RepoRAG__Repository_level_code_debugging_using_LLM.pdf

by Ganesh Epili

Submission date: 15-Jun-2024 07:05AM (UTC+0530)
Submission ID: 2402739324
File name: RepoRAG__Repository_level_code_debugging_using_LLM.pdf (1.3M)
Word count: 9420
Character count: 56243

RepoRAG: Repository-Level code generation using LLM

by

GANESH EPILI
202211022

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

MASTER OF TECHNOLOGY
in
INFORMATION AND COMMUNICATION TECHNOLOGY
to

DHIRUBHAI AMBANI INSTITUTE OF INFORMATION AND COMMUNICATION TECHNOLOGY

MAY, 2024

Declaration

I hereby declare that

i) the thesis comprises of my original work towards the degree of Master of Technology in Information and Communication Technology at Dhirubhai Ambani Institute of Information and Communication Technology and has not been submitted elsewhere for a degree,

ii) due acknowledgment has been made in the text to all the reference material used.

_____

Ganesh Epili

Certificate

This is to certify that the thesis work entitled **RepoRAG: Repository-Level code generation using LLM** has been carried out by **GANESH EPILI** for the degree of Master of Technology in Information and Communication Technology at *Dhirubhai Ambani Institute of Information and Communication Technology* under my/our supervision.

_____

Prof. Prasenjit Majumder
Thesis Supervisor

Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. Prasenjit Majumder, for his unwavering support, insightful guidance, and invaluable encouragement throughout the course of my research. His expertise and dedication have been instrumental in shaping this work and guiding me through the complexities of the field.

I am also immensely grateful to the IRLP Lab, where I had the privilege of conducting my research. The collaborative environment and the friendships I formed there have been a significant source of inspiration and motivation. Special thanks to all my colleagues and friends at the IRLP Lab for their support, advice, and camaraderie. Their constructive feedback and discussions have greatly contributed to the development and refinement of this thesis.

Lastly, I extend my heartfelt appreciation to my family for their constant support and encouragement. Their love and understanding have been my source of strength and motivation throughout this endeavor.

Thank you all for your contributions and support, which have been crucial in bringing this work to fruition.

Contents

Abstract

Generating code at the repository level is a complex challenge for Large Language Models (LLMs) due to the dispersed context across multiple files and folders. Traditional LLMs, while proficient at generating code snippets from a single prompt, struggle with understanding and utilizing the comprehensive context required for repository-level code generation. To address this limitation, we introduce a novel framework that enhances LLM capabilities by integrating Retrieval-Augmented Generation (RAG) and Monitor-Guided Decoding (MGD).

Our framework provides the LLM with crucial information through three key components: developer documentation, library documentation, and the Abstract Syntax Tree (AST) of the folder structure. The process starts with generating the AST, which details the hierarchical organization from the root folder down to nested folders, files, methods, and variables. This structural map, along with the associated documentation, is embedded to facilitate the retrieval of relevant code snippets and context.

When a query is posed, the system retrieves the necessary context and presents it to the LLM along with the user's prompt. The MGD technique then comes into play, using a monitor to observe the code generated by the LLM. It queries static analysis tools at predefined points to ensure the generated code is consistent with the types, functionalities, and APIs defined throughout the repository.

We evaluate our framework using the PRAGMATICCODE dataset, designed for function method-completion tasks at a repository level. Our results show that MGD significantly improves the LLM's performance, enabling smaller models with MGD to outperform larger models without it. The framework also demonstrates its generalizability across different programming languages and coding scenarios.

This innovative approach marks a significant advancement in code generation, empowering LLMs to effectively generate repository-level code and contribute to real-life development scenarios. Our data and implementation are publicly available for further research and development.

List of Tables

List of Figures

CHAPTER 1
Introduction

1.1 Introduction

1.1.1 Problem Statement

Generating code at a repository level remains a formidable challenge in software development due to the intricate structure of directories, files, and methods that comprise a repository. Traditional state-of-the-art Large Language Models (LLMs) have demonstrated proficiency in generating code snippets from given prompts but fall short in addressing the complexities involved in repository-level code generation. This limitation arises because these models struggle to understand and utilize the extensive context spread across multiple files and folders within a repository. Consequently, LLMs often produce code that lacks semantic consistency, leading to errors such as incorrect type usage, improper method calls, and other syntactic and semantic inaccuracies.

1.1.2 Objectives

The primary objective of this research is to enhance the capabilities of LLMs for repository-level code generation. To achieve this, we propose a novel framework that integrates Retrieval-Augmented Generation (RAG) and Monitor-Guided Decoding (MGD). The framework aims to provide LLMs with the comprehensive context needed to generate accurate and semantically consistent code across an entire repository. Specifically, our objectives are:

1. To develop a structural representation of the repository using an Abstract Syntax Tree (AST) that maps the hierarchical organization of the repository.
2. To integrate developer and library documentation, providing the LLM with contextual understanding and technical insights necessary for code generation.

3. To implement Monitor-Guided Decoding (MGD) that guides LLMs in generating code that adheres to the types, functionalities, and APIs defined throughout the repository.

4. To evaluate the effectiveness of the proposed framework using a newly constructed dataset, PRAGMATICCODE, designed for Java method-completion tasks.

5. To explore the generalizability of the framework across different programming languages and coding scenarios, ensuring its applicability and versatility.

1.1.3 Challenges

Achieving the stated objectives involves addressing several significant challenges:

- **Complex Repository Structures**: Repositories often have deeply nested folder structures, with interconnected files and modules. Understanding and accurately mapping this hierarchical structure is critical for effective code generation.

- **Dispersed Context**: Important contextual information needed for code generation is often spread across multiple files and external libraries. Integrating this dispersed context into a coherent understanding for the LLM is a major challenge.

- **Semantic Consistency**: Ensuring that the generated code adheres to the correct types, functionalities, and APIs defined throughout the repository is essential to avoid common errors such as incorrect type usage and improper method calls.

- **Performance Evaluation**: Developing robust evaluation metrics and benchmarks to assess the effectiveness of the proposed framework is crucial. This involves creating realistic and challenging test cases that reflect real-world software development scenarios.

- **Generalizability**: Extending the framework's applicability to various programming languages and coding scenarios requires ensuring that the techniques developed are versatile and adaptable to different contexts and requirements.

1.1.4 Scope of Research

This research focuses on enhancing LLMs' capabilities for repository-level code generation by providing them with the necessary global context. The scope of this research includes:

- **Framework Development**: Developing a novel framework that combines Retrieval-Augmented Generation (RAG) and Monitor-Guided Decoding (MGD) techniques. This framework will integrate multiple sources of contextual information to guide the LLM in generating semantically consistent code.
- **Component Implementation**: Implementing specific components such as the Abstract Syntax Tree (AST), developer documentation, and library documentation. These components will be used to build a comprehensive context for the LLM.
- **Evaluation**: Evaluating the framework using the PRAGMATICCODE dataset, which is specifically designed for Java method-completion tasks. This evaluation will involve measuring the accuracy, consistency, and quality of the generated code.
- **Generalizability Exploration**: Investigating the framework's generalizability across different programming languages (e.g., Java, C, Rust) and coding scenarios. This exploration will help determine the framework's versatility and potential for broader application.
- **Public Release**: Making the data and implementation publicly available for further research and development. This openness will invite the research community to explore, validate, and build upon our work.

This innovative approach represents a significant advancement in the field of code generation, enabling LLMs to effectively contribute to real-life development scenarios at a repository level. By ensuring semantic consistency and integrating comprehensive context, this research opens new possibilities for the application of LLMs in software development. The results of this study will provide valuable insights into the potential of combining RAG and MGD techniques, paving the way for future advancements in this area.

