UC Irvine

UC Irvine Electronic Theses and Dissertations

Title

Toward More Reliable Large Language Models for Code-Related Tasks

Permalink

https://escholarship.org/uc/item/517224k1

ISBN

9798173737748

Author

Chen, QiHong

Publication Date

2026-11-04

Copyright Information

This work is made available under the terms of a Creative Commons Attribution-NonCommercial-NoDerivatives License, available at https://creativecommons.org/licenses/by-nc-nd/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

Toward More Reliable Large Language Models for Code-Related Tasks

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Software Engineering

by

QiHong Chen

Dissertation Committee:
Iftekhar Ahmed, Chair
Ian Harris
Thomas Zimmermann

2026

© 2026 QiHong Chen

DEDICATION

I dedicate this dissertation to my Ph.D. advisor, Iftekhar Ahmed. I met Professor Ahmed in 2020, right after my graduation. His guidance has shaped my growth as a researcher ever since. Through this experience, he taught me how to conduct research and guided me in discovering my research interests. Presenting and writing papers have always been difficult for me, as I have struggled to put myself in a reader's shoes. With Professor Ahmed's patience and guidance, I have slowly grown into this skill. I am deeply grateful for his mentorship. Being his student has been the greatest pleasure of my academic journey.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **vii**

ACKNOWLEDGMENTS **viii**

VITA **x**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**
1.1 Dissertation Structure 6

2 From Bias To Improved Prompts: A Case Study of Bias Mitigation of Clone Detection Models **7**
2.1 Introduction 7
2.2 Background & Related Works 11
2.2.1 The Evolution of Code Clone Detection 11
2.2.2 Large Language Models Use Cases 12
2.2.3 Large Language Models Limitations 13
2.3 Experimental Setup 13
2.3.1 Model Selection 14
2.3.2 Prompt Bias Mistake Categories Generation 17
2.3.3 Calculating Prevalence of Prompt Bias Mistake Category: 19
2.3.4 Leveraging Prompt Bias Lessons to Improve Model Performance . . . 22
2.3.5 Method Evaluation 23
2.4 Results 24
2.4.1 RQ1: In the context of within-language and cross-language clone detection tasks, what are the prompt bias mistake categories? 25
2.4.2 RQ2: How prevalent are the mistake categories and what is the most frequent category? 30
2.4.3 RQ3: How can we harness these categories to improve the model performance? 32
2.5 Discussion and Implications 34

2.6 Threats to Validity . . . 36
2.7 Conclusion . . . 37
2.8 Data Availability . . . 38

3 When LLMs Go Wrong: A Study of Code Generation Mistakes 39
3.1 Introduction . . . 39
3.2 Related Works . . . 42
3.2.1 LLMs for Code Generation . . . 42
3.2.2 Mistakes in LLM Generated Code . . . 42
3.2.3 Why LLMs Make Mistakes in Code Generation . . . 43
3.3 Experimental Setup . . . 44
3.4 RQ1:What mistakes do LLMs make in the code generation? . . . 46
3.4.1 Methodology to Answer RQ1 . . . 46
3.4.2 Answer to RQ1: Non-Syntactic Mistake Analysis . . . 49
3.4.3 Mistake Severity Categorization . . . 54
3.4.4 Analysis of Mistake Categories and Severity . . . 55
3.5 RQ2: What are the underlying reasons for the LLM's mistakes in the code generation task? . . . 56
3.5.1 Methodology to Answer RQ2 . . . 56
3.5.2 Answer to RQ2: Reasons of Non-syntactic Mistakes . . . 61
3.6 RQ3: How effectively can LLMs identify the mistakes and the underlying reasons in their generated code? . . . 64
3.6.1 Methodology to Answer RQ3 . . . 64
3.6.2 Answer to RQ3: Performance of Mistake and Reason Identification . 67
3.7 Threats to Validity . . . 68
3.8 Conclusion . . . 69

4 ProjAgent: Procedural Similarity Retrieval for Repository-Level Code Generation 70
4.1 Introduction . . . 70
4.2 Related Work . . . 74
4.3 Background . . . 76
4.4 ProjAgent Pipeline . . . 78
4.4.1 Offline Preprocessing . . . 80
4.4.2 Procedural Context Retrieval . . . 84
4.4.3 Semantic Retrieval . . . 87
4.4.4 Code Generation With Feedback . . . 88
4.5 Experimental Setup . . . 90
4.5.1 Benchmark and Metric . . . 90
4.5.2 Baselines . . . 90
4.5.3 Projection Effectiveness Setup . . . 90
4.5.4 Ablation Setup for RQ3 . . . 92
4.6 Result . . . 93
4.7 Threat To Validity . . . 97
4.8 Conclusion . . . 98

5 Conclusion **100**

Bibliography **103**

LIST OF FIGURES

Page

2.1 Overall Approach 14
2.2 Default Prompt 17
2.3 Confidence Extraction Prompt 18
2.4 Rational Extraction Prompt 19
2.5 Example of Operational Ambiguities category 26
2.6 Example of Confusion Over Varied Approaches category 28
2.7 Best Performing Prompt 33

3.1 LLM code-generation mistake examples: IUE, IMKE, ARHE, IDS 50
3.2 LLM code-generation mistake examples: IRE, CME 50
3.3 Example reasons for LLM code-generation mistakes. 61

4.1 Overview of the ProjAgent pipeline. The offline preprocessing stage decomposes the functions in the repository and collects the projection. The procedural and semantic retrieval stages retrieve procedural and semantic contexts. The code generation stage generates code using the retrieved contexts and static analysis feedback. 79

LIST OF TABLES

Page

2.1 Details of Datasets 15
2.2 Models Selection Result 25
2.3 Confidence Level Of PaLM Models 26
2.4 Frequency of Each Category 31
2.5 Overview Of Prompt Bias Lessons 31
2.6 Prediction Shifts for PoolC and Avatar: Default Prompt vs. With All Lessons Prompt 34
2.7 The Performances of PaLM Model With Prompt Bias Lesson on Avatar Dataset 34
2.8 The Performances of PaLM Model With Prompt Bias Lesson On PoolC Dataset 35

3.1 Mistake Categories, Mistake Types and Severity Frequency (highlighted types are newly identified) 49
3.2 GPT-4 Reason Identification F1 Score 67

4.1 Tools available to the ProjAgent agentic seed finder (See 4.4.2). 86
4.2 Pass@1 (%) on REPOCOD for all methods using Qwen2.5-Coder-14B-Instruct. 94
4.3 Precision, recall, and F1 score of the promoted and leftover groups across six threshold configurations. Config B (bolded) achieves the highest Promoted-F1, indicating the best precision-recall balance. 94
4.4 Ablation on Different Components of ProjAgent 96

ACKNOWLEDGMENTS

When I first started my academic journey at UC Irvine as an undergraduate, my life plan was to graduate and work as a software development engineer. I had no research experience and did not expect to pursue research at all. Looking at this from where I stood five years ago, it would have been hard to imagine that I could one day earn a Ph.D. degree. As I reflect on the past five years, it is hard to believe that I persisted through difficulties such as paper rejections, self-doubt, health issues, and family issues. I am grateful to my younger self for not giving up and for continuing to fight for a better future. In addition, I want to thank everyone who accompanied me on this journey, and especially my mom, whose support made the completion of this Ph.D. dissertation possible.

I would first like to express my gratitude to my advisor, Professor Iftekhar Ahmed. I would not have entered the field of software engineering research if I had not met him in 2020. Over the past six years, he has been my "academic father" and role model. His guidance, patience, and support have become invaluable, helping me develop essential skills and the ability to conduct independent research. Meeting Professor Ahmed and having him as my advisor has been the most fortunate event in my life so far.

Next, I would like to express my gratitude to my mom, JingFen Chen. Without her, I would not have the chance to come to the US to pursue my educational dream. She has supported me unconditionally, both emotionally and financially, throughout every stage of my life. Her sacrifices gave me the opportunity to study abroad and build a life I could not have built on my own. Even though she cannot speak English and does not know my research, her belief in me has carried me through the life barriers I encountered and whenever I doubt myself. Beyond the academic difficulties, she has been my strongest support whenever I suffer from gout. As a chronic condition that causes extreme, recurring pain, gout frequently makes me unable to focus on my work and requires patient care. Through all this, my mom has always been the one taking care of me, and without her, I would not have been able to make it to this point.

I also appreciate the members of my dissertation committee, Ian Harris and Thomas Zimmermann, for their insightful feedback and constructive criticism, which helped refine this work.

A number of my high school and college friends have supported me in this journey. I thank my friend Peng Peng for being by my side through the cultural shock when I first came to the United States. I thank my college friend JieCheng Deng for his continued support throughout both my undergraduate and graduate years. We first got to know each other while taking classes together as undergraduates at UCI, and later collaborated on research during graduate school. I appreciate my internship manager, Qiang Zhou, for his invaluable guidance on industry practices and for helping me understand the differences between industry and academic research. I appreciate Md Rakib Hossain for discussing research with me and teaching me various static code analysis techniques during the first year of my Ph.D. life.

I also want to express my gratitude to the members of my research group at the STAIRS lab for the publications that we have collaborated on: Jiri Gesi, Jiawei Li, and Jina Chun. I would also like to thank everyone in our lab for the countless hours we spent together discussing research ideas.

Finally, I would like to acknowledge the financial support from the UC Irvine Informatics department, which has enabled me to pursue my doctoral studies.

VITA

QiHong Chen

EDUCATION

Doctor of Philosophy in Software Engineering **2021-2026**
University of California, Irvine *Irvine, California*

Master of Software Engineering **2021-2024**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computational Sciences **2017-2019**
University of California, Irvine *Irvine, California*

RESEARCH EXPERIENCE

Graduate Research Assistant **2021–2026**
University of California, Irvine *Irvine, California*

Applied Scientist Internship **2025**
Amazon Science *Santa Clara, California*

Applied Scientist Internship **2024**
Amazon Science *Seattle, Washington*

TEACHING EXPERIENCE

Teaching Assistant **2025–2026**
University of California, Irvine *Irvine, California*

Teaching Assistant **2021–2022**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

A Deep Dive into Large Language Model Code Generation Mistakes: What and Why? **2026**
(Under Review) Information and Software Technology (IST)

ProjAgent: Repository Level Code Generation Using Procedurally Similar Contexts **2026**
(Under Review) ACM Transactions on Software Engineering and Methodology (TOSEM)

Different, really! A comparison of highly-configurable systems and single systems **2022**
Information and Software Technology (IST)

Evaluating and improving static analysis tools via differential mutation analysis **2021**
IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)

REFEREED CONFERENCE PUBLICATIONS

The smelly eight: An empirical study on the prevalence of code smells in quantum computing **2023**
2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)

Leveraging feature bias for scalable misprediction explanation of machine learning models **2023**
2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)

REFEREED Other PUBLICATIONS

Is multi-agent debate (mad) the silver bullet? an empirical analysis of mad in code summarization and translation **2025**
arXiv

Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks **2025**
arXiv

Does the Order of Fine-tuning Matter and Why? **2024**
arXiv

From Bias To Improved Prompts: A Case Study of Bias Mitigation of Clone Detection Models **2024**
arXiv

ABSTRACT OF THE DISSERTATION

Toward More Reliable Large Language Models for Code-Related Tasks

By

QiHong Chen

Doctor of Philosophy in Software Engineering

University of California, Irvine, 2026

Iftekhar Ahmed, Chair

Large language models (LLMs) have demonstrated strong capabilities on a wide range of code-related tasks. When performing such a task, an LLM must first understand the given input, then produce an output based on that understanding. Depending on the task, this output can take the form of a label or free-form text such as source code. For example, in clone detection, the LLM takes two code snippets as input and produces a label indicating whether they are semantically equivalent. To give a correct label, the LLM must understand two code snippets and have the correct understanding of what is semantically equivalent. In code generation, the LLM must understand the natural-language specification and generate source code that fulfills those requirements. Despite their strong performance, LLMs can make mistakes at either stage (understanding and generation), which can undermine users' trust. Therefore, understanding and mitigating LLMs' mistakes at both stages is key to improving their performance on code-related tasks.

Among various code understanding tasks, we select clone detection because it is widely used to support software maintenance activities, including plagiarism detection, refactoring, and bug identification. The first work aims to understand LLMs' mistakes in clone detection and improve their performance. Existing work has mainly evaluated LLMs' performance on clone detection but has not systematically investigated the underlying reasons for their

misclassifications. To investigate this, we manually analyzed the rationale for each LLM's misclassification and found that LLMs' understanding of clone code differs from that of human annotators. For example, we found that LLMs often fail to recognize Type-4 clones, code fragments that are functionally equivalent but syntactically dissimilar, as clones, revealing a narrower notion of clone code than the one intended by human annotators. In total, we identified eight distinct mistakes LLMs made during clone detection. For each mistake, we derived corresponding natural-language guidance that can be inserted into the prompt to mitigate LLMs' errors. We evaluated the effectiveness of the proposed guidance and found that it can improve the LLM's performance on clone detection, achieving up to a 10.81% improvement in F1 score.

As the first work focuses on the LLM's mistakes in the code understanding stage, the second work focuses on the mistakes during the generation stage. Among various code-related generation tasks, we selected code generation due to its popularity and importance, as writing source code is one of the most fundamental activities in software engineering. Despite LLMs' strong performance on code generation, they still produce incorrect code. Thus, the second work aims to understand the mistakes in the generated code. Prior work analyzing mistakes in LLM-generated code has largely relied on earlier, less capable models and small-scale, Python-only datasets, thereby restricting the scope of the identified mistakes. Furthermore, prior work does not investigate the reasons for those mistakes. To fill these gaps, we analyze code generated by two state-of-the-art LLMs across two large-scale, multilingual benchmarks comprising 2,268 standalone coding questions in Python and Java. Through extensive manual analysis, we identified 17 types of non-syntactic mistakes, organized into seven high-level categories, ten of which were overlooked by prior literature. We further identified six underlying reasons for these mistakes. Finally, we assessed LLMs' ability to automatically identify mistakes in generated code and the underlying reasons for these mistakes. Results indicate that top-performing models like GPT-4 achieve high precision of 0.95 in detecting mistakes and an F1 score of up to 0.78 in identifying the underlying reasons. These findings enriched

the taxonomy of mistakes in LLM-generated code, providing developers with practical guidance for using LLMs more effectively and a deeper understanding of the reasons behind these mistakes.

While the second work focuses on mistakes in generated code and their underlying reasons, it does not propose a way to improve LLM performance. The third work aims to do so. Instead of focusing on standalone code generation, the third work focuses on repository-level code generation, as it better reflects real-world software development, where new code is written within the context of an existing codebase. Learning from our second work, context plays a central role in successful code generation, especially for repository-level code generation, as the generated code may call existing functions in the repository or be called by them. Therefore, the third work aims to improve repository-level code generation by identifying better contexts. Existing work often retrieves contexts based on lexical or semantic similarity to the function to be completed, overlooking functions that share similar logic. We refer to these overlooked contexts as procedurally similar contexts. To fill this gap, we propose ProjAgent, a system that retrieves procedurally similar contexts through agentic search over the codebase. These contexts serve as the necessary logic to complete the target function. Furthermore, ProjAgent retrieves semantically similar contexts from the codebase that serve as the ingredients for implementing the logic. Our evaluation shows that ProjAgent improves code generation performance by up to 41.14% over baseline approaches.

In summary, the contributions in this dissertation advance the understanding of LLMs' mistakes in code-related tasks and their underlying reasons, and propose methods to improve LLM performance on these tasks.

Chapter 1

Introduction

Source code is the primary artifact developers create, maintain, and evolve over a software system's lifetime [106]. In recent years, large language models (LLMs) have demonstrated strong capabilities on a wide range of code-related tasks, reshaping how software is written and reviewed, and have been increasingly adopted in both research and industry to support software development [138]. These tasks require the LLM to interpret the input and produce an output based on that understanding. For instance, clone detection is a task centered on the understanding stage [154]. To correctly identify the clone code, the LLM needs to correctly understand the given code snippet pair and have a correct definition of clone code. Code generation, in contrast, is a task in which LLMs must produce source code that meets the requirements [32]. Despite their strong performance [73, 45], LLMs remain prone to mistakes, including hallucinating nonexistent APIs and producing plausible-looking but incorrect code [163, 82], which undermine users' trust. Therefore, understanding LLMs' mistakes at both understanding and generation stages and mitigating them is essential.

To understand LLMs' mistakes at the understanding stage, we selected clone detection, as it plays a central role in supporting software maintenance activities [154, 108, 183]. While

prior work has mainly evaluated LLMs' performance on clone detection [42], it has not systematically investigated the mistakes and reasons behind LLMs' misclassifications, nor designed mitigation strategies to improve their performance in clone detection. We took the first step to fill this gap by understanding why LLMs misclassify code pairs during clone detection and using this understanding to improve their performance.

To understand the reason for LLMs' misclassification, we prompt the LLM to provide a rationale when generating the label for the given pair of code snippets. We then manually analyzed the rationales for each misclassification. Our analysis reveals that LLMs' mistakes in clone detection are mainly due to a mismatch between the LLM's understanding of what constitutes a clone and the definition intended by human annotators. For example, we found that LLMs often fail to recognize Type-4 clones, code fragments that are functionally equivalent but syntactically dissimilar [153], as clones at all, revealing a different definition of clone code than the one intended by human annotators. In total, we identified eight distinct mistakes LLMs make during clone detection, spanning issues such as variable initialization and naming ambiguity and overemphasis on textual similarity. For each mistake, we derived a corresponding natural-language guidance that can be inserted into the prompt to correct the underlying misunderstanding. We evaluated the effectiveness of the proposed guidance and found that it improves LLM performance on clone detection, achieving up to a 10.81% improvement in F1 score.

As the first work focuses on LLMs' mistakes in the understanding stage, the second work turns to LLMs' mistakes in the generation stage. Among the various code-related generation tasks, we selected code generation due to its popularity [73, 45]. For instance, over 25% of Google's code is now generated by AI [52], and GitHub Copilot has surpassed 20 million all-time users [64]. However, LLMs still produce plausible-looking but incorrect code [118]. Therefore, understanding these mistakes and their underlying reasons becomes essential. Although prior work has also analyzed mistakes in LLM-generated code, it has mostly focused

on code produced by earlier, less capable models and small-scale, Python-only benchmarks [160, 170, 86]. These settings potentially limit the diversity and comprehensiveness of the identified mistakes. To make matters worse, as LLMs' abilities evolve, LLMs' mistakes evolve too. Therefore, it is critical to keep the taxonomy of mistakes up to date. Furthermore, prior work has not investigated the underlying reasons behind these mistakes.

To fill these gaps, we analyze code generated by two state-of-the-art LLMs (Qwen2.5-coder-14b-instruct [79] and GPT-4 [8]) across two large-scale, multilingual benchmarks (HumanEval-X [203] and MBXP [18]), comprising 2,268 standalone coding questions in Python and Java. Through extensive manual analysis, we identify 17 types of non-syntactic mistakes, organized into seven high-level mistake categories. Among the 17 mistakes, 10 had not previously been reported in the literature. We further identify six underlying reasons for these mistakes, such as LLMs' sensitivity to the wording of the coding question specification. Finally, we assess LLMs' ability to automatically identify mistakes in generated code and the underlying reasons behind them. Our results show that top-performing models such as GPT-4 achieve a high precision of 0.95 in detecting mistakes and an F1 score of up to 0.78 in identifying the underlying reasons behind them. These findings enrich the taxonomy of mistakes in LLM-generated code, providing developers with practical guidance for using LLMs more effectively and a deeper understanding of why these mistakes occur.

While the second work characterizes LLMs' mistakes in code generation and their underlying reasons, it does not propose a way to improve LLM performance in code generation. The third study focuses on repository-level code generation. Compared to standalone function generation, repository-level code generation better reflects real-world software development, where generated code needs to call existing functions in the repository and may be called by them. As noted in prior work [97, 140], context retrieved from the repository plays a central role in successful code generation. The third study therefore aims to improve repository-level code generation by identifying better sources of context. Existing approaches

typically retrieve context based on lexical or semantic similarity to the target function (the function to be completed) [194, 144, 198]. However, they are limited to functions within modules or packages that share the same identifiers. Inspired by how human developers write code within a repository, we posited that successful repository-level code generation requires the code logic and the ingredients, such as existing functions, classes, or variables, available to implement that logic [155]. However, existing retrieval signals based on lexical or semantic similarity primarily surface the latter, leaving the reasoning-level guidance largely uncaptured. We refer to functions that share similar logic with the target function as the procedurally similar contexts.

To address this gap, we adopt the projection method proposed by Hu et al. [75]. This method captures an LLM's reasoning by projecting its hidden states into a low-dimensional reasoning subspace via singular value decomposition (SVD) [58]. We then use projection similarity within this subspace as a filter to efficiently remove procedurally dissimilar context. Building on these findings, we introduce ProjAgent, a system that retrieves procedurally similar context through agentic search over the codebase and expands the identified context using projection similarity as an efficient filter. While procedurally similar context conveys the logic needed to implement the target function, we further retrieve semantically similar context from the codebase to serve as the ingredients for implementing that logic. Our evaluation shows that ProjAgent improves code generation performance by up to 41.14% over baseline approaches.

The main contributions of this dissertation are:

- **Understanding LLM Mistakes in Clone Detection:** We took the first step by manually analyzing LLM misclassifications in clone detection and found that LLMs' understanding of clone code differs from the definition intended by human annotators. We identify eight distinct mistakes underlying this mismatch.

- **Guidance for Improving LLM-Based Clone Detection:** For each of the eight identified mistakes, we derive a corresponding natural-language guidance that can be inserted into the prompt to correct the underlying misunderstanding, improving LLM performance on clone detection by up to 10.81% in F1 score.

- **A Comprehensive Taxonomy of LLM Code Generation Mistakes:** We analyze code generated by two state-of-the-art LLMs across two large-scale, multilingual benchmarks and identify 17 types of non-syntactic mistakes across seven high-level mistake categories, ten of which were previously unidentified in the literature.

- **Underlying Reasons Behind LLM Code Generation Mistakes:** We identify six underlying reasons behind these 17 mistakes and empirically validate each. We further show that top-performing LLMs, such as GPT-4, can identify the mistakes in the generated code and the reasons behind them with high precision and F1 score, respectively.

- **Procedurally Similar Context:** We identify a new source of context for repository-level code generation, which we call procedurally similar context. Those context functions share similar reasoning steps or logic with the target function, which is overlooked by existing lexical- and semantic-similarity-based retrieval approaches.

- **ProjAgent:** We propose ProjAgent, a system that retrieves procedurally similar context, conveying the logic needed to implement the target function, along with semantically similar context, providing the symbols and ingredients needed to implement that logic, and combines both to instruct the LLM during code generation.

1.1 Dissertation Structure

The rest of this dissertation is organized as follows. In Chapter 2, I present the study that investigates LLMs' mistakes in clone detection and proposes natural-language guidance to mitigate them. This chapter is based on the paper "From Bias to Improved Prompts: A Case Study of Bias Mitigation of Clone Detection Models." In Chapter 3, I present an empirical study that characterizes the mistakes LLMs make during standalone code generation and the underlying reasons behind them. This chapter is based on the paper "When LLMs Go Wrong: A Study of Code Generation Mistake" In Chapter 4, I introduce ProjAgent, a system that improves repository-level code generation by retrieving procedurally and semantically similar context. This chapter is based on the paper "ProjAgent: Repository Level Code Generation Using Procedurally Similar Contexts." Finally, Chapter 5 concludes the dissertation with implications for researchers and practitioners.

Chapter 2

From Bias To Improved Prompts: A Case Study of Bias Mitigation of Clone Detection Models

2.1 Introduction

Code clones are source code fragments that exhibit either syntactic or semantic similarity [98, 21, 43, 94]. They are prevalent in traditional software development practices, often arising from the practice of copying and pasting source code snippets found on online platforms like Stack Overflow [149, 195] and duplicated repositories on platform such as GitHub [123]. However, clone codes have the potential to detrimentally affect the quality of traditional software, leading to issues in software maintenance and debugging [196]. Moreover, from a software system security perspective, clone codes could lead to vulnerability propagation if a vulnerable code fragment is cloned [108]. Consequently, clone detection has emerged as a crucial field in software engineering, tasked with predicting whether two code snippets are indeed clone codes. This area of research has gained significant traction due

to its practical applications, including code reusability enhancement, aspect mining, plagiarism detection, copyright infringement investigation, code optimization, software evolution analysis, code quality assessment, bug detection, virus detection, and improvement of code maintainability [154].

In the realm of traditional software, clone codes have been a prevalent occurrence. However, the advent of cutting-edge large language models (LLMs), exemplified by ChatGPT, has significantly amplified the prevalence of clone codes in both traditional and machine-learning-based software. This amplification can be attributed to LLMs' exceptional capabilities in source code generation and human-language comprehension. These remarkable features have captivated users across a spectrum of coding proficiency levels, empowering them to generate source code from natural language requirements, incomplete code fragments, and even code written in other programming languages [122, 91, 117]. Consequently, this exacerbates the issue of clone codes because LLMs are bound by the source code they have encountered during their training phase. When users prompt LLMs with similar requirements, they generate similar source code, leading to a proliferation of similar code instances. Moreover, as reported by Liu et al. [122], the quality of LLMs-generated source code heavily relies on the difficulty of the requirement, program size and the programming language. Thus, when these LLM-generated source codes are incorporated into machine-learning-based software, its quality can be significantly compromised. Furthermore, Allamanis et al. [13] underscore the impact of clone code on machine learning models. They highlight that, in certain scenarios, a machine learning model's performance can experience inflation of up to 100% when evaluated on datasets containing duplicated code, as opposed to its performance on datasets where duplicated code has been eliminated. Therefore, with the ascendancy of LLMs, the importance of clone detection has been elevated to a paramount position in software development domain.

Over the years, researchers have devised a multitude of methodologies, encompassing text-

based, lexical-based, tree-based, metric-based, semantic-based, hybrid-based, and machine learning-based approaches for clone detection [154, 156, 10]. Among these approaches, the machine learning paradigm exhibits a particularly strong alignment with clone detection task due to its heightened emphasis on comprehending the nuanced meaning embedded within source code. Numerous machine-learning approaches have been proposed for clone detection. For instance, Fang et al.[47] employed a deep neural network (DNN) to identify function clones, capturing syntax and semantic information through abstract syntax trees (ASTs) and control flow graphs (CFGs). Wang et al. [178] proposed an approach combining graph neural networks and control/data flow information that outperformed state-of-the-art techniques using the AST subtree-based approach for clone detection. An essential aspect of clone code detection is the ability to comprehend the source code, and research has shown that Large Language Models (LLMs) have robust code comprehension capabilities. This makes LLMs a perfect candidate for clone detection. However, current research has not investigated the benefit of using LLms for clone detection and we aim to fill this gap in this research.

