TRUSTWORTHY PROGRAM GENERATION IN THE KEY PHASES OF THE SOFTWARE DEVELOPMENT LIFE CYCLE

by

CAI YUFAN
(B.S., Shanghai Jiao Tong University)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

in the

GRADUATE DIVISION

of the

NATIONAL UNIVERSITY OF SINGAPORE

2024

Supervisor:
Professor Dong Jin Song

Examiners:
Associate Professor Chin Wei Ngan
Associate Professor Mohan Gurusamy

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Cai Yufan

30 July 2024

Acknowledgments

First and foremost, I would like to express my heartfelt gratitude to my supervisor, Dr. Dong Jin Song, for his invaluable guidance, insightful suggestions, and constant encouragement throughout my doctoral journey. His unwavering support has been instrumental in helping me navigate numerous challenging research topics.

I am deeply thankful to my Thesis Advisory Committee members, Dr. Kan Min Yen and Dr. Mohan Gurusamy, for their thoughtful feedback and constructive comments, which have significantly shaped my research work.

I am profoundly grateful to Dr. Lin Yun for introducing me to the fascinating and dynamic field of code generation. My sincere thanks also go to Dr. Hou Zhe, who inspired my exploration of the domain of programming languages. I would like to extend special appreciation to Dr. David Miguel Sanan Baena, Dr. Sun Jun, Dr. Gong Yeyun, Dr. Liu Yang, and other collaborators for their invaluable contributions and joint efforts to enhance my research.

I am especially thankful to my fellow student, Liu Chenyan, for our close collaboration on code-related tasks. I am also deeply appreciative of the unwavering support and friendship of my friends: Dr. Zhang Hangsheng, Dr. Zhang Yedi, Dr. Xu Ming, Dr. Qi Binhang, Yang Xianglin, Zhang Yifan, Sun Changsheng, Lin Yuxi, Zuo Xinyue, and Liu Zhaoyu, whose presence has enriched my Ph.D. experience.

I am immensely grateful for the financial support provided by the National University of Singapore Graduate School for Integrative Sciences and Engineering Programme. Additionally, the School of Computing generously supported my participation in several international conferences, for which I am truly thankful. I am also honored to have received the Research Achievement Award and the President's Graduate Fellowship, which have been significant recognitions of my efforts.

Finally, I would like to extend my deepest gratitude to my family for their boundless love and unwavering support throughout these years. Their encouragement has been my greatest source of strength and inspiration.

Contents

Abstract

Trustworthy Program Generation in the Key Phases of the Software Development Life Cycle

by

Cai Yufan

Doctor of Philosophy in Computer Science

National University of Singapore

The advent of large language models (LLMs) has transformed software engineering by enabling the automatic generation of code from natural language specifications. Despite their remarkable capabilities, LLMs lack reliability guarantees and often produce code that may contain errors. This issue is exacerbated by their inherent tendency to hallucinate, introducing significant risks into the software development process. Furthermore, the process by which LLMs transform specifications into code is largely opaque, functioning as an uncontrolled black box. This lack of transparency hinders users' comprehension and trust of the generated outputs. Additionally, the absence of explainability in the generated code makes it challenging for programmers to understand and debug.

To address these challenges, this thesis introduces innovative methodologies to enhance trustworthiness in program generation across key phases of the software development life cycle. In this context, "trustworthiness" is defined by three core attributes: verifiability, explainability, and reliability. These attributes are essential for ensuring that the generated programs are functional and dependable.

This thesis addresses the key phases of the software development life cycle, including the software requirements, software design and verification, and software maintenance and evolution. Each chapter is devoted to one or more of these phases, presenting pioneering techniques that blend advancements in artificial intelligence and formal methods with robust software engineering practices. By doing so, this thesis seeks to establish a foundation for creating trustworthy systems that bridge the gap between automated code generation and practical software development.

The following introduces four key contributions:

Program Refinement. We build the LLM Aided Program Refinement (LLM4PR) approach to transform formal specifications into verified and reliable code through program refinement. It combines formal program refinement techniques with LLM-driven code generation. We integrate the formal verification system with LLM and address the reliability issues of the generated program.

Program Documentation. We propose the structure-centric and context-fusion code summarization tool called CProSum to improve the code comprehension capabilities of neural network models. By creating a contextual knowledge graph from existing codebases, CProSum achieves a more explainable and accurate code summary. It substantially enhances the quality of automated code comments and effectively narrows the divide between complex code scenarios and their summaries.

Program Evolution. We explore code evolution through intelligent edit recommendations that interact with developers to evolve the code project. The proposed tool CoEdPilot improves existing code by considering historical edits and contextual project information, employing LLMs to predict and apply precise code modifications accurately and reliably with the interaction of the users.

Program Adaptation. We introduce the adaptive model generation tool Adacom, a dynamic system that adjusts code and comment generation models in real-time. Adacom tailors its training dataset by focusing on the most beneficial data subsets, significantly boosting the efficacy of code and comment generators and demonstrating the practicality of model adaptation in various settings.