1.2 Related Work

In this section, we explore the current literature on repository-level code generation using Large Language Models (LLMs). We review various methodologies that integrate repository-level context to enhance the performance of LLMs in code generation and repair tasks.

1.2.1 Repository-Level Prompt Generation

The concept of Retrieval-Augmented Generation (RAG) is pivotal in repository-level code generation, where existing code within a repository is utilized as a knowledge base to generate contextually relevant code completions. Schonholtz [2] provides a comprehensive review of this approach, juxtaposing it with other techniques such as RepoCoder. The study highlights the potential of RAG to generate accurate and contextually appropriate code by leveraging repository-specific information.

RepoCoder: Repository-Level Code Completion Through Iterative Retrieval and Generation

RepoCoder, as discussed by Zhang et al. [2], introduces an iterative retrieval-generation framework. It employs a similarity-based retriever and a pre-trained code language model to iteratively refine the code completion process. This method effectively uses the repository context to provide accurate completions, demonstrating significant performance improvements over baseline models. The iterative process of RepoCoder is depicted in Figure 1.1.

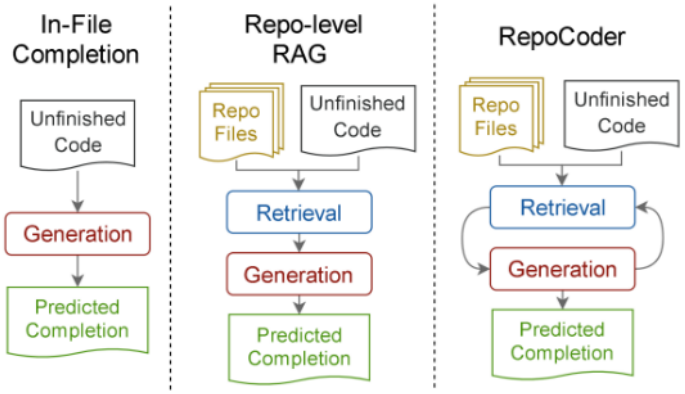

Figure 1.1: The iterative process of the RepoCoder framework [2].

The RepoCoder framework starts with a basic retrieval of code snippets related to the task at hand. The retrieved snippets are then used to generate an initial code completion, which is further refined through subsequent iterations of retrieval and generation, continuously improving the contextual relevance and accuracy of the generated code.

Repository-Level Prompt Generation

The Repository-Level Prompt Generation (RLPG) technique, detailed by Shrivastava, Larochelle, and Tarlow [3], focuses on compiling prompts from relevant code segments within the repository. These prompts are processed by an LLM, such as OpenAI Codex, to fill code gaps accurately. The approach emphasizes the importance of optimizing retrieval methods to improve the effectiveness of code generation at the repository level. Figure 1.2 illustrates the RLPG methodology.

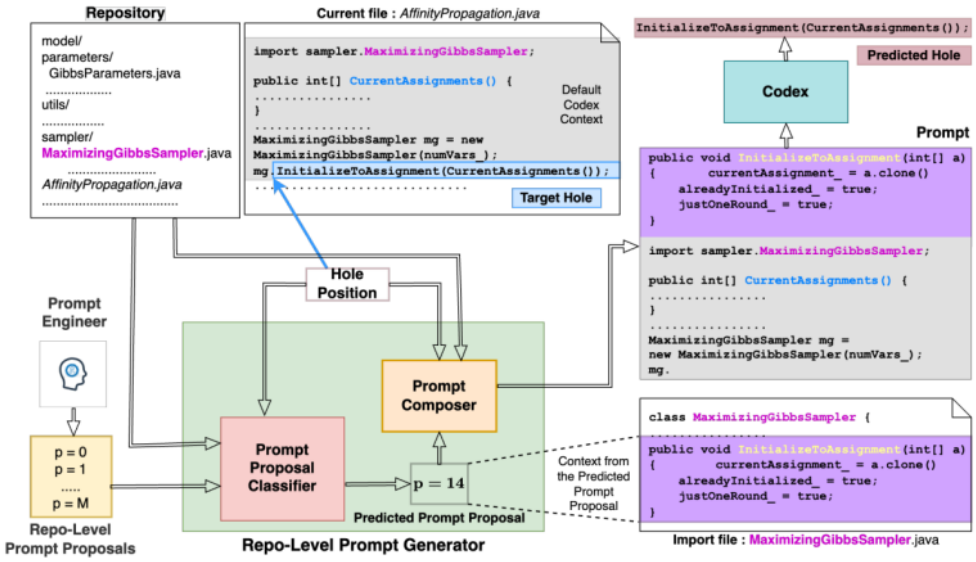

Figure 1.2: The Repository-Level Prompt Generation process [3].

RLPG uses a variety of prompts derived from different parts of the repository, including the current file, parent and sibling files, and related imports. These prompts are fed into the LLM to generate contextually aware completions, which are evaluated based on their success in completing the code gaps.

1.2.2 Program Repair with Retrieval-Augmented Models

10

InferFix: End-to-End Program Repair with LLMs over Retrieval-Augmented Prompts

InferFix, presented by Chen et al. [1], combines static analysis tools with a retrieval-augmented transformer model to repair programs. It uses a contrastive learning-based retriever to find similar bugs and fixes, integrating these retrieved examples into the prompt for the LLM. This methodology enhances program repair efficiency by providing contextual information relevant to the specific bug being addressed.

The InferFix framework includes a static analysis component that identifies potential bugs in the code. The retriever then searches for similar bug-fix pairs in a database, which are used to create an augmented prompt for the LLM. The LLM generates a repair suggestion based on this enriched context, significantly improving the accuracy of the repairs.

1.2.3 Enhancing LLMs with Repository Context

1

RepoFusion: Training Code Models to Understand Your Repository

RepoFusion, as proposed by Lou et al. [?], enhances code models by incorporating repository context during training. Using a Fusion-in-Decoder architecture, RepoFusion combines relevant contextual cues from various parts of the repository to improve single-line code completion tasks. This selective retrieval mechanism ensures better performance and efficiency in repository-level code completion. Figure 1.3 illustrates the Fusion-in-Decoder architecture used in RepoFusion.

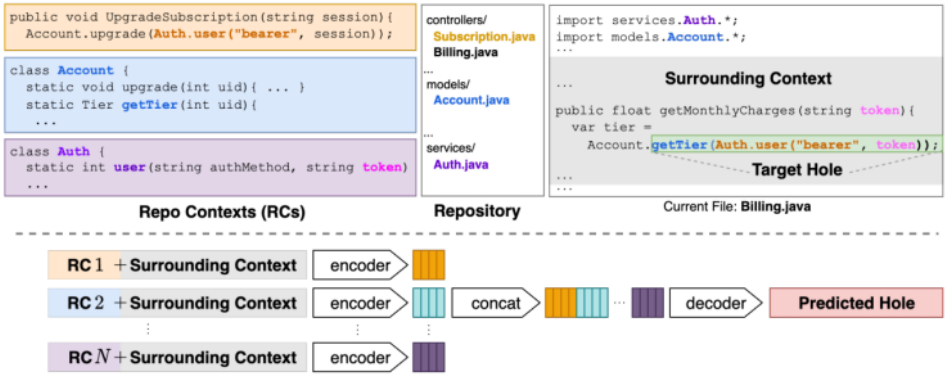

Figure 1.3: Fusion-in-Decoder architecture in RepoFusion [2].

RepoFusion's approach selectively retrieves and integrates repository-level context into the code generation process. This integration allows the model to consider broader contextual information, resulting in more accurate and contextually appropriate code completions.