While LLMs possess remarkable source comprehension abilities, they suffer from issues like prompt bias [208] and hallucination. Prior research has highlighted that LLMs' outputs fluctuate substantially when prompts are slightly modified. Zhou et al. [208] introduced an approach to uncover improved prompts by incorporating descriptive context alongside desired input/output, thereby instructing the GPT model to generate the desired prompt. However, this method has limitations as it does not address the root cause of the output variations. We assert that the fundamental source of prompt bias can be traced back to the errors made by the model. By comprehending these errors, we can refine the prompt to optimize the performance of LLMs. Consequently, we seek to bridge this gap by applying LLMs to clone detection using an innovative approach to mitigate the prompt bias inherent in LLMs.

In this study, our objective is to employ LLMs for both within-language and cross-language clone detection tasks. We have made the deliberate choice not to confine our investigation solely to within-language clone detection. This choice is motivated by the recognition that cross-language clone codes represent a particularly challenging category, being semantically similar yet textually distinct. We contend that clone codes of this category hold greater relevance in real-world scenarios. Furthermore, by demonstrating the effective capability of LLMs in identifying both within and cross-language clone codes, we aim to underscore the pivotal role of LLMs in shaping future trends in clone detection research. To achieve this goal, we conduct experiments to answer the following research questions:

- RQ1: In the context of within-language and cross-language clone detection tasks, what are the prompt bias mistake categories?
- RQ2: How prevalent are the mistake categories and what is the most frequent category?
- RQ3: How can we harness these categories to improve the model performance?

Specifically, this paper makes the following contributions:

1. This study represents the first exploration of LLMs within the domain of clone detection.
2. This is the first study that marks the pioneering effort to delve into the prompt bias of LLMs from the model's error perspective.
3. We introduce a comprehensive framework meticulously crafted to augment the performance of GPT models in both within-language and cross-language clone detection tasks.

2.2 Background & Related Works

In this section, we commence by offering an overview of the historical evolution of research on clone detection.

2.2.1 The Evolution of Code Clone Detection

Clone code is a pair of similar source codes that are either identical or appear like one another within a software system. Roy et al. [153] outline the fundamental types of clone code as following:

Exact Clones (Type 1): Identifical code segments except for changes in comments, layouts and whitespaces.

Renamed Clones (Type 2): Code segments which are syntactically or structurally similar other than changes in comments, identifiers, types, literals, and layouts. These clones are also called parameterized clones.

Near Miss Clones (Type 3): Copied pieces with further modication such as addition or removal of statements and changes in whitespaces, identifiers, layouts, comments, and types but outcomes are similar. These clones are also known as gapped clones.

Semantic Clones (Type 4): More than one code segments that are functionally similar but implemented by different syntactic variants.

In the realm of clone detection research, various methodologies have been devised, encompassing text-based, lexical-based, tree-based, metric-based, semantic-based, hybrid-based, and machine learning-based approaches. However, it is widely recognized among researchers that a mere focus on syntactic similarity falls short of capturing the essence of code functionality and the crux of effective clone detection lies in comprehending the functional aspects of source code. An innovative breakthrough in this pursuit was introduced by Yu et al. [191], who employed a semantics learning method utilizing Control Flow Graphs (CFG) and Program Dependency Graphs (PDG) to delve into deeper semantic intricacies. Subsequent

research endeavors expanded upon this approach by creating additional source code representations. For instance, CCGraph [167] harnessed the power of Program Dependency Graphs (PDGs) in conjunction with near graph-matching algorithms to detect code clones. DeepSim [202] took this a step further by encoding control and data flow as semantic matrices, offering a more comprehensive representation of code semantics. These machine learning-based hybrid approaches outperformed approaches that relied on techniques using the AST subtree-based approach for clone detection. Despite the divergence in these approaches regarding the representations they employ, their common objective remains consistent: aiding BERT models in better grasping source code semantics.

2.2.2 Large Language Models Use Cases

In contrast to BERT models, LLMs such as ChatGPT, whose behaviors are guided by prompts, exhibit unparalleled capabilities across a range of software engineering research areas. These include machine translation [39, 71, 148], content summarization [148, 49, 59], sentiment analysis [188, 199], and query resolution [148]. LLMs have acquired these exceptional skills through extensive training on vast textual datasets and innovative training methodologies, such as reinforcement learning from human feedback (RLHF) [137]. These approaches enable LLMs to grasp intricate linguistic patterns and structures. Given their capabilities, prior research has primarily focused on integrating LLMs into various domains, including conversational agents like ChatGPT [1] and AI-assisted drawing, exemplified by Midjourney [3]. Remarkably, the source code comprehension ability of LLMs aligns with the insights of clone detection researchers, although there has been limited exploration into the suitability of LLMs for clone detection tasks. In our study, we address this gap and empirically assess the performance of LLMs in the context of clone detection.

2.2.3 Large Language Models Limitations

While LLMs have demonstrated impressive capabilities across various domains, they are not without their limitations. Previous research has highlighted certain shortcomings inherent to machine learning models, such as dataset biases, encompassing issues like class imbalance and feature bias [56, 9, 24, 28, 29, 69, 171]. Additionally, LLMs are prone to a phenomenon known as "hallucination," where they generate text that seems logically coherent and contextually relevant but may contain factual inaccuracies or entirely fabricated information [12]. To address this hallucination issue, various techniques have been developed, such as self-consistency [179] and chain of thought [184], which aim to guide GPT models in producing reliable outputs by fostering internally consistent reasoning. Another significant challenge faced by LLMs is "prompt bias," which occurs when slight modifications to the input prompt result in substantially different model outputs. Zhou et al. [208] attempted to tackle prompt bias by introducing an approach that identifies improved prompts. They achieved this by providing descriptive context alongside the desired input/output, encouraging the GPT model to generate the desired prompt. However, this approach primarily focuses on mitigating prompt bias by generating better prompts from the model's perspective. However, their approach does not delve into the underlying causes. In our study, we aim to bridge this gap by conducting an in-depth analysis of the factors driving LLM behavior and using the gained insights to address prompt bias effectively in the domain of clone detention, which, to the best of our knowledge, is the first of its kind.

2.3 Experimental Setup

In this section, we describe our process to answer each research question(RQs): (A) We choose the model for our later analysis. (B) We generate prompt bias mistake categories based on prompted model mistakes. (C) We calculate the prevalence of each prompt bias mistake category. (D) We propose a method to improve model's performances using the

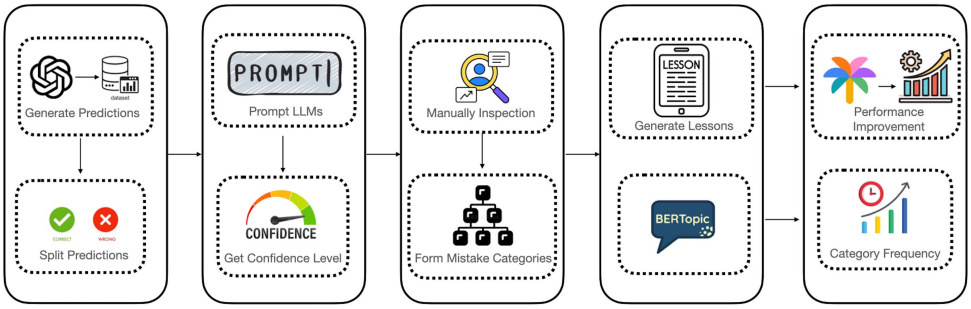


Figure 2.1: Overall Approach

prompt bias lessons. (E) We evaluate the proposed method. An outline of our methodology is presented in Figure 2.1.

2.3.1 Model Selection

In this section, we outline our model selection process, covering dataset specifics, candidate model information, and the procedure for identifying the top-performing model.

Datasets Introduction:

The primary goal of this paper is to investigate LLMs performance on clone detection for both within and cross-language scenarios. Among many available datasets, we picked the poolC dataset [5] for within-language clone detection and created an avatar dataset for cross-language clone detection based on a language translation project named avatar [7].

We decided to use poolC datset [5] since it has been extensively used in clone-related research. We opted not to use the well-known BigCloneBench (BCB) dataset due to issues related to unbalanced data labels and ground truth quality [162, 99]. The avatar dataset originally served as a code translation dataset [7], comprising code snippets with identical functionality but composed in distinct programming languages. We leveraged this dataset and transformed it into a cross-language clone detection dataset. The original avatar dataset comprised three JSONL files, where each line within the file represented a dictionary. The dictionary structure

Table 2.1: Details of Datasets

Dataset	Clone Detection Type	Dataset Size	Sampled Size	Language
Avatar with comment	Cross Language	1.7k	1.7K	Java, Python
Avatar without comment	Cross Language	1.7k	1.7K	Java, Python
PoolC with comment	Within Language	6,710k	16K	Python
PoolC without comment	Within Language	6,710k	16K	Python

was as follows: {"java": Java code snippet, "python": Python code snippet, idx: idx value}. The "idx" value denoted the theme of the code snippets, and Java and Python code snippets were considered to share the same functionality *iff* they possessed the same "idx" value. Given the consistent format across all three JSONL files, we consolidated them into a single JSONL file named "combined.jsonl," containing 9,515 lines. Subsequently, we constructed our avatar dataset from "combined.jsonl," comprising a "data.jsonl" file, a training dataset name "train.txt", a validation dataset "valid.txt", and a testing dataset named "test.txt". A comprehensive overview of these datasets can be found in Table 2.1.

Notably, the code snippets in the poolC and avatar datasets include comments. While many non-machine learning-based approaches tend to discard comments as they are often seen as non-essential to the functionality of source code, we intentionally chose to retain comments. Our reasoning is that the effectiveness of machine learning models relies on their ability to understand source code comprehensively. Comments, which serve as explanations of the code written by developers, have the potential to assist models in better comprehending the code. We created two versions for both the avatar and poolC datasets: one with comments and one without comments.

Candidate Models Introduction:

Among all available deep learning models and LLMs, we picked CodeT5 [181], a BERT model with 220 million parameters, Turbo-3.5 [4] and Text-002 models [4], both GPT models developed by OpenAI with 175 billion parameters, and PaLM model [39], another GPT model by Google boasting a substantial 540 billion parameter size. We included models

from both Bidirectional Encoder Representations from Transformers (BERT) and Generative Pre-trained Transformer (GPT) model families because of the absence of sufficient evidence indicating the superior performance of GPT models over BERT models in the context of clone detection task. Since Google and OpenAI do not offer direct access to their model weights, we interact with these models using prompts. In contrast, codeT5 model had been pre-trained, and we subsequently fine-tuned it on the dataset to maximize the performance.

Selecting Model:

To choose one model for further analysis, we ran every model on every dataset and picked the top-performing model. Given the substantial size of each dataset and the considerable amount of time and budget required to evaluate models on the entire dataset, we opted to establish a benchmark dataset comprising 200 instances. For every dataset, we randomly sampled 200 instances from the validation set. To mitigate any potential adverse effects stemming from class imbalance, we took great care to maintain class balance within the benchmark dataset, ensuring an equal distribution of clone and non-clone instances. Next, we evaluated every model on every crafted benchmark dataset and recorded the model performances in terms of precision, recall, accuracy and F1 score. For CodeT5, we finetuned it on the training set and evaluate it on the test set. For GPT model, we employed the default prompt defined with a temperature setting of 0 to evaluate the model performance on the test set. We configured the temperature to be 0, ensuring that the model's output is deterministic and our results are reproducible. To construct the default prompt, we adopted Zhou et al's [208] approach and formulated our default prompt using GPT-4. Specifically, we feed GPT-4 model with several exemplary input/output instances and request it to generate the default prompt. Note that we deliberately include programming languages of the code snippets in the prompt because we posit that programming language information could assist the model in better understanding the code snippet [122]. The default prompt we used is shown in Figure 2.2. Lastly, we picked the top-performing model in terms of F1 score.

You are a helpful assistant that takes two pieces of code snippets and detects whether these two pieces of code snippets are clone code or not. The first code snippet is written in <language1>, and the second code snippet is written in <language2>. Here are the two code snippets: Code Snippet1: <code1>. Code Snippet2: <code2>. If you think any part of the two code snippets contains clones, answer Yes, otherwise answer No. You must answer either Yes or No. Your Response:[INSERT].

Figure 2.2: Default Prompt

2.3.2 Prompt Bias Mistake Categories Generation

In this section, we describe the procedure for generating prompt bias categories. Specifically, this includes generating model predictions, getting model confidence levels, and generating prompt bias mistake categories.

Generate Model Predictions:

Due to the substantial size of the poolC dataset, we employed a sampling approach [40] using a sampling size calculator [6] with a 99% confidence level and a 1% margin of error to construct the 16k size poolC with comment dataset and 16k size poolC without comment dataset. Next, we ran the selected model on every dataset and recorded the model performance. For BERT model, we finetune it on the training set and, subsequently, apply it to the test set to evaluate its performance. For the GPT model, we employed the default prompt defined in 2.3.1 with a temperature setting of 0 to evaluate the model performance on the test set.

Get Model Confidence Level:

We then divided the model's predictions into correct and incorrect prediction groups. The correct prediction group consists of instances where the prediction is the same as the ground truth label and the incorrect prediction group consists of instances where the prediction is different from the ground truth label. To study the mistakes of the model, we focused on the

incorrect prediction group. To mitigate the negative effect caused by hallucination [105], we first extracted the confidence level of the model regarding its prediction using the prompt shown earlier where we explicitly asked *Tell me how much confidence you have for your answer scale from 0 to 100.* The prompt we used to get confidence level of the model is shown in Figure 2.3.

> *"You are a helpful assistant that takes two pieces of code snippets and detects whether these two pieces of code snippets are clone code or not. The first code snippet is written in <language1>, and the second code snippet is written in <language2>. Here are the two code snippets: Code Snippet1: < code1 >. Code Snippet2: <code2>. If you think any part of the two code snippets contains clones, answer Yes; otherwise answer No. You must answer either Yes or No. Your Response:[INSERT]." I have documented your answer as <prediction>. Tell me how much confidence you have for your answer scale from 0 to 100.*

Figure 2.3: Confidence Extraction Prompt

We employed this confidence level to indicate the reliability of model's responses. Therefore, we retained only those incorrect predictions where the model exhibited 80% confidence in its responses, as these instances represented cases where the model made mistakes with a high degree of confidence. We call them reliable incorrect predictions. We picked 80% as a cutoff point because, as we observed, the model's confidence level is either higher than 80% (from 80% to 100%) indicating highly confident or 0% indicating not confident. Therefore, picking 80% as a cutoff point was a reasonable choice. We provide the statistical detail of selecting threshold value later in section 2.4.2.

Prompt Bias Mistake Categories Generation:

To understand the reasons behind the model's incorrect predictions, we extracted the high-level rationale behind the incorrect predictions. The prompt we used is shown in Figure 2.4. These high-level rationales encompass the model's thought process during the prediction phase. To understand the root cause behind those rationales, we randomly selected 100

examples and reviewed them. To mitigate human bias, two authors independently reviewed the 100 instances together with the code snippets meticulously, with the objective of summarizing the errors made by the model that are specifically related to the misunderstanding of the nature of the clone detection task [166]. Subsequently, the two authors discussed the list of model mistakes that pertained to misinterpretations of the task and reached an inter-rater reliability Cohen's Kappa (k) [65] of 0.85, which indicates almost perfect agreement. These identified model errors were coined as "prompt bias mistake categories."

"You are a helpful assistant that takes two pieces of code snippets and detects whether these two pieces of code snippets are clone code or not. The first code snippet is written in <language1>, and the second code snippet is written in <language2>. Here are the two code snippets: Code Snippet1: <code1>. Code Snippet2: <code2>. If you think any part of the two code snippets contains clones, answer Yes, otherwise answer No. You must answer either Yes or No. Your Response:[INSERT]." Your answer was <model_prediction>. However, the correct answer is <label>. Your answer was wrong, I want you to give me the reason why you made the decision in high level.

Figure 2.4: Rational Extraction Prompt

2.3.3 Calculating Prevalence of Prompt Bias Mistake Category:

As we have defined the prompt bias mistake categories, next we want to understand the prevalence of each category. We adopted a topic modeling tool named BERTopic [60] to calculate the frequency of each category. Our applied procedure was as follows:

1. **Generated Topics:** We adopted the methods mentioned above in subsection 2.3.2 to generate rationales for every incorrect instance of every dataset. Those rationales are then fed into BERTopic as the input data. BERTopic is designed to be modular, meaning each algorithm phase may be customized. As such, ensuring the appropriate configuration at each step is of the utmost importance. The first phase in the BERTopic pipeline is calculating embedding vectors for the input documents. Various embedding models are supported in this step. Thus, our first variable that can signif-

icantly affect the final topics is the embedding model employed. We opted to use the default embedding model which is all-MiniLM-L6-v2.

In the second phase, the dimensionality of the embedding vectors is reduced to counteract the curse of the dimensionality problem in the clustering phase. To handle large amounts of documents from poolC dataset, cuML's UMAP implementation can be used to enable GPU utilization in this phase. Although other dimensionality reduction algorithms could be used, we adopted UMAP as its superiority is discussed in previous work [60, 131]. CuML's UMAP offers a range of tunable parameters, but among them, two variables highlighted by BERTopic have a substantial impact on the algorithm: the **number of neighbors** and the **number of components**. The number of neighbors, which can be set between 2 and 100, plays a crucial role in determining the preservation of local data. Smaller values favor the retention of more localized data. Additionally, the number of components represents the dimensionality of the reduced vectors. Increasing the value of this variable reduces the impact of dimensionality reduction on clustering. In addition to the aforementioned hyperparameters, there is another crucial parameter to consider in cuML UMAP: the **minimum distance** between the reduced vectors. Ranging from 0 to 1, this parameter plays a pivotal role in the balance between global and local structure preservation. When set to smaller values, it leads to a more tightly clustered embedding space, where vectors in close proximity on the manifold are drawn even closer together. We initiated with a conservative minimum value of 5 for both hyperparameters. Subsequently, we incremented these values in steps of 5, gradually increasing them to 20. This enabled the exploration of larger values in cases where we observed an upward trend in the topic model's performance.

In the third phase, reduced vectors are clustered. Similar to previous phases, BERTopic supports customized clustering algorithms. One such algorithm, used as the default in BERTopic, is HDBSCAN [130]. HDBSCAN simplifies the task by preventing the need

to manually select the number of clusters and excelling at automatically clustering dense areas of varying densities. To harness GPU acceleration benefits, we opted for cuML's HDBSCAN implementation.

2. **Seed Topics:** Different variations of topic modeling are supported by BERTopic, such as dynamic, multimodal, hierarchical, online, and semisupervised approaches. Using vanilla BERTopic, the generated topics may be biased toward terms that are not relevant to the prompt bias mistake categories. To address this issue and guide the topic model towards creating topics that align more closely with our areas of interest, we can introduce a weighting mechanism for specific words within the dataset. This approach ensures that the resulting topic model contains topics whose representations are more in line with our criteria. BERTopic introduces a semisupervised variation of topic modeling known as Guided Topic Modeling, which facilitates this process by allowing us to guide the model's topic creation using weighted terms called *seed*, thereby producing topics that better match our intended criteria. Seed is a nested list of terms, where each inner list consists of phrases relevant to one of our desired topics. The prompt bias mistake categories generated as a result of the method described in subsection 2.3.2 were used as the seed topics.

3. **Mapping Seed Topics:** Mapping our predefined seed topics, which represent prompt bias mistake categories, to the topics generated by the model was a crucial step in assessing the relevance and coverage of our predefined categories within the actual topics identified by the model. This process was essential to ensure that the topics discovered by the model closely aligned with the principles we manually analyzed. To achieve this, we processed each prompt bias mistake category (detailed shown in section 2.4.1) into individual words to match BERTopic's output format. Subsequently, we used the `model.find_topic` function to identify the most closely matching generated topic, thus establishing a connection between our predefined categories and the topics generated

by the model.

After this procedure, the BERTopic model yields the count of each category within the generated high-level rationales. We calculated the prevalence of each category:

$$\text{Category_Prevalence} = \frac{category_count}{\sum generated_rationales} \tag{2.1}$$

2.3.4 Leveraging Prompt Bias Lessons to Improve Model Performance

To address prompt bias mistakes made by the model during the prediction phase, we propose a method where the model considers these biases when performing the task. Since we don't have access to the model's weights or knowledge of its training data, our approach centers on the prompt itself. Specifically, we suggest that the model can take prompt biases into account by adding a lesson to the prompt. This lesson instructs the model on what to focus on during the task and is referred to as a "prompt bias lesson." In essence, each lesson serves as a high-level encapsulation of the model's confusion, offering valuable insights into potential solutions for rectifying prompt bias mistakes. To generate these prompt bias lessons, two authors conducted an independent review of each prompt bias mistake category and manually assigned corresponding lessons. Following this independent assessment, the two authors discussed the crafted prompt bias lessons. After discussion, both authors reached a unanimous consensus, with an inter-rater reliability Cohen's Kappa (k) [65] of 1, indicating perfect agreement. Finally, we appended the lesson to the default prompt to create the new prompt.

2.3.5 Method Evaluation

Finally, we evaluated the proposed method by rerunning the model again on all datasets with the modified prompts. To illustrate the effectiveness of the model, we report the performance of the model using standard precision, recall, accuracy, and F1 (harmonic mean of precision and recall) because those metrics show the reliability and effectiveness of the model. Moreover, these metrics were used to evaluate the performance of clone detection models in prior work [178, 49]. We then use those metrics to compare the performance of the model before and after adopting the proposed method. Moreover, to evaluate whether the improvement is significant or not, we adopted the t-test [92, 25, 25] and recorded the p-value. We give the definition of each metric below.

Precision:

Precision is a measure that quantifies the number of correct positive predictions made. It is defined as the ratio of true positives (tp) to the sum of true positives and false positives (fp). The formula for precision is given by:

$$\text{Precision (P)} = \frac{tp}{tp + fp} \tag{2.2}$$

Recall:

Recall, often referred to as the true positive rate, measures the percentage of actual positives that were correctly classified. It is the ratio of true positives to the sum of true positives and false negatives (fn). The formula for recall is:

$$\text{Recall (R)} = \frac{tp}{tp + fn} \tag{2.3}$$

Accuracy:

Accuracy is a metric that measures the proportion of true results (both true positives and true negatives) in the dataset. It provides a holistic view of the model's performance across all classes. The formula for accuracy, considering true negatives (tn) as well, is:

$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn} \tag{2.4}$$

F1-score:

The F1-score is the harmonic mean of precision and recall. It provides a balance between the two metrics, especially when there is an uneven class distribution. The formula for the F1-score is:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.5}$$

2.4 Results

In this section, we discuss the results of our study by placing them in the context of three research questions, which are defining the prompt bias mistake categories (RQ1), finding the prevalence of each prompt bias mistake category and the most frequent category (RQ2), and whether we can improve the model performance on clone detection task by leveraging prompt bias mistake categories (RQ3).

Table 2.2: Models Selection Result

Model	Benchmark Dataset Name	Precision	Accuracy	Recall	F1 Score
Turbo-3.5	avatar without comment	1.00	0.66	0.32	0.48
	avatar with comment	1.00	0.58	0.17	0.29
	poolC with comment	0.96	0.91	0.87	**0.91**
	poolC without comment	0.96	0.84	0.72	0.82
Text_002	avatar without comment	0.51	0.53	1.00	0.68
	avatar with comment	0.51	0.52	1.00	0.68
	poolC with comment	0.53	0.53	0.99	0.69
	poolC without comment	0.54	0.55	1.00	0.70
PaLM	avatar without comment	0.96	0.92	0.88	**0.92**
	avatar with comment	0.99	0.87	0.75	**0.86**
	poolC with comment	1.00	0.90	0.81	0.89
	poolC without comment	1.00	0.85	0.71	**0.83**
CodeT5	avatar without comment	0.55	0.54	0.57	0.56
	avatar with comment	0.49	0.50	0.56	0.52
	poolC with comment	0.47	0.0.45	0.45	0.46
	poolC without comment	0.58	0.56	0.62	0.60

2.4.1 RQ1: In the context of within-language and cross-language clone detection tasks, what are the prompt bias mistake categories?

In this section, we present the models' selection results, the default prompt we generated from GPT-4, the selected model's baseline performance, the confidence level of the model's incorrect predictions, and prompt bias mistake categories.

Model Selection Results:

We use the default prompt shown in section 2.3.1 to select the highest-performing model and to establish the model's baseline performance.

Subsequently, we present each model's performance across all benchmark datasets in Table 2.2. Upon a meticulous review of these results, a consistent trend emerged, with the PaLM model consistently surpassing the other models across nearly all benchmark datasets. The only exception to this pattern was observed in the case of the poolC with comment benchmark dataset, where we observed a marginal 2% higher F1 score for the Turbo-3.5 model compared to the PaLM model. Nonetheless, considering that the PaLM model outperformed the Turbo-3.5 model across all other instances and that the 2% discrepancy in F1 score is

Table 2.3: Confidence Level Of PaLM Models

Confidence Level	poolC with comment frequency	poolC without comment frequency	avatar with comment frequency	avatar without comment frequency
0	100.00%	99.72%	76.33%	86.44%
60	0.00%	0.05%	0.00%	0.00%
80	0.00%	0.05%	6.28%	0.00%
90	0.00%	0.19%	15.94%	6.78%
95	0.00%	0.00%	0.00%	6.78%
99	0.00%	0.00%	0.48%	0.00%
100	0.00%	0.00%	0.97%	0.00%

```python
# code snippet 1

X,Y,a,b,c = map(int,input().split())
if X < a + c or 0 > a-c:
    print("No")
elif Y < b + c or 0 > b- c:
    print("No")
else :
    print("Yes")

```

```python
# code snippet 2:

if __name__== "__main__":
    W, H,x,y,r = input().split()
    W,H,x,y,r = int(W),int(H),int(x),int(y),int(r)

    if (x-r) < 0 or W < (x+r) or (y-r)< 0 or H < (y+r):
        print("No")
    else:
        print("Yes")


```

Figure 2.5: Example of Operational Ambiguities category

relatively modest, we ultimately selected the PaLM model as our primary choice for further experimentation.