Above all, this thesis tackles the practical hurdles in program-related tasks. We combine program refinement and verification methods, retrieval augmented generation (RAG) methods, and real-time adaptation techniques to build a trustworthy, verifiable, explainable, and reliable program generation system. The contributions of this thesis have been published in several top conferences.

The experiment results show that our tools can generate a more trustworthy program than the state-of-the-art models. Through these contributions, this thesis addresses some practical challenges associated with automating code generation tasks and lays a robust foundation for developing trustworthy automated tools that assist developers throughout the software development life cycle.

Key words: Trustworthy AI, Program Generation, Program Refinement, Program Verification, Program Documentation, Program Evolution, Real-time Adaptation, Language Model, Neural Network, Machine Learning

List of Figures

List of Tables

Chapter 1

Introduction

The phenomenon of hallucination in large language models (LLMs)—where these models generate plausible yet incorrect responses—is a notoriously challenging issue. This issue is particularly acute in program generation, as LLMs often introduce subtle, hard-to-detect bugs. In this thesis, we address these challenges by developing a sophisticated, trustworthy program generation system. Our approach integrates several advanced techniques to enhance the reliability of the generated code in several key stages of the software development life cycle.

Firstly, we utilize formal methods to formalize the user requirements, build the formal specifications, and then refine them to the programs, ensuring they meet rigorous correctness standards. Secondly, we incorporate retrieval-augmented generation (RAG) methods, which leverage a knowledge graph to provide relevant information during the code comment generation process, increasing the explainability for further software product release and deployment. Additionally, we emphasize the role of user interaction, allowing for adjustments based on the user's feedback to evolve the software. Finally, our system is designed to adapt dynamically during the test time for domain-specific knowledge, increasing its explainability and reliability. Together, these methodologies form a comprehensive framework for creating software that not only functions correctly but also maintains a high level of trustworthiness with the users throughout the software development life cycle.

1.1 Motivation and Goals

Recent advancements in neural networks and language models have significantly improved the mathematics, reasoning, and programming capabilities of the deep learning models, as detailed in comprehensive surveys and studies [235, 174]. Notable industrial applications such as GPT-4 [155] and Copilot [70] have provided substantial assistance to programmers, even achieving performances that surpass the 50th percentile in programming competitions. Despite these successes, these models still face significant challenges, particularly the issue of *hallucination*, where they generate plausible yet factually incorrect information. User studies such as [196, 50] have highlighted that programmers struggle to trust and debug code generated by LLMs due to the non-transparent and uncontrollable nature of the generation processes. This lack of transparency is a major barrier to their reliability. Moreover, it has been demonstrated that a significant portion of responses from ChatGPT to programming queries are inaccurate, with over half containing errors [97]. Some mathematical analyses such as [212] have proven that *hallucinations* in LLMs are an unavoidable phenomenon, underscoring the need for ongoing research to mitigate these issues and enhance model reliability.

In response, this thesis presents a series of innovative methodologies designed to enhance the trustworthiness of program generation through language models. The methodologies integrate the latest advancements in rigorous formal methods, software engineering practices, and advanced artificial intelligence techniques to verify and explain the result of the language models. This integration approach ensures the reliable creation and modification of code throughout various essential stages of the software development life cycle. By employing this comprehensive strategy, we mitigate the risks associated with neural network outputs and establish a new benchmark for developing and maintaining reliable, explainable, and verifiable software systems. Specifically, we address four critical issues: (i) program refinement, (ii) program documentation, (iii) program evolution, and (iv) program adaptation.

Program Refinement The refinement calculus [140, 11, 28, 180, 193] is one kind of formal methods that stepwisely translate the formal specifications into programs. It entails correctness-preserving transformations of formal abstract specifications into executable programs guided by a calculus-based approach. However, this transformation, based

on program refinement calculus, is largely carried out manually, making it both time-intensive and prone to errors. Therefore, integrating LLMs and a proof assistant into the refinement process is a logical progression. The system further extends the goal of developing tool support for refinement calculus that is accessible to people without deep expertise in program refinement or theorem proving. Besides, many refinement steps require handling sub-derivations on sub-components of the program [26]. This segmented reasoning requires library-based reasoning, which can manage complexity by encapsulating functionality within functional abstractions. Traditional verification tools for program refinement are highly interactive and offer limited automation. This thesis presents an automated tool for program refinement called LLM4PR that combines the LLM with the formal program refinement methods to generate verified code step by step.