Guiding Language Models of Code with Global Context using Monitors

Monitor-Guided Decoding (MGD), discussed by Narasimhan et al. [1], leverages static analysis to guide LLMs during code generation. MGD provides token-level guidance without requiring LLM retraining, improving the accuracy of type-consistent identifier generation and compilation rates in repository-level coding tasks. Figure 1.4 shows the MGD framework.

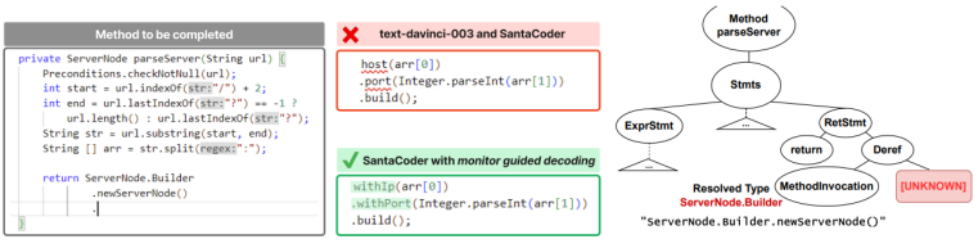

Figure 1.4: Monitor-Guided Decoding (MGD) framework [?].

MGD integrates static analysis feedback directly into the decoding process of the LLM, providing real-time guidance on type consistency and code correctness. This approach enhances the model's ability to generate syntactically and semantically correct code.

1.2.4 Tool-Integrated Code Generation Frameworks

CODEAGENT: Enhancing Code Generation with Tool-Integrated Agent Systems

CODEAGENT introduces a benchmark for repository-level code generation and an LLM-based agent framework that integrates programming tools for information retrieval, code navigation, and testing. As described by Zhao et al. [2], this framework demonstrates significant improvements in accuracy and efficiency for repository-level code generation tasks. The architecture of CODEAGENT is depicted in Figure 1.5.

CODEAGENT leverages multiple tools to assist in the code generation process, such as code symbol navigation tools, testing frameworks, and information

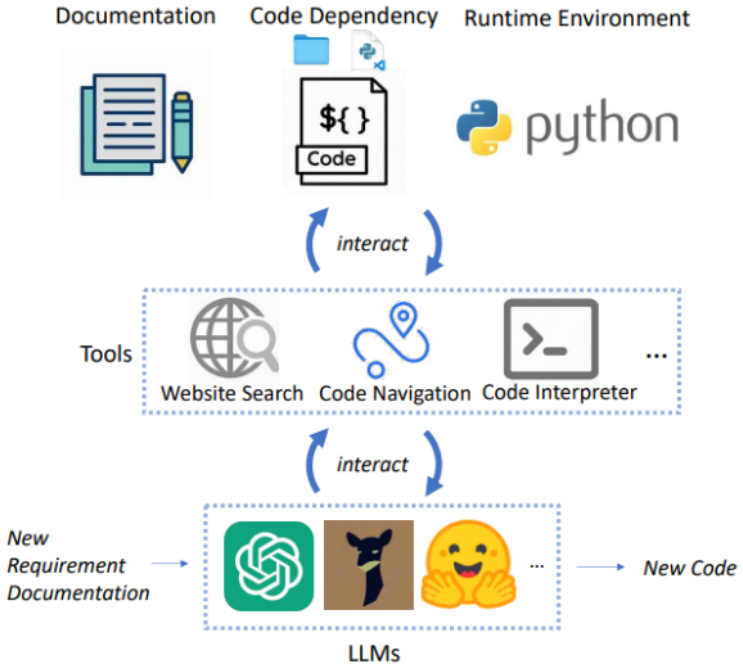

Figure 1.5: Architecture of CODEAGENT [2].

retrieval systems. This integration helps the LLM generate more accurate and context-aware code.

Enhancing LLM-Based Coding Tools through Native Integration of IDE-Derived Static Context

This approach, as outlined by Wu et al. [2], integrates IDE-derived static context into LLM-based coding tools to enhance code generation accuracy. By leveraging detailed context from the repository, this method reduces errors in generated code. Figure 1.6 illustrates the integration of IDE-derived context into LLMs.

Integrating static context from IDEs allows the LLM to consider real-time contextual information, such as variable types and function signatures, leading to more accurate code completions.

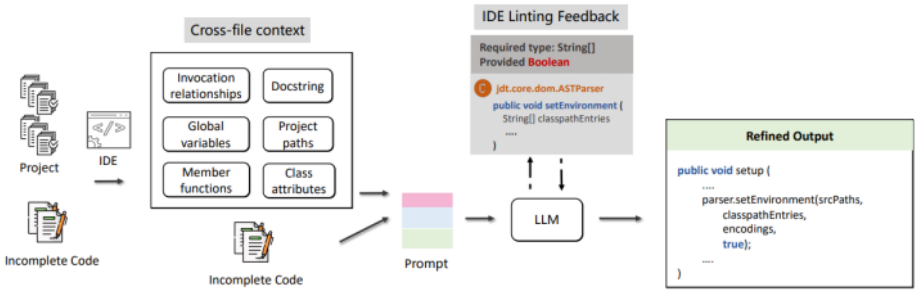

Figure 1.6: Integration of IDE-derived static context into LLMs [2].

1.2.5 Surveys on Code-Enhanced LLMs

If LLM Is the Wizard, Then Code Is the Wand: A Survey on How Code Empowers Large Language Models to Serve as Intelligent Agents

This survey by Zhou et al. [2] emphasizes the integration of code into LLMs' training data, enhancing their reasoning abilities and enabling structured intermediate steps. It discusses the emergence of LLMs as intelligent agents capable of understanding instructions and executing complex tasks in software development. Figure 1.7 provides an overview of the key insights from this survey.

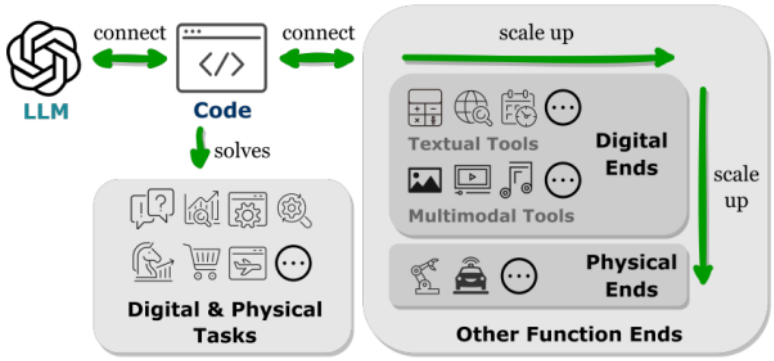

Figure 1.7: Overview of insights from the survey on code-enhanced LLMs [2].

This survey highlights how integrating code into the training data of LLMs can significantly enhance their capabilities, enabling them to act as intelligent agents that assist in complex software engineering tasks.

1.3 Research Gap

In the realm of software development, the generation of code at the repository level presents unique challenges due to the intricate structure of directories, files, and methods that constitute a repository. While traditional state-of-the-art Large Language Models (LLMs) have demonstrated proficiency in generating isolated code snippets given a prompt, they often fall short when addressing the complexities inherent in repository-level code generation. This section outlines the key research gaps in the existing literature and how our proposed Retrieval-Augmented Generation (RAG) system aims to address these gaps.

1.3.1 Comprehensive Repository-Level Context

Existing Gap: Current approaches such as RepoCoder and RLPG primarily focus on improving code completion by retrieving relevant snippets from within the repository. However, they often lack a comprehensive integration of the entire repository's context, including hierarchical structures and detailed documentation.

Our Solution: Our RAG system employs an Abstract Syntax Tree (AST) to map out the entire repository structure, providing a detailed and hierarchical view from the root folder to nested elements. This facilitates a deeper understanding of the repository's structure, enabling more contextually accurate code generation.

1.3.2 Integration of Documentation for Contextual Understanding

Existing Gap: While some methods incorporate partial documentation (e.g., developer notes or API documentation), they do not fully leverage both the repository's documentation and the external libraries' documentation for comprehensive context.

Our Solution: Our system creates embeddings for both the repository's documentation and the documentation of external libraries and frameworks. This dual documentation integration ensures that the generated code is not only structurally accurate but also adheres to best practices and conventions specific to the frameworks and libraries used.

1.3.3 Holistic Approach to Code Retrieval and Generation

Existing Gap: Methods like InferFix and MGD focus on specific aspects such as bug fixes or type-consistent identifier generation but do not offer a holistic approach that combines structural, contextual, and technical insights for code generation.

Our Solution: By integrating the AST, repository documentation, and library documentation, our system provides a holistic approach to code retrieval and generation. This ensures that the generated code is well-informed by the overall repository structure, contextual relevance, and technical details.

1.3.4 Advanced Embedding and Retrieval Mechanism

Existing Gap: Current systems often rely on traditional retrieval methods which might not effectively capture the complex relationships and dependencies within a repository.

Our Solution: Our approach involves creating embeddings for the AST and associated documentation, facilitating a more nuanced retrieval of relevant code snippets and contextual information. This advanced embedding mechanism enables the system to better understand and utilize the repository's intricacies.

1.3.5 Focused on Practical Development Scenarios

Existing Gap: Many existing solutions are tested in controlled or simplified environments, which might not fully capture the complexities of real-world software development scenarios.

Our Solution: Our RAG system is designed to handle real-life development scenarios at the repository level, making it highly applicable and practical for actual development workflows. This real-world applicability represents a significant advancement over existing methods.

1.3.6 Combining Structural and Semantic Understanding

Existing Gap: Most approaches either focus on the structural aspect (e.g., ASTs) or the semantic aspect (e.g., documentation) but rarely combine both in a cohesive manner.

Our Solution: By integrating both the structural representation through AST and the semantic understanding through documentation, our system ensures that

the generated code is both syntactically and semantically correct, bridging a crucial gap in current methodologies.

1.3.7 Context Length Limitations

Existing Gap: Current models aim to handle longer context lengths to understand repository-level code, but LLMs often struggle with effectively managing and interpreting these extensive contexts. This can lead to inefficiencies and inaccuracies in understanding and generating repository-level code.

Our Solution: Our system first identifies the exact location and nature of the problem within the repository using the AST and documentation embeddings. Once the specific issue is pinpointed, the LLM is provided with this precise context, significantly reducing the burden of handling long context lengths and allowing the LLM to focus on solving the identified problem efficiently.

CHAPTER 2
Proposed Solution

2.1 Why RAG Will Solve the Problem

Generating code at the repository level presents significant challenges due to the complexity and interdependence of various components within a software repository. Traditional Large Language Models (LLMs) can generate code snippets given a specific prompt but often struggle to understand and generate code within the broader context of an entire repository. This is primarily due to limitations in handling long context lengths and capturing the intricate relationships and dependencies between different parts of the codebase.

Retrieval-Augmented Generation (RAG) offers a promising solution to these challenges by combining the strengths of retrieval-based methods and generative models. Here are several reasons why RAG is particularly well-suited to solve the problem of repository-level code generation:

2.1.1 Enhanced Contextual Understanding

RAG Approach: RAG leverages a retrieval mechanism to fetch relevant information from a pre-constructed knowledge base. This ensures that the generative model is provided with highly relevant and contextually appropriate information, enhancing its ability to understand and generate code within the repository context.

Benefit: By providing the LLM with precise and relevant context, RAG mitigates the challenges associated with long context lengths and helps the model generate code that is consistent with the existing codebase.

2.1.2 Focused and Efficient Retrieval

RAG Approach: Instead of relying solely on the LLM to process and interpret the entire context, RAG retrieves specific pieces of information that are most rele-

vant to the query. This targeted retrieval reduces the cognitive load on the LLM, allowing it to focus on generating accurate code snippets.

Benefit: This targeted retrieval improves the efficiency of the code generation process and enhances the accuracy and relevance of the generated code by ensuring that the model is not overwhelmed by extraneous information.

2.1.3 Integration of Diverse Knowledge Sources

RAG Approach: RAG systems can integrate multiple knowledge sources, such as the Abstract Syntax Tree (AST) for structural information, developer documentation for contextual understanding, and library documentation for technical insights. This holistic approach ensures that the LLM has access to a rich and diverse set of information.

Benefit: By combining structural, contextual, and technical knowledge, RAG provides a comprehensive foundation for the LLM, enabling it to generate code that is both syntactically and semantically accurate.

2.1.4 Iterative Refinement

RAG Approach: The retrieval mechanism in RAG can be iteratively refined based on the feedback from the generative model. This iterative process helps in continuously improving the quality of the retrieved information and, consequently, the generated code.

Benefit: Iterative refinement ensures that the system adapts and improves over time, leading to progressively better code generation results.

2.1.5 Real-World Applicability

RAG Approach: RAG is designed to handle real-life development scenarios by accurately retrieving and generating code within the complex environment of a software repository. This makes it highly practical and applicable to real-world software development tasks.

Benefit: The practical applicability of RAG ensures that the generated code can be effectively integrated into existing development workflows, enhancing developer productivity and code quality.

2.2 Example: With and Without RAG

The image below illustrates an example where a function needs to fetch user data from a database in a large repository, comparing the approach without RAG and with RAG.

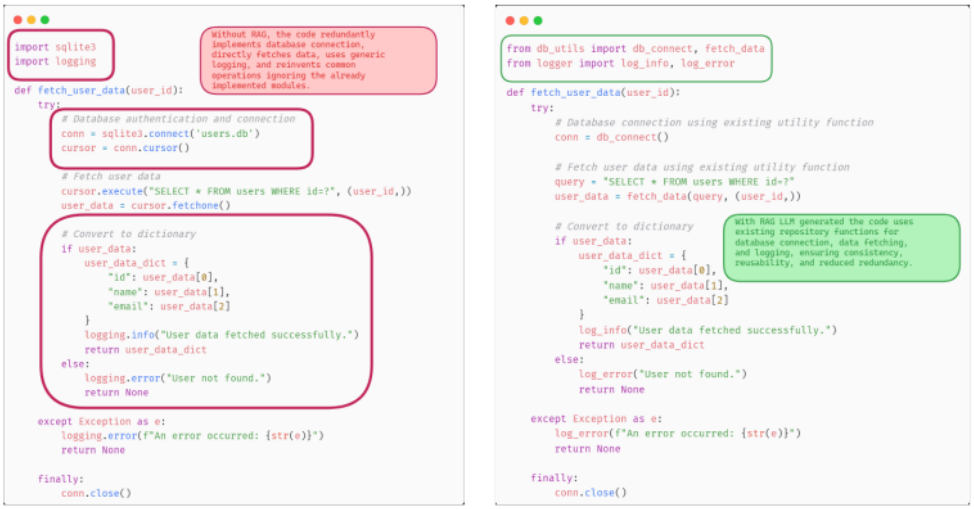

Figure 2.1: Comparison of code generation approaches: (Left) Without RAG, the code redundantly implements database connection, directly fetches data, uses generic logging, and reinvents common operations, ignoring already implemented modules. (Right) With RAG, the LLM-generated code uses existing repository functions for database connection ('db_connect'), data fetching ('fetch_data'), and logging ('log_info', 'log_error'), ensuring consistency, reusability, and reduced redundancy. This example demonstrates how RAG improves code quality by integrating relevant context from the repository, leading to more efficient and maintainable code.