Lastly, we present the confidence level of the PaLM model for all incorrect predictions in both the poolC and avatar datasets, as detailed in Table 2.3. Upon a comprehensive analysis of these results, a clear and consistent pattern emerges. Specifically, when the confidence level equals or exceeds 80, the PaLM model demonstrates a notably high level of confidence in its predictions. This is evident from the majority of cases where the confidence level consistently remains above 80 or precisely at 80. It is worth noting that there are rare instances where the confidence level drops to 60. However, this occurrence is limited to the poolC without comment dataset and is exceptionally infrequent, constituting only 0.05% of all instances. As a result, we can confidently affirm that the PaLM model demonstrates a robust level of confidence in its predictions when its confidence level reaches or exceeds 80. This finding serves as strong validation for our decision to employ 80 as the threshold to identify reliable incorrect predictions, as discussed in section 2.3.2.

Prompt Bias Mistake Categories:

Next, we present our defined eight prompt bias mistake categories along with their definition:

- *Misconception Of Clone Code Definition:* This category refers to LLMs' confusion regarding the definition of clone code. Their interpretation of clone code aligns precisely with Type 1 clone code, which denotes code snippets that are identical [153]. Consequently, LLMs categorize all clone codes falling under Type 2-4 as non-clone codes [153].

- *Operational Ambiguities:* This category pertains to operators. LLMs tend to classify code snippets as non-clone codes when they exhibit variations in the order of operations. Additionally, LLMs often misclassify code snippets that differ in statements containing semantically equivalent but textually distinct operators as non-clone codes. In both cases related to operations, these code snippets are semantically identical. An illustrative example is provided in Figure 2.5, where variables X, Y, a, b, c are equated to W, H, x, y, r, respectively. LLMs incorrectly categorize this instance as non-clone code due to their confusion arising from the textual differences between "(x-r) $<$ 0" and "0 $>$ a-c," where "a=x" and "r=c." In essence, when LLMs make mistakes in this category, their ability to identify clone codes becomes limited to Type 2 clones [153].

- *Variable Initialization And Naming Ambiguity:* This category pertains to variables, encompassing aspects such as the sequence of variable initialization and variable names. Regarding the order of variable initialization, LLMs often categorize code snippets as non-clone codes when they differ in the sequence of variable initialization. Conversely, in situations where two code snippets employ distinct variable names, but the roles and values of these variables in the source code remain unchanged, LLMs inaccurately classify them as non-clone codes. In sum, when LLMs make errors within this category, their capability to detect clone codes is confined to Type 1 and 2 clones [153].

```python
# Code Snippet 1

import sys

alpha = input()

text = 'abcdefghijklmnopqrstuvwxyz'

for k in range(26):
    if text[k] == alpha:
        print('a')
        sys.exit()
    else:
        if k == 25:
            print('A')
```

```python
# Code Snippet 2

moji = input()



if moji.isupper() == True:
  print("A")
else:
  print("a")

```

Figure 2.6: Example of Confusion Over Varied Approaches category

- *Data Structure-Based Misclassification:* This category arises when two code snippets differ in their choice of data structures, a divergence that does not impact the underlying logic of the code snippets. For instance, when code snippet 1 utilizes a list to store elements, while code snippet 2 employs a set for the same purpose. LLMs often make mistakes by inaccurately categorizing such code snippets as non-clone codes. In essence, when LLMs make mistakes within this category, their capacity to identify clone codes is constrained primarily to Type 1 and Type 2 clones [153].

- *Misinterpretation Of Function And Library API Nomenclature:* This category pertains to the nomenclature of functions, encompassing both user-defined functions and APIs sourced from libraries. Within code snippets, it's common to encounter user-defined functions that may possess dissimilar function names but exhibit identical or akin functionalities. Conversely, APIs originating from distinct libraries often share similar functionalities. As an illustration, consider the functions torch.matmul() from the PyTorch library and np.matmul() from the NumPy library, which serve the same purpose despite having different names. LLMs make erroneous classifications when confronted with such code snippets. In essence, when LLMs err in this category, their capacity to identify clone codes becomes confined to Type 1 clones [153].

- *Thematic Distraction In Semantics:* This category pertains to situations in which two code snippets perform semantically identical tasks, yet they differ in their thematic content. For instance, consider code snippet 1, which employs binary search to locate a specific card within a poker card deck, and code snippet 2, which utilizes binary search to find a number within a list of numbers. If we disregard the thematic variations between these two code snippets, they essentially accomplish the same task. However, LLMs often become perplexed by the thematic distinctions and consequently misclassify them as non-clone code instances. As a result, when LLMs make mistakes in this category, their capability to identify clone codes is limited to Type 1, 2, and 3 clones [153].

- *Erroneous Code Comprehension:* This category pertains to instances where LLMs encounter challenges in comprehending the code snippet or misconstrue its context. When faced with such situations, the LLMs begin producing arbitrary explanations for the code snippet, akin to the issue of hallucination as described by Lee et al.[105]. Consequently, when LLMs make mistakes within this category, their capacity to detect clone codes is confined primarily to Type 1 clones [153].

- *Confusion Over Varied Approaches:* This classification pertains to scenarios in which two code snippets exhibit comparable logic but differ in their textual representation, especially when two code snippets adopt slightly different approaches. A clone code example of this category is shown in Figure 2.6. Both code snippets essentially perform the same function: they print 'a' if the input value is a lowercase letter and 'A' if the input value is an uppercase letter. However, PaLM classified them as non-clone code due to their textual dissimilarity. LLMs often overlook the similarity in code logic between these two snippets and instead make predictions primarily based on the textual variations. In essence, when LLMs err in this category, their capability to discern clone codes is limited to Type 1, 2, and 3 clones [153].

Finding 1: Majority of the mistakes made by LLM limits its capability to identify Type 4 clones.

2.4.2 RQ2: How prevalent are the mistake categories and what is the most frequent category?

In this section, we provide an overview of the frequency of each prompt bias mistake category drawn from BERTopic model's output. For better illustration purposes, we average the frequency of each category across with and without comment versions of poolC and avatar datasets. We present the results in Table 2.4. From the result, we observed that the category "Misinterpretation Of Function and Library API Nomenclature" has high frequency in both poolC and avatar datasets indicating that it is the most prevalent category (65.39%) in all datasets.

Moreover, when we contrast the avatar dataset with the poolC dataset, we observe a significant increase in frequency values across all categories. This variance can be attributed to the computation method employed for determining frequency values, which takes into account the number of topics generated by the BERTopic model from the incorrect predictions. Remarkably, the PaLM model displayed outstanding performance when applied to the avatar dataset. Consequently, the BERTopic model could only produce three topics due to small number of wrong predictions. As a result, this led to the merging of several categories into the same topics, thereby causing the observed rise in frequency values.

Finding 2: Misinterpretation Of Function and Library API Nomenclature is the most frequent prompt bias mistake category found in within and cross language clone detection datasets.

Table 2.4: Frequency of Each Category

Prompt Bias Mistake Category Name	**poolC**	**avatar**	**Across Datasets**
	Frequency (in%)	Frequency (in %)	Average Frequency (in %)
Misinterpretation Of Function and Library API Nomenclature	**36.38%**	**94.40%**	**65.39%**
Variable Initialization And Naming Ambiguity	35.42%	82.09%	58.76%
Operational Ambiguities	29.11%	48.13%	38.62%
Misconception Of Clone Code Definition	27.14%	48.13%	37.64%
Erroneous Code Comprehension	27.24%	42.91%	35.08%
Overemphasis On Textual Similarity	9.12%	60.45%	34.79%
Data Structure-based Misclassification	1.28%	47.39%	24.34%
Thematic Distraction In Semantics	8.70%	48.13%	28.42%

Table 2.5: Overview Of Prompt Bias Lessons

Lesson ID	**Prompt Bias Lessons**
1	Clone code is defined by similar functionalities; therefore, the code does not necessarily need to be identical.
2	Differences in function, and method names should not be criteria for determining whether code is cloned.
3	Differences in variable names should not be criteria for determining whether code is cloned.
4	Variations in approach should not be criteria for determining whether code is cloned.
5	Differences in data structure and implementation details should not be criteria for determining whether code is cloned.
6	Code syntax should not be a predominant factor in determining clones.
7	Code logic holds greater significance than minor code differences in determining whether code is cloned.
8	Variations in the order of operations should not be used to determine code clones.

2.4.3 RQ3: How can we harness these categories to improve the model performance?

In this section, we present our findings pertaining to the response to RQ3. Initially, we showcase the baseline performance of the PaLM model on each dataset using the default prompt shown in section 2.4.1, in Tables 2.7 and 2.8 indicated as "deafult". Subsequently, we provide an overview of the prompt bias lessons that were developed based on the categories of prompt bias mistakes, as detailed in Table 2.5. The two authors performed qualitative analysis [166] to derive the lessons from prompt bias mistake categories. Details were described in section 2.3.4.

To assess the effectiveness of our proposed approach, we carried out an ablation study [134] to investigate the impact of individual prompt bias lesson on the performance of the PaLM model. The outcomes for both the poolC and avatar datasets are summarized in Table 2.7 and Table 2.8. To facilitate a clearer understanding of the improvement brought about by each lesson, both tables contain a column labeled Δ_{F1}, indicating the change in the model's F1 score. A "/" denotes instances where no discernible improvement was achieved from a particular lesson.

Upon analyzing the Δ_{F1} column, we observed an average improvement of approximately *3.99%* for the avatar without comment dataset and *5.16%* for the avatar with comment dataset. For the poolC dataset, the average improvement was *3.25%* for without comment dataset and *3.03%* for with comment dataset. Additionally, observing from the Δ_{F1} columns in both Table 2.7 and Table 2.8, lesson 7 consistently brings substantial improvements to the performance of the PaLM model in all four datasets, indicating that lesson "Code logic holds greater significance than minor code differences in determining whether code is cloned." is the most effective lesson.

To determine the maximum performance boost achievable with our proposed method, we

ran the model with all lessons incorporated into the default prompt. The results are also displayed in Table 2.7 and Table 2.8. Notably, our analysis showed that the peak improvements of our methods can be up to *10.81%* in terms of F1 score for the avatar dataset and *9.77%* for the poolC dataset.

Furthermore, we conducted a t-test to assess the statistical significance of these improvements. The reported p-values shown in Table 2.8 and Table 2.7 demonstrated that all improvements were statistically significant, as they were below the threshold of 0.05 [67]. In summary, our proposed approach demonstrates its effectiveness and validity through the significant enhancements achieved exclusively through the incorporation of additional lessons. Figure 2.7 shows the prompt that resulted in the best performance.

Finding 3: Our proposed method is effective and brings an improvement up to 10.81% and 9.77% in terms of F1 score for avatar and poolC datasets, respectively.

Finding 4: "Code logic holds greater significance than minor code differences in determining whether code is cloned." is the most effective individual lesson.

You are a helpful assistant that takes two pieces of code snippets and detects whether these two pieces of code snippets are clone code or not. The first code snippet is written in <language1>, and the second code snippet is written in <language2>. <prompt bias lessons 1>. <prompt bias lessons 2>. <prompt bias lessons 3>. <prompt bias lessons 4>. <prompt bias lessons 5>. <prompt bias lessons 6>. <prompt bias lessons 7>. <prompt bias lessons 8>. Here are the two code snippets: Code Snippet1: <code1>. Code Snippet2: <code2>. If you think any part of the two code snippets contains clones, answer Yes; otherwise answer No. You must answer either Yes or No. Your Response:[INSERT].

Figure 2.7: Best Performing Prompt

Moreover, in order to gain a deeper understanding of the effectiveness of our approach, we conducted a comparative analysis between the predictions made by the PaLM model using the default prompt and those using the prompt enriched with all_lessons (as illustrated

in Figure 2.7). The results of this analysis are presented in Table 2.6. To be specific, we quantified the number of instances that were initially misclassified by the PaLM model when using the default prompt but were subsequently correctly classified when employing the prompt containing all_lessons (column 2). Conversely, we calculated the number of instances that were initially correctly classified by the PaLM model when using the default prompt but were subsequently incorrectly classified when employing the prompt containing all_lessons (column 3).

Table 2.6: Prediction Shifts for PoolC and Avatar: Default Prompt vs. With All Lessons Prompt

Dataset Name	**Incorrect to Correct Predictions Count**	**Correct To Incorrect Predictions Count**
poolC without comment	1,356	77
poolC with comment	1,166	101
avatar without comment	128	25
avatar with comment	175	20

Our findings indicate that through the adoption of our proposed method, the PaLM model makes a greater number of correct predictions, as evident from the significant values in column 2. Importantly, this enhanced performance does not negatively affect the previously correct predictions, as reflected in the small values in column 3.

2.5 Discussion and Implications

This section presents the implications of our findings for researchers and practitioners who consider using LLMs in their work.

Table 2.7: The Performances of PaLM Model With Prompt Bias Lesson on Avatar Dataset

Prompt Lesson ID	**avatar without comment**						**avatar with comment**					
	Precision	Accuracy	Recall	F1 score	$\Delta_{F1}(in\%)$	P-value	Precision	Accuracy	Recall	F1 score	$\Delta_{F1}(in\%)$	P-value
1	97.10	93.72	90.36	93.61	4.31%	<5.76e-12*	98.03	93.36	88.63	93.09	6.94%	<1.76e-20*
2	97.12	93.33	89.49	93.15	3.85%	<2.29e-13*	98.17	92.67	87.14	92.33	6.18%	<7.12e-18*
3	97.45	92.97	88.43	92.72	3.42%	<5.29e-13*	98.14	91.91	85.65	91.47	5.32%	<4.95e-14*
4	98.41	86.08	73.54	84.18	/	/	98.16	84.37	70.17	81.84	/	/
5	97.93	90.73	83.55	90.17	0.87%	<1.48e-03*	98.25	88.99	79.62	87.96	1.81%	<1.31e-06*
6	97.92	90.86	83.77	90.29	0.99%	<7.73e-03*	98.42	89.65	80.85	88.77	2.62%	<3.89e-09*
7	97.36	94.85	92.30	**94.77**	**5.47%**	<2.09e-21*	98.08	94.39	90.66	**94.23**	**8.08%**	<2.78e-26*
8	98.49	88.33	78.04	87.08	/	/	98.45	87.27	75.87	85.70	/	/
all_lessons	95.48	96.39	97.51	**96.48**	**7.18%**	<1.65e-27*	96.28	96.92	97.64	**96.96**	**10.81%**	<3.39e-27*
default	98.20	90.08	82.01	89.30	0.00	/	98.18	87.53	76.75	86.15	0	/

Table 2.8: The Performances of PaLM Model With Prompt Bias Lesson On PoolC Dataset

Prompt Lesson ID	**poolC without comment**						**poolC with comment**					
	Precision	Accuracy	Recall	F1 score	Δ_{F1} (in %)	P-value	Precision	Accuracy	Recall	F1 score	Δ_{F1} (in%)	P-value
1	98.91	90.76	82.23	89.81	5.27%	<2.13e-17*	98.54	91.64	84.48	90.97	4.56%	<1.26e-12*
2	99.03	87.85	76.30	86.19	1.65%	<3.44e-05*	98.78	89.21	79.26	87.95	1.54%	<8.38e-05*
3	99.06	87.13	74.81	85.25	0.71%	<9.02e-03*	98.75	88.76	78.38	87.39	0.98%	<7.41e-07*
4	99.50	78.95	57.84	73.16	/	/	99.21	82.11	64.44	78.13	/	/
5	99.45	83.26	66.61	79.78	/	/	99.23	85.55	71.45	83.08	/	/
6	99.51	81.17	62.44	76.73	/	/	99.13	84.31	68.97	81.35	/	/
7	98.92	90.81	82.37	**89.89**	**5.35%**	<5.30e-17*	98.54	92.06	85.28	**91.43**	**5.02%**	<5.06e-23*
8	99.70	78.88	57.67	73.07	/	/	99.48	82.17	64.41	78.20	/	/
all_lessons	98.26	94.57	90.67	**94.31**	**9.77%**	<9.23e-34*	97.77	94.75	91.58	**94.55**	**8.14%**	<8.70e-24*
default	98.96	86.59	73.79	84.54	0	/	98.62	87.97	76.89	86.41	0	/

Implications for practitioners: In comparison to BERT models, LLMs exhibit impressive source code comprehension abilities, making them a well-suited choice for the task of clone detection. However, practitioners who are either contemplating the adoption of LLMs or are currently utilizing them need to be aware of the notable performance variability of LLM and implement the lessons outlined in Table 2.5 to mitigate prompt bias.

Implications for researchers: In this study, we conducted a comprehensive comparison between CodeT5, a high-performing BERT model, and several popular GPT models within the context of clone detection. Our findings revealed that GPT models consistently outperformed CodeT5, underscoring the superiority of GPT models in this domain. Therefore, we strongly advocate for increased attention from researchers toward GPT models as a result of these impressive performances. Additionally, our research sheds light on the issue of prompt bias in GPT models and introduces a methodology involving prompt bias lessons to alleviate this problem. We encourage researchers to prioritize investigations into this specific concern related to GPT models and develop further solutions for mitigating prompt bias. Furthermore, we recommend clone detection researchers consider incorporating LLMs into their work as our work has shown the supreme performance of LLMs over other machine learning models.

As our work have shown, erroneous code comprehension is one potential reason for LLMs' incorrect predictions. Hence, future research should concentrate on addressing this issue. A promising avenue for exploration involves assisting LLMs in their comprehension through

prompting or providing them with explicit explanations of the source code. Furthermore, we manually generate prompt bias mistake categories and lessons. However, it would be more efficient and pragmatic if future research could automate this step. Lastly, as we have shown that prompt bias lessons deriving from prompt bias mistake categories have the potential to mitigate model prompt bias, an intriguing avenue for future research would involve uncovering additional categories and devising supplementary prompt bias lessons.

Moreover, within this work, we conducted an ablation study to assess the performance enhancements gained from individual lessons and calculated the cumulative performance improvement when applying all lessons simultaneously. However, we posit that there may exist optimal combinations of lessons that could yield even better results. Therefore, we encourage researchers to explore and identify more efficient ways of determining the most effective lesson combination.

2.6 Threats to Validity

In this study, our pipeline encompasses several key steps: model selection, the generation of rationales for selected model's incorrect predictions, defining categories for prompt bias mistakes, crafting prompt bias lessons, and enhancing the selected model through modified prompts. We have taken meticulous measures to mitigate potential threats that might affect the validity of our research. Nevertheless, it is important to acknowledge the possibility of flaws in our pipeline.

During the model selection phase, we observed outstanding performance from GPT models on both the poolC and avatar datasets. However, it's essential to note that this performance may not necessarily extend to clone detection datasets written in different programming languages.

In the process of defining prompt bias mistake categories, our approach mainly relied on

the model's incorrect prediction groups within the poolC dataset, with a limited sample size of 100 instances. Consequently, the prompt bias mistake categories we identified may be confined to the characteristics specific to the source code within the poolC dataset, and they may not encompass all potential prompt bias mistakes.

Furthermore, while applying the prompt bias lessons led to notable improvements of 10.81% and 9.77% for the avatar and poolC datasets, respectively, it's important to exercise caution when generalizing these findings to other clone detection datasets. However, we contend that our approach still provides valuable insights into mitigating prompt bias in large language models as we have shown the generalizability of the method in avatar dataset.

Lastly, the definition of prompt bias mistake categories and the generation of prompt bias lessons were conducted manually. To minimize the introduction of unintentional bias, both authors independently reviewed the model-generated mistakes, formulated the prompt bias categories, and crafted the prompt bias lessons. The high Cohen's kappa of 0.9 indicates a high level of reliability in this process.

2.7 Conclusion

Clone detection has remained a longstanding and prominent research domain due to the persistence of clone codes in traditional software development, where developers frequently employ code snippets from online sources. The advent of LLMs, notably exemplified by ChatGPT, has brought code generation into the forefront of their capabilities. However, this capability has the potential to lead to an increase in clone code instances, as LLMs generate code patterns they've encountered during their training. Consequently, the significance of clone detection has become more pronounced than ever.

In this study, we explored the feasibility and suitability of utilizing LLMs for clone detection and found that LLMs are well-suited for this task. Additionally, we delved into prompt

bias in LLMs and introduced a comprehensive framework for its mitigation. Our findings illuminated the fact that although LLMs perform admirably in clone detection, they are not infallible. Our analysis unveiled categories of prompt bias mistake that underscore some of the weaknesses in LLMs, including misconception of clone code definition, ambiguities related to the order of operations and semantically equivalent operations, uncertainties in variable initialization and naming, misclassification based on data structures, misinterpretations of function and API nomenclature, thematic distraction in semantics, erroneous code comprehension, and confusion over varied Approaches. Our proposed method shows possible directions for improving the weakness of the LLMs.

These findings offer valuable insights for researchers and practitioners considering the integration of LLMs into their work. As we chart the course for the future of software engineering, we posit that fully harnessing the potential of LLMs in diverse fields will be a prevailing trend and our work takes a significant first step towards that direction.

2.8 Data Availability

As part of our commitment to open science policy, all data collected for this study are made available as supplemental material. We provide our replication package in [2].

Chapter 3

When LLMs Go Wrong: A Study of Code Generation Mistakes

3.1 Introduction

Recent advancements in Large Language Models (LLMs) have garnered significant attention from both industry and academia, leading to a surge in LLM-based applications and research publications [193, 46, 187, 83, 22]. These models have been trained with large data corpora that contain natural language texts and source code, showing high competency in following human instructions in a wide range of downstream tasks [55, 63, 48]. One popular application of LLM is code generation, which utilizes LLMs to automatically generate source code for a given natural language requirement (i.e., coding question) [203, 135, 41, 116]. LLMs have shown impressive performance on this task [204, 77]. Thus, developers have begun integrating LLMs into their daily coding tasks in various stages of software development [174, 27, 55, 63].

While LLMs are showing impressive performance, research indicates that they can produce factually incorrect outputs [37, 119]. These mistakes pose a significant threat to the relia-

bility of LLMs [118, 150, 30]. For example, LLMs could recommend nonexistent third-party libraries to software developers, contributing to the distribution of software prone to exploitation [176]. Furthermore, companies such as Google have adopted LLMs to write codes for their software [132]. Consequently, mistakes in those automatically generated codes could harm the quality of the software. Therefore, uncovering and understanding the mistakes in LLM responses and their root causes has become urgent.

In recent years, the Software Engineering (SE) community has started investigating the mistakes in LLM-generated code and the underlying reasons. Fan et al. [46] identified four categories of syntactic mistakes in LLM-generated codes, while Song et al. [160] found seven categories of syntactic and non-syntactic mistakes. Although the mistakes identified in previous studies are valuable, several significant limitations remain. These studies only identified mistakes of LLMs from relatively small datasets, including LeetCode and HumanEval, which limits the completeness of their findings. Additionally, those works have focused exclusively on mistakes in Python LLM-generated code, neglecting mistakes in other languages. However, identifying mistakes in LLM-generated code beyond Python is crucial because LLMs may have different understandings across different languages [136, 207].

To address these shortcomings, our study examines Python and Java code generated by top-performing LLMs using datasets that include nearly 11 times as many coding questions, enabling us to identify more mistakes than previous studies [160, 46, 165]. We focused on Python and Java because code written in those two languages has been heavily deployed in software development [14, 36, 157, 133]. As LLMs' training data and parameter size change rapidly, their outputs, behavior, and biases also shift [104, 169, 72, 81, 143]. For example, Perez et al. [143] reported an inverse-scaling effect on the goal preservation evaluation in which larger models more often insist on keeping their goals unchanged, which is scored as undesirable, and thus accuracy declines as parameter size increases. They also documented sycophancy, showing that larger models more frequently chose the option aligned with a

user's stated view rather than the correct answer. Likewise, Tal et al.[169] reported that the ratio of stereotypical to anti-stereotypical errors grows with parameter size. Since LLM's coding mistakes stem from the generated code, patterns that were held for earlier models may no longer apply—or may manifest differently in subtle but important ways. Thus, we extended the prior mistake categories to reflect these evolving behaviors [160]. In addition to identifying additional mistakes, our study investigates the underlying reasons for the mistakes. This analysis is crucial, as it deepens our understanding of these mistakes and helps developers mitigate them when prompting LLMs to generate code. Finally, unlike previous studies [165] that relied solely on manual discussions to validate the reasons, we introduce a novel semi-automated verification method to substantiate that the identified reasons are indeed responsible for the mistakes, thereby enhancing the reliability of our findings. Specifically, our experiments aimed to address the following research questions.

- RQ1: What mistakes do LLMs make in code generation?
- RQ2: What are the underlying reasons for the LLM's mistakes in the code generation task?
- RQ3: How effectively can LLMs identify the mistakes and the underlying reasons in their generated code?

Specifically, this paper makes the following contributions:

1. A derived list of LLMs' code generation mistakes.
2. A derived list of reasons behind LLMs' mistakes in the code generation task.
3. A benchmark of 202 verified coding questions that LLMs failed to solve, each paired with a validated failure reason.
4. An empirical investigation on LLM's ability to identify the mistakes and underlying

reasons in its generated code.

3.2 Related Works

3.2.1 LLMs for Code Generation

Recent advancements in Natural Language Processing have led to the development of powerful LLMs, such as GPT-4 [8] and Claude 3 [16], which have demonstrated exceptional performance in both general NLP tasks [30] and Software Engineering (SE) tasks like Code Generation. Code-specific LLMs (CodeLLMs), such as StarCoder [124], DeepSeek-Coder [62], and Qwen2.5-Coder [79], are trained primarily on source code corpora with objectives designed to capture syntactic and semantic code features. Thanks to their extensive training data, these models exhibit strong performance on downstream tasks through prompt engineering alone, without requiring fine-tuning. Recent studies have achieved state-of-the-art Code Generation results by leveraging LLM prompting techniques [78, 205, 204, 110]. Building on this research, we prompted LLMs to generate source code given problem specifications from widely used Code Generation benchmarks. We then analyzed the LLM-generated code to identify mistakes and their underlying causes.

3.2.2 Mistakes in LLM Generated Code

Researchers have employed sound verifiers [164] to assess the correctness of LLM-generated code by enforcing strict rules or test cases. However, these verifiers are task-specific and require significant manual effort to design and implement. More recently, researchers have explored prompting LLMs to critique their own outputs, such as self-critique [125] and self-refinement [129], in general NLP tasks. However, Code Generation differs fundamentally from these tasks as it requires understanding both natural language and source code. Fan et al. [46] found that Codex [32] makes errors similar to those of humans when solving LeetCode problems. Song et al. [160] identified syntactic and semantic mistakes in LLM-

generated code on the HumanEval dataset. Zhang et al. [201] manually analyzed Python3 CoderEval-generated code, categorizing eight types of mistakes, while Tambon et al. [170] examined 333 bugs from LLM-generated code on the same dataset, identifying ten mistake categories.

Despite these insights, existing studies have key limitations. First, they primarily analyze small datasets, limiting the comprehensiveness of identified mistakes. Second, they focus on earlier LLMs like ChatGPT and Codex, raising concerns about the relevance of their findings as newer, more capable models are introduced. Finally, they examine only Python3-generated code, overlooking potential variations in mistakes across programming languages. To address these gaps, we analyzed mistakes in LLM-generated Python and Java code across two datasets containing 2,268 coding questions.

3.2.3 Why LLMs Make Mistakes in Code Generation

Several studies have explored the underlying reasons for LLM-generated code mistakes. Jesse et al. [86] found that LLMs produce Simple and Stupid Bugs (SStuBs) due to quality issues in their training data. Kou et al. [96] attributed errors to a misalignment between LLMs' and human coders' attention, analyzing mistakes through model attention patterns. Mu et al. [135] reported that ambiguous or insufficient coding requirements contribute to errors and introduced ClarifyGPT, which prompts LLMs to ask clarifying questions to mitigate misunderstandings. In this study, we take the first step toward a comprehensive analysis of the reasons behind non-syntactic mistakes in LLM-generated code for both standalone and production-level functions. Unlike prior work, which examines mistakes from a single perspective, we consider all previously identified factors. Furthermore, we conduct experiments to verify that these factors contribute to the observed mistakes.

3.3 Experimental Setup

Non-syntactic Mistakes: As LLMs grow larger, incorporating more parameters and extensive training data, they tend to produce fewer *syntactic mistakes*—errors that prevent code from compiling— [127, 158]. While syntactic mistakes are easily detected through compilation, *non-syntactic mistakes* are more elusive, manifesting only in specific test cases or remaining undetected if test coverage is insufficient. We argue that *non-syntactic mistakes* reveal not only limitations in LLMs' programming capabilities but also flaws in their reasoning and understanding abilities. Addressing these issues could inform the development of more reliable LLMs for code generation. Therefore, this study focuses on *non-syntactic mistakes*, which fall into two categories: (1) the generated code compiles but encounters runtime errors (i.e., crashes during execution) and (2) the generated code compiles and runs but fails to meet the functional requirements specified, resulting in test failures.

Code Generation Datasets: To thoroughly study LLM's code generation mistakes, we selected two widely used multilingual datasets, namely HumanEval-X [203] and MBXP [18]. The HumanEval-X dataset comprises 328 coding problems—164 in Python and 164 in Java—with an average of seven test cases per coding question. In contrast, the MBXP dataset includes 1,940 coding questions, with 974 in Python and 966 in Java, and an average of three test cases per coding question. We selected datasets supporting Python and Java due to their extensive use in industry and open-source projects [173]. Coding questions in these datasets are self-contained and, therefore, may not fully reflect real-world software development workflows. We deliberately excluded repository-level datasets since prior work has shown that LLMs struggle to extract and integrate necessary context from large codebases [113, 97]. Until this challenge is addressed, evaluating mistakes in repository-level code generation conflates context acquisition failures with true mistake identification.

Model Selection: To ensure the generalizability of our results, we selected two LLMs with cutting-edge performance in Code Generation [120, 210]: **GPT-4**: GPT-4 is one of the

most powerful general-purpose LLMs developed by OpenAI. It has demonstrated exceptional performance in numerous SE tasks, including Code Generation. We adopt GPT-4-0125-preview (GPT-4) [8] in our experiments. **Qwen2.5-Coder**: Qwen2.5-Coder is a CodeLLM series with different model sizes, which have shown state-of-the-art performance in a wide range of SE tasks such as Code Generation. In our study, we select qwen2.5-coder-14b-instruct [79].

Data Preparation: To collect LLM-generated code snippets with *non-syntactic mistakes*, we prompted GPT-4 and Qwen2.5-Coder to solve coding questions from the selected datasets (Section 3.3). Following best practices [70, 77], we designed our prompts to include task instructions, coding question specifications, function signatures, and input-output examples demonstrating the required functionality. To ensure reproducibility, we set the LLMs' temperature to 0 and executed three times for each coding problem to obtain the most deterministic outputs [142].

Next, we executed all test cases associated with the coding questions (detailed performance metrics for each LLM on these datasets, measured by Pass@1 score, are provided in our replication package [51]). We then filtered out code snippets that failed to compile due to *syntactic mistakes* and those that passed all test cases (i.e., correct solutions). In total, the LLMs produced incorrect solutions for 513 coding questions in MBXP and 71 in HumanEval-X. We discarded 27 LLM-generated code snippets with syntactic mistakes, corresponding to 25 coding questions. We collected their actual outputs and retained their test failure information for the remaining code snippets that failed at least one test case or encountered runtime errors (*non-syntactic mistakes*).

3.4 RQ1:What mistakes do LLMs make in the code generation?

3.4.1 Methodology to Answer RQ1

Obtain Correct Code: To identify non-syntactic mistakes in the LLM-generated code, we compared the incorrect outputs against correct reference solutions. However, we observed that LLMs often produce code that diverges from the ground truth in terms of algorithm or data structure choices. This is reflected in the low Jaccard similarity scores between the generated and ground truth code (0.24–0.28 for Qwen2.5-Coder and 0.22–0.44 for GPT-4). Discarding all such divergent yet plausible solutions would substantially reduce the dataset size, limiting the breadth and impact of our analysis.

To address this, we applied Automated Program Repair (APR) techniques to fix the incorrect code while preserving its original algorithmic structure. Specifically, we used CHATREPAIR [185], a state-of-the-art APR tool that iteratively refines code through conversational interactions with the LLM, guided by compiler errors and test failures. We followed the default hyperparameter settings recommended by Xia et al. [185], allowing up to 30 trials per repair. If CHATREPAIR failed to produce a functionally correct solution (as determined by the test cases) within this limit, we defaulted to the original ground truth.

To ensure the repaired code retained the LLM's intent, we measured Jaccard similarity between the APR-fixed code and the original incorrect version. The results showed high similarity—0.88 for Qwen2.5-Coder and 0.72–0.86 for GPT-4—indicating that CHATREPAIR preserved the underlying approach while correcting the errors. Further evidence supporting the efficacy of this process is provided in our replication package [51]. For each LLM-generated sample with non-syntactic mistakes, we compared the Jaccard similarity of both the APR-fixed and ground truth code to the original LLM output. In 99% of cases, the APR-fixed version was more similar to the original than to the ground truth, validating

our decision to use it as a reference.

Mistake Identification: To identify mistakes, we followed a two-step process. First, each author independently reviewed the incorrect LLM-generated code and its correct version, focusing on critical control statements (e.g., 'if' and 'while') that affected program logic. We also analyzed test case failures, variable values during execution, and runtime logs. Each author created an overview of the mistakes, although some mistakes might have been missed. In the second step, the authors (each with at least five years of Python and Java programming experience) met to discuss and validate the mistakes, proposing valid inputs to trigger each identified issue. This process resulted in a comprehensive list of mistakes for each incorrect LLM-generated code.

Mistake Categorization: After identifying mistakes in the LLM-generated code, we categorized mistakes throughout the following two phases. First, four authors conducted open coding [90] using a negotiated agreement process [68] to categorize them. Specifically, we began by considering the likely process an LLM follows when solving a coding problem—interpreting the specification, devising a solution, and implementing the code. Since these steps involve reasoning, algorithm selection, and programming skills (e.g., data structures, operators, APIs), each author independently analyzed the mistakes and proposed categories based on these aspects.

After that, we met to discuss and align our categorizations to mitigate individual bias. For each mistake, all authors presented their views, followed by a group discussion focusing on core elements such as algorithms and code constructs. If disagreements remained, a fifth author, with the most senior experience, reviewed the arguments and made the final decision [66, 128]. After reaching a consensus, we examined the categories to merge semantically similar ones, with the fifth author resolving any unresolved debates. Finally, we refined category names to ensure clarity and informativeness.

In the second phase, we merged the mistake categories identified in Phase 1 with existing mistake categories from prior works [160]. Each author independently (i) reviewed the definitions of existing mistake categories from prior works and the definitions of the mistake categories identified in Phase 1, and then (ii) compared, for every Phase 1 category, its definition against all existing mistake category definitions to decide whether it overlapped with an existing category or represented a new category. To mitigate individual bias, we followed the same process in phase 1. Specifically, all authors met and presented their decisions, followed by a group discussion focusing on the details of the category definition, category boundaries, and representative examples. If disagreements remained, a fifth author, with the most senior experience, reviewed the arguments and made the final mistake category merging [66, 128].

Mistake Severity Identification: After identifying errors in LLM-generated code, we assessed the severity of each mistake based on its impact on functional correctness. This analysis offers two key benefits: (1) it helps practitioners prioritize debugging efforts by focusing on critical issues, (2) it serves as an indicator of code quality, enabling more objective comparisons of LLM performance.

Each author reviewed the incorrect code to understand its overall structure, focusing on key elements such as parameters, outputs, and comments to infer intended functionality. We analyzed the function in segments, starting with variable initialization and data structures, followed by control flows (e.g., conditionals, loops) and exception handling to examine how the code processes different inputs. Combining these observations, each author formed a detailed understanding of the adopted algorithm and its functionality, which was then compared against the coding question's specification to assess deviations. Based on this gap, each author independently categorized the severity of the algorithmic mistakes.

Following the individual analyses, four authors met to discuss and agree on the severity of each incorrect code. Each author presented their assessment, including the identified algo-

Table 3.1: Mistake Categories, Mistake Types and Severity Frequency (highlighted types are newly identified)

Mistake Categories	Mistake Types	FADE	PADE	DADE	GPT-4 Mistake	Qwen2.5 Mistake
Condition Error	Conditional Misalignment Error (15.65%) Missing Condition Error (14.24%)	45.55%	49.74%	4.71%	30.49%	31.01%
Index Issue	Indexing Convention Mismatch Error (2.50%) Index Utilization Error (0.78%) Miscalculated Index Error (2.19%)	48.57%	37.14%	14.29%	2.74%	7.28%
Return Issue	Missing Return Error (1.88%) Minor Output Format Mismatch Error (8.92%) Wrong Variable Return (0.47%)	95.83%	4.17%	0%	12.50%	9.81%
Math Knowledge Issue	Incorrect Math Knowledge Error (14.24%)	7.69%	20.88%	71.43%	16.77%	11.39%
API Usage	API Return Handling Error (1.10%) Inappropriate Data Structure (3.44%) API Misuse Error (10.80%)	40.82%	55.10%	4.08%	17.68%	12.66%
Erroneous Pattern	Incorrect Mask Error (1.56%) Incorrect Regex Error (1.56%)	15%	80%	5%	3.35%	2.85%
Incorrect Functionality	Flawed Implementation (1.72%) Partial Implementation (9.23%) Completely Incorrect Implementation (9.70%)	8.33%	44.70%	46.97%	16.46%	25.00%

rithm and its implemented functionality, and the group compared these against the intended functionality. If disagreements remained, a fifth author with the most senior experience was consulted to make the final determination [66, 128]. After reaching a consensus, we reviewed and merged semantically similar categories, with the fifth author resolving any remaining disputes. Lastly, we refined category names to ensure clarity and informativeness.

3.4.2 Answer to RQ1: Non-Syntactic Mistake Analysis

The overall mistake categories are summarized in Table 3.1. Through manual inspection, we identified seven high-level mistake categories: *Condition Error, Index Issue, Return Issue, Math Knowledge Issue, API Usage, Erroneous Pattern, and Incorrect Functionality*, which are further divided into 17 specific mistake types. Due to space constraints, we present examples for only a subset of these mistakes; complete examples are available in our replication package [51]. Among the 17 mistake types, ten types were overlooked by prior studies [160, 201, 170] (highlighted in Table 3.1). The oversight can be explained by two main factors. First, our analysis not only considered smaller datasets such as HumanEval but also included LLM-generated code from the MBXP dataset, which comprises 1,948 coding questions—nearly 11 times the 164 questions in the HumanEval dataset. This more diverse

```
def large_product(nums1, nums2, N):
    """Find the specified number of largest products from two given lists"""
    nums1.sort(reverse=True)
    nums2.sort(reverse=True)
    products = []
    for i in range(min(len(nums1), len(nums2))):
        products.append(nums1[i] * nums2[i])
    products.sort(reverse=True)
    return products[:N]
```

(a) Index Utilization Error Example

```
def sd_calc(data):
    """Calculate the standard deviation"""
    mean = sum(data) / len(data)
    variance = sum((x - mean) ** 2 for x in data) / len(data)
    standard_deviation = variance ** 0.5
    return standard_deviation
```

(b) Incorrect Math Knowledge Error Example

```
public List<String> listSort(List<String>lst){
    List<String>evenLengthString = lst.stream().filter(s->
        s.length()%2==0).toList();
    evenLengthString.sort(Comparator.comparingInt(String::length));
    return evenLengthString;
}
```

(c) API Return Handling Error Example

```
def Repeat(x):
    """find duplicates from the given list >>>Repeat([1, 2, 3, 1, 2,]) [1, 2]"""
    seen, duplicates = set(), set()
    for item in x:
        if item in seen:
            duplicates.add(item)
        else:
            seen.add(item)
    return list(duplicates)
```

(d) Inappropriate Data Structure Example

Figure 3.1: LLM code-generation mistake examples: IUE, IMKE, ARHE, IDS

```
def check_char(string):
    """check whether given string starts and ends with same character"""
    pattern = r'^.(.).*\1$|^.$'
    if re.match(pattern, string):
        return "Valid"
    else:
        return "Invalid"
```

(a) Incorrect Regex Error Example

```
def is_sorted(lst):
    """If list has more than 1 duplicate of the same number, return False"""
    if len(lst) != len(set(lst)):
        return False
    return lst == sorted(lst)
```

(b) Conditional Misalignment Error Example

Figure 3.2: LLM code-generation mistake examples: IRE, CME

dataset choice enabled us to explore a broader range of coding scenarios, thereby uncovering additional mistakes. Second, by analyzing LLM-generated code in both Python and Java, our study was able to identify mistakes that may be unique to specific programming languages. Below, we provide definitions and examples of these ten non-syntactic mistake types:

Indexing Convention Mismatch Error (ICME) (2.50%): arises when the LLM misaligns its assumed index system (often 0-based) with the specified one. This could lead to LLM returning the wrong index or making out-of-bounds access.

Index Utilization Error (IUE) (0.78%): arises when the LLM misuses indices (not due to index arithmetic calculation or index convention system) to access the search space—either under-utilizing them, which leads to exploring only a subset of the necessary candidate

solutions, or over-utilizing them, by examining a broader range than required. In both cases, improper index usage overlooks valid solutions or expends unnecessary computational effort, ultimately leading to suboptimal or incorrect outcomes. An example is shown in Figure 3.1(a). The coding question requires the LLM to identify the specified number of largest products from two given lists. To do so, it must first compute the products for all pairs formed from the two lists. However, the LLM only considers pairs with matching indices (marked in red), thereby excluding other combinations that might yield larger products.

Missing Return Error (MRE) (1.88%): arises when the LLM correctly implements the desired functionality but does not have the return statement to return the result in the code.

Minor Output Format Mismatch Error (MOFME) (8.92%): arises when the generated code is functionally correct but the output deviates from the specification only in terms of its representation. This may involve differences in data type, numeric precision, or other formatting details, resulting in a mismatch between the expected and actual output format.

Wrong Variable Return (WVR) (0.47%): arises when the generated code is functionally correct, and the result is stored in a designated variable. However, instead of returning this variable at the end of the code, the incorrect code returns a different one, leading to an incorrect output.

Incorrect Math Knowledge Error (IMKE) (14.24%): arises when the LLM uses incorrect mathematical concepts, formulas, or properties to solve the coding question. Consequently, the generated code deviates from the intended mathematical reasoning required to solve the problem. It can manifest in multiple ways, such as improperly sequencing operations (e.g., performing addition before multiplication, contrary to the standard order of operations) or using the wrong formula to calculate the math sequences. An example is shown in Figure 3.1(b). The coding question requires the LLM to calculate the standard deviation. The LLM-generated code follows the standard approach by computing the mean

and variance. However, the code incorrectly calculates the variance by dividing len(data) instead of len(data)-1 (marked in red).

API Return Handling Error (ARHE) (1.10%): arises when the output of an API call is mishandled due to incorrect assumptions about its data type or mutability. In essence, the mistake involves treating the returned value of an API call as if it were of a different type or had different properties than it actually had. As shown in Figure 3.1(c), an LLM incorrectly uses a *stream* call and then attempts to convert it to a list object (in red). The *stream* call returns a non-mutable type, so the subsequent *toList* function call fails, highlighting the LLM's misuse of the *toList* method on the *stream's* return object.

Inappropriate Data Structure (IDS) (3.44%): arises when the LLM does not adequately account for the intrinsic properties of the adopted data structure, such as automatic duplicate removal or the absence of order. An example is shown in Figure 3.1(d). The coding question requires the LLM to identify duplicate integers from a list while preserving their original order. To accomplish this, the LLM uses a set to store duplicates (marked in red). Although a set effectively eliminates redundant entries, its inherent lack of order results in the loss of the original sequence of duplicates.

Incorrect Mask Error (IME) (1.56%): arises when the LLM uses the incorrect mask to perform the bit operation.

Incorrect Regex Error (IRE) (1.56%): arises when the LLM uses an incorrect regex pattern to match the given string. An example is shown in Figure 3.2(a). The coding question specifies the LLM to check whether the given string starts and ends with the same character using regex. However, the LLM adopted a regex pattern (marked in red) that checks for the second character and the last character instead.

Besides those ten mistakes types, we have also identified seven mistake types that align with prior studies [160, 201, 170]. Below, we describe those seven types:

Conditional Misalignment Error (CME) (15.65%): arises when the LLM generates a condition that misaligns with the coding question, resulting in flawed conditional logic. This mistake is not limited to if or else-if statements but also includes conditions in loops, such as while and for. For example, as shown in Figure 3.2(b), the coding question requires returning false only if a number appears three or more times in the list. However, the LLM-generated code incorrectly returns false when a number appears just twice, as seen in the faulty condition if len(lst) != len(set(lst)) (highlighted in red).

Missing Condition Error (MCE) (14.24%): arises when the LLM-generated code omits a crucial condition outlined in the specification or needed to address flaws in the approach, such as handling specific input scenarios or corner cases. Unlike the *Conditional Misalignment Error*, where the condition deviates from the specification, the Missing Condition Error arises when the required condition is entirely absent.

Miscalculated Index Error (MIE) (2.19%): arises when the generated code performs an incorrect index calculation, leading to accessing an uninitialized or incorrect index. Unlike an *Indexing Convention Mismatch Error*, where the model uses a different indexing system than specified, in this case, the LLM adheres to the correct indexing convention but miscalculates the index.

API Misuse Error (AME) (10.80%): arises when the LLM uses the wrong API call or incorrectly invokes the correct API with the wrong parameters or incorrect number of arguments. For example, a coding question specifies that for a given number a and a digit value *digits*, the LLM should round a to the specified digits. However, the LLM mistakenly uses Python's round function with the expression *round(a + 10**(-len(str(a))-1), digits)*. In Python, the round function does not always round up, depending on the specified digit value. Since the question requires the number to always round up, the LLM should have used the ceil function instead.

Flawed Implementation (FI) (1.72%): arises when the adopted algorithm in the generated code is mostly correct with only some minor logical errors, and its implemented functionality mostly aligns with the required functionality.

Partial Implementation (PI) (9.23%): arises when the adopted algorithm is on the right track, but with noticeable flaws that need to be addressed.

Completely Incorrect Implementation (CII) (9.70%): arises when the LLM-generated code is completely incorrect. Specifically, the functionality implemented in the LLM-generated code completely differs from the desired functionality described in the coding question specification.

3.4.3 Mistake Severity Categorization

This subsection presents the identified severity levels. Due to the space constraint, we include one example for each mistake severity in our replication package [51].

Flawed Algorithm Design Error (FADE): arises when the LLM's approach is generally on the right track—showing reasonable reasoning—but the algorithm contains minor flaws. These minor flaws may allow the code to produce correct or nearly correct outputs for some inputs, yet it fails on others due to overlooked corner cases or mishandles the explicitly mentioned subtle requirement that causes incorrect results.

Partial Algorithm Design Error (PADE): arises when the LLM's approach has a significant gap but remains aligned with the goal of solving the coding question. However, part of the required functionality—clearly specified in the problem description or examples—is either completely missing or incorrectly implemented.

Divergent Algorithm Design Error (DADE): arises when the LLM's implementation is entirely off track, substantially deviating from the intended functionality described in the

coding question specification and examples. This includes cases where the code performs the opposite of what is asked, attempts to solve an unrelated problem, or overlooks most or all required functionalities.

3.4.4 Analysis of Mistake Categories and Severity

In Table 3.1, columns 3–5 show the severity distribution (Flawed, Partial, Divergent) for each mistake category, while the last two columns report how often each LLM made mistakes in each category.

Model Mistake Analysis: Based on the comparison of the last two columns, we highlighted five key observations. First, both LLMs frequently make conditional errors, underscoring the need for practitioners to verify condition handling rigorously. Second, incorrect application of mathematical formulas is common, with GPT-4 making such mistakes slightly more often; both models struggle to recall precise formulas tied to specific concepts (e.g., sequences or series). In contrast, both LLMs consistently handle masking techniques and regular-expression patterns correctly, indicating strong proficiency in these basic constructs. Additionally, return-related mistakes are frequent in both LLMs, suggesting that they often overlook formatting details—seemingly minor issues that may lead to cascading failures in larger systems. Lastly, incorrect API usage remains prevalent, with GPT-4 slightly more prone to such errors, though this is expected given Qwen2.5-Coder's stronger coding-specific training. Regardless, careful validation of API usage remains crucial.

Comparison with Prior Categories: Compared to the mistake categories identified by Song et al [160], we identified 2 entirely new mistake categories, namely *math knowledge issue* and *erroneous pattern*. Unlike the mistake category "Operation/Calculation Error" identified by Song et al[160], which only includes arithmetic and comparison mistakes, the newly identified *Math Knowledge Issue* mistake category revealed LLM's conceptual misunderstandings of mathematical facts and properties (see Figure3.1(b)). The newly identified *erroneous pat-*

tern reveals that even state-of-the-art LLMs often generate faulty regular expressions and masking rules. In addition, we redefined the *return error* as a semantic error, which was previously defined as a syntactic error. Specifically, we observed that LLMs can miss return statements, produce mismatched output formats, or return the wrong variable—cases that reflect deeper semantic misunderstandings. Lastly, the newly identified subcategory in *API Return Handling Error* revealed that LLM's misunderstandings of the API are not limited to the API's functionality, but also to the return data type of the API call (see Figure 3.1(c)).

Cross Language Analysis: We analyzed the frequency of mistake categories across Python and Java LLM-generated codes and observed that category frequencies were broadly similar, and no category appeared exclusively in one language.

Mistake Severity Analysis: Our analysis reveals that condition errors, return issues, API misuse, and pattern-related mistakes largely stem from flawed or incomplete algorithm designs, whereas mathematical errors are often linked to divergent algorithms. To assess code quality further, we examined mistake severity across models. GPT-4's errors are more frequently classified as divergent, while Qwen2.5-Coder's are predominantly flawed. This aligns with GPT-4's higher rate of mathematical knowledge errors (16.77% vs. 11.39%), and Qwen2.5-Coder's lower incidence of return formatting mistakes (9.81% vs. 12.50%).

3.5 RQ2: What are the underlying reasons for the LLM's mistakes in the code generation task?

3.5.1 Methodology to Answer RQ2

Reason Identification: To identify the reasons behind non-syntactic mistakes, we analyzed the typical process LLMs follow when solving coding problems: interpreting the coding question, devising a logical strategy, and implementing the code. We hypothesized that four

factors influence this process: *(i) Phrasing used in the prompt*, *(ii) Presentation structure*. Additionally, input-output demonstrations help the LLM confirm its understanding and verify its approach [34, 53], leading to our third hypothesis: *iii) Input-output pair demonstration quality*. Finally, when generating code, the LLM relies on programming skills acquired during training [86]. This suggests a fourth contributing factor: *iv) Training-Induced mistakes*. To identify the underlying causes of mistakes, we employed an open coding process [90] combined with a negotiated agreement approach [68]. Four authors independently reviewed incorrect LLM-generated code, reference code, and coding questions to determine the reasons for errors, followed by group discussions to resolve disagreements. Once validated, the reasons were categorized and empirically tested by modifying coding questions based on the identified factors and prompting the LLM to regenerate the code. We deemed a reason validated only when the regenerated code passed all test cases; if any test failed, the reason was not verified. Below, we detail the analysis performed for the four hypothesized factors.

i) Phrasing Used In The Prompt: The LLM's comprehension of the coding question specification plays a crucial role in the quality of its generated code. One key factor influencing this comprehension is the phrasing and word choices used in the question. Ambiguities in the coding question can lead to misunderstandings, which manifest in three primary ways: **Ambiguous Wording:** The question may be phrased in a way that allows multiple interpretations, leading to inconsistencies in the LLM's response. **Misalignment with Function Names:** The intended meaning of the question may not fully align with the provided function name, potentially causing misinterpretation of the task. **Misleading Parameter Names:** Parameter names in the function signature may inadvertently suggest meanings that do not exist. For example, consider a function intended to calculate the area of a parallelogram, with the signature:*def calculate_ area(b, h)*. While the variable names b and h might suggest "base" and "height," they may actually represent two arbitrary sides of the parallelogram. This discrepancy can mislead the LLM, affecting its interpretation and leading to incorrect implementations.

To identify mistakes introduced due to this hypothesis, all authors independently reviewed the referenced correct code, coding question specifications, and identifiers of functions and parameters in the function signature. The analysis followed a structured four-step process: **Identifying the requirements of the coding question** – Each author independently determined what the coding question was asking for. **Understanding the LLM's mistake:** Authors examined what the LLM actually did when it made the mistake. **Group discussion:** The team discussed discrepancies between the intended requirements and the LLM's actual implementation, particularly focusing on whether the phrasing in the prompt contributed to the misunderstanding. **Generating paraphrased versions:** Sentences containing potentially ambiguous phrases were paraphrased using GPT-4, with explicit instructions to preserve their original semantic meaning.

GPT-4 was chosen for paraphrasing due to its strong natural language understanding capabilities [89]. The temperature was set to 0.5, ensuring a balance between diversity and controlled randomness, allowing creative yet semantically consistent variations [168, 145]. To validate the paraphrased sentences, all authors reviewed the original and modified versions to confirm that the intended meaning remained unchanged. If most authors found that a paraphrase altered the meaning, it was excluded from the analysis.

ii) Presentation Structure: LLMs are built on transformer architectures that use attention mechanisms to represent the focus. Insufficient attention allocation can cause LLMs to overlook certain content. To address this issue, it is crucial to enhance the LLM's attention to the overlooked sections of the specification. We hypothesized that the position of information within the input influences the LLM's attention distribution. Thus, repositioning relevant information—while preserving its semantic meaning—may encourage the LLM to allocate more attention to it. Importantly, the *presentation structure* emphasizes the position of the overlooked specification in the prompt, whereas the *phrasing used in the prompt* centers on the specific wording of the specification.

To identify mistakes introduced due to this hypothesis, each author individually and then through negotiated agreement [68] reached a consensus on the *missing functionality*. Then, we moved the description of the missing functionality to a before and after location in the coding question (see Figure 3.3(d) as an example). To ensure this adjustment did not change the meaning of the coding question, all authors read and compared the descriptions before and after the change. In cases where changing the position of the word phrases would affect the meaning of the coding question, we instead added the phrase "note that" immediately before the missing functionality in the original description to force the LLM to pay more attention.

iii) Input-output pair demonstration quality: Besides the specification, the input-output pair demonstrations also help the LLM understand the intended functionality and the input/output format. Consequently, non-syntactic mistakes made by LLMs can often be traced back to the quality of these demonstrations—for example, how thoroughly they convey the desired functionality, specify the output format, or address potential corner cases.

We first gathered all LLM-generated codes that passed demonstration examples but failed other tests to identify mistakes caused by this issue. Using program slicing [186, 23, 204], we broke the incorrect code into smaller segments. For each, we traced intermediate variable values via program tracing [101] and analyzed the logic to understand why it succeeded on demonstrations but failed other tests. Each author independently performed this analysis, then reached consensus on the LLM's reasoning about the input-output pairs through negotiated agreement [68]. Finally, we compared this reasoning to the original question to determine if the LLM misinterpreted the problem or overfitted to the demonstrations.

To verify this reason, we employed two strategies. First, we augmented the coding question with additional examples to improve the LLM's understanding of the task. These examples were generated by prompting GPT-4 to design new test inputs, after which we executed the ground truth code to obtain the expected outputs. We selected GPT-4 for test case

design due to its strong code understanding ability [8]. To ensure the quality of these demonstrations, we discarded any inputs for which the ground truth code failed to produce an output. Second, we revised the prompt to instruct the LLM to consider not only the provided inputs but also other valid inputs.

iv) Training-Induced: LLMs acquire programming skills by learning frequent patterns in their training data. However, this reliance on common patterns can lead to mistakes when subtle, language-specific features are not well captured during training. These features include operators (e.g., negation, inversion) and API calls to built-in or third-party libraries. A key issue arises when an LLM confuses the functionality of a method or operator across different programming languages. For example, the function call *round(-5.5)* is syntactically correct in both Python and Java, and in most cases, their outputs align. However, for certain inputs—such as -5.5—the results differ: Python rounds it to -6, while Java rounds it to -5. Such errors likely stem from limitations in training data, either due to insufficient exposure to these nuances or an inadequate number of relevant examples. As a result, when generating code, the LLM may inadvertently apply the semantics of one language to another, leading to non-syntactic mistakes.

To investigate whether LLMs make non-syntactic mistakes due to failing to recognize subtle differences between identically named functions and operators across programming languages, we first compiled a list of functions and operators from Python and Java documentation that share the same name but differ in functionality [146, 84], return types, or signatures. We then identified coding questions where the LLM's implementation was correct in one language (e.g., Python) but incorrect in the other (e.g., Java). Using AST, we pinpointed lines containing function calls and operators in both implementations and excluded those not on our list. We analyzed the LLM's use of functions, including parameter types, number of arguments, and return value handling, along with its use of operators. When the LLM invoked functions or used operators similarly across both languages—despite subtle dif-

```
def same_chars(s0, s1):
    """check if two words have the
       same characters"""
    dict0, dict1 = {}, {}
    for char in s0:
        if char in dict0:
            dict0[char] += 1
        else:
            dict[0][char] = 1
    for char in s1:
        if char in dict1:
            dict1[char] += 1
        else:
            dict1[char] = 1
    return dict0 == dict1
```

```
def same_chars(s0, s1):
    """check if two words have the
       same set of unique characters"""

    return set(s0) == set(s1)
```

(a) Misleading Coding Question Specification Example

```
You'll be given a string of words, and your task is to count the number of boredoms.
A boredom is a sentence starts with the word "I". Sentences are delimited by '.', '?' or '!'.
>>> is_bored("The sky is blue. The sun is shining. I love this weather") 1
```

```
def is_bored(S):
    sentences = S.split('.|?|!')
    bored_count = 0
    for sentence in sentences:
        words = sentence.split()
        if len(words) > 0 and words[0] == 'I'
            bored_count += 1
    return bored_count
```

```
public int isBored(String S){
    int count = 0;
    String [] sentences = S.split("[.?!]")
    for (String sentence: sentences){
        String [] words = sentence.split(" ");
        if(words.length>0 && words[0].equals("I")){
            count ++;
        }
    }
    return count;
}
```

(b) Incorrect Trained Knowledge Example

```
public static String[] concatenateStringArrays(String[] array1, String[] array2) {
    /*Concatenate two string arrays into one. You must consider all possible cases*/
    if ((array1==null || array1.length==0) && (array2==nullllarray2.length==0)){
        return new String[0];
    }
    if (array1==nullllarray1.length==0){
        return array2;
    }
    if (array2==nullllarray2.length==0){
        return array1;
    }
    String[] concatenatedArray = new String[array1.length + array2.length];
    System.arraycopy(array1, 0, concatenatedArray, 0, array1.length);
    System.arraycopy(array2, 0, concatenatedArray, array1.length, array2.length);
}
```

(c) Edge Case Example

```
Write a function vowels_count which takes
a string representing a word as input and
returns the number of vowels in the string.
Vowels in this case are 'a', 'e', 'I', 'o', 'u'.
Here 'y' is also a vowel, but only when it is
at the end of the given word.
```

```
def vowels_count(s):
    vowels = 'aeiouy'
    count = 0
    for letter in s.lower():
        if letter in vowels:
            count += 1
    if s[-1] == 'y':
        count -= 1
    return count
```

```
Write a function vowels_count which takes
a string representing a word as input and
returns the number of vowels in the string.
Here 'y' is also a vowel, but only when it is at
the end of the given word.
Vowels in this case are 'a', 'e', 'I', 'o', 'u'.
```

```
def vowels_count(s):
    vowels = 'aeiouy'
    count = 0
    for letter in s.lower():
        if letter in vowels:
            count += 1
        elif letter=='y' and s.index(letter)==len(s)-1:
            count += 1
    return count
```

(d) Positional Sensitivity Example

Figure 3.3: Example reasons for LLM code-generation mistakes.

ferences—it suggested that the LLM either overlooked or was unaware of these distinctions. Furthermore, if the LLM processed a function's return value as identical across languages, despite differences, this indicated an incorrect understanding of the function's return details.

3.5.2 Answer to RQ2: Reasons of Non-syntactic Mistakes

In this section, we report reasons identified with newly identified reasons using an asterisk (*).

1. Misleading Coding Question Specification (MCQS) (56.19%) refers to the cases where LLMs are confused by specific phrases used in the coding question specification. We found that such ambiguities can lead to a range of mistakes, including condition errors, index issues, missing return errors, and minor output format mismatches. For example, consider Figure 3.3(a), where the LLM was given the following coding question specification: *"Check if two words have the same characters."* Upon analyzing the incorrect code generated by the

LLM, we observed that it used two dictionaries to separately count the frequency of characters in each word. This suggested that the LLM interpreted the question as checking whether the two words were anagrams—that is, whether they contained the same characters with the same frequencies. Recognizing this misinterpretation, we hypothesized that the confusion arose from the phrase *"same characters"*, as the word *"same"* may have led the LLM to consider character frequencies rather than just character presence. To verify our hypothesis, we prompted GPT-4 to rephrase the instruction to reduce ambiguity. It generated the revised phrase: *"same set of unique characters."*. We then asked the LLM to regenerate the code using the revised question specification:*"Check if two words have the same set of unique characters."* The newly generated code passed all test cases, confirming our hypothesis that the phrase *"same characters"* introduced ambiguity, leading to the LLM's misinterpretation.

2. Poor Input-output Demonstration* (PIOD) (21.26%) occurs when low-quality or insufficient input-output examples lead to LLM misunderstandings, affecting code correctness. Ambiguous or limited demonstrations can result in multiple interpretations, causing the LLM to generate flawed yet seemingly correct outputs. This aligns with Tambon et al.'s findings on LLM bias towards provided examples [170]. Additionally, when coding questions lack sufficient examples or explicit output format specifications, LLM-generated code often exhibits minor formatting errors, likely due to the model's inability to infer the expected return format.

3. Incorrect Trained Knowledge (ITK) (5.71%) refers to cases where LLMs generate incorrect code due to flawed knowledge learned during training, supporting the *Training-Induced* hypothesis. Specifically, LLMs sometimes mislearn details of language-specific features like APIs, libraries, and operators, leading to errors such as API misuse, return handling issues, and indexing mismatches. The underlying reason is that, during training, LLMs typically receive only the file contents—not filenames, extensions, or other path metadata that would explicitly signal the programming language [112, 124]. As a result, the model

loses important contextual signals about whether code belongs to Python, Java, or another language. An example of LLM misusing the API call is shown in Figure 3.3(b). In this example, the LLM generated one Python code (left) and one Java code (right) for the same coding question. However, the Python code mistakenly uses the *split* function since LLM incorrectly assumed that the *split* function in Python is the same as that in Java, which can split the string by regular expression. This issue reflects the limitations of LLM training on vast, unlabeled datasets, where the lack of language context in training examples can induce persistent misconceptions, a "necessary evil" [148].

4. Misleading Function Signature* (MFS) (4.44%) refers to cases where LLMs are misled by function names in the specification that conflict with the intended functionality, aligning with the *phrasing used in the prompt* hypothesis. When function names suggest a different task than described, LLMs may generate incorrect code, causing errors like condition mistakes, index issues, or output mismatches. For instance, in a task requiring the removal of all elements that appear more than once, the function name *remove_ duplicates* led GPT-4 to mistakenly remove only duplicate instances. Renaming it to *removeALLElementsOccurMoreThanOnce* resulted in correct code, confirming that misleading function signatures can impair LLM performance [189].

5. Edge Case (EC) (4.86%) occurs when LLMs fail to account for all possible corner cases. This relates to the *Input-output pair demonstration quality* hypothesis because LLMs could be biased by the input-output pair demonstration and overlook the possible edge cases. As a result, LLM-generated code may work for provided examples but fail on edge cases, leading to errors like conditional misalignment or missing conditions. For instance, as shown in Figure 3.3(c), GPT-4's code failed when an input array was null. However, adding the phrase *"You must consider all possible cases"* to the prompt led to correctly handling null and empty inputs.

6. Positional Sensitivity* (PS) (4.12%) refers to cases where LLMs overlook or misin-

terpret parts of a coding question due to lack of attention [34], supporting the *Presentation structure* hypothesis. This often occurs in longer prompts, leading to partial implementations or missing condition errors. For example, as shown in Figure 3.3(d), when asked to count vowels and treat 'y' as a vowel only if it appears at the end (as shown in the left figure, marked in red), the LLM misapplied this rule. By simply repositioning the instruction about 'y' to an earlier part of the prompt (as shown in the right figure, marked in green), the LLM produced correct code, indicating that non-syntactic mistakes can result from positional sensitivity to how information is presented.

3.6 RQ3: How effectively can LLMs identify the mistakes and the underlying reasons in their generated code?

Given the strong reasoning abilities of LLM [93, 57], we explore their potential for automating the identification and uncovering causes of mistakes.

3.6.1 Methodology to Answer RQ3

Mistake Identification

To evaluate LLM's abilities in identifying the mistakes in the generated code, we selected GPT-4 because of its widespread adoption by researchers [125] for critiquing the LLM-generated responses and its strong code understanding ability [8].

Prompt Construction: We provided GPT-4 with the coding question, incorrect LLM-generated code, and test case failure information. The failure information included the test code, error message, expected output, and actual output of the LLM-generated code. We then instructed GPT-4 to identify potential non-syntactic mistakes (the full prompt is included in our replication package [51]). Afterward, we manually reviewed GPT-4's responses and compared them to the mistakes identified in RQ1 (Section 3.4). For responses

that did not align with our identified mistakes, we verified them using a procedure similar to that in Section 3.4.

Evaluation Metric: To evaluate GPT-4's performance, we use precision and Coverage Rate (CR). Precision was calculated based on true positives (actual mistakes reported by the LLM) and false positives (hallucinated mistakes). The coverage rate (CR) is defined as:

$$\text{CR} = \frac{\#\text{ Identified True Mistakes}}{\#\text{ Total Mistakes}} \tag{3.1}$$

where Identified True Mistakes represent the actual mistakes reported by LLM, while Total Mistakes are true mistakes identified by either human or LLM.

Reason Identification

To evaluate LLM's abilities in identifying the reasons for mistakes, we created a benchmark of 202 instances containing incorrect LLM-generated code, corresponding coding question, and the human-labeled reasons. We created this benchmark based on the reasons identified in Section 3.5. Subsequently, we experimented with three approaches using GPT-4: *Base Prompt*, *Advanced Prompt*, and *Advanced Prompt+ReAct*. The prompts are included in our replication package [51]. Below, we describe each prompt.

(i) **Base Prompt:** We prompted GPT-4 to identify the reasons behind non-syntactic mistakes in LLM-generated incorrect code. The prompt included the coding question specification, the incorrect code, its actual output, and test failure information. It also contained instructions directing the model to analyze how the LLM may have been misled and to provide a comprehensive explanation. Importantly, we did not include any human-labeled reasons or their definitions in the prompt

(ii) **Advanced Prompt:** In addition to the information in the *Base Prompt*, this prompt

included definitions of all six human-labeled reasons (Section 3.5.2). We expected these definitions to help GPT-4 identify reasons with the intended granularity. The LLMs were instructed to report the reasons and explain their reasoning steps.

(iii) **Advanced Prompt+ReAct:** We employed the ReAct prompting technique [190], which integrates reasoning and actions to enable LLMs to iteratively retrieve and process relevant information. This method allows GPT-4 to dynamically select and apply contextual information, thereby enhancing its reasoning capabilities. Given its demonstrated effectiveness in tasks requiring complex, context-aware reasoning and generation [190], ReAct is particularly well-suited for our reason identification task. To support GPT-4 in identifying the underlying causes of non-syntactic mistakes, we developed three specialized tools.

Description of the tools implemented:

(i) **Function Call Analysis Tool:** The function call analysis tool identifies third-party and built-in function calls in incorrect LLM-generated code and retrieves their official documentation to help detect misuse. We used the *ast*[17] and *javalang*[85] modules to parse Python and Java code, respectively. For Python, documentation was retrieved via *pydoc*'s *render_doc* function, and for Java, we scraped Oracle API docs using *BeautifulSoup*. This tool replicates the manual process of consulting API documentation.

(ii) **Function Signature Explainer Tool:** This tool automates the manual steps from Section 3.5.2 to detect misleading function signatures. It extracts the function signature and parameter names from the coding question and prompts GPT-4 to summarize the function's purpose, and compares this summary with the coding question specification to evaluate how clearly the function signature conveys its functionality.

(iii) **Coding Question Specification Ambiguity Check Tool:** This tool assesses the ambiguity of a coding question using the self-consistency method [180]. If an LLM consistently understands a question, its repeated summaries should be similar. We prompt GPT-4

Table 3.2: GPT-4 Reason Identification F1 Score

Reason	Base Prompt	Advanced Prompt	Advanced ReAct
Misleading Coding Question Specification	0.87	0.91	0.93
Poor Input Output Demonstration	0.67	0.72	0.87
Edge Case	0.78	0.81	0.83
Misleading Function Signature	0.62	0.63	0.68
Positional Sensitivity	0.25	0.62	0.65
Incorrect Trained Knowledge	0.64	0.68	0.70
Average	**0.64**	**0.73**	**0.78**

to generate five summaries of the question (using temperature 0.5) given the original question and a human-written summary. These are converted into embeddings using *all-mpnet-base-v2* [151], and we compute pairwise cosine similarities. The question is deemed ambiguous if most scores fall below 0.8 [206].

Evaluation: To evaluate our approaches, all authors independently reviewed the LLM-generated reasons and explanations, assessing their semantic equivalence to human-labeled reasons in our benchmark. Disagreements were resolved through consensus using the negotiated agreement method [68]. A reason was deemed correct if it semantically aligned with the corresponding benchmark reason.

3.6.2 Answer to RQ3: Performance of Mistake and Reason Identification

1) Mistake Identification Result: Overall, GPT-4 performs comparably to human evaluators on HumanEval-X and MBXP, achieving precision scores of 0.97 and 0.95, respectively, with a coverage rate (CR) of 0.94 on both datasets. Humans attained perfect precision (1.0) and slightly higher CRs (0.98 on HumanEval-X and 0.99 on MBXP). These results indicate that LLMs like GPT-4 are effective at detecting mistakes in LLM-generated code, including subtle non-syntactic errors that humans might miss.

2)Reason Identification Result: The performance of GPT-4 is summarized in Table 3.2.

Due to space constraints, we report only F1 scores here, with additional metrics provided in the replication package [51]. For the *Base Prompt*, GPT-4 achieved an average 0.64 F1 score. This increased to 0.73 for *Advanced Prompt*. The Advanced Prompt led to substantial gains in identifying *positional sensitivity* (from 0.25 to 0.62) and *incorrect training knowledge* (from 0.64 to 0.68). We attribute these improvements to including human-identified reasons, which helped steer the model's attention toward subtle and previously overlooked aspects. The *Advanced Prompt+ReAct* approach showed further improvement, with the F1 score rising from 0.73 to 0.78, with further improvement in identifying misleading coding question specification (from 0.91 to 0.93). These results indicate that LLMs can successfully identify most of the reasons, except for issues related to positional sensitivity.

3.7 Threats to Validity

Construct validity The quality of LLM-generated code may be influenced by prompt design. While we followed best practices [53, 80, 20], we did not employ all possible advanced prompting techniques. Additionally, we relied on test cases from the datasets to assess the correctness of the generated and APR-repaired code, which might not cover all possible edge cases. To address these concerns, we used widely adopted code generation benchmarks that are adopted by existing code generation studies.

Internal validity Our experiment relies heavily on manual examination, which may introduce limitations such as incorrect understanding and overlooked mistakes or reasons. To mitigate these risks, we used established practices such as open coding and negotiated agreement [50], involving multiple authors to reach a consensus on the findings. In RQ3, both advanced prompts include manually labeled reasons intended to help the LLM identify the underlying reasons for the mistake. We recognized a potential bias, whereby the LLM might favor the provided reasons and overlook alternatives. To mitigate this risk, the prompts explicitly present the labeled reasons as guidance rather than directives and instruct

the LLM to consider them alongside other plausible reasons and to justify its final choice independently.

External validity Our findings may not be generalizable to LLM-generated code from other benchmarks, all programming languages, and all available LLMs. However, since GPT-4 and Qwen2.5-Coder come from different organizations with potentially different training data and model parameters, their mistakes offer insights applicable to a broader range of LLMs. Furthermore, coding questions from the HumanEval-X and MBXP datasets may have been used to train both LLMs, posing risks of data leakage. However, this does not weaken our conclusions because the persistence of mistakes under potential exposure suggests these mistakes are model-intrinsic rather than unseen-problem effects. Moreover, we study cross-language LLM mistakes only using two multilingual benchmarks with two programming languages. However, we do not claim exhaustiveness, and models may exhibit different mistake patterns in other programming languages.

3.8 Conclusion

In this work, we identified 17 types of mistakes, underlying reasons, and LLM's ability to identify the mistakes and underlying reasons. Our findings are valuable for practitioners using LLM-generated code and for researchers. Our replication package is available at [51].

Chapter 4

ProjAgent: Procedural Similarity Retrieval for Repository-Level Code Generation

4.1 Introduction

Large Language Models (LLMs) have demonstrated strong performance across a wide range of Software Engineering (SE) tasks. From automated bug fixing to test case generation, LLM-based approaches have achieved promising results on established benchmarks such as SWE-bench [87] and BigCodeBench [211]. These advances have accelerated the adoption of LLMs in software development workflows, supporting activities including code writing, code review, and software maintenance. Among these applications, code generation has attracted particular attention due to its broad practical utility and increasing adoption in real-world development [212, 159].

Despite these advances, repository-level code generation remains substantially more challeng-

ing than standalone code generation [32, 19, 177]. Unlike standalone benchmarks, real-world software development rarely involves implementing isolated functions. Instead, developers work within repositories where functions depend on utilities, type definitions, APIs, and project-specific conventions distributed across multiple files [109, 194]. When relevant repository context is unavailable, LLMs frequently hallucinate APIs, invoke nonexistent functions, or generate implementations that violate project conventions, resulting in incorrect or non-executable code [102]. Prior work has shown that removing repository context consistently degrades code generation performance across models [109]. Consequently, repository-level code generation depends not only on the generation capability of an LLM, but also on its ability to retrieve repository context that is useful for solving the target programming task.

Existing context retrieval methods for repository-level code generation, such as BM25 and dense embedding search, rely primarily on lexical or semantic similarity [194, 61, 152]. These methods were originally developed for code search, where retrieving surface-level similar examples is often sufficient [102]. In repository-level code generation, however, critical context for the target function (i.e., the function to be generated) may come from helper functions that it depends on, even when those functions differ substantially in naming, data types, or domain vocabulary. Useful context may also come from functions that implement similar logical procedures without sharing a direct dependency relationship. Because such functions often exhibit little lexical or semantic similarity to the target function, they are likely to be overlooked by existing retrieval methods. We refer to functions whose steps share the same underlying reasoning patterns as the target function's steps as a procedurally similar context.

Figure 4.1 illustrates this challenge. The coding problem *BlackBody.evaluate* contains a step that checks whether the inputs x and temperature are *astropy.units.Quantity* instances and raises ValueError if the temperature is negative. The retrieved context step is from the method *FLRW.m_nu*, which checks that the input value has the correct shape and

is non-negative, and raises a ValueError if either condition fails. The coding problem is in *astropy/modeling/physical_models.py*, and the retrieved context is in *astropy/cosmology/flrw/base.py*. Although they are located in different modules and serve different purposes, they share a common computational pattern: validating inputs for shape, type, and value constraints. Existing retrieval signals, such as BM25, dense embedding, or data flow analysis, fail to retrieve this context step for this coding problem step because the two steps share no lexical overlap (BM25 similarity of 0.38), low semantic surface similarity (embedding similarity of 0.59), and no direct call dependency between the two modules. As a result, surface similarity-based retrieval can miss critical dependencies while retrieving context that appears relevant but serves a different purpose, ultimately degrading generation quality [102, 61].

This observation suggests that repository-level retrieval should consider not only lexical and semantic similarity but also *procedural similarity*, the extent to which two functions implement similar computational procedures regardless of naming or domain-specific vocabulary. Procedurally similar functions may arise because one function serves as a building block for another or because both independently implement comparable implementation patterns, such as input validation, state transformation, or unit conversion. Identifying such relationships requires representations that capture implementation behavior beyond surface-level textual similarity. Recent work has shown that intermediate LLM representations encode rich semantic and functional properties of source code [75], motivating their use for identifying procedurally related functions.

To leverage this insight, we propose ProjAgent, a repository-level code generation system that combines procedural, lexical, and semantic retrieval signals to construct richer repository context. Rather than relying solely on representation similarity, ProjAgent employs an agentic retrieval workflow that first identifies and validates a small set of procedurally related functions, then expands this set to retrieve additional context using hidden-state projection

```python
def evaluate(self, x, temperature, scale):
    # Check if the input temperature is a valid type (astropy.units.Quantity)
    if not isinstance(temperature, u.Quantity):
        in_temp = u.Quantity(temperature, u.K)
    else:
        in_temp = temperature

    # Check if the input x is a valid type (astropy.units.Quantity)
    if not isinstance(x, u.Quantity):
        in_x = u.Quantity(x, self.input_units["x"])
    else:
        in_x = x

    # Check if temperature is not zero
    if np.any(temp < 0):
        raise ValueError(f"Temperature should be positive: {temp}")
    ...
```

(a) Coding Problem Step: `BlackBody.evaluate` (step_0) from `astropy/modeling/physical_models.py`

```python
@m_nu.validator
def m_nu(self, param, value):
    if (nneutrinos := floor(self.Neff)) == 0 or self.Tcmb0.value == 0:
        return None  # None, regardless of input

    value = _validate_with_unit(self, param, value)

    # Check values and data shapes and raise ValueError if the shape is incorrect
    if value.shape not in ((), (nneutrinos,)):
        raise ValueError(
            "unexpected number of neutrino masses "
            f"expected {nneutrinos}, got {len(value)}."
        )
    elif np.any(value.value < 0): # check if any mass is negative
        raise ValueError("invalid (negative) neutrino mass encountered.")
    ...
```

(b) Retrieved Context Step: `FLRW.m_nu` (step_2) from `astropy/cosmology/flrw/base.py`

Listing 4.1: Example of procedurally similar steps. Both steps implement the same guard-clause pattern, validating numerical inputs against domain-specific constraints and raising `ValueError` on violation, yet share low lexical overlap between their descriptions ("temperature/scale/Quantity" vs. "neutrino mass/shape/negative").

similarity. Because successful repository-level code generation requires both procedural guidance and repository-specific knowledge, ProjAgent complements procedural retrieval with conventional lexical and semantic retrieval to capture project APIs, variables, and structural dependencies [194, 109]. Finally, ProjAgent incorporates a static-analysis feedback loop that iteratively refines generated code to improve correctness. We evaluate ProjAgent through the following research questions:

- **RQ1:** To what extent does ProjAgent improve repository-level code generation performance compared to existing retrieval-augmented generation baselines?
- **RQ2:** How effectively does projection similarity identify procedurally related context for repository-level code generation?
- **RQ3:** What is the contribution of each retrieval component to ProjAgent's overall performance?

Our contributions are as follows:

- We introduce *procedural similarity* as a retrieval dimension for repository-level code generation, complementing existing lexical and semantic retrieval signals.
- We leverage LLM hidden-state projections to represent procedural similarity, enabling the retrieval of procedurally related context across naming and domain boundaries.
- We propose ProjAgent, a repository-level code generation system that uses an agentic workflow to identify procedural context and combines it with lexical and semantic retrieval for broader context coverage.
- We conduct a comprehensive evaluation of ProjAgent on repository-level code-generation benchmarks, comparing it against existing retrieval-augmented generation baselines and analyzing the contributions of its retrieval components.

The rest of the paper is structured as follows. Section 4.2 reviews related work. Section 4.3 introduces the technique for extracting LLM's reasoning. Section 4.4 describes the details of the ProjAgent system. Section 4.5 presents the experimental setup for answering each research question. Section 4.6 reports the results, and Section 4.7 discusses the threat to validity.

4.2 Related Work

Repository-Level Code Generation and Context Retrieval: Repository-level code generation extends traditional code generation by requiring models to generate code that is consistent with cross-file dependencies, project-specific APIs, and repository conventions. Benchmarks such as DevEval [109], RepoExec [103], and REPOCOD [114] have highlighted the importance of retrieving relevant repository context for accurate code generation. Consequently, a large body of work has focused on improving repository context retrieval.

Early retrieval-augmented approaches primarily relied on lexical and semantic similarity. RepoCoder [194] introduced an iterative retrieve-then-generate framework based on BM25 and dense retrieval, establishing retrieval-augmented generation as a standard paradigm for repository-level code generation. Subsequent work explored richer repository context beyond textual similarity. CatCoder [140] emphasized type-related information, including API definitions and class hierarchies, while RepoExec [103] demonstrated that carefully selected context can outperform providing the entire repository. More recent methods have incorporated structural information to improve retrieval. DraCo [38] and GraphCoder [121] leveraged program dependency structures to capture cross-file relationships, whereas RepoHyper [144] and CodeRAG [198] combined multiple complementary retrieval signals, including sparse retrieval, dense embeddings, graph representations, and dataflow analysis.

Several recent approaches have also moved beyond retrieving entire functions. Alliance-Coder [61] represents repository functions using natural-language descriptions and retrieves context by matching implementation steps to these descriptions. Hydra [102] further demonstrated that retrieval based primarily on surface similarity can introduce misleading context that degrades generation quality. These findings suggest that effective repository-level code generation depends not only on retrieving semantically relevant code but also on retrieving context that is useful for implementing the target function.

Our work extends this line of research by introducing *procedural similarity* as a complementary retrieval signal. Unlike existing approaches that primarily rely on lexical, semantic, structural, or functional similarity, ProjAgent retrieves functions that implement similar computational procedures, enabling the discovery of useful context even when functions differ substantially in naming, APIs, implementation details, or application domains.

Agentic Workflows for Repository-Level Code Generation: Recent work has increasingly adopted agentic workflows to improve repository-level code generation by allowing LLMs to actively explore repositories, retrieve context, and iteratively refine generated code.

CodeAgent [197] introduced an autonomous framework equipped with repository exploration tools, including document retrieval and symbol search. TENET [76] incorporated repository navigation into a test-driven development loop, enabling agents to iteratively retrieve context and repair generated code using execution feedback. SpecAgent [126] further decomposed repository exploration across specialized agents responsible for context retrieval, planning, and implementation.

Although these systems demonstrate the benefits of agentic workflows for repository understanding and iterative code generation, their retrieval strategies primarily rely on repository navigation, structural dependencies, or surface-level relevance. In contrast, ProjAgent employs an agentic workflow to identify and validate procedurally similar functions before expanding retrieval using hidden-state projection similarity. Rather than introducing another repository exploration strategy, our work introduces a new retrieval signal that complements existing lexical and semantic retrieval methods within an agentic generation framework.

4.3 Background

Measuring procedural similarity between functional steps requires representations that capture implementation behavior beyond surface-level lexical similarity. To obtain such representations, we build on the reasoning-subspace projection technique proposed by Hu et al. [75]. We adopt this approach because it isolates hidden-state components associated with reasoning-related behavior, while suppressing linguistic variation. Prior studies have shown that LLM hidden states encode rich semantic and structural information about source code [213, 111], making them suitable representations for downstream software engineering tasks.

Hu et al. [75] observed that the hidden-state space of an LLM can be approximately decomposed into a semantic subspace and a reasoning-related subspace. The semantic subspace

encodes the linguistic expression, and the reasoning subspace encodes the LLM's internal reasoning process. To identify these subspaces, they perform Singular Value Decomposition (SVD) [58] on the parameter matrix of the unembedding layer $W_{unemb} = U\Sigma V^\top$, which maps hidden states to output-token logits. Since the unembedding layer encodes both semantic and reasoning information, the right singular vectors (columns of V) corresponding to the dominant singular values span the semantic subspace, while the remaining vectors span the reasoning subspace $\mathcal{S}_{Reasoning}$. The projection of the hidden state h_l onto the reasoning subspace is computed as:

$$\text{proj}_R(h_l) = V_R^\top \cdot h_l \tag{4.1}$$

where h_l is the hidden state at the last layer prior to the final layer normalization, extracted from the response tokens. The resulting representation emphasizes reasoning-related information while suppressing surface-level linguistic variation. We reuse the subspace construction settings from Hu et al. [75], except for the energy threshold, which we increase from 0.95 to 0.98. In the HARP paper, Hu et al. [75] reported that the reasoning subspace should occupy approximately 5% of the hidden state dimension, calculated as $\frac{d-k}{d} \times 100\%$. Here, d is the model's hidden-state dimensionality (model-dependent), and k is determined by the energy threshold. Thus, the goal is to adjust the energy threshold so that the resulting k could yield approximately 5% of the hidden state dimension for our model. We empirically tested different energy threshold values and found that a threshold of 0.98 could yield 5% of the hidden state dimension.

While Hu et al. [75] use reasoning-subspace projections as features for hallucination detection, we use them to represent the underlying logic of functional steps and measure procedural similarity through cosine similarity between projected representations. However, we found that the cosine similarity between any pair of projections remained uniformly high even for unrelated functional steps. This behavior is consistent with the anisotropy phenomenon

observed in neural representations [44, 172], where vectors are dominated by a common direction that is largely independent of their content.

To mitigate this issue, we introduce a Principal Component Analysis (PCA) [141] based debiasing step that is not present in the original method of Hu et al. [75]. PCA identifies the dominant direction within a set of representations by computing the mean μ and the first principal component PC_1, which together summarize the common variation shared across all representations. The dominant direction in the representations can then be removed by subtracting both from each representation. Specifically, we computed μ and PC_1 from a representative set of projected representations as:

$$\mu = \frac{1}{N}\sum_{i=1}^{N} \mathrm{proj}_R(h_l), \quad \mathrm{PC}_1 = \mathrm{PCA}_1\left(\{\mathrm{proj}_R(h_l)\}_{i=1}^{N}\right) \tag{4.2}$$

To debias each projected representation, we first mean-center it by subtracting μ and then remove its component along the first principal component PC_1:

$$\tilde{h} = (\mathrm{proj}_R(h_l) - \mu) - \left((\mathrm{proj}_R(h_l) - \mu)^\top \cdot \mathrm{PC}_1\right) * \mathrm{PC}_1 \tag{4.3}$$

Removing the dominant shared direction reduces anisotropy and substantially improves the discriminative ability of cosine similarity when comparing procedural representations.

4.4 ProjAgent Pipeline

ProjAgent is a repository-level code generation system that retrieves procedurally and semantically similar contexts from a repository to provide an LLM with repository-specific guidance for code generation. Given a coding question consisting of an incomplete Python function and its docstring, along with the corresponding repository, ProjAgent generates the completed function. We use Qwen2.5-Coder-14B-Instruct (Qwen2.5) as the backbone model

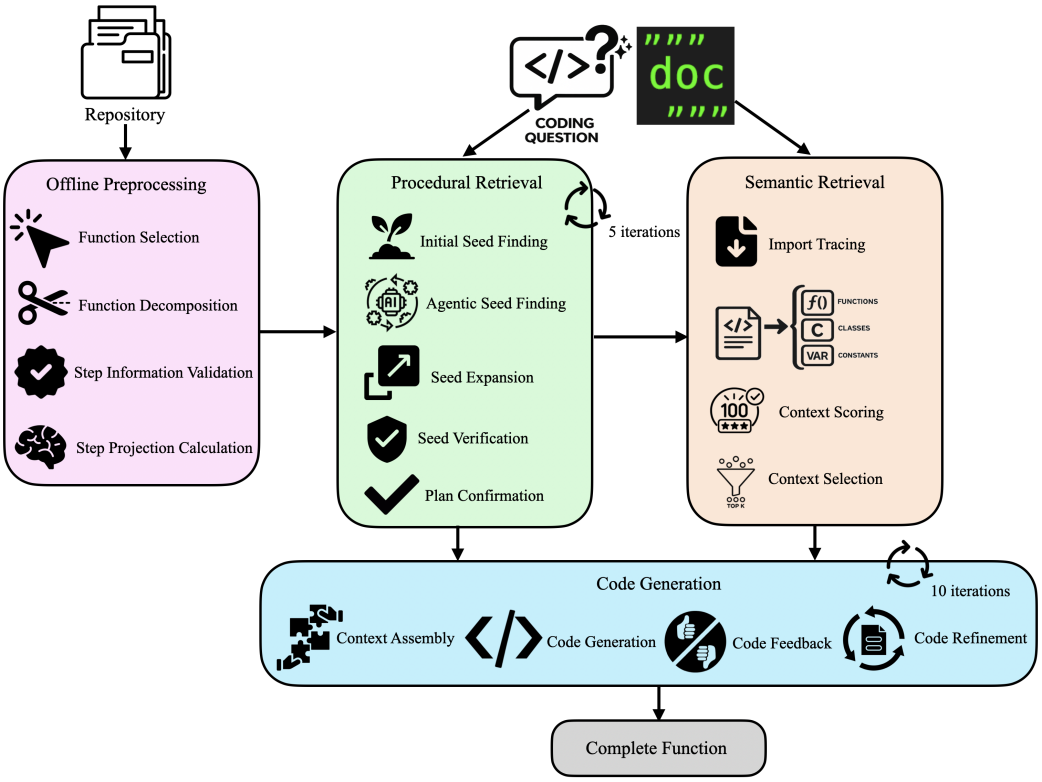


Figure 4.1: Overview of the ProjAgent pipeline. The offline preprocessing stage decomposes the functions in the repository and collects the projection. The procedural and semantic retrieval stages retrieve procedural and semantic contexts. The code generation stage generates code using the retrieved contexts and static analysis feedback.

for its strong code-generation capabilities [79]. Throughout this paper, the function to be completed is referred to as the *target function*, while all other functions in the repository are referred to as *context functions*, which serve as candidate contexts for completing the *target function*.

Figure 4.1 provides an overview of the ProjAgent pipeline. In the offline preprocessing stage, a subset of repository context functions is decomposed into procedural steps (context steps), and a projection is computed for each step (Section 4.4.1). During procedural retrieval, an agentic workflow identifies procedurally similar contexts (Section 4.4.2). Semantic retrieval then retrieves semantically related contexts, including APIs, classes, and utility functions (Section 4.4.3). Finally, the code generation stage completes the target function using the retrieved procedural and semantic contexts and iteratively refines the generated code based

on static analysis feedback (Section 4.4.4).

4.4.1 Offline Preprocessing

Function Selection: The offline preprocessing stage processes repository context functions to construct the projection representations used during procedural retrieval. However, large repositories may contain thousands of context functions (e.g., Astropy contains over 16,000), making preprocessing every function computationally expensive. Moreover, Hu et al. [74] showed that most repository files are irrelevant to a given coding question. Therefore, ProjAgent first selects a subset of context functions for preprocessing.

We first identify all Python files in the repository. For each file f, we compute two scores: $N_{target}(f)$, the number of target functions in f, and $N_f(f)$, the total number of functions in f. Files are ranked in descending order by $N_{target}(f)$, with ties broken by $N_f(f)$. This ranking prioritizes files containing benchmark target functions and favors those with richer functional content, consistent with prior work showing that functions within the same file often provide useful contextual information for code generation [109]. We then select the top 20 files, extract all functions, remove the target functions (i.e., the benchmark coding questions), and retain the remaining context functions. The subsequent preprocessing steps operate only on these selected context functions.

Function Decomposition: After selecting the context functions, we decompose each into a sequence of logical steps, where a logical step represents a distinct operation within the implementation (e.g., input validation). We provide the LLM with the function signature, docstring, and body, and instruct it to identify these logical steps. For each step, the model produces (1) a natural language description capturing its intent and (2) the corresponding code snippet implementing that step. We then validate each generated step by verifying that its code snippet exists in the original function and accurately corresponds to its description, as detailed next.

Step Information Validation: To verify that a generated context step is grounded in its context function, we propose a three-stage validation process. First, we check whether the function docstring contains the step description. The docstring is extracted using `ast.get_docstring` [147], and both the docstring and step description are normalized following prior work [107, 11] by converting text to lowercase, replacing newlines with spaces, removing punctuation, and collapsing consecutive whitespace. If the normalized step description appears as a substring of the normalized docstring, the step is considered valid. Otherwise, the second stage computes the ROUGE-L score [115] between the normalized texts. Following prior work [88], a step is accepted if its ROUGE-L score exceeds 0.7. If this stage also fails, we prompt the LLM with the function body and docstring as the source text, and the step description as the claim, asking it to determine whether the claim is supported by the source text. The model's judgment serves as the final validation decision. The resulting steps are referred to as *valid context steps*.

Next, we validate the code snippet associated with each valid context step. We first verify that every line in the snippet exists in the original function body. If any line is missing, we prompt the LLM with the step description and function body, asking it to re-identify the corresponding code snippet. We then encode the step description and code snippet using *google/embeddinggemma-300m* [175], which is trained on both natural language and code and provides an explicit retrieval mode suitable for cross-modal similarity [175]. The cosine similarity between the two embeddings is computed, and following prior work [31], the snippet is considered valid if the similarity exceeds 0.75. Only context steps with valid code snippets are retained for the remainder of the pipeline; we refer to them as *grounded context steps*.

Step Projection Construction: For each grounded context step, we construct a reasoning representation using the reasoning subspace projection technique described in Section 4.3. Specifically, we prompt the LLM with the step description and the step snippet from the

Algorithm 1: Stable μ and PC_1 Estimation

Input: Ranked repository files, initial projections $\mathcal{P}$
Output: Stable μ, PC_1
$(\mu_{\text{old}},\ \mathrm{PC}_{1,\text{old}}) \leftarrow$ BUILD_MU_$PC_1(\mathcal{P})$;
is_stable $\leftarrow$ False, $i \leftarrow 0$;
while not is_stable **do**
  Select next batch of files;
  Decompose functions in the new batch into steps;
  Compute those steps' projections;
  $\mathcal{P} \leftarrow \mathcal{P} \cup$ new projections;
  $(\mu_{\text{curr}},\ \mathrm{PC}_{1,\text{curr}}) \leftarrow$ BUILD_MU_$PC_1(\mathcal{P})$;
  if CHECKSTABILITY$(\mu_{\text{old}}, \mu_{\text{curr}})$ and CHECKSTABILITY$(\mathrm{PC}_{1,\text{old}}, \mathrm{PC}_{1,\text{curr}})$ **then**
    is_stable $\leftarrow$ True;
  end
  if *all functions have processed* **then**
    is_stable $\leftarrow$ True
  end
  $\mu_{\text{old}} \leftarrow \mu_{\text{curr}}$, $\mathrm{PC}_{1,\text{old}} \leftarrow \mathrm{PC}_{1,\text{curr}}$;
  $i \leftarrow i + 1$;
end
$(\mu,\ \mathrm{PC}_1) \leftarrow$ BUILD_MU_$PC_1(\mathcal{P})$;
return $\mu,\ \mathrm{PC}_1$;

grounded context step, and instruct it to reason about its implementation. We then extract the hidden states from the last layer for all response tokens and project them onto the reasoning subspace, which encodes the model's internal reasoning process. The last layer is chosen because Hu et al. [75] showed that shallow-layer vectors are primarily represented in the semantic subspace, whereas deep-layer vectors are more concentrated in the reasoning subspace. We selected response tokens rather than prompt tokens because the prompt shares the common instruction and system message. When the step description of the grounded context step is short, the projection obtained from the prompt tokens contains mostly the model's reasoning about the instruction and the system message. We refer to this set of projections as P.

As mentioned in Section 4.3, the projections in P suffered from the anisotropy phenomenon [44, 172]. Therefore, we design an incremental stabilization algorithm to compute a stable μ

and PC_1, shown in Algorithm 1. The algorithm takes the ranked repository files (obtained from Section 4.4.1) and P as input. We then compute the μ and PC_1 from the projection set P. After that, we select a new batch of 5 Python files from the ranked repository files (ones not selected in Section 4.4.1). Following the same procedure, we extract all context functions within those 5 files (Line 4), decompose them into context steps (see Section 4.4.1), and compute their projections (Line 6). We then merge those new projections into P (Line 7). We compute μ_{curr} and $PC_{1,curr}$ from P (line 8). Following prior work [192], we check the stability of μ (line 9) using the relative change formula:

$$\frac{\|\mu_{\text{curr}} - \mu_{\text{old}}\|}{\|\mu_{\text{old}}\|} \leq \tau_{\mu} \tag{4.4}$$

with a threshold of 1e-3 and the stability of PC_1 (line 9) using the absolute cosine similarity formula:

$$|\cos(\text{PC}_{1,\text{old}},\ \text{PC}_{1,\text{curr}})| \geq \tau_{\text{PC}_1} \tag{4.5}$$

with a threshold of 0.99. Both the relative change formula and the absolute cosine similarity formula measure how much the current estimates of μ and PC_1 have changed relative to earlier iterations. The while loop stops when either both μ and PC_1 are stable (line 10), or all context functions in the repositories have been processed (line 13).

After obtaining stable μ and PC_1, we process each projection in P to remove the shared dominant direction. Specifically, we apply mean pooling [151] to the projection as we extract hidden states for all response tokens. Finally, we apply Equation 4.3 in Section 4.3 to the resulting projection.

4.4.2 Procedural Context Retrieval

Figure 4.1 illustrates the procedural context retrieval pipeline. ProjAgent first decomposes the target function into logical steps (target steps) using its signature and docstring, and computes a projection for each step. It then retrieves procedurally similar contexts through an iterative workflow. Each iteration begins by retrieving an initial set of context steps (Section 4.4.2), followed by an agentic search to identify additional candidates (Section 4.4.2). The retrieved contexts are then expanded (Section 4.4.2), validated (Section 4.4.2), and evaluated using an LLM-based plan confirmation step (Section 4.4.2) to determine whether sufficient procedural context has been collected for each target step. The process repeats until all target steps have sufficient context or a maximum of five iterations is reached.

Initial Seed Finding: ProjAgent first identifies an initial set of procedurally similar context steps using projection similarity. We refer to each context step from the processed context functions as a *candidate step*. For each target step, we compute the cosine similarity between its projection and the projection of every candidate step, and rank the candidates accordingly. Candidate steps with projection similarity below 0.75 are discarded. We selected this threshold through empirical calibration on five randomly selected target steps from the *Astropy* repository by evaluating thresholds of 0.7, 0.75, and 0.8, and found that 0.75 best separated procedurally similar candidates from dissimilar ones. To further reduce computational cost, we retain only the top 20 remaining candidates. The retained candidates are then verified using an LLM. Given the target step description, candidate step description, and corresponding code snippet, the model determines whether the candidate performs the same computational operation as the target step, regardless of its application domain. This verification complements projection similarity, which captures coarse procedural similarity but may not fully distinguish semantically equivalent computational patterns. Candidate steps confirmed by the LLM are retained as *initial seeds*.

Agentic Seed Finding: While projection similarity retrieves procedurally similar contexts

from previously decomposed functions, it cannot discover functions that have not yet been processed. To address this limitation, ProjAgent employs an agentic repository exploration strategy in which an LLM navigates the repository using a set of tools, progressively accumulating knowledge to identify promising context functions.

The agent is initialized with the target step description, target file location, repository structure, and the initial seeds identified in Section 4.4.2. At each iteration, it reviews the interaction history, reasons about which function is most likely to contain a procedurally similar step, determines what additional information is needed, and selects the next tool call. To explore the repository, the agent is provided with the *ls*, *execute_bash*, *read_func*, *read_lines*, and *search_func* tools [200]. Once it identifies a promising function, it invokes *propose_func*. To encourage sufficient exploration, the agent is instructed to avoid proposing functions prematurely. Table 4.1 summarizes all available tools.

When a function is proposed, it is decomposed following Section 4.4.1 if it has not been processed previously. We then verify whether any of its steps are procedurally similar to the target step through a four-stage filtering process. First, candidate steps with projection similarity below 0.75 are discarded. Second, candidate steps whose projection similarity to any existing seed exceeds 0.95 are removed to avoid redundancy. Third, the remaining candidates are evaluated by the LLM for procedural similarity. If none are accepted, the agent receives feedback on the reasons for rejection, along with the previously rejected functions, allowing it to refine subsequent exploration. If multiple candidates are accepted, we retain the one with the highest projection similarity while ensuring it remains sufficiently distinct from the existing seeds. The accepted step is then added to the initial seed set.

Repository exploration may require many iterations, causing the conversation history to exceed the model's context window and potentially leading to out-of-memory (OOM) failures [182]. To mitigate this issue, we adopt an incremental context-compaction strategy inspired by recent context engineering work [26, 209]. The conversation history is partitioned into

Tool	Description
`ls`	List directory contents
`execute_bash`	Pattern-based grep search
`read_func`	Read a specific function
`read_lines`	Read specific line ranges
`search_func`	Locate functions by name or class
`propose_func`	Propose a candidate function as seed

Table 4.1: Tools available to the ProjAgent agentic seed finder (See 4.4.2).

four components: the system prompt, the initial user message, exploration messages, and proposal messages. The system prompt, initial user message, and proposal messages are preserved in full because they define the task and record the agent's decisions. Only the exploration history is compressed using an LLM-based summarizer [139], which preserves all explored files, directories, and referenced context functions while reducing the token footprint.

Seed Expansion: While the identified seeds in the *initial seeds* are valuable, they alone may not be sufficient to cover all implementation details needed by the model. However, exhaustively running the agentic workflow across the entire repository is too expensive. Therefore, we use projection similarity to efficiently filter a large pool of processed context steps into a small set of promising candidate steps without LLM calls. We then expand the procedurally similar contexts from the remaining promising candidate steps. For each context function, we determine whether it contains a procedurally similar step to the target step by computing the cosine similarity between the projection of each seed step and that of each context step (candidate step) in the context function. After that, we retain only context steps whose cosine similarity exceeds the threshold of 0.65 with at least 2 seeds. We determined these two thresholds empirically (See details in Section 4.5.3). Requiring agreement across multiple seeds improves robustness, as a candidate step that scores highly on only one seed may be coincidentally similar rather than genuinely procedurally similar. We then collect the remaining context steps and refer to them as the *expand context steps*. Finally, we run an LLM-based verification process on *expand context steps* (covered next).

Our rationale is that the remaining context steps still contain false positives, particularly for generic steps such as updating a state value or validating an input, which can superficially resemble unrelated context steps.

Verification: To control costs, we only run the LLM-based verification process for the top 30 context steps in the *expand context steps*. For each target step, the confirmed seeds are presented to the LLM as calibration examples illustrating what counts as procedurally similar, alongside the problem statement and the context steps to be verified, batched dynamically within a token budget. The LLM independently determines whether each context step is procedurally similar to the target step. We add the verified context steps into the *initial seeds* to form the retrieved context for the target step.

Plan Confirmation: After collecting procedurally similar context, we present the step descriptions of each retrieved context and the target step descriptions to the LLM and instruct it to indicate whether the provided context is sufficient to implement the target step. We designate a target step with sufficient contexts for implementation as the confirmed step, and unconfirmed steps otherwise. For each unconfirmed step, the LLM also produces a refined search query that identifies the missing context required to correctly implement it. We update the unconfirmed step's description by adding the search query at the beginning. We then recalculate its projection using the updated step description. Finally, confirmed steps are saved with their retrieved contexts, while unconfirmed steps proceed to the next iteration, where the updated projection is used to search for new procedurally similar contexts.

4.4.3 Semantic Retrieval

While procedurally similar contexts can teach the LLM about the target steps' logic, the LLM still needs contexts that are semantically related to the retrieved procedurally similar contexts to produce a correct implementation. Those contexts serve as the ingredients needed to turn the logic into the correct implementation. This is essential because procedurally similar

contexts are retrieved based on logical similarity to the target step and may come from anywhere in the repository (as shown in Figure 4.1), making them potentially inaccessible within the target function's scope.

Starting from the import statements in the Python file containing the target function (target file), ProjAgent finds all accessible files by transitively tracing them using *ast.Import* and *ast.ImportFrom* [147]. ProjAgent then constructs a symbol pool containing functions, classes, and constants from those accessible files, representing the complete set of symbols the target function can directly access at runtime.

To select semantic contexts that help the model implement the target step, ProjAgent builds an enriched query by combining each target step's description with the step description of each retrieved procedural context. To score each symbol, we compute a score v that captures both lexical and semantic relevance between the enriched query q and symbol s:

$$v = 0.5 \cdot \widehat{\text{BM25}}(q, s) + 0.5 \cdot \widehat{\text{dense}}(q, s) \tag{4.6}$$

where BM25(q, s) is the BM25 score, representing lexical relevance, and dense(q, s) is the embedding similarity calculated using *sentence-transformers/all-mpnet-base-v2* model [95], representing semantic relevance. The $\hat{\cdot}$ operation enforces the BM25 and dense scores to be in $[0, 1]$. Lastly, we rank the symbols by v in descending order and select the top 20 symbols as the semantic contexts for each target step.

4.4.4 Code Generation With Feedback

After retrieving procedural and semantic contexts, ProjAgent formats them separately to maximize target-step coverage within the prompt budget. Each procedural context includes a natural-language step description and its corresponding code snippet, while each semantic

context includes the retrieved code symbols. For each target step, ProjAgent prioritizes confirmed procedural contexts when available and otherwise uses the best unconfirmed context. The same strategy is applied to semantic contexts. All contexts are explicitly labeled as confirmed or unconfirmed to indicate their reliability. Once each target step is covered by at least one procedural and one semantic context, any remaining prompt budget is filled with additional confirmed contexts, prioritizing procedural contexts before semantic ones. ProjAgent then constructs the generation prompt using the problem statement, target-step descriptions, and formatted procedural and semantic contexts, and instructs the LLM to generate the implementation.

ProjAgent further refines the generated code using static-analysis feedback. It first performs an AST-based syntax check [147]; if a syntax error is found, the error message is returned to the LLM. Otherwise, ProjAgent applies four semantic consistency checks inspired by prior work [35]: (1) method call checking verifies that invoked methods are accessible from the current class hierarchy and called with the correct number of arguments; (2) field access checking verifies that referenced fields are defined in the current class hierarchy; (3) variable-method checking infers the receiver type at a call site and verifies that the invoked method exists in the corresponding type hierarchy; and (4) standalone function call checking verifies that called functions resolve to accessible functions and are invoked with the correct number of arguments.

Because Python's dynamic typing prevents complete static resolution, the feedback loop is conservative: an issue is reported only when it can be confirmed. ProjAgent then constructs a refinement prompt containing the previous implementation, problem statement, target-step descriptions, formatted contexts, and static-analysis feedback, and asks the LLM to revise the code. This loop continues until no feedback is produced or the maximum number of iterations (10) is reached.

4.5 Experimental Setup

4.5.1 Benchmark and Metric

To measure the effectiveness of ProjAgent (RQ1) and its components (RQ3), we evaluated ProjAgent on REPOCOD [114], a benchmark for repository-level code generation consisting of 980 problems drawn from 11 real-world repositories. Each coding problem requires generating a complete target function given the function signature and docstring. We evaluated the generated code using the test cases provided in the REPOCOD dataset and reported the Pass@1 metric. To control LLM's non-determinism, we used greedy decoding [161] by setting temperature 0 and a maximum of 4096 output tokens for code generation.

4.5.2 Baselines

We compare ProjAgent against four baselines: dense retrieval (DENSE), sparse retrieval (SPARSE), same-file retrieval (SAME_FILE), and SpecAgent [126]. DENSE, SPARSE, and SAME_FILE were selected because they represent complementary repository retrieval strategies, and we follow their implementation as described in REPOCOD [114]. For DENSE, we use the *sentence-transformers/all-mpnet-base-v2* embedding model [95]. We also compare against SpecAgent, the current state-of-the-art repository-level code generation system [126]. As its replication package is unavailable, we have reimplemented the approach using the methodology described in the paper and the prompts provided in its appendix. To ensure a fair comparison, all baselines use Qwen2.5-Coder-14B-Instruct as the backbone model.

4.5.3 Projection Effectiveness Setup

To evaluate how effectively projection similarity identifies procedurally related contexts (RQ2), we designed a three-step process: (1) construct a labeled dataset, (2) evaluate different projection similarity configurations, and (3) compare their ability to distinguish procedurally similar from dissimilar contexts.

Dataset Construction: We constructed the dataset using all target functions from the *Astropy* repository in REPOCOD [114]. For each target step, we executed the initial seed finding, agentic seed finding, and seed expansion stages (Section 4.4.2). For every target-context step pair, we recorded the projection similarity scores between the context step and each expanded seed of the target step. The resulting dataset maps each (target step, context step) pair to its corresponding list of similarity scores. Overall, it contains 9,598 target-context step pairs spanning all 85 target functions and every context function in the repository.

Dataset Labeling: We labeled each of the 9,598 pairs using Claude Sonnet 4.6 [15]. For each pair, Claude received the target and context step descriptions, along with the context step snippet, and determined whether the context step was procedurally similar to the target step. To clarify the notion of procedural similarity, we employed contrastive in-context learning [54] using one manually selected positive and one negative example from the *Seaborn* repository.

To validate the automated labels, two authors independently annotated a statistically significant random sample of 370 pairs (95% confidence level, 5% margin of error). Each author assessed whether the context step exhibited a computational pattern similar to that of the target step, based on the step descriptions and the code snippet. Disagreements were resolved through discussion. The resulting inter-rater agreement was $\kappa = 0.86$, indicating almost perfect agreement [100]. Comparing the consensus labels with Claude's predictions yielded $\kappa = 0.82$, demonstrating strong agreement.

Evaluation: For each projection similarity configuration, defined by the similarity threshold τ and agreement threshold k, we partitioned the dataset into a *promoted group* and a *leftover group*. A target-context pair was assigned to the promoted group if at least k projection similarity scores exceeded τ; otherwise, it was assigned to the leftover group. We evaluated six (τ, k) configurations and report precision, recall, and F1 score for both groups using the manually validated labels as ground truth. We restricted the analysis to a single repository

for two reasons: manually validating all candidate pairs across the 11 REPOCOD repositories is infeasible, and selecting in-context examples from the evaluation repository could introduce information leakage that biases Claude's judgments.

Experimental Scope: Due to the computational cost of repository-level code generation, all experiments were conducted using a single backbone model (Qwen2.5-Coder-14B-Instruct). A complete evaluation of one system requires generating and executing code for all 980 REPOCOD tasks across 11 repositories. Beyond the main evaluation, this work includes multiple baselines, six projection-similarity configurations for RQ2, and four system variants for the RQ3 ablation study, resulting in several thousand end-to-end code generation and test-execution runs. Repeating the entire experimental pipeline across multiple backbone models would increase the computational cost proportionally while substantially extending the experimental time. We therefore focus on a single, strong open-source code model to enable a comprehensive evaluation of the proposed retrieval methodology under a consistent experimental setting.

4.5.4 Ablation Setup for RQ3

To understand the effectiveness of each retrieval component, we conducted an ablation study on all 85 coding problems from the *Astropy* repository in REPOCOD. We evaluated three variations: **w/o procedural**, which removes the contexts retrieved from the procedurally similar signal; **w/o semantic**, which removes the contexts retrieved from the semantic retrieval signal; and **w/o feedback loop**, which uses contexts collected from both procedurally similar and semantic retrieval signals, but disables the static analysis feedback loop. In each variant, only the specified component was removed, while all others remained identical to those in the full pipeline. We report the Pass@1 score for each variant.

4.6 Result

Answer to RQ1: Table 4.2 summarizes the code generation performance of ProjAgent and the baselines on the REPOCOD benchmark. ProjAgent achieves the highest Pass@1 among all methods. Compared to SPARSE, DENSE, and SAME_FILE, ProjAgent improves Pass@1 by 12.31%, while outperforming SpecAgent by 6.62%.

We posit that these gains stem from the quality of the retrieved procedural and semantic contexts. However, because repository exploration is constrained by a fixed search budget, the agent may fail to discover highly relevant procedural contexts. To assess the impact of search coverage, we conducted an additional experiment on the *Astropy* repository. Instead of decomposing context functions on demand, we decomposed every context function offline and applied the initial seed finding, seed expansion, verification, and plan confirmation pipeline (Section 4.4.2) to all of them. Under this full-search setting, ProjAgent achieved a Pass@1 of 25.12%, compared to 21.16% under the budget-constrained agentic search, representing a 3.96% improvement. These results indicate that search coverage is a key factor in ProjAgent's effectiveness and suggest that future work should focus on improving repository exploration under fixed search budgets.

RQ1 Key Takeaway: Procedural retrieval substantially (**6.62%**) improves repository-level code generation. While ProjAgent outperforms all baselines, additional gains are observed as repository search coverage increases, suggesting that efficient exploration of large repositories is a primary bottleneck for further performance improvements.

Answer to RQ2: Table 4.3 reports the number of pairs per group (promoted or leftover), the precision, recall, and F1 score of each group across the six configurations, with the best configuration (Configuration B) shown in bold. From the precision, recall, and F1 score for the *leftover group*, we observed that these scores remained consistently high across all configurations. This indicates that the projection similarity-based method reliably excludes

Method	Pass@1
Sparse	26.58%
Dense	28.83%
Same_File	14.98%
SpecAgent	34.52%
ProjAgent (Ours)	**41.14%**

Table 4.2: Pass@1 (%) on REPOCOD for all methods using Qwen2.5-Coder-14B-Instruct.

		Promoted				Leftover			
Cfg	**(τ, k)**	**#Pairs**	**Prec.**	**Recall**	**F1**	**#Pairs**	**Prec.**	**Recall**	**F1**
A	(0.60, 2)	2,036	3.6%	36.7%	0.065	7,562	98.3%	79.1%	0.877
B	**(0.65, 2)**	**997**	**5.4%**	**27.1%**	**0.090**	**8,601**	**98.3%**	**90.0%**	**0.940**
C	(0.65, 3)	677	5.8%	19.6%	0.089	8,921	98.2%	93.2%	0.956
D	(0.70, 2)	410	6.6%	13.6%	0.089	9,188	98.1%	95.9%	0.970
E	(0.70, 3)	294	6.1%	9.0%	0.073	9,304	98.1%	97.1%	0.976
F	(0.80, 2)	45	6.7%	1.5%	0.025	9,553	97.9%	99.6%	0.987

Table 4.3: Precision, recall, and F1 score of the promoted and leftover groups across six threshold configurations. Config B (bolded) achieves the highest Promoted-F1, indicating the best precision-recall balance.

procedurally dissimilar context steps regardless of the chosen configuration. On the other hand, the precision, recall, and F1 score of the promoted group across all six configurations are low. We suspect these low scores are due to two factors. First, not every target step has a procedurally similar context step in the repository, because some target steps implement highly specific logic unique to their functions. To validate this, we took the retrieved procedurally similar contexts for each coding question across all repositories and calculated the ratio of query steps that have at least one procedurally similar context. We found that only 35.1% of query steps have at least one genuine procedurally similar context step. Second, many target and context steps describe generic operations, such as validating input and checking input. These steps share similar computational patterns, but only at a level of abstraction that is too coarse to be procedurally similar. This leads to false positives in the promoted group, directly lowering its precision. We further observed that precision

increases monotonically as the thresholds τ and k become stricter, from 3.6% (configuration A) to 6.7% (configuration F), while recall drops sharply from 36.7% (configuration A) to 1.5% (configuration F). This indicates that the stricter configuration discards the vast majority of procedurally similar context steps. Since the precision, recall, and F1 scores of the *leftover group* remain consistently high, we select the best configuration based on the F1 score of the *promoted group* and found that configuration B (τ of 0.65 and agreement k of 2) performs the best. The low precision, recall, and F1 score in the *promoted group* also justifies the verification step described in Section 4.4.2, as it filters out the false positives in the promoted group and improves the quality of the retrieved procedurally similar contexts.

Overall, these results suggest that projection similarity is most effective as a *candidate generation* mechanism rather than a standalone procedural similarity classifier. While the promoted group contains many false positives, the consistently high precision of the leftover group indicates that projection similarity effectively eliminates a large portion of procedurally unrelated context steps from consideration. This substantially reduces the search space while retaining most candidate procedural matches, allowing the subsequent verification stage to focus on a much smaller set of potentially relevant contexts. The complementary strengths of projection similarity and verification, therefore, motivate the two-stage retrieval design adopted by ProjAgent.

RQ2 Key Takeaway: The projection similarity-based method reliably filters out procedurally dissimilar context steps from a large pool of context functions (leftover precision $\geq$ 97.9% across all configurations). **Configuration B** (τ=0.65, $k \geq 2$) achieves the best F1 score and is adopted as the default configuration of ProjAgent.

Answer to RQ3: Table 4.4 presents the results of the ablation study. Removing procedurally or semantically similar contexts results in a substantial drop in Pass@1, confirming that both types of contexts are critical to ProjAgent's performance. When dropping the procedurally similar contexts, the performance drops from 41.14% to 25.76%. On the other hand,

Method	**Pass@1**
ProjAgent (All Components)	**41.14%**
wo/procedural	25.76%
wo/symbolic	32.29%
wo/feedback loop	40.29%

Table 4.4: Ablation on Different Components of ProjAgent

when semantically similar contexts are dropped, performance drops from 41.14% to 32.29%. Comparing *wo/procedural* with *wo/semantic*, removing procedurally similar contexts results in greater performance degradation for ProjAgent.

We attribute this difference to the distinct roles of the two retrieval mechanisms. Procedural retrieval provides examples of implementation procedures that guide how the target functionality should be implemented, whereas semantic retrieval supplies repository-specific symbols, such as functions, classes, and constants, that enable those procedures to be translated into executable code. Together, these retrieval signals provide both procedural guidance and repository-specific knowledge, explaining why combining them yields substantially better performance than using either retrieval signal alone.

In contrast, removing the static-analysis feedback loop produces only a modest decrease in performance. We attribute this to the conservative design of the feedback mechanism, which reports only errors that can be determined with high confidence. Consequently, many ambiguous issues, such as those involving dynamically inferred argument types, are intentionally ignored to avoid introducing incorrect feedback. While this design minimizes false corrections, it also limits opportunities for iterative refinement. Future work could investigate richer feedback mechanisms, including runtime execution feedback and lightweight type inference, to improve the effectiveness of the refinement stage.

RQ3 Key Takeaway: Procedural and semantic retrieval provide complementary information for repository-level code generation, and removing either substantially reduces Pass@1. Within the current ProjAgent pipeline, removing procedural retrieval results in the larger performance degradation, while the conservative static-analysis feedback loop provides a modest but measurable improvement.

4.7 Threat To Validity

External Validity: We evaluated ProjAgent on REPOCOD, a repository-level code generation benchmark consisting exclusively of Python repositories. Consequently, our findings may not generalize to repositories written in other programming languages or to other repository-level code generation benchmarks. In addition, ProjAgent was implemented using Qwen2.5-Coder-14B-Instruct as the backbone model. The effectiveness of procedural retrieval may vary across LLMs with different hidden-state representations, potentially affecting the quality of reasoning-subspace projections. We limited our evaluation to a single backbone model because repository-level code generation is computationally expensive: a single evaluation requires generating and executing code for all 980 tasks in REPOCOD. Our study also includes four baselines, six projection-similarity configurations for RQ2, and four variants for the RQ3 ablation study. Extending this evaluation to multiple backbone models would increase the computational cost proportionally. Finally, the ablation study for RQ3 was conducted on a single repository (*Astropy*); therefore, the relative contribution of individual retrieval components may differ for repositories with different structures, domains, or dependency characteristics. In addition, our experiments used a fixed retrieval budget and context window. The effectiveness of procedural retrieval may differ under alternative retrieval budgets or models with substantially larger context windows.

Internal Validity: In RQ2, we used Claude as an LLM judge to label candidate procedural relationships because manually annotating the large number of (target function step, con-

text function step) pairs required for evaluation was prohibitively time-consuming. Although LLM judges may produce incorrect labels, we mitigated this threat by having two researchers independently manually inspect a statistically significant random sample of 370 pairs and verify the assigned labels. In addition, ProjAgent's performance is sensitive to several configuration choices, including the projection-similarity threshold, seed-agreement count, and the number of iterations used for agentic seed discovery. While we conducted ablation studies to assess the contributions of individual components, we did not exhaustively explore all possible parameter configurations due to the substantial computational cost of evaluating repository-level code generation systems. Consequently, different parameter settings may lead to different performance outcomes.

4.8 Conclusion

In this paper, we introduced *procedural similarity* as a complementary retrieval signal for repository-level code generation and presented ProjAgent, a retrieval-augmented generation system that integrates procedural, lexical, and semantic retrieval within an agentic workflow, together with a static-analysis feedback loop. Unlike existing retrieval approaches that primarily rely on lexical, semantic, or structural similarity, ProjAgent identifies repository functions that implement similar computational procedures, thereby providing LLMs with context useful for solving the target programming task.

Our evaluation on REPOCOD demonstrates that ProjAgent achieves state-of-the-art repository-level code generation performance, outperforming existing baselines. We further show that reasoning-subspace projections provide an effective representation for identifying procedurally related functions and that combining procedural retrieval with conventional retrieval signals consistently improves generation performance. Ablation studies further confirm that each component of the retrieval pipeline contributes to the system's overall effectiveness.

More broadly, our results suggest that retrieval quality depends not only on identifying code that is lexically or semantically similar, but also on retrieving code that follows similar computational procedures. We hope this work motivates future research on behavior-aware retrieval methods for software engineering tasks and on richer representations of procedural knowledge in large language models. All code, data, and replication materials are available in the supplementary material [33].

Chapter 5

Conclusion

Source code is central to software development as it is the primary artifact developers create, maintain, and evolve over a software system's lifetime. In the past few years, LLMs have demonstrated strong capabilities on a wide range of code-related tasks, with the potential to fundamentally change how software is developed. However, our understanding of LLMs' limitations on these tasks remains limited. In this dissertation, I found that many LLM misclassifications in clone detection, a representative code-understanding task, stem from a mismatch between the model's understanding of what constitutes clone code and the definition intended by human annotators. By manually analyzing these misclassifications, I identified eight distinct mistakes underlying this mismatch and derived corresponding natural-language guidance to correct each one, improving LLM performance on clone detection by up to 10.81% in F1 score. These findings in Chapter 2 demonstrate that LLMs' clone detection mistakes stem from misunderstandings of clone code definitions rather than fundamental limitations in the model's ability to understand code snippets.

While Chapter 2 focuses on LLMs' mistakes in code understanding, Chapter 3 turns to code generation, another major class of LLM-based code-related tasks. Although LLMs have

shown strong capabilities in generating code, they remain prone to errors. Prior studies of these errors were largely limited to earlier, less capable models evaluated on small, Python-only benchmarks. To fill this gap, I examined code generated by two state-of-the-art LLMs across two large-scale, multilingual benchmarks spanning 2,268 standalone coding questions in Python and Java. This analysis surfaced 17 types of non-syntactic mistakes across seven categories, ten of which had not been previously documented, along with six underlying reasons behind them. I also found that top-performing models such as GPT-4 can identify mistakes in generated code along with the reasons behind them with high precision and F1 score, respectively. Taken together, these findings in Chapter 3 substantially broaden the known taxonomy of LLM code generation mistakes, providing developers with practical guidance for using LLMs more effectively and a clearer picture of why such mistakes occur in the first place.

While Chapter 3 characterizes LLMs' mistakes in code generation and their underlying reasons, it does not propose a way to improve LLM performance on the task. Chapter 4 aims to do so by focusing on repository-level code generation rather than standalone generation, since modern LLMs already perform well at the latter. Furthermore, repository-level code generation better reflects real-world software development, where new code is written within the context of an existing codebase. In this dissertation, I identified a new source of context for repository-level code generation, which I call procedurally similar context. This refers to code that shares similar reasoning steps or logic with the target function, regardless of its lexical or semantic similarity. Existing repository-level code generation approaches that rely on lexical or semantic similarity to retrieve context overlook this signal entirely. I further found that projection similarity, computed by projecting an LLM's hidden states into a reasoning subspace, is highly effective at filtering out procedurally dissimilar context.

Building on this finding, I introduced ProjAgent, a system that retrieves procedurally similar context through agentic search over the codebase. To complement the retrieved procedurally

similar contexts, ProjAgent further retrieves semantically similar context, which serves as the ingredients needed to implement the logic conveyed by the procedurally similar contexts. Our evaluation shows that ProjAgent achieves a Pass@1 score up to 41.14%. Together, these results in Chapter 4 establish procedurally similar context as a valuable, previously overlooked signal for repository-level code generation. These results suggest that future work on context selection for LLM-based code generation should look beyond lexical and semantic similarity.

Bibliography

[1] chatgpt, howpublished = https://chat.openai.com/, note = Accessed: 2023-09-23.

[2] figshare-replication package, howpublished = https://figshare.com/s/4e885d949f72dd4b1840, note = Accessed: 2023-9-28.

[3] Midjourney, howpublished = https://www.midjourney.com/home/?callbackurl=%2fapp%2f, note = Accessed: 2023-09-23.

[4] OpenAI-models, howpublished = https://platform.openai.com/docs/models/gpt-3-5, note = Accessed: 2023-09-23.

[5] PoolC-5-fold-clone-detection-600k-5fold, howpublished = https://huggingface.co/datasets/poolc/5-fold-clone-detection-600k-5fold, note = Accessed: 2023-09-23.

[6] surveyMonkey-sample size calculator, howpublished = https://www.surveymonkey.com/mp/sample-size-calculator/, note = Accessed: 2023-09-23.

[7] wasiahmad-AVATAR, howpublished = https://github.com/wasiahmad/avatar, note = Accessed: 2023-09-23.

[8] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[9] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha. Black box fairness testing of machine learning models. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 625–635, 2019.

[10] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.

[11] A. A. Aliero, S. A. Bashir, H. O. Aliyu, A. G. Tafida, U. K. Bashar, and M. D. Nasiru. Systematic review on text normalization techniques and its approach to non-standard words. 2023.

[12] H. Alkaissi and S. I. McFarlane. Artificial hallucinations in chatgpt: implications in scientific writing. *Cureus*, 15(2), 2023.

[13] M. Allamanis. The adverse effects of code duplication in machine learning models of code. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 143–153, 2019.

[14] M. Altherwi. *An Empirical Study of Programming Languages in Open-Source Software Projects based on Mining Software Repositories*. PhD thesis, University of Southampton, 2021.

[15] Anthropic. Claude sonnet 4.6, 2025. Accessed: 2025.

[16] A. Anthropic. The claude 3 model family: Opus, sonnet, haiku. *Claude-3 Model Card*, 1:1, 2024.

[17] AST — Abstract Syntax Tree. https://docs.python.org/3/library/ast.html.

[18] B. Athiwaratkun, S. K. Gouda, Z. Wang, X. Li, Y. Tian, M. Tan, W. U. Ahmad, S. Wang, Q. Sun, M. Shang, et al. Multi-lingual evaluation of code generation models. *arXiv preprint arXiv:2210.14868*, 2022.

[19] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[20] awesome-chatgpt-prompts. https://github.com/f/awesome-chatgpt-prompts, 2023. 2023.

[21] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE, 1998.

[22] B. R. Beaulieu-Jones, M. T. Berrigan, S. Shah, J. S. Marwaha, S.-L. Lai, and G. A. Brat. Evaluating capabilities of large language models: Performance of gpt-4 on surgical knowledge assessments. *Surgery*, 2024.

[23] D. W. Binkley and K. B. Gallagher. Program slicing. *Advances in computers*, 43:1–50, 1996.

[24] S. Biswas and H. Rajan. Do the machine learning models on a crowd sourced platform exhibit bias? an empirical study on model fairness. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 642–653, 2020.

[25] J. M. Bland and D. G. Altman. Multiple significance tests: the bonferroni method. *Bmj*, 310(6973):170, 1995.

[26] N. D. Bui. Building effective ai coding agents for the terminal: Scaffolding, harness, context engineering, and lessons learned. *arXiv preprint arXiv:2603.05344*, 2026.

[27] D. Castelberg and F. Flury. *LLM Assisted Development*. PhD thesis, OST Ostschweizer Fachhochschule, 2024.

[28] J. Chakraborty, K. Peng, and T. Menzies. Making fair ml software using trustworthy explanation. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1229–1233, 2020.

[29] J. Chakraborty, T. Xia, F. M. Fahid, and T. Menzies. Software engineering for fairness: A case study with hyperparameter optimization. *arXiv preprint arXiv:1905.05786*, 2019.

[30] Chang, Yupeng and Wang, Xu and Wang, Jindong and Wu, Yuan and Yang, Linyi and Zhu, Kaijie and Chen, Hao and Yi, Xiaoyuan and Wang, Cunxiang and Wang, Yidong and others. A survey on evaluation of large language models. *ACM Transactions on Intelligent Systems and Technology*, 15(3):1–45, 2024.

[31] J. Chen, J. Zhang, J. He, Y. Tang, and W. Zou. Bridging the programming language gap: Constructing a multilingual shared semantic space through ast unification and graph matching. *arXiv preprint arXiv:2605.07788*, 2026.

[32] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[33] Q. H. Chen, A. Imani, and I. Ahmed. Supplementary Material for ProjAgent, 2026.

[34] X. Chen, R. A. Chi, X. Wang, and D. Zhou. Premise order matters in reasoning with large language models. *arXiv preprint arXiv:2402.08939*, 2024.

[35] X. Chen, M. Lin, N. Schärli, and D. Zhou. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pages 8746–8825, 2024.

[36] Y. Chen, R. Dios, A. Mili, L. Wu, and K. Wang. An empirical study of programming language trends. *IEEE software*, 22(3):72–79, 2005.

[37] Chen, Yuyan and Fu, Qiang and Yuan, Yichen and Wen, Zhihao and Fan, Ge and Liu, Dayiheng and Zhang, Dongmei and Li, Zhixu and Xiao, Yanghua. Hallucination detection: Robustly discerning reliable answers in large language models. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*, pages 245–255, 2023.

[38] W. Cheng, Y. Wu, and W. Hu. Dataflow-guided retrieval augmentation for repository-level code completion. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7957–7977, 2024.

[39] A. Chowdhery, S. Narang, J. Devlin, M. Bosma, G. Mishra, A. Roberts, P. Barham, H. W. Chung, C. Sutton, S. Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[40] R. B. Dell, S. Holleran, and R. Ramakrishnan. Sample size determination. *ILAR journal*, 43(4):207–213, 2002.

[41] Y. Dong, X. Jiang, Z. Jin, and G. Li. Self-collaboration code generation via chatgpt. *arXiv preprint arXiv:2304.07590*, 2023.

[42] S. Dou, J. Shan, H. Jia, W. Deng, Z. Xi, W. He, Y. Wu, T. Gui, Y. Liu, and X. Huang. Towards understanding the capability of large language models on code clone detection: A survey. arxiv 2023. *arXiv preprint arXiv:2308.01191*, 2023.

[43] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*, pages 109–118. IEEE, 1999.

[44] K. Ethayarajh. How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*, pages 55–65, 2019.

[45] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[46] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1469–1481. IEEE, 2023.

[47] C. e. a. Fang. Functional code clone detection with syntax and semantics fusion learning. In *ISSTA 2020*, page 516–527. ACM, 2020.

[48] S. Feng and C. Chen. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[49] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[50] J. Fereday and E. Muir-Cochrane. Demonstrating rigor using thematic analysis: A hybrid approach of inductive and deductive coding and theme development. *International journal of qualitative methods*, 5(1):80–92, 2006.

[51] figshare replication package. https://figshare.com/s/10e27d42bf537f6321f7.

[52] Fortune. Over 25% of Google's code is written by AI, Sundar Pichai says. `https://fortune.com/2024/10/30/googles-code-ai-sundar-pichai/`, October 2024.

[53] S. Gao, X.-C. Wen, C. Gao, W. Wang, H. Zhang, and M. R. Lyu. What makes good in-context demonstrations for code intelligence tasks with llms? In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 761–773. IEEE, 2023.

[54] X. Gao and K. Das. Customizing language model responses with contrastive in-context learning. In *Proceedings of the aaai conference on artificial intelligence*, volume 38, pages 18039–18046, 2024.

[55] M. Geng, S. Wang, D. Dong, H. Wang, G. Li, Z. Jin, X. Mao, and X. Liao. Large language models are few-shot summarizers: Multi-intent comment generation via in-context learning. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[56] J. Gesi, J. Li, and I. Ahmed. An empirical examination of the impact of bias on just-in-time defect prediction. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, 2021.

[57] F. Gilardi, M. Alizadeh, and M. Kubli. Chatgpt outperforms crowd workers for text-annotation tasks. *Proceedings of the National Academy of Sciences*, 120(30):e2305016120, 2023.

[58] G. H. Golub and C. F. Van Loan. *Matrix computations*. JHU press, 2013.

[59] T. Goyal, J. J. Li, and G. Durrett. News summarization and evaluation in the era of gpt-3. *arXiv preprint arXiv:2209.12356*, 2022.

[60] M. Grootendorst. Bertopic: Neural topic modeling with a class-based tf-idf procedure, 2022.

[61] W. Gu, J. Chen, Y. Wang, T. Jiang, X. Li, M. Liu, X. Liu, Y. Ma, and Z. Zheng. What to retrieve for effective retrieval-augmented code generation? an empirical study and beyond. *arXiv preprint arXiv:2503.20589*, 2025.

[62] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[63] Q. Guo, J. Cao, X. Xie, S. Liu, X. Li, B. Chen, and X. Peng. Exploring the potential of chatgpt in automated code refinement: An empirical study. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[64] M. Gupta, R. Agrawal, V. Shrivastava, A. K. Choudhary, M. Kumar, and S. Arya. A comprehensive study on the adoption of large language models across industries. *Enterprise Development and Microfinance*, 36(3s):222–231, 2026.

[65] K. L. Gwet. Computing inter-rater reliability and its variance in the presence of high agreement. *British Journal of Mathematical and Statistical Psychology*, 61(1):29–48, 2008.

[66] S. N. Halpin. Inter-coder agreement in qualitative coding: Considerations for its use. *American Journal of Qualitative Research*, 8(3):23–43, 2024.

[67] L. G. Halsey, D. Curran-Everett, S. L. Vowler, and G. B. Drummond. The fickle p value generates irreproducible results. *Nature methods*, 12(3):179–185, 2015.

[68] K. Hansen, K. S. Johannsen, and A. Larsen. Recommendations for negotiated agreements. *Environment and Planning C: Government and Policy*, 20(1):19–37, 2002.

[69] F. Harel-Canada, L. Wang, M. A. Gulzar, Q. Gu, and M. Kim. Is neuron coverage a meaningful measure for testing deep neural networks? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–862, 2020.

[70] J. He, M. Rungta, D. Koleczek, A. Sekhon, F. X. Wang, and S. Hasan. Does prompt formatting have any impact on llm performance? *arXiv preprint arXiv:2411.10541*, 2024.

[71] A. Hendy, M. Abdelrehim, A. Sharaf, V. Raunak, M. Gabr, H. Matsushita, Y. J. Kim, M. Afify, and H. H. Awadalla. How good are gpt models at machine translation? a comprehensive evaluation. *arXiv preprint arXiv:2302.09210*, 2023.

[72] J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. de Las Casas, L. A. Hendricks, J. Welbl, A. Clark, et al. Training compute-optimal large language models (2022). *arXiv preprint arXiv:2203.15556*, 2022.

[73] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology*, 33(8):1–79, 2024.

[74] C. Hu, W. Zeng, Y. Shi, B. Shen, and X. Gu. In line with context: Repository-level code generation via context inlining. *arXiv preprint arXiv:2601.00376*, 2026.

[75] J. Hu, G. Tu, S. Cheng, J. Li, J. Wang, R. Chen, Z. Zhou, and D. Shan. Harp: Hallucination detection via reasoning subspace projection. *arXiv preprint arXiv:2509.11536*, 2025.

[76] Y. Hu, N. Jiang, S. Liang, Y. Wu, and L. Tan. Tenet: Leveraging tests beyond validation for code generation. *arXiv preprint arXiv:2509.24148*, 2025.

[77] Y. Hu, Q. Zhou, Q. Chen, X. Li, L. Liu, D. Zhang, A. Kachroo, T. Oz, and O. Tripp. Qualityflow: An agentic workflow for program synthesis controlled by llm quality checks. *arXiv preprint arXiv:2501.17167*, 2025.

[78] D. Huang, Q. Bu, J. M. Zhang, M. Luck, and H. Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

[79] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024.

[80] IBM Global AI Adoption Index 2022. https://www.ibm.com/watson/resources/ai-adoption, 2022. 2022.

[81] D. Jain, P. Kumar, S. Gehman, X. Zhou, T. Hartvigsen, and M. Sap. Polyglotoxicityprompts: Multilingual evaluation of neural toxic degeneration in large language models. *arXiv preprint arXiv:2405.09373*, 2024.

[82] N. Jain, R. Kwiatkowski, B. Ray, M. K. Ramanathan, and V. Kumar. On mitigating code llm hallucinations with api documentation. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 237–248. IEEE, 2025.

[83] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[84] Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html.

[85] Pure Python Java parser and tools. https://github.com/c2nes/javalang.

[86] K. Jesse, T. Ahmed, P. T. Devanbu, and E. Morgan. Large language models and simple, stupid bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 563–575. IEEE, 2023.

[87] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *International Conference on Learning Representations*, volume 2024, pages 54107–54157, 2024.

[88] M. Joo and H. Cho. Cleanse: Uncertainty estimation approach using clustering-based semantic consistency in llms. In *Proceedings of the Fourth Workshop on Generation, Evaluation and Metrics (GEM²)*, pages 291–301, 2025.

[89] N. Karanikolas, E. Manga, N. Samaridi, E. Tousidou, and M. Vassilakopoulos. Large language models versus natural language understanding and generation. In *Proceedings of the 27th Pan-Hellenic Conference on Progress in Computing and Informatics*, pages 278–290, 2023.

[90] S. H. Khandkar. Open coding. *University of Calgary*, 23(2009):2009, 2009.

[91] R. Khoury, A. R. Avila, J. Brunelle, and B. M. Camara. How secure is code generated by chatgpt? *arXiv preprint arXiv:2304.09655*, 2023.

[92] T. K. Kim. T test as a parametric statistic. *Korean journal of anesthesiology*, 68(6):540–546, 2015.

[93] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213, 2022.

[94] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *International static analysis symposium*, pages 40–56. Springer, 2001.

[95] D. Kosenko and D. Zharikova. Krgp: Knowledge-based response generation with persona. *Journal of Mathematical Sciences*, 285(1):49–59, 2024.

[96] B. Kou, S. Chen, Z. Wang, L. Ma, and Z. Tianyi. Do large language models pay similar attention like human programmers when generating code? *Proceedings of the ACM on Software Engineering*, 1(FSE):2261–2284, 2024.

[97] A. Kovrigin, A. Eliseeva, Y. Zharov, and T. Bryksin. On the importance of reasoning for context retrieval in repository-level code editing. *arXiv preprint arXiv:2406.04464*, 2024.

[98] J. Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.

[99] J. Krinke and C. Ragkhitwetsagul. Bigclonebench considered harmful for machine learning. In *2022 IEEE 16th International Workshop on Software Clones (IWSC)*, pages 1–7. IEEE, 2022.

[100] J. R. Landis and G. G. Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[101] J. R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, 1993.

[102] M. Le-Anh, H. Nguyen, K. A. Tran, N. L. Hai, L. N. Van, N. D. Bui, and B. Le. Do not treat code as natural language: Implications for repository-level code generation and beyond. *arXiv preprint arXiv:2602.11671*, 2026.

[103] N. Le Hai, D. M. Nguyen, and N. D. Bui. Repoexec: Evaluate code generation with a repository-level executable benchmark. *arXiv preprint arXiv:2406.11927*, 2024.

[104] K. Lee, D. Ippolito, A. Nystrom, C. Zhang, D. Eck, C. Callison-Burch, and N. Carlini. Deduplicating training data makes language models better. *arXiv preprint arXiv:2107.06499*, 2021.

[105] M. Lee. A mathematical investigation of hallucination and creativity in gpt models. *Mathematics*, 11(10):2320, 2023.

[106] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 2005.

[107] G. Leusch, N. Ueffing, D. Vilar, and H. Ney. Preprocessing and normalization for automatic evaluation of machine translation. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization*, pages 17–24, 2005.

[108] H. Li, H. Kwon, J. Kwon, and H. Lee. Clorifi: software vulnerability discovery using code clone verification. *Concurrency and Computation: Practice and Experience*, 28(6):1900–1917, 2016.

[109] J. Li, G. Li, Y. Zhao, Y. Li, H. Liu, H. Zhu, L. Wang, K. Liu, Z. Fang, L. Wang, et al. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 3603–3614, 2024.

[110] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin. Acecoder: An effective prompting technique specialized in code generation. *ACM Transactions on Software Engineering and Methodology*, 2024.

[111] K. Li, A. K. Hopkins, D. Bau, F. Viégas, H. Pfister, and M. Wattenberg. Emergent world representations: Exploring a sequence model trained on a synthetic task. *arXiv preprint arXiv:2210.13382*, 2022.

[112] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

[113] S. Liang, Y. Hu, N. Jiang, and L. Tan. Can language models replace programmers? repocod says' not yet'. *arXiv preprint arXiv:2410.21647*, 2024.

[114] S. Liang, N. Jiang, Y. Hu, and L. Tan. Can language models replace programmers for coding? repocod says 'not yet'. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24698–24717, 2025.

[115] C.-Y. Lin. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out*, pages 74–81, 2004.

[116] F. Lin, D. J. Kim, et al. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852*, 2024.

[117] C. Liu, X. Bao, H. Zhang, N. Zhang, H. Hu, X. Zhang, and M. Yan. Improving chatgpt prompt for code generation. *arXiv preprint arXiv:2305.08360*, 2023.

[118] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

[119] H. Liu, W. Xue, Y. Chen, D. Chen, X. Zhao, K. Wang, L. Hou, R. Li, and W. Peng. A survey on hallucination in large vision-language models. *arXiv preprint arXiv:2402.00253*, 2024.

[120] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36:21558–21572, 2023.

[121] W. Liu, A. Yu, D. Zan, B. Shen, W. Zhang, H. Zhao, Z. Jin, and Q. Wang. Graphcoder: Enhancing repository-level code completion via code context graph-based retrieval and language model. *arXiv preprint arXiv:2406.07003*, 2024.

[122] Y. Liu, T. Le-Cong, R. Widyasari, C. Tantithamthavorn, L. Li, X.-B. D. Le, and D. Lo. Refining chatgpt-generated code: Characterizing and mitigating code quality issues. *arXiv preprint arXiv:2307.12596*, 2023.

[123] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek. Déjàvu: a map of code duplicates on github. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[124] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

[125] L. Luo, Z. Lin, Y. Liu, L. Shu, Y. Zhu, J. Shang, and L. Meng. Critique ability of large language models. *arXiv preprint arXiv:2310.04815*, 2023.

[126] G. Ma, A. Koul, Q. Chen, Y. Wu, S. Kuhar, Y. Yu, A. Sengupta, V. Kumar, and M. K. Ramanathan. Specagent: A speculative retrieval and forecasting agent for code completion. *arXiv preprint arXiv:2510.17925*, 2025.

[127] W. Ma, S. Liu, Z. Lin, W. Wang, Q. Hu, Y. Liu, C. Zhang, L. Nie, L. Li, and Y. Liu. Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138*, 2023.

[128] K. M. MacQueen, E. McLellan, K. Kay, and B. Milstein. Codebook development for team-based qualitative analysis. *Cam Journal*, 10(2):31–36, 1998.

[129] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

[130] L. McInnes, J. Healy, and S. Astels. hdbscan: Hierarchical density based clustering. *J. Open Source Softw.*, 2(11):205, 2017.

[131] L. McInnes, J. Healy, and J. Melville. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426*, 2018.

[132] G. McKenna. Over 25% of google's code is now written by ai—and ceo sundar pichai says it's just the start. https://fortune.com/2024/10/30/googles-code-ai-sundar-pichai/. Accessed: 2025-02-10.

[133] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 1–18, 2013.

[134] R. Meyes, M. Lu, C. W. de Puiseau, and T. Meisen. Ablation studies in artificial neural networks. *arXiv preprint arXiv:1901.08644*, 2019.

[135] F. Mu, L. Shi, S. Wang, Z. Yu, B. Zhang, C. Wang, S. Liu, and Q. Wang. Clarifygpt: Empowering llm-based code generation with intention clarification. *arXiv preprint arXiv:2310.10996*, 2023.

[136] G. Orlanski, K. Xiao, X. Garcia, J. Hui, J. Howland, J. Malmaud, J. Austin, R. Singh, and M. Catasta. Measuring the impact of programming language distribution. In *International Conference on Machine Learning*, pages 26619–26645. PMLR, 2023.

[137] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

[138] S. Overflow. Ai| 2024 stack overflow developer survey. *URL https://survey. stackoverflow. co/2024/ai*, 2024.

[139] C. Packer, V. Fang, S. Patil, K. Lin, S. Wooders, and J. Gonzalez. Memgpt: towards llms as operating systems. 2023.

[140] Z. Pan, X. Hu, X. Xia, and X. Yang. Enhancing repository-level code generation with integrated contextual information. *arXiv preprint arXiv:2406.03283*, 2024.

[141] K. Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin philosophical magazine and journal of science*, 2(11):559–572, 1901.

[142] K. Peng, L. Ding, Q. Zhong, L. Shen, X. Liu, M. Zhang, Y. Ouyang, and D. Tao. Towards making the most of chatgpt for machine translation. *arXiv preprint arXiv:2303.13780*, 2023.

[143] E. Perez, S. Ringer, K. Lukosiute, K. Nguyen, E. Chen, S. Heiner, C. Pettit, C. Olsson, S. Kundu, S. Kadavath, et al. Discovering language model behaviors with model-written evaluations. In *Findings of the association for computational linguistics: ACL 2023*, pages 13387–13434, 2023.

[144] H. N. Phan, H. N. Phan, T. N. Nguyen, and N. D. Bui. Repohyper: Search-expand-refine on semantic graphs for repository-level code completion. In *2025 IEEE/ACM Second International Conference on AI Foundation Models and Software Engineering (Forge)*, pages 14–25. IEEE, 2025.

[145] Y. Poon, J. S. Y. Lee, et al. Few-shot question generation for reading comprehension. In *Proceedings of the 10th SIGHAN Workshop on Chinese Language Processing (SIGHAN-10)*, pages 21–27, 2024.

[146] pydoc — Documentation generator and online help system. https://docs.python.org/3/library/pydoc.html.

[147] Python Software Foundation. ast — abstract syntax trees, 2025. Accessed: 2025.

[148] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[149] C. Ragkhitwetsagul, J. Krinke, M. Paixao, G. Bianco, and R. Oliveto. Toxic code snippets on stack overflow. *IEEE Transactions on Software Engineering*, 47(3):560–581, 2019.

[150] V. Rawte, A. Sheth, and A. Das. A survey of hallucination in large foundation models. *arXiv preprint arXiv:2309.05922*, 2023.

[151] N. Reimers and I. Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084*, 2019.

[152] S. Robertson and H. Zaragoza. *The probabilistic relevance framework: BM25 and beyond*, volume 4. Now Publishers Inc, 2009.

[153] C. K. Roy and J. R. Cordy. A survey on software clone detection research. *Queen's School of computing TR*, 541(115):64–68, 2007.

[154] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[155] C. Sadowski, K. T. Stolee, and S. Elbaum. How developers search for code: a case study. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*, pages 191–201, 2015.

[156] N. Saini, S. Singh, et al. Code clones: Detection and management. *Procedia computer science*, 132:718–727, 2018.

[157] L. Schibelius, A. Ross, and A. Katz. An empirical study of programming languages specified in engineering job postings. In *2022 ASEE Annual Conference & Exposition*, 2022.

[158] A. Sergeyuk, Y. Golubev, T. Bryksin, and I. Ahmed. Using ai-based coding assistants in practice: State of affairs, perceptions, and ways forward. *arXiv preprint arXiv:2406.07765*, 2024.

[159] A. Shah, A. Chernova, E. Tomson, L. Porter, W. G. Griswold, and A. G. Soosai Raj. Students' use of github copilot for working with large code bases. In *Proceedings of the 56th ACM Technical Symposium on Computer Science Education V. 1*, pages 1050–1056, 2025.

[160] D. Song, Z. Zhou, Z. Wang, Y. Huang, S. Chen, B. Kou, L. Ma, and T. Zhang. An empirical study of code generation errors made by large language models. *MAPS '23, December 3, 2023, San Francisco, CA, USA*, 2023.

[161] Y. Song, G. Wang, S. Li, and B. Y. Lin. The good, the bad, and the greedy: Evaluation of LLMs should not ignore non-determinism. In L. Chiruzzo, A. Ritter, and L. Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4195–4206, Albuquerque, New Mexico, Apr. 2025. Association for Computational Linguistics.

[162] T. Sonnekalb, B. Gruner, C.-A. Brust, and P. Mäder. Generalizability of code clone detection on codebert. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–3, 2022.

[163] J. Spracklen, R. Wijewickrama, A. N. Sakib, A. Maiti, and B. Viswanath. We have a package for you! a comprehensive analysis of package hallucinations by code generating {LLMs}. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3687–3706, 2025.

[164] K. Stechly, M. Marquez, and S. Kambhampati. Gpt-4 doesn't know it's wrong: An analysis of iterative prompting for reasoning problems. *arXiv preprint arXiv:2310.12397*, 2023.

[165] B. Steenhoek, M. M. Rahman, M. K. Roy, M. S. Alam, H. Tong, S. Das, E. T. Barr, and W. Le. To err is machine: Vulnerability detection challenges llm reasoning. *arXiv preprint arXiv:2403.17218*, 2024.

[166] A. Strauss and J. Corbin. Basics of qualitative research techniques. 1998.

[167] M. Sudhamani and L. Rangarajan. Code clone detection based on order and content of control statements. In *2016 2nd International Conference on Contemporary Computing and Informatics (IC3I)*, pages 59–64. IEEE, 2016.

[168] J. Sun, C. Shaib, and B. C. Wallace. Evaluating the zero-shot robustness of instruction-tuned language models. *arXiv preprint arXiv:2306.11270*, 2023.

[169] Y. Tal, I. Magar, and R. Schwartz. Fewer errors, but more stereotypes? the effect of model size on gender bias. *arXiv preprint arXiv:2206.09860*, 2022.

[170] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol. Bugs in large language models generated code: An empirical study. *Empirical Software Engineering*, 30(3):1–48, 2025.

[171] Y. Tian, Z. Zhong, V. Ordonez, G. Kaiser, and B. Ray. Testing dnn image classifiers for confusion & bias errors. In *Proceedings of the acm/ieee 42nd international conference on software engineering*, pages 1122–1134, 2020.

[172] W. Timkey and M. Van Schijndel. All bark and no bite: Rogue dimensions in transformer language models obscure representational quality. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 4527–4546, 2021.

[173] TIOBE Index. https://www.tiobe.com/tiobe-index/.

[174] R. Tóth, T. Bisztray, and L. Erdodi. Llms in web-development: Evaluating llm-generated php code unveiling vulnerabilities and limitations. *arXiv preprint arXiv:2404.14459*, 2024.

[175] H. S. Vera, S. Dua, B. Zhang, D. Salz, R. Mullins, S. R. Panyam, S. Smoot, I. Naim, J. Zou, F. Chen, et al. Embeddinggemma: Powerful and lightweight text representations. *arXiv preprint arXiv:2509.20354*, 2025.

[176] J. Vijayan. Pervasive LLM Hallucinations Expand Code Developer Attack Surface. `https://www.darkreading.com/application-security/pervasive-llm-hallucinations-expand-code-developer-attack-surface`.

[177] R. Wang, J. Guo, C. Gao, G. Fan, C. Y. Chong, and X. Xia. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proceedings of the ACM on Software Engineering*, 2(ISSTA):1955–1977, 2025.

[178] W. Wang, G. Li, B. Ma, X. Xia, and Z. Jin. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 261–271. IEEE, 2020.

[179] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[180] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. arxiv 2022. *arXiv preprint arXiv:2203.11171*, 2022.

[181] Y. Wang, W. Wang, S. Joty, and S. C. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[182] Y. Wang, Z. Wang, Y. Shi, S. Chen, X. Wang, Y. Wang, B. Shen, L. Li, X. Gu, J. McAuley, et al. Context compression for llm agents: A survey of methods, failure modes, and evaluation. 2026.

[183] Y. Wang, Y. Ye, Y. Wu, W. Zhang, Y. Xue, and Y. Liu. Comparison and evaluation of clone detection techniques with different code representations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 332–344. IEEE, 2023.

[184] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837, 2022.

[185] C. S. Xia and L. Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831, 2024.

[186] B. Xu, J. Qian, X. Zhang, Z. Wu, and L. Chen. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes*, 30(2):1–36, 2005.

[187] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

[188] H. Xu, B. Liu, L. Shu, and P. S. Yu. Dombert: Domain-oriented language model for aspect-based sentiment analysis. *arXiv preprint arXiv:2004.13816*, 2020.

[189] G. Yang, Y. Zhou, W. Yang, T. Yue, X. Chen, and T. Chen. How important are good method names in neural code generation. *A Model Robustness Perspective. ArXiv abs/2211.15844*, 2022.

[190] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

[191] D. Yu, Q. Yang, X. Chen, J. Chen, and Y. Xu. Graph-based code semantics learning for efficient semantic code clone detection. *Information and Software Technology*, 156:107130, 2023.

[192] X. Yu, Y. Xin, Y. Zhang, W. Zhang, C. Liu, H. Zhao, C. Liu, X. Hu, Z. Qiao, H. Tang, et al. Modality gap-driven subspace alignment training paradigm for multimodal large language models. *arXiv preprint arXiv:2602.07026*, 2026.

[193] A. Zamfiroiu, D. Vasile, and D. Savu. Chatgpt–a systematic review of published research papers. *Informatica Economica*, 27(1):5–16, 2023.

[194] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. Chen. Repocoder: Repository-level code completion through iterative retrieval and generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 2471–2484, 2023.

[195] G. Zhang, X. Peng, Z. Xing, and W. Zhao. Cloning practices: Why developers clone and what can be changed. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 285–294. IEEE, 2012.

[196] H. Zhang and K. Sakurai. A survey of software clone detection from security perspective. *IEEE Access*, 9:48157–48173, 2021.

[197] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. *arXiv preprint arXiv:2401.07339*, 2024.

[198] S. Zhang, Y. Ding, S. Lian, S. Song, and H. Li. Coderag: Finding relevant and necessary knowledge for retrieval-augmented repository-level code completion. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 23289–23299, 2025.

[199] T. Zhang, B. Xu, F. Thung, S. A. Haryono, D. Lo, and L. Jiang. Sentiment analysis for software engineering: How far can pre-trained transformer models go? In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 70–80. IEEE, 2020.

[200] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 1592–1604, 2024.

[201] Z. Zhang, Y. Wang, C. Wang, J. Chen, and Z. Zheng. Llm hallucinations in practical code generation: Phenomena, mechanism, and mitigation. *arXiv preprint arXiv:2409.20550*, 2024.

[202] G. Zhao and J. Huang. Deepsim: deep learning code functional similarity. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 141–151, 2018.

[203] Q. Zheng, X. Xia, X. Zou, Y. Dong, S. Wang, Y. Xue, L. Shen, Z. Wang, A. Wang, Y. Li, et al. Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 5673–5684, 2023.

[204] L. Zhong, Z. Wang, and J. Shang. Ldb: A large language model debugger via verifying runtime execution step-by-step. *arXiv preprint arXiv:2402.16906*, 2024.

[205] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. Language agent tree search unifies reasoning acting and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

[206] K. Zhou, K. Ethayarajh, D. Card, and D. Jurafsky. Problems with cosine as a measure of embedding similarity for high frequency words. *arXiv preprint arXiv:2205.05092*, 2022.

[207] X. Zhou, K. Kim, B. Xu, J. Liu, D. Han, and D. Lo. The devil is in the tails: How long-tailed code distributions impact large language models. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 40–52. IEEE, 2023.

[208] Y. Zhou, A. I. Muresanu, Z. Han, K. Paster, S. Pitis, H. Chan, and J. Ba. Large language models are human-level prompt engineers. *arXiv preprint arXiv:2211.01910*, 2022.

[209] N. Zhu, H. Wang, J. Zhou, F. Chen, S. Zhang, G. Chen, C. Liu, J. Wu, W. Chen, X. Mou, et al. Semaclaw: A step towards general-purpose personal ai agents through harness engineering. *arXiv preprint arXiv:2604.11548*, 2026.

[210] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

[211] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations*, volume 2025, pages 66602–66656, 2025.

[212] A. Ziegler, E. Kalliamvakou, X. A. Li, A. Rice, D. Rifkin, S. Simister, G. Sittampalam, and E. Aftandilian. Productivity assessment of neural code completion. In *Proceedings of the 6th ACM SIGPLAN international symposium on machine programming*, pages 21–29, 2022.

[213] A. Zou, L. Phan, S. Chen, J. Campbell, P. Guo, R. Ren, A. Pan, X. Yin, M. Mazeika, A.-K. Dombrowski, et al. Representation engineering: A top-down approach to ai transparency. *arXiv preprint arXiv:2310.01405*, 2023.