complex Instructions. <https://doi.org/10.48550/arXiv.2406.15877> arXiv:2406.15877 [cs]

A QUESTIONS USED FOR THE EVALUATION OF THE APPROACH

Table 18. Questions with parameters and corresponding intents and difficulty levels

#	Question	Intent	Level
1	How many commits happened in [DATE RANGE]?	Commits By Date Period	1
2	What is the latest commit?	Commits By Date	1
3	Can you tell me the details of the commits between [DATE RANGE]?	Commits By Date Period	1
4	Return a commit message on [DATE]?	Commits By Date	1
5	Show me the changes for [FILENAME] file?	File Commits	2
6	Give me all the commits for [FILENAME] file?	File Commits	2
7	Determine the developers that had the most unfixed bugs?	Overloaded Dev	2
8	Which developer has most number of bugs yet to be fixed?	Overloaded Dev	2
9	Determine the developers that fixed the most bugs in [FILENAME]?	Experienced Dev Fix Bugs	3
10	Who did most fixed bugs in [FILENAME]?	Experienced Dev Fix Bugs	3
11	Determine the files that introduce the most bugs?	Buggy Files	2
12	What are the most buggy files?	Buggy Files	2
13	What are the buggy commits that happened on [DATE]?	Buggy Commits By Date	2
14	What commits were buggy on [DATE]?	Buggy Commits By Date	2
15	Commit(s) that fixed the bug ticket [ISSUE ID]?	Fix Commit	2
16	Which commit fixed the bug ticket [ISSUE ID]?	Fix Commit	2
17	Determine the bug(s) that were introduced because of commit hash [COMMIT HASH]?	Buggy Commits	2
18	What are the bugs caused by commit [COMMIT HASH]?	Buggy Commits	2
19	Determine the percentage of the fixing commits that introduced bugs on [DATE]?	Buggy Fix Commits	3
20	How many fixing commits caused bugs on [DATE]?	Buggy Fix Commits	3

B QUESTION ANSWERED CORRECTLY AND INCORRECTLY IN EACH PROJECT IN RQ1

Table 19. Questions answered in each project. ✓ indicates the question was answered correctly and ✗ indicates the question was answered incorrectly or not answered

#	Question	AutoGPT	Bootstrap	Ohmyzsh	React	Vue
1	How many commits happened in [DATE RANGE]?	✓	✓	✓	✓	✓
2	What is the latest commit?	✓	✓	✓	✓	✓
3	Can you tell me the details of the commits between [DATE RANGE]?	✓	✓	✓	✓	✗
4	Return a commit message on [DATE]?	✓	✗	✗	✓	✗
5	Show me the changes for [FILENAME] file?	✓	✓	✓	✓	✓
6	Give me all the commits for [FILENAME] file?	✓	✓	✓	✓	✓
7	Determine the developers that had the most unfixed bugs?	✗	✗	✗	✗	✗
8	Which developer has most number of bugs yet to be fixed?	✗	✓	✓	✗	✗
9	Determine the developers that fixed the most bugs in [FILENAME]?	✓	✓	✓	✓	✓
10	Who did most fixed bugs in [FILENAME]?	✓	✗	✓	✓	✓
11	Determine the files that introduce the most bugs?	✓	✗	✓	✗	✗
12	What are the most buggy files?	✗	✓	✓	✓	✓
13	What are the buggy commits that happened on [DATE]?	✓	✓	✗	✗	✓
14	What commits were buggy on [DATE]?	✓	✓	✗	✗	✗
15	Commit(s) that fixed the bug ticket [ISSUE ID]?	✓	✓	✗	✓	✓
16	Which commit fixed the bug ticket [ISSUE ID]?	✓	✓	✓	✓	✓
17	Determine the bug(s) that were introduced because of commit [COMMIT HASH]?	✓	✓	✓	✓	✓
18	What are the bugs caused by commit [COMMIT HASH]?	✓	✗	✗	✓	✓
19	Determine the percentage of the fixing commits that introduced bugs on [DATE]?	✗	✗	✗	✗	✗
20	How many fixing commits caused bugs on [DATE]?	✗	✗	✗	✗	✗

C QUESTION ANSWERED CORRECTLY AND INCORRECTLY IN EACH PROJECT IN RQ3

Table 20. Questions answered in each project with few-shot chain-of-thought prompting. ✓ indicates the question was answered correctly and ✗ indicates the question was answered incorrectly or not answered

#	Question	AutoGPT	Bootstrap	Ohmyzsh	React	Vue
1	How many commits happened in [DATE RANGE]?	✓	✓	✓	✓	✓
2	What is the latest commit?	✓	✓	✓	✓	✓
3	Can you tell me the details of the commits between [DATE RANGE]?	✓	✓	✓	✓	✓
4	Return a commit message on [DATE]?	✓	✗	✗	✓	✗
5	Show me the changes for [FILENAME] file?	✓	✓	✓	✓	✓
6	Give me all the commits for [FILENAME] file?	✓	✓	✓	✓	✓
7	Determine the developers that had the most unfixed bugs?	✗	✗	✗	✗	✗
8	Which developer has most number of bugs yet to be fixed?	✓	✓	✓	✓	✓
9	Determine the developers that fixed the most bugs in [FILENAME]?	✓	✓	✗	✓	✓
10	Who did most fixed bugs in [FILENAME]?	✓	✗	✓	✓	✓
11	Determine the files that introduce the most bugs?	✗	✗	✓	✓	✗
12	What are the most buggy files?	✓	✓	✓	✓	✓
13	What are the buggy commits that happened on [DATE]?	✓	✓	✓	✓	✓
14	What commits were buggy on [DATE]?	✓	✓	✗	✗	✗
15	Commit(s) that fixed the bug ticket [ISSUE ID]?	✓	✓	✓	✓	✓
16	Which commit fixed the bug ticket [ISSUE ID]?	✓	✓	✓	✓	✓
17	Determine the bug(s) that were introduced because of commit [COMMIT HASH]?	✓	✓	✓	✓	✓
18	What are the bugs caused by commit [COMMIT HASH]?	✓	✓	✓	✓	✓
19	Determine the percentage of the fixing commits that introduced bugs on [DATE]?	✓	✓	✓	✓	✓
20	How many fixing commits caused bugs on [DATE]?	✓	✓	✓	✓	✓

D RESULTS OF THE ZERO-SHOT CHAIN-OF-THOUGHT PROMPTING ACROSS THE SELECTED PROJECTS AND DIFFICULTY LEVELS

Table 21. Comparison of the accuracy of the zero-shot chain-of-thought prompting across the selected projects

Project	Questions	Answered	Correct	Accuracy
AutoGPT	20	20	14	0.70
Bootstrap	20	19	15	0.75
Ohmyzsh	20	19	15	0.75
React	20	20	15	0.75
Vue	20	19	11	0.55
Overall	100	97	70	0.70

Table 22. Comparison of the accuracy of the zero-shot chain-of-thought prompting based on the difficulty level of the questions

Level	Questions	Answered	Correct	Accuracy
1	20	20	18	0.90
2	60	58	40	0.67
3	20	19	12	0.60
Overall	100	97	70	0.70

E ADDITIONAL RESULTS ON THE SELECTED PROJECTS AND DIFFICULTY LEVELS WHEN THE REASONING OF THE LLM ON THE AMBIGUOUS QUESTION ARE EVALUATED AS CORRECT

Table 23. Results by project showing the chain-of-thought responses where the LLM's reasoning was considered correct in ambiguous questions

Project	Questions	Answered	Correct	Accuracy
AutoGPT	20	20	20	1.00
Bootstrap	20	20	19	0.95
Ohmyzsh	20	18	18	0.90
React	20	19	19	0.95
Vue	20	18	18	0.90
Overall	100	95	94	0.94

Table 24. Results by level showing the chain-of-thought responses where the LLM’s reasoning was considered correct in ambiguous questions

Level	Questions	Answered	Correct	Accuracy
1	20	18	17	0.85
2	60	57	57	0.95
3	20	20	20	1.00
Overall	100	95	94	0.94

F NUMBER OF TASKS COMPLETED BY EACH PARTICIPANT AND THEIR CORRECT ANSWERS WITH AND WITHOUT THE CHATBOT

Table 25. Task Completion and Correctness per Participant (Manual vs. Chatbot).

Participant	Tasks Assigned	Completed Manually	Completed with Chatbot	Manual Correct	Chatbot Correct
Participant 1	5	0	5	0	5
Participant 2	5	4	5	4	5
Participant 3	5	4	5	3	5
Participant 4	5	5	4	2	4
Participant 5	5	2	5	2	5
Participant 6	5	3	5	2	4
Participant 7	5	5	5	4	5
Participant 8	5	5	5	4	5
Participant 9	5	4	5	3	5
Participant 10	5	4	4	0	3
Participant 11	5	4	4	2	4
Participant 12	5	3	4	1	1
Participant 13	5	4	5	1	5
Participant 14	5	4	5	2	4
Participant 15	5	4	5	2	4
Participant 16	5	4	5	1	5
Participant 17	5	3	5	1	5
Participant 18	5	3	5	1	5
Participant 19	5	0	0	0	0
Participant 20	5	4	5	1	5
Total	100	69	91	36	84

G AD-HOC EXAMPLE DEMONSTRATION

Question:

Who added the most lines of code in December, 2023?

Cypher Text (Output of Cypher Generator):

To determine who added the most lines of code in December 2023, we need to focus on the `additions` property of the `Commit` node. The relevant nodes are the `User` and `Commit` nodes, and the relevant relationship is the `author` relationship, which connects a `User` to a `Commit`. We also need to filter the commits to only include those made in December 2023.

First, we identify the relevant nodes and relationships:

- `User` node represents the developer.
- `Commit` node represents the commit made by the developer.
- `author` relationship connects the `User` to the `Commit`.

Next, we filter the commits to those made in December 2023. The `committedDate` property of the `Commit` node is used for this purpose, and we format the date using a regex to match the desired month and year.

Finally, we aggregate the total number of lines added by each user and sort the results to find the user with the most additions.

The complete query is as follows:

```
...
cypher
MATCH (u:User)-[:author]->(c:Commit)
WHERE c.committedDate =~ '2023-12-.*'
RETURN u.name AS developer, SUM(c.additions) AS totalAdditions
ORDER BY totalAdditions DESC
LIMIT 1
...
```

This query finds the user who added the most lines of code in December 2023 by summing the `additions` for each user's commits during that month and ordering the results in descending order. The `LIMIT 1` ensures that we only get the top result.

Cypher Query (Output of Cypher Query Extraction):

```
cypher
MATCH (u:User)-[:author]->(c:Commit)
WHERE c.committedDate =~ '2023-12-.*'
RETURN u.name AS developer, SUM(c.additions) AS
totalAdditions
ORDER BY totalAdditions DESC
LIMIT 1
```

Results (Output of Query Executor):

```
[[{'developer': 'XXXX XXXX', 'totalAdditions': 1423}]]
```

Response (Output of Response Generator):

The developer who added the most lines of code in December 2023 is XXXX XXXX, with a total of 1,423 additions.

Fig. 9. Demonstration of the approach answering a question not in the evaluation set. To protect privacy, we have redacted the developer's name.