Program Documentation Program comprehension forms a critical aspect of software engineering. With the advent of neural network-powered technologies, source code summarization and documentation are increasingly aiding programmers in understanding their code projects. Traditionally, general natural language translation tasks include translating text from one language to another, assuming that the input contains sufficient information to deduce the output. Similarly, the prevalent one-code-to-one-comment approach treats code summarization as a translation problem, where the deep language models translate a piece of code into a piece of natural language comment. In contrast to conventional comment generators, our approach capitalizes on the insight that structural context can serve as a prompt to guide deep language models toward generating more accurate comments. This thesis implements an explainable documentation generator CProSum that transforms a code project into a code knowledge graph, effectively defining the scope of contextual information for a target piece of code. This method enhances the precision of the generated comments and aligns more closely with how developers interact with complex codebases. By leveraging structured contextual data, our system offers a more nuanced and reliable tool for program comprehension, setting a new benchmark for comment-generation technologies.

Program Evolution It has been empirically observed that program evolution with incremental code edits is commonplace. The latest approaches to code editing often frame the task as generating edits based on *known* relevant prior edits and contextual

information and then formatted them into prompts to train language models. However, the reality of practical code editing is often more complex. Several challenges complicate the effective implementation of code edits: (i) Unknown relevant edits: in many cases, the relevant prior edits might not be known upfront. An editing session may involve multiple relevant and irrelevant edits to the modified code. (ii) Complex ripple effects: inferring subsequent edits involves understanding the extensive ripple effects that an edit could have across the entire project. This complexity adds a layer of difficulty in predicting and managing changes effectively. (iii) Interactive edits: edits during a session are often not isolated but interact with one another. This interaction requires sophisticated modeling to ensure that the system can accurately interpret and respond to the dynamic nature of code modifications. Addressing these challenges requires a nuanced approach that goes beyond the current methods of training LLMs with static prompts. It necessitates the development of more adaptive and context-aware systems that can handle the intricate dynamics of real-world software development environments. This thesis proposes the CoEdPilot to recommend the next code edit with the interaction of the users based on the selection of prior edits and contextual information.

Program Adaptation Large-scale deep learning models are typically trained on extensive datasets comprising code-comment pairs, effectively handling program generation and summarization tasks. To manage and generalize from an exceedingly diverse training corpus, leading industry companies continue to increase the scale of these models—from millions to billions of neurons, as seen with GPT-3 and GPT-4. While scaling up to tens of billions of neurons proves beneficial, it comes with substantial organizational training and maintenance costs and a high barrier for everyday users. This thesis introduces a novel and lightweight approach, Adacom, for enhancing the performance of small-scale deep neural networks through on-the-fly model adaptation. It addresses potential distribution shifts that could affect model performance, ensuring that the trained model remains reliable under varied inputs. Our idea is based on the observation that smaller models often compromise their predictive accuracy for specific samples due to their limited scale. By focusing on dynamic model adaptation, Adacom seeks to mitigate these limitations, enabling relatively small models to deliver more consistent and accurate outputs without the extensive resource requirements of their larger counterparts.

1.2 Summary of Contributions

The contributions of this thesis can be summarized as follows:

1. We propose LLM4PR —a framework that integrates program refinement to guide LLMs and validate their generated code, transforming traditional program refinement into a more accessible and flexible process. The primary objectives of LLM4PR are as follows:

    a) Formally refine specifications to ensure clarity and rigor in defining desired program behavior.

    b) Automatically prompt and guide the LLM using the refinement calculus to streamline the generation process.

    c) Interact with the LLM to generate code that adheres to the refined specifications.

    d) Verify the generated code to confirm it satisfies all specified constraints, thereby ensuring correctness.

    e) Learn and develop advanced refinement laws to enhance and extend the refinement calculus, improving efficiency and applicability.

    We have implemented LLM4PR using GPT-4 and Coq, leveraging the generative capabilities of advanced LLMs alongside the formal verification strengths of proof assistants. To evaluate LLM4PR, we compared it against state-of-the-art baselines in program refinement and LLM benchmarks. Experimental results demonstrate that LLM4PR efficiently generates more reliable code while significantly reducing the time required for refinement and verification.

2. We propose CProSum to generate code comments and documents from ubiquitous contextual information, assuming that the structural context of the code is helpful to fill the information gap to create code summaries and documents. CProSum consists of one context evaluator model and one comment generator model, which are interactively trained for comment generation. On the one hand, the comment generator learns to generate comments using the context selected by the context evaluator. On the other hand, the context evaluator learns to score the contexts

and output useful contexts, allowing the comment generator to achieve the best performance.

To facilitate such two models, we build a graph dataset that contains more than 7.4M nodes and 8.8M edges from the top 100 open-source Java projects by stars. Extensive experimental results with eight baseline approaches show that CProSum achieves significant improvement (16%-35% on BLEU4, 14%-29% on METEOR, and 11%-24% on ROUGE-L) on the generated comment by effectively utilizing the contextual information.

3. We built CoEdPilot, a language model-based method designed to predict possible code edits by identifying relevant, useful edits and estimating their contributions throughout the project. This tool harnesses the power of several transformer-based neural networks to determine *what* to edit and *how* to implement code evolution within a project, focusing on both the location and content of changes.