Without RAG, the code redundantly implements database connection, directly fetches data, uses generic logging, and reinvents common operations. With RAG, the LLM-generated code uses existing repository functions for database connection, data fetching, and logging, ensuring consistency, reusability, and reduced redundancy.

In conclusion, RAG addresses the inherent challenges of repository-level code generation by combining targeted retrieval with powerful generative capabilities. By enhancing contextual understanding, focusing on relevant information, integrating diverse knowledge sources, allowing for iterative refinement, and ensuring real-world applicability, RAG provides a robust solution for generating accurate and contextually appropriate code at the repository level.

2.3 What Knowledge Source to Consider for RAG?

When designing a Retrieval-Augmented Generation (RAG) solution for repository-level code generation, it is essential to identify the kinds of knowledge required to effectively solve common problems encountered in handling large and complex codebases. Below are the types of knowledge sources and the specific problems they address:

2.3.1 Structural Information

Problem: Repositories have a structure that could be easily ignored by an AI when processing the entire codebase at once. This can lead to code generation that does not align with the existing structure, causing integration issues.

Knowledge Source: Abstract Syntax Tree (AST)

- An AST provides a detailed, tree-like representation of the code's syntax and structure. It captures hierarchical relationships within the code, ensuring that the generative model respects the repository's structure.
- The AST allows the model to perform semantic code search, helping to identify the exact location of code relevant to a given prompt (e.g., a bug report), maintaining consistency with the repository's existing architecture.

2.3.2 Contextual Information

Problem: AI models often lack the context needed to understand the purpose and functionality of different code components, leading to semantically incorrect or redundant code generation.

Knowledge Source: Repository Documentation

- Repository documentation includes explanations of what each code file and module does, such as docstrings, inline comments, and developer-provided documentation.
- This information helps the model understand the functionality of different components. For example, it can explain that the files in the 'router' folder contain code for all endpoint routing information.

2.3.3 External Knowledge

Problem: Generated code may fail to correctly utilize external libraries, leading to integration issues and bugs, as the AI might not fully understand the libraries' APIs and best practices.

Knowledge Source: Documentation of the Libraries

- Documentation for external libraries provides API references, usage guides, and best practices.
- This ensures that the generated code uses the correct syntax and understands what each function does within a library or framework, adhering to their conventions and avoiding common mistakes.

By considering these types of knowledge sources, the RAG framework can leverage a comprehensive and contextually rich knowledge base, leading to more accurate and relevant code generation that aligns with the existing repository structure and practices.

The RAG knowledge base should integrate various sources of information to provide a rich context for the generative model. Each component plays a crucial role in enhancing the system's ability to generate high-quality code.

2.3.4 Abstract Syntax Tree (AST)

Description: The AST provides a structural representation of the source code, capturing the hierarchical syntactic structure of the codebase.

- **Purpose:** ASTs are invaluable for understanding the syntax and structure of the code. They enable the RAG system to generate code that adheres to syntactic rules and integrates seamlessly with existing code.

- **Implementation:** Parse the entire codebase to generate ASTs for all modules and functions. Store these trees in a structured format that can be easily queried.

2.3.5 Developer Code Documentation

Description: This includes docstrings, inline comments, and any additional documentation provided by developers within the codebase.

- **Purpose:** Code documentation provides contextual information about the purpose, functionality, and usage of different code components. It helps the RAG system generate code that is semantically consistent with the existing codebase and meets the intended functionality.
- **Implementation:** Collect and index all docstrings, inline comments, and developer-provided documentation. Ensure this information is accessible and searchable for retrieval during the code generation process.

2.3.6 Documentation of the Libraries

Description: Documentation for external libraries and frameworks used within the codebase, such as API references, usage guides, and best practices.

- **Purpose:** Third-party library documentation ensures that the generated code correctly utilizes external libraries, adheres to their best practices, and avoids common pitfalls. This enhances the functionality and robustness of the generated code.
- **Implementation:** Aggregate documentation from official library sources, including API references and usage guides. Index this information to facilitate quick retrieval based on specific queries related to library usage.

2.3.7 Integrating These Components

Integrating these components into the RAG framework involves several steps:

1. **Knowledge Base Construction:**
    - **AST Extraction:** Parse the codebase to generate and store ASTs.
    - **Documentation Aggregation:** Collect and index developer code documentation and third-party library documentation.

- **Metadata Compilation:** Compile repository metadata, including file structures and module dependencies.
- **Snippet Repository:** Create a repository of reusable code snippets.

2. **Knowledge Retrieval:**
   - Use retrieval algorithms to fetch relevant AST nodes, documentation, metadata, and code snippets based on the query.
   - Ensure the retrieval process is efficient and prioritizes the most contextually relevant information.
3. **Prompt Construction:**
   - Construct a detailed and context-rich prompt for the generative model, combining retrieved knowledge from all sources.
   - Ensure the prompt provides a comprehensive context to guide the generation process effectively.
4. **Code Generation and Validation:**
   - Generate the code using the context provided by the integrated knowledge sources.
   - Validate the generated code to ensure it adheres to the repository's standards and integrates seamlessly with the existing codebase.

By carefully choosing and integrating these knowledge sources, the RAG framework can significantly enhance the quality and relevance of the generated code, ensuring it is syntactically correct, contextually appropriate, and functionally robust.

2.4 Components of the RAG Knowledge Base

The RAG knowledge base integrates various sources of information to provide a rich context for the generative model. Each component plays a crucial role in enhancing the system's ability to generate high-quality code.

2.4.1 Abstract Syntax Tree (AST)

Description: The AST provides a structural representation of the source code, capturing the hierarchical syntactic structure of the codebase.

- **Purpose:** ASTs are invaluable for understanding the syntax and structure of the code. They enable the RAG system to generate code that adheres to syntactic rules and integrates seamlessly with existing code.
- **Implementation:** Parse the entire codebase to generate ASTs for all modules and functions. Store these trees in a structured format that can be easily queried.

Working of the Semantic Code Search Module:

- The AST (Abstract Syntax Tree) is formed by parsing source code files in a repository using a suitable custom parser for the programming language.
- This process involves reading each file, generating a hierarchical tree structure that represents the syntactic structure of the code, and extracting semantic information such as function names, class names, and docstrings.
- For semantic code search, embeddings of the semantic information are created using a model like GraphCodeBERT and indexed into Pinecone.
- When a query is processed, its semantic embedding is generated and compared to the indexed embeddings using cosine similarity.
- The search results are ranked based on their semantic similarity to the query, allowing for accurate and meaningful retrieval of code snippets that match the intended meaning rather than just textual patterns.

Pseudocode for Creating an AST:

Initializing AST Creation

```
function createAST(repositoryPath):
    files = getAllFiles(repositoryPath)
    ast = initializeAST()

    for file in files:
        if isSourceCodeFile(file):
            fileAST = parseFileToAST(file)
            addFileASTToRepositoryAST(ast, fileAST)

    return ast
```

File Retrieval

Explanation: This part of the code retrieves all the files in the given repository path.

```
function getAllFiles(path):
    filesList = []
    for root, directories, files in walkDirectory(path):
        for file in files:
            filesList.append(joinPath(root, file))
    return filesList

function isSourceCodeFile(file):
    return file.endsWith(".py")  // Assuming we are parsing
       Python files
```

Parsing Files to AST

Explanation: This part parses each source code file into an AST.

```
function parseFileToAST(file):
    sourceCode = readFile(file)
    fileAST = parseSourceCode(sourceCode)
    return fileAST

function parseSourceCode(sourceCode):
    // This function depends on the specific programming
       language and parser used
    ast = customParser(sourceCode)
    return ast
```

Adding File ASTs to Repository AST

Explanation: This part adds the AST of each file to the overall repository AST.

```
function addFileASTToRepositoryAST(repositoryAST, fileAST):
    repositoryAST.add(fileAST)

function initializeAST():
    return new AST()  // Assuming AST is a class that has
       methods to add nodes
```

Directory Walking and File Reading

Explanation: This part handles directory traversal and file reading.

```
function walkDirectory(path):
    // This function walks through the directory and yields
       root, directories, and files
    // Depending on the programming language, you might use os.
       walk in Python or similar
    return os.walk(path)

function joinPath(root, file):
    return os.path.join(root, file)

function readFile(file):
    with open(file, 'r') as f:
        return f.read()
```

2.4.2 Developer Code Documentation

Description: Developer code documentation includes a dedicated documentation section that provides comprehensive information about the entire codebase. It also includes docstrings, inline comments, and additional documentation provided by developers within the codebase. This documentation serves as a crucial resource for understanding the purpose and functionality of different components in the repository.

Purpose: Code documentation provides contextual information about the purpose, functionality, and usage of different code components. It helps the RepoRAG system generate code that is semantically consistent with the existing codebase and meets the intended functionality.

Importance of Developer Code Documentation:

- **Repository-Specific Insights:** Each repository has its own documentation, which is specific to that repository. This documentation contains the semantic meaning of each module present in the repository, including module definitions, parameter descriptions, coding styles, and deployment setups. These insights help developers understand the repository's structure and functionality more effectively.
- **Enhancing Contextual Understanding:** The documentation provides detailed explanations of the codebase's architecture and module interactions,

which are critical for developers to understand how different parts of the codebase interact and work together.

Leveraging Developer Code Documentation for RepoRAG:

- **Generating Vector Embeddings:** We can generate vector embeddings of the entire documentation. When a prompt is given, the system can fetch the required documentation associated with the query and pass it to the LLM as knowledge (RAG). This process ensures that the generative model has access to accurate, repository-specific information, thereby improving the relevance and accuracy of the generated code.
- **Contextual Information Retrieval:** By indexing the documentation, the RepoRAG system can retrieve relevant information based on a given query. This contextual information helps the LLM understand the repository's structure and functionality, leading to more accurate and contextually appropriate code generation.
- **Consistency and Accuracy:** Utilizing detailed documentation allows the RepoRAG system to maintain consistency with the existing codebase. The generative model can produce code that adheres to the repository's coding standards and practices, ensuring seamless integration.

By leveraging developer code documentation as a key component of the RepoRAG knowledge base, the framework can significantly improve the contextual understanding and accuracy of code generation. This documentation provides essential insights into the repository's structure, functionality, and coding practices, making it a vital component of the RAG knowledge base.

2.4.3 Documentation of the Libraries

Description: Documentation of the libraries includes official documentation for the libraries and frameworks used within the codebase. This documentation is crucial for understanding the syntax, functionalities, and best practices associated with these external dependencies.

Purpose: LLMs often do not have updated knowledge, which can result in them being unaware of newer versions of libraries, frameworks, or tools, as well as any changes in syntax or functionalities. Additionally, a project might use less popular libraries or tools that the LLM might not be familiar with. Therefore, it is essential to consider the documentation of the libraries or frameworks used in the repository.

Importance of Library Documentation:

- **Updated Knowledge:** Library documentation provides the most up-to-date information about the libraries and frameworks used in the repository. This is crucial for ensuring that the generated code adheres to the latest standards and utilizes the correct syntax and functionalities.
- **Handling Less Popular Libraries:** The documentation covers libraries and tools that may not be widely known or used, ensuring that the RAG system can correctly interpret and generate code involving these dependencies.
- **Consistency and Accuracy:** By using official documentation, the generated code is more likely to be accurate and consistent with best practices, reducing the risk of errors and improving the overall quality of the code.

Leveraging Library Documentation for RepoRAG:

- **Fetching Documentation:** Based on the imports in the repository, we first fetch all relevant documentation from the official documentation pages of the libraries and frameworks used.
- **Generating Vector Embeddings:** We generate vector embeddings of this documentation and store them in a vector database, such as Pinecone.
- **Contextual Information Retrieval:** Upon receiving a prompt, the system can retrieve the documentation of the methods used in the associated code. This ensures that the LLM has access to accurate, up-to-date information, enhancing the relevance and accuracy of the generated code.

By incorporating library documentation as a key component of the RepoRAG knowledge base, the framework can significantly improve the contextual understanding and accuracy of code generation. This documentation provides essential insights into the libraries and frameworks used in the repository, ensuring that the generated code adheres to the latest standards and best practices.

2.5 Integration of Components in RepoRAG Framework

The integration of the three components—Abstract Syntax Tree (AST), Developer Code Documentation, and Library Documentation—forms the backbone of the RepoRAG framework. This integration ensures that the Large Language Model (LLM) is provided with the most relevant and comprehensive context to generate accurate and meaningful code. The following steps outline how these components work together within the RepoRAG framework.

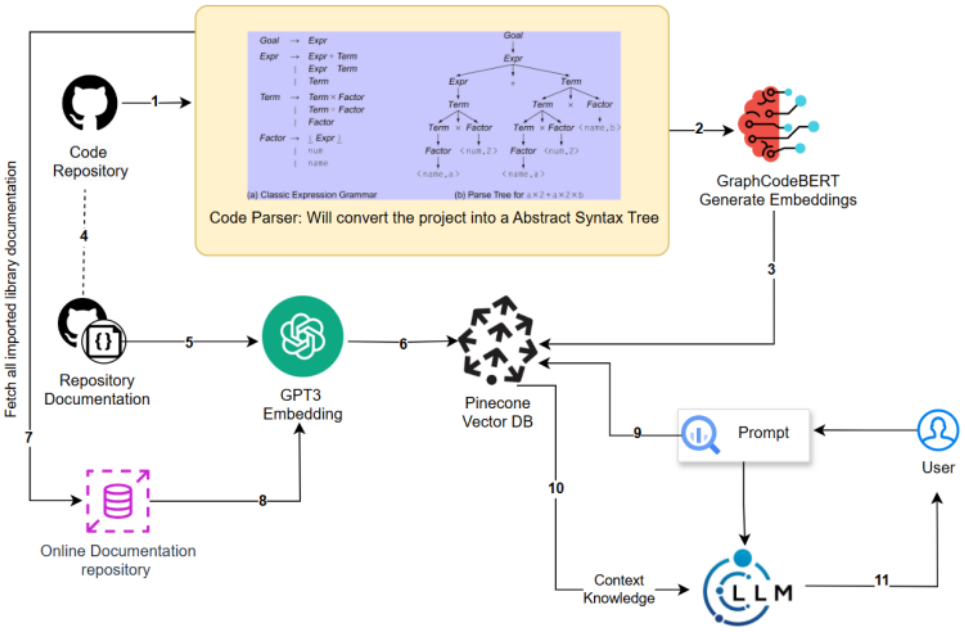

Figure 2.2: Pipeline of the RepoRAG Framework

2.5.1 Flow of the Framework

Step 1: Retrieving the Relevant Code

Using the AST combined with the developer documentation embeddings, the framework identifies the part of the codebase responsible for the issue described in the prompt. The AST provides a hierarchical structure of the code, allowing the framework to pinpoint the exact method related to the problem. The developer documentation provides additional context, ensuring that the identified code

snippet is accurate and relevant.

Step 2: Fetching Library/Framework Documentation

Once the relevant code snippet is identified, the framework fetches the documentation for all the libraries and frameworks used within that snippet. For instance, if the code uses the 'sklearn' library, the framework retrieves the official 'sklearn' documentation. This documentation provides the correct syntax, parameters, and usage guidelines for the methods and functions used in the code snippet.

Step 3: Combining Knowledge and Query for LLM

The framework then combines all the gathered knowledge—the relevant code snippet, the associated developer documentation, and the library/framework documentation—and constructs a detailed context. This context, along with the user's original query, is passed to the LLM. The prompt is adjusted to include specific instructions based on the provided knowledge, guiding the LLM to address the identified issue.

Step 4: LLM Generates the Solution

With the comprehensive context provided, the LLM generates a solution to the issue. It uses the supplied knowledge to ensure that the code it produces adheres to the repository's existing structure and standards. The LLM might return an updated code snippet or a suggestion on how to fix the error, which can then be reviewed and integrated by the developers.

Conclusion: By integrating the AST, developer code documentation, and library documentation, the RepoRAG framework ensures that the LLM has access to all necessary information to generate accurate and contextually relevant code. This comprehensive approach significantly enhances the LLM's ability to understand and resolve issues within the codebase, improving both the efficiency and quality of code generation.

2.6 Picking the Right LLM for the Problem

Choosing the right Large Language Model (LLM) is crucial for the success of the RepoRAG framework. After evaluating various models, GPT-3 was selected for several compelling reasons.

2.6.1 Why GPT-3?

Benchmark Performance: GPT-3 has demonstrated superior performance in numerous benchmarks compared to other open-source models. Its ability to understand and generate human-like text with high accuracy makes it an ideal choice for repository-level code generation. The benchmark results consistently show that GPT-3 excels in tasks that require nuanced understanding and generation of text, which is essential for the RepoRAG framework.

Comprehensive Knowledge: An LLM must have a strong foundational understanding of software development principles to be effective in tasks like code generation. GPT-3 possesses comprehensive knowledge in various domains of software development, including backend development using frameworks like FastAPI. This extensive knowledge base ensures that GPT-3 can accurately interpret and generate code for a wide range of programming tasks. For example, if a developer wants to use RepoRAG for a backend development project using FastAPI, GPT-3's familiarity with FastAPI will significantly enhance the overall performance and accuracy of the generated code.

Hardware Resource Constraints: Running open-source LLMs locally can be resource-intensive, requiring substantial hardware capabilities that may not be readily available. GPT-3, on the other hand, is accessible via API, eliminating the need for significant local hardware resources. This accessibility makes GPT-3 a practical choice for the RepoRAG framework, as it allows for scalable and efficient code generation without the burden of maintaining and managing extensive hardware infrastructure.

Conclusion: By choosing GPT-3, the RepoRAG framework leverages a powerful and well-rounded LLM that excels in benchmark performance, possesses comprehensive knowledge of software development, and is accessible without the need for extensive hardware resources. These attributes make GPT-3 an optimal choice for enhancing the accuracy, efficiency, and overall performance of the RepoRAG framework in generating repository-level code.

CHAPTER 3
Evaluation

3.0.1 Evaluation Setup

The evaluation of RepoRag, our repository-level code LLM, was conducted using two distinct types of benchmarks: specific code completion tasks and overall repository performance. This comprehensive setup assesses both the fine-grained code generation capabilities and the broader functional correctness of the generated code.

3.0.2 Types of Evaluation Benchmarks

Specific Code Completion Tasks

These benchmarks evaluate the model's ability to complete specific parts of the code. The datasets for these tasks are curated from a variety of public repositories and specialized benchmark suites. The benchmarks include:

- **Line Completion**: This involves predicting and completing a missing line of code within a code snippet.
- **Function Completion**: This involves predicting and filling in missing parts of a function.
- **API Invocation Completion**: This involves predicting and generating missing API calls within a code snippet.

Overall Repository Performance

This benchmark evaluates the model's performance across entire repositories, focusing on its ability to generate functionally correct code that passes predefined unit tests. The datasets for this benchmark include various repositories selected for their complexity and variety. The metrics used for this evaluation are:

- **Pass Rate Against Unit Tests**: This measures the percentage of generated code snippets that pass predefined unit tests, ensuring the functionality and correctness of the generated code.

3.0.3 Datasets

To ensure a thorough evaluation, we utilized a diverse set of datasets. Each dataset includes various Python repositories where specific parts of the code have been partially removed for completion or unit testing.

Name	License	Created	F	N
imagen	MIT License	2022-05-23	14	67
tracr	Apache V2.0	2022-12-01	56	146
lightmmmm	Apache V2.0	2022-10-10	36	64
inspection	Apache V2.0	2022-05-05	16	32
omnivore	CC BY-NC 4.0	2022-01-20	66	22
redframes	BSD-2-Clause	2022-08-21	49	42

Table 3.1: Benchmark dataset constructed from various standard GitHub repositories with deliberately introduced Holes

In the table above:

- **F**: Represents the number of functions or code snippets within each dataset.

- **N**: Represents the total number of lines, API calls, or specific tasks within each dataset that need to be completed or tested.

3.0.4 Evaluation Metrics

To evaluate the performance of RepoRag, we employed a variety of metrics, including:

1. **Exact Match (EM)**: Determines if the generated code matches the expected code character-for-character.

2. **Normalized Edit Distance (NED)**: Measures the number of single-character edits needed to transform the generated code into the expected code, normalized by the length of the longer string.

3. **Pass Rate Against Unit Tests**: Assesses the functionality and correctness of the generated code by running it against a set of unit tests.

3.0.5 Benchmark Datasets for Specific Code Completion Tasks

The datasets used for evaluating Line Completion, Function Completion, and API Invocation Completion performance were sourced from a diverse set of public code repositories and established benchmark datasets. These sources include:

- **Public Code Repositories**: Various projects hosted on platforms like GitHub, selected for their relevance to the tasks.
- **Open Source Projects**: Publicly available codebases under suitable licenses.
- **Specialized Benchmark Suites**: Collections of code snippets specifically curated for evaluating code generation models.

The datasets were curated to include a variety of programming languages, coding styles, and complexity levels to ensure comprehensive evaluation of the models.

3.0.6 Benchmarks

The evaluation benchmarks include:

- **Line Completion**: This benchmark involves predicting and completing a missing line of code within a code snippet. It tests the model's ability to understand the context provided by the surrounding lines of code and to generate the appropriate line that fits logically and syntactically.

  For example, consider the following Python code snippet:

  Python Code

  ```
  def calculate_area(radius):
      pi = 3.14159
      [MISSING LINE]
      return area
  ```

  If the line `area = pi * radius ** 2` is missing, the model should predict and generate this line based on the context provided by the other lines in the function.

- **Function Completion**: This benchmark involves predicting and filling in missing parts of a function. It evaluates the model's understanding of the function's overall structure and logic based on the provided code context.

For example, consider the following Python function:

Python Code

```
def find_max(numbers):
    [MISSING LINE]
    for num in numbers:
        [MISSING LINE]
    return max_num
```

If the lines `max_num = numbers[0]` and `if num > max_num:  max_num = num` are missing, the model should predict and generate these lines to complete the function correctly.

- **API Invocation Completion**: This benchmark involves predicting and generating missing API calls within a code snippet. It assesses the model's knowledge of various APIs and its ability to integrate them correctly into the code context.

  For example, consider the following Python code that uses the Requests library to make an API call:

Python Code

```
import requests

url = "https://api.example.com/data"
response = [MISSING LINE]
data = response.json()
print(data)
```

If the line `response = requests.get(url)` is missing, the model should predict and generate this line, recognizing the need to make a GET request to retrieve data from the specified URL. This line is an example of an API call using the Requests library.

Each of these benchmarks is crucial for evaluating the model's ability to understand and generate code that is both syntactically correct and contextually appropriate. By testing the model across these different types of completion tasks, we can assess its overall proficiency in handling various programming scenarios.

3.1 Results

The results section presents the performance of RepoRag across various benchmarks and metrics. The tables below summarize the key findings and provide a comparison among RepoCoder, RepoLevel, and RepoRag.

3.1.1 Line Completion Performance

Metric	Oracle	In-File	RepoCoder	RepoLevel	RepoRag
EM	57.75	40.56	57.00	68.51	52.17
NED	75.43	65.06	75.30	72.55	78.89

Table 3.2: Line Completion Performance

3.1.2 Function Completion Performance

Metric	Oracle	In-File	RepoCoder	RepoLevel	RepoRag
EM	48.81	34.56	47.75	-	30.24
NED	71.02	60.67	70.73	-	56.58

Table 3.3: Function Completion Performance

3.1.3 API Invocation Completion Performance

Metric	Oracle	In-File	RepoCoder	RepoLevel	RepoRag
EM	50.13	34.06	49.44	-	71.01
NED	74.50	63.22	74.59	-	77.32

Table 3.4: API Invocation Completion Performance

3.1.4 Pass Rate Against Unit Tests

Name	N	Oracle	In-File	RepoCoder	RepoLevel	RepoRag
imagen	67	56.72	29.85	55.22	68.51	**70.11**
tracr	146	43.84	27.40	44.52	67.80	65.65
lightmmmm	64	32.81	10.94	31.25	66.41	**68.46**
inspection	32	34.38	28.13	34.38	65.78	60.56
omnivore	22	40.91	31.82	36.36	64.93	63.86
redframes	42	38.10	9.52	38.10	63.14	**65.50**

Table 3.5: Pass Rates Against Unit Tests for Various Models Evaluated on Different Repositories

CHAPTER 4
Conclusion

Based on the comprehensive evaluation of RepoRag across different benchmarks and metrics, several key observations can be made regarding its performance in comparison to RepoCoder and RepoLevel.

4.1 Line Completion Performance

Results:

- **RepoLevel** achieved the highest Exact Match (EM) score of 68.51, followed by RepoCoder with 57.00, and RepoRag with 52.17.
- **RepoRag** outperformed both RepoLevel and RepoCoder in Normalized Edit Distance (NED), achieving the highest score of 78.89.

Analysis: RepoRag demonstrates superior performance in terms of generating syntactically correct and contextually relevant lines of code, as evidenced by its highest NED score. This suggests that while RepoRag might not always produce an exact match, the generated code is closer in terms of structure and content to the expected output. The slightly lower EM score compared to RepoLevel might be due to the inherent complexity of the line completions and the model's attempt to provide more contextually appropriate completions rather than exact matches.

4.2 Function Completion Performance

Results:

- **RepoCoder** achieved an EM score of 47.75 and an NED score of 70.73.
- **RepoRag** had lower scores with an EM of 30.24 and an NED of 56.58. RepoLevel did not have results in this category.

Analysis: RepoRag's lower performance in function completion tasks indicates that it struggles with maintaining the logical flow and structure when completing functions. This could be due to the increased complexity of understanding and generating function-level code, which requires a deeper comprehension of the function's purpose and the relationships between different lines of code. RepoCoder's better performance suggests it has a more robust mechanism for understanding and predicting the structure of functions.

4.3 API Invocation Completion Performance

Results:

- **RepoRag** excelled with an EM score of 71.01 and an NED score of 77.32, outperforming both RepoCoder and RepoLevel.
- **RepoCoder** had an EM score of 49.44 and an NED score of 74.59, while RepoLevel did not have results in this category.

Analysis: RepoRag's superior performance in API invocation completion tasks highlights its strength in recognizing and correctly generating API calls. This can be attributed to its likely extensive training on various API patterns and its ability to understand the context in which specific API calls are made. This capability is crucial for modern software development, where API integrations are common, and accurately predicting these invocations significantly enhances developer productivity.

4.4 Pass Rate Against Unit Tests

Results:

- **RepoRag** showed strong performance across different repositories, achieving the highest pass rates in several cases, such as imagen (70.11), lightmmm (68.46), and redframes (65.50).
- **RepoLevel** also demonstrated high pass rates, with notable performances in repositories like tracr (67.80) and inspection (65.78).

Analysis: RepoRag's ability to generate code that passes unit tests across various repositories indicates its effectiveness in producing functionally correct and reliable code. The high pass rates suggest that RepoRag is not only generating

syntactically correct code but also ensuring that the code adheres to the functional requirements and passes real-world validation through unit tests. This robustness is essential for practical applications, where the correctness of the generated code is paramount.

4.5 Overall Conclusion

RepoRag exhibits a mixed performance across different benchmarks:

- **Strengths**: It excels in Normalized Edit Distance for line completions, indicating a strong ability to generate contextually appropriate code. It also demonstrates superior performance in API invocation completion, which is critical for modern software development involving numerous API integrations. Additionally, RepoRag's high pass rates against unit tests across various repositories highlight its reliability and correctness.
- **Weaknesses**: RepoRag falls short in function completion tasks, where it struggles to maintain logical flow and structure, resulting in lower Exact Match and NED scores compared to RepoCoder. This indicates a potential area for improvement, possibly through enhanced training on more complex function-level tasks and better handling of function semantics.

4.6 Future Work

To further enhance RepoRag's performance, future work could focus on:

- **Improving Function Completions**: Developing more sophisticated techniques to understand and generate function-level code, possibly through more extensive training data and advanced modeling approaches.
- **Enhanced Context Understanding**: Incorporating more context from surrounding code to improve the accuracy of line and function completions.
- **Robust API Prediction**: Continuing to refine the model's ability to predict API calls accurately, leveraging more diverse datasets and scenarios.

By addressing these areas, RepoRag can become an even more powerful tool for code generation, significantly aiding developers in various programming tasks.

References

[1] M. Chen et al. Inferfix: End-to-end program repair with llms over retrieval-augmented prompts. *Proceedings of the ACM Conference on Programming Language Design and Implementation*, X:11–20, 2023.

[2] Douglas Schonholtz. A review of repository level prompting for llms. *arXiv preprint arXiv:2312.10101v1*, 2023.

[3] D. Shrivastava, H. Larochelle, and D. Tarlow. Repository-level prompt generation for large language models of code. *arXiv preprint arXiv:2312.10101v1*, 2023.

CHAPTER A

More research

CHAPTER B

Even more research

RepoRAG__Repository_level_code_debugging_using_LLM.pdf

ORIGINALITY REPORT

6%	5%	1%	3%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	arxiv.org Internet Source	2%
2	Submitted to Dhirubhai Ambani Institute of Information and Communication Student Paper	2%
3	Nazia Bibi, Ayesha Maqbool, Tauseef Rana. "Enhancing source code retrieval with joint Bi-LSTM-GNN architecture: A comparative study with ChatGPT-LLM", Journal of King Saud University - Computer and Information Sciences, 2023 Publication	1%
4	Submitted to Liverpool John Moores University Student Paper	<1%
5	ml4code.github.io Internet Source	<1%
6	Submitted to University of the Aegean Student Paper	<1%
7	fastercapital.com Internet Source	

		<1%
8	Submitted to The University of Manchester Student Paper	<1%
9	lists.openembedded.org Internet Source	<1%
10	export.arxiv.org Internet Source	<1%
11	ir.juit.ac.in:8080 Internet Source	<1%
12	rosap.ntl.bts.gov Internet Source	<1%
13	files.library.northwestern.edu Internet Source	<1%
14	Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, Tao Xie. "CoderEval: A Benchmark of Pragmatic Code Generation with Generative Pre-trained Models", Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, 2024 Publication	<1%

Exclude quotes	Off	Exclude matches	< 8 words
Exclude bibliography	On		