   The operation of CoEdPilot is structured through a series of sophisticated components:

   - **Initial Edit Analysis:** After a programmer performs a code edit, optionally accompanied by the description of the edit, the *Subsequent Edit Analysis* will identify the most related files across all the files within the project and predict the next edits that may occur.
   - **Edit-Type Detection:** An *edit-type detector* infers the specific types of changes for each line of code (e.g., no change, insertion, replacement) within the identified relevant files.
   - **Edit Prediction:** The *Edit-content Generator* generates specific edit suggestions for code lines, utilizing the related prior edits identified by the *Edit-dependency Analyzer*.
   - **Feedback Integration:** The *Subsequent Edit Analysis* and *Edit-content Generator* iteratively refine their recommendations by incorporating feedback from relevant prior edits.

   This framework ensures a comprehensive and interactive approach to managing and recommending code edits effectively across projects. Our experiments comprehensively demonstrate that CoEdPilot achieves high accuracy in predicting edits, with

an edit location prediction accuracy ranging from 70.8% to 85.3%, an exact match rate for edit content of 41.8%. Furthermore, a user study involving 18 participants across three editing tasks revealed that CoEdPilot significantly outperforms tools like Copilot in assisting users with code edits. The study also provides valuable insights for future enhancements of the tool's design, confirming CoEdPilot 's effectiveness and potential for advancing code editing solutions.

4. We design Adacom to detect instances where the model might exhibit compromised performance on a sample (i.e., source code) and re-adapt the model on the fly by training with the most contributing training samples. Technically, Adacom re-trains the model dynamically using these samples to enhance performance on the given test sample.

   Our extensive experiments conducted on seven deep comment generators and four public datasets demonstrate the following:

   a) Adacom accurately detects, on average, 61.5% of samples with compromised performance.

   b) Adacom significantly boosts the performance of comment generation, achieving an average improvement of 14.9% in BLEU4, 12.2% in METEOR, and 7.4% in ROUGE-L scores.

   c) The entire adaptation process for an individual code sample incurs minimal runtime overhead, requiring only 1.46 seconds for small-sized models and 3.16 seconds for medium-sized models, making it well-suited for on-the-fly adaptation.

   d) Adacom effectively adapts to out-of-distribution code samples, demonstrating robustness in diverse scenarios.

1.3 Thesis Outline and Overview

This section briefly presents the thesis outline and overview of each chapter.

Chapter 2 begins by establishing a background of several key stages in the software development life cycle, which helps delineate the role of our advanced models. First, the program refinement process transforms the formal specifications into executable code

while preserving correctness. It highlights the balance between maintaining the specification's original intent and ensuring the resultant code is reliable. Second, program documentation is recognized as vital. It facilitates future software maintenance by providing precise and accessible information about the code's functionality and architecture. This stage ensures the software remains understandable and manageable over time, aiding current and future developers. Third, the software evolution stage addresses the continuous development of software after its initial release, responding to evolving stakeholder needs and software engineering's new demands. It focuses on adapting and enhancing the software to meet these changing requirements following the prior edits while maintaining system integrity. Finally, real-time adaptation aims to bridge the gap between the training environment and real-world datasets by adapting models based on specific testing examples.

Chapter 3 presents our innovative approach for program refinement, which leverages LLMs in conjunction with rigorous, correctness-preserving refinement calculus. This chapter includes the critical phases of software design, implementation, and verification in the software development life cycle. We explore how LLMs' flexibility and computational power can be harmoniously integrated with traditional program refinement methods to enhance the accuracy and reliability of the software development process. It discusses the theoretical underpinnings and practical implementations of combining these two distinct paradigms to achieve more reliable software. We also improve the efficiency of the refinement process and ensure that the final product adheres closely to its original specifications while maintaining high standards of correctness.

Chapter 4 details our methodology for generating code documentation, a crucial aspect of software development that enhances understanding and facilitates future code maintenance. This chapter underscores the importance of bridging the gap between natural language and programming languages, which is fundamental to creating adequate documentation. We explore techniques involving knowledge graphs and retrieval-augmented generation methods to produce documentation that is not only more explainable but also highly reliable. By integrating these technologies, we aim to generate comments that provide more precise insights into the code's functionality and design rationale.

Chapter 5 delves into an LM-driven solution to revolutionize how code edits are recommended within software projects. The advanced system enhances the decision-making process by carefully discriminating relevant edits, exploring the interactive dynamics

between these edits, and meticulously estimating their potential ripple effects throughout the project. By integrating a deep understanding of the code's context and the interdependence within the project structure, the solution aims to provide highly accurate and contextually appropriate recommendations to the users. This chapter explains the technological underpinnings of this approach. It showcases its application in real-world scenarios, demonstrating its capacity to significantly improve code project development by optimizing the code evolution process.

In Chapter 6, we explore innovative real-time adaptation methods to enhance neural network models' performance through on-the-fly model adaptation. We tackle the problem that small-size neural models often face challenges in maintaining consistent predictive accuracy across various samples. Due to their limited size and capacity, these models typically compromise on the quality of predictions with large training datasets. By implementing dynamic adaptation techniques, we aim to mitigate these shortcomings, enabling these compact models to adapt and improve their performance in real-time. This approach boosts their effectiveness and ensures they remain competitive with larger models, providing more reliable and precise applications in downstream tasks.

We finally summarize the contributions of the thesis and discuss our future research plan in Chapter 7.

1.4 Publications from the Thesis

Most of the work presented in this thesis has been published or accepted in international conference proceedings. The work in Chapter 3 is accepted at *The* 52^{nd} *ACM SIGPLAN Symposium on Principles of Programming Languages*. The work in Chapter 5 is published at *The* 33^{rd} *ACM SIGSOFT International Symposium on Software Testing and Analysis* [123]. The work in Chapter 6 is published at the *Proceedings of the* 37^{th} *International Conference on Neural Information Processing Systems* [27]. The work in Chapter 4 has been submitted for publication. I have contributed most to theory development and tool implementation for all the above work.

Chapter 2

Background

This chapter introduces the essential background of the key phases in the software development life cycle. First, program refinement transforms the formal specification into executable code while preserving correctness. Second, program documentation is essential in providing readable information for future software maintenance. Thirdly, program evolution is the continual development of software after its initial release to meet the changing requirements. Finally, program adaptation seeks to adapt the program generation model in real-time to the specific domain to tackle potential distribution shifts between training and testing.

2.1 Trustworthy Program Generation

This section examines existing works on trustworthy program generation.

Language Model-based Program Generation. Recent years have seen many works in program generation based on neural network models [214, 186]. Researchers mainly focus on training a deep learning model that inputs a specification and outputs the demanding code with different model sizes, structures, input styles, and prompts. It starts from sequence-based and tree-based approaches [122, 192] on the recurrent neural network model [55, 83, 35] and gravitates to pre-trained (large) language models based on transformer model [198], such as BERT [48], GPT [23], T5 [172, 80], GraphCodeBERT [74], CodeT5+ [203], CodeBERT [57], CodeT [32], CodeT5 [204], StarCoder [114], and Incoder [65].

CodeBERT [57] is a pre-trained bimodal model that bridges programming languages (PLs) and natural language (NLs), enabling downstream tasks like code comment gen-

eration and code search across these modalities. CodeBERT follows the architecture of RoBERTa [127], which uses the encoder of the transformer[90]. CodeBERT's pre-training tasks include Masked Language Modeling (MLM) and Replaced Token Detection (RTD), in which the natural language text and code sequence are concatenated by a unique token $[SEP]$ and fed into the encoder. The pre-training tasks mask or replace random tokens, and then the decoder recovers the masked tokens or detects the replaced tokens. Following their work, GraphCodeBERT [74] has the same RoBERTa [127] architecture. Compared with CodeBERT, GraphCodeBERT's input includes code tokens and a code data flow graph among the variables. In the graph, each node represents a variable, and the edge refers to the direction of data flow. GraphCodeBERT designs a graph-guided masked attention, where a variable can attend to another variable only if a directed edge exists in the data flow. Besides, code tokens and graph nodes can attend to each other if they represent the same variable.

CodeT5 [204], unlike CodeBERT and GraphCodeBERT, focuses on a unified representation of programming languages (PLs) and natural languages (NLs). Recently, Incoder [65] has proposed to be trained by switching the traditional left-to-right generation model to an infilling training method. That involves randomly masking and rearranging code fragments to handle arbitrary code that fills a bidirectional context. It effectively improves the performance of tasks such as type inference, comment generation, and variable renaming. StarCoder [114] has been trained with over eight programming languages, and the training materials include Git commits and GitHub issues. It outperforms existing open-sourced LLMs and matches closed models such as code-cushman-001 (the original Codex [151]) model. The pre-training tasks are designed, including randomly masking spans with arbitrary length and predicting words within the span, tagging the identifiers in the snippet, predicting the masked identifiers' names, and bimodal dual generation.

Mitigating LLM's Hallucination Hallucination, a critical challenge in LLMs, refers to generating fabricated or misleading information. To address this issue, prior research has explored various strategies to guide and verify LLM outputs, aiming to mitigate hallucinations effectively.

Prompting techniques such as Chain-of-Thought [207], Tree-of-Thought [218], and Graph-of-Thoughts [17] have been employed to provide structured reasoning or knowledge-passing procedures, enhancing the reliability of LLM responses. Similarly, [91] introduces

a self-monitoring and iterative prompting framework that leverages formal methods to detect hallucinations and steer the LLM towards the correct specification. Furthermore, [31] proposes specialized prompts based on counterexamples derived from model checking, enabling LLM-driven code debugging and repair. Another notable contribution, Language Model Programming (LMP) [18], extends traditional prompting techniques by integrating text with scripting, allowing constraints to be explicitly specified over the model's outputs.

Retrieval-based methods utilize external knowledge graphs or databases to correct factual inaccuracies in LLM-generated content [133, 234], supplementing the model with verified information beyond its training data. Additionally, collaborative approaches involve using multiple LLMs to solve a single problem, relying on mechanisms such as majority voting or consensus after natural language dialogues. Other studies employ LLMs to produce diverse outputs, including specifications, code, and test cases, and subsequently evaluate the consistency across these outputs [1, 137]. Another innovative direction involves using additional LLMs to critically review and challenge the target model's outputs in a debate-like format, fostering a process of refinement through disagreement and resolution [232].

2.2 Program Refinement

This section examines existing work on program refinement and program synthesis.

Traditional Program Refinement. Program refinement is a systematic approach to developing programs through stepwise refinement, inherently involving an interactive process alternating between programming steps and proof steps, each feeding back into the other. As a long-established discipline, the concept of program refinement can be traced back to foundational works such as [49, 61, 82]. The underlying theories[140, 11] are grounded in the calculus of weakest preconditions.

Recent advancements have introduced formalizations of refinement calculus through interactive theorem provers. For instance, [62] explores its application in Isabelle, while [8, 180] leverages Coq[14] for similar purposes. The work in [8] focuses on deriving imperative programs by applying validated refinement rules within proof mode. Consequently, the final program design inherently integrates intermediate refinement steps along with their correctness proofs. Users in [8] are required to specify loop invariants. In

contrast, [180] introduces an alternative approach by allowing the specification of loop bodies as input-output relations and employing the weakest pre-specifications rather than the weakest preconditions to compute proof obligations.

The refinement calculus has also been applied to diverse domains. For instance, [52] demonstrates its use in compositional modeling and reasoning about reactive systems. Similarly, [102] applies the refinement approach to develop correct software product lines within object-oriented and feature-oriented programming paradigms. Additionally, [56] employs refinement techniques for requirements engineering, integrating argumentation theory to address complex design requirements.

Program Synthesis The program synthesis community has developed numerous approaches, each leveraging unique methodologies to tackle this complex problem. Synthesis based on verification, as exemplified in [188], involves an interplay between verification and synthesis. Programs are transformed into predicates that must hold for all inputs, ensuring correctness through a systematic verification process. Counterexample Guided Inductive Synthesis (CEGIS), such as Sketch [187], employs a generate-and-check paradigm. In this approach, the synthesizer generates candidate programs, which are then validated using an off-the-shelf checking procedure, iteratively refining the solution based on counterexamples. Synthesis utilizing refinement types like [165] allows for specifying complex program properties, catching errors early—even before the program is fully implemented. This method benefits from round-trip type checking, which efficiently prunes the search space at each step, and condition abduction, which incrementally generates programs.

Deductive synthesis systems, while powerful, are often challenging to implement due to the difficulty of selecting appropriate rules. The Spiral system [169], designed for signal processing kernels, overcomes this challenge by embedding extensive domain knowledge into its rules and application strategies. This enables the fully automated transformation of high-level specifications into efficient implementations. Similarly, [46] introduces the Fiat system, built on Coq, which leverages Coq's tactic language to achieve a high degree of automation in deductive synthesis. Fiat facilitates the derivation of correct-by-construction programs, streamlining the development process.

Recent advancements, such as [54, 22], focus on synthesizing library functions that encapsulate standard functionality from a corpus of programs. These approaches rely on the structure hypothesis, which posits that program search becomes tractable by reducing

the search space of low-level code within a domain-specific language. This hypothesis has proven effective in simplifying program synthesis tasks while maintaining generalizability.

Trustworthy LLM with Formal Methods. Recent advancements in formal mathematical proof generation leverage machine learning techniques for proof search and premise selection. Several existing approaches, such as GPT-f [167], PACT [79], and Expert Iteration [166], employ LLMs to generate actions, which are then used by search engines to identify possible correct steps in a proof. Other methods like HTPS [106] and DT-Solver [200], combine machine learning techniques to enhance the search. Thor [92] integrates neural policy models with automated theorem provers (ATPs) to assist in proving theorems. LeanDojo [217] facilitates interaction with the proof assistant Lean [141].

2.3 Program Verification

This section examines existing work on program verification and auto-formalization.

Theorem Proving. In theorem proving, users define a system using appropriate mathematical logic, and the theorem prover determines whether a given goal can be inferred from a set of axioms based on that logic. Theorem proving tools are broadly categorized into two types: Interactive Theorem Provers (ITPs) and Automated Theorem Provers (ATPs) [146].

Interactive Theorem Provers (ITPs)—also referred to as proof assistants—are designed to assist users in constructing and developing proofs interactively. Examples include Isabelle [162], Coq [14], Lean [141], Metamath [135], Atelier B [111], Twelf [67], Agda [149], Mizar [176], HOL [150], RedPRL [9], ACL2 [99], and PVS [157]. These tools rely on user guidance and interaction to build proofs.

Automated Theorem Provers (ATPs), on the other hand, are designed to automatically prove goals without requiring user intervention. Examples include E-prover [183], CVC4 [15], Vampire [104], and Z3 [45]. Additionally, some ITPs integrate ATP capabilities to enhance automation in proof construction. For example, Isabelle uses Sledgehammer [20], and Coq incorporates CoqHammer [43] to bridge the gap between interactive and automated theorem proving.

Trustworthy LLM with Verification. LLMs exhibit impressive proficiency in generating coherent and contextually relevant text based on their training data. Nonetheless, a significant limitation of LLMs is their propensity to "hallucinate," producing not only incorrect but often fabricated and misleading information. To address this issue, [91] introduces a self-monitoring and iterative prompting approach. This method leverages formal techniques to identify hallucinations and redirect the LLM toward generating accurate specifications. Similarly, [31] employs specialized prompts informed by counterexamples obtained through model checking. These counterexamples guide LLMs in debugging and repairing code, enhancing their reliability and alignment with desired outputs.

Autoformalization. To utilize the rich context of informal proofs, researchers try to translate the informal natural language text into formal language using large language models, including [201, 210, 60, 236]. Recently, Lego-prover [211] established an advanced learning paradigm that utilizes refined structural informal proof and retrieved lemma to formalize problems of escalating complexity progressively. It performs better than the current state-of-art models based on the ChatGPT.

2.4 Program Documentation

This section examines existing work on program documentation on input representation, model architecture, and some training techniques.

Input Representation Though deriving from NL machine translation, code comment generation requires a specialized representation of the model input. Early summarization techniques still take the code text as a token sequence to generate comments of programming language (e.g., C# and SQL code) [89]. Following this approach, more specific code information, such as AST (abstract syntax tree), CFG (control flow graph), and DFG (data flow graph), is considered. Specifically, code token sequence and AST information are fed into the model for comment prediction [109, 108]. Sequence encoders with a graph component are designed to capture the code relation such as co-reference [58]. Moreover, AST can also be decomposed to several AST-node paths so that their representation can serve as the model input [7].

Model Architecture Generally, the model architecture follows an encoder-decoder structure. Treating code text as sequential data, various backbone models such as CNNs [6], LSTMs [89, 205], and GRUs [109] have been employed. With advancements in modeling techniques, attention mechanisms [109] and transformers [3, 38] have been adopted to further enhance performance. Additionally, graph neural networks have been utilized to capture abstract syntax trees (ASTs) [108] and data flow graphs [58] within a piece of code.

Training Techniques Similar to pre-trained model as BERT [48] and RoBERTa [127], CodeBERT [57] is proposed to pre-train the code token embedding with tasks mask language model tasks and replace token detection on collected NL-PL pairs. Following their work, GraphCodeBERT is further proposed to train a better embedding by considering code structure (e.g., data flow). On the other hand, duality training is emerging [205, 219], which considers comment-to-code translation and code-to-comment translation as a dual problem. Thus, two-model structures are designed to train the models on the two tasks simultaneously, further enhancing the summarization performance.

Prompt Learning and Context Fusion Prompt-based learning emerges as the large language model gains popularity. Generally, such approaches consists of prompt-template engineering [168, 44, 117], prompt-answer engineering [182, 94], multi-prompt learning [98, 181], and prompt-based training [228, 117, 128].

Regarding context utilization for comment generation, [208] search and reuse comments from cloned code, while other comment-reuse techniques [206, 144] borrow the comment of a similar method and adapt them to fit as a new comment through neural networks. Those techniques require a code search engine to look for similar code with available comments, constrained by the database's domain shifts. Different from their approaches, we use the whole code project itself as the context information, to derive a new comment on the target method. A more relevant work is proposed by Bansal et al. [13], which selects a set of files or methods as its context based on some heuristic. Even though the context is considered, the coarse definition of context provides insufficient support for the model. Likewise, in [69], code-comment examples are retrieved as few-shot examples for LLMs. Although the retrieved samples are semantically similar, they lack awareness of the project structure.

Code Comment Generator Recent advancements in code comment generation have evolved from template-based techniques [138] to statistical language models [142], and ultimately to neural network-powered approaches. Modern techniques focus on designing various deep learning models to improve summarization performance, with innovations in input structure representation, model architecture, training methodologies, and code token embedding.

Deep learning models have gained significant traction in automating comment generation from source code, aiding in tasks like code documentation [134], program comprehension [85], and reverse engineering [78]. By treating code-to-comment generation as a language translation or summarization task, advanced models such as CodeBERT [57], GraphCodeBERT [74], CodeT5 [204], and CodeGPT [73] have set new benchmarks. Moreover, ChatGPT [71] has proven effective in producing human-like interpretations and concise summaries of code.

Retrieval Augmented Generator Given the prevalence of code duplication in large-scale repositories, retrieval-based techniques have played a significant role. Early methods, such as the vector space model [78] and code clone detection [209], were used to identify the most similar code-comment pairs from a database. These approaches often relied on manually designed rules to filter the closest matches and reuse their comments.

Recent research combines retrieval systems with neural network models to enhance comment generation. Neural networks serve as semantic feature extractors to retrieve comments from the most similar training samples. For instance, CCGIR [216] integrates CodeBERT with the BERT-whitening operation [189] for retrieval. Other studies, such as [87] and [224], propose dynamic strategies to choose between retrieval-based and neural network-based methods. Re^{2}Com [206] employs BM25 to identify the most semantically similar sample and uses four encoders to process target code, similar code, the comment of the similar code, and the code's abstract syntax tree (AST). An attention mechanism fuses the outputs of these encoders, which are then fed into a decoder to generate the final comment. Similarly, [221] introduces a fusion layer to combine CodeBERT outputs for both target and retrieved code. Zhang et al. [225] retrieve samples as prompts to guide the generator, encouraging the use of locally important yet globally rare words.

2.5 Program Evolution

Code editing is a specific case of code generation that involves generating subsequent code edits based on either the code to be edited or a natural language comment. Among the works addressing code editing [30, 238, 29, 233, 93, 116, 226, 121], Codit [30] was the first to introduce tree-based neural networks for predicting code edits. Codit predicts the syntax tree of the edited code and then concretizes the code based on the predicted tree structure, outperforming traditional information retrieval methods. Building on this tree-based structure, Recoder [238] incorporates an abstract syntax tree (AST) reader alongside a code reader, further improving Codit's performance.

MODIT [29] enhances model performance by incorporating code context and natural language guidance into the encoder as additional input. With the advent of pre-trained models, researchers have explored their applicability for code editing tasks. For instance, CURE [93] leverages pre-trained models for automatic program repair, while CodeReview [116] tailors pre-training models for code review scenarios. CoditT5 [226] extends the pre-training of the CodeT5 [204] base model using inputs that combine natural language comments with edited code hunks while generating outputs in the form of edit plans. GRACE [75] leverages a prompting-based large language model [223] trained with carefully crafted prompts to incorporate relevant code updates. Overwatch [233] employs symbolic analysis of edit sequence patterns by encoding them into rules derived from previous program transformations. These works have paved the way for significant advancements in industrial code editing tools, such as Cursor and Devin.

2.6 Program Adaptation

Adaptation refers to the process of generalizing from one data distribution to another. Transfer learning, as discussed in seminal works[51, 220], involves transferring parameters from a pre-trained model to a new model tailored to a specific target dataset. This approach leverages the knowledge encapsulated in the model's parameters to expedite the learning process on new data. Initially, the model is trained on a source dataset and then fine-tuned with the target dataset's training data. The fine-tuning way prepares the model to make predictions on the target dataset's unlabeled test data, necessitating some knowledge of the target data's labels and distribution.

Domain adaptation techniques, highlighted in [66], focus on enhancing model performance in a different domain (the target domain), aiming to match the performance achievable in the original domain (the source domain). It requires access to the source data and labels but only the data from the target domain. The training involves a cross-domain loss that combines supervised loss from the source domain with a distribution shift loss between the source and target data, thus minimizing the disparity in feature space and fostering the development of domain-invariant features. Additionally, the test-time training technique, as introduced in [191], updates the model during the testing phase using unlabeled data self-supervised. The model initially trains using both supervised and self-supervised losses on the source data and later adapts using only the self-supervised loss from the unlabeled target data. In contrast, Wang et al. propose a novel adaptation method in [199] that operates independently of the training data and loss. The model is trained to minimize test entropy during the testing phase, allowing it to self-adjust based on the feedback from its predictions, thus enhancing its adaptability and performance under new conditions.

Chapter 3

Program Refinement: From Specification to Program

3.1 Introduction

Recent advancements in LLMs have significantly enhanced their capabilities in mathematics, logical reasoning, and programming [235, 174]. Tools such as GPT-4 [155] and Copilot [70] have been widely adopted in industry, providing substantial support for coding tasks and achieving performance levels surpassing the 50th percentile in programming competitions. Despite these achievements, LLMs still face a major challenge in hallucination, where they generate outputs that seem credible but are factually inaccurate.

Besides, user studies [196, 50] have shown that programmers often struggle to trust and debug LLM-generated code due to its opaque generation process, which lacks transparency and control. Even more concerning, research indicates that over half of ChatGPT's responses to programming questions contain inaccuracies [97]. Furthermore, recent mathematical proofs suggest that hallucination in LLMs is fundamentally unavoidable [212].

To illustrate this issue, we analyze a classic example: the square root algorithm, as shown in Figure 3.1. Using the prompt:

Find the square root of N within the error bound e.

Both GPT-4 [155] and Copilot [70] generate code that is almost correct but contains subtle bugs. Specifically, both programs fail in the case $N < 1$ as $N * N < N$. Mathematically, the variable `high` should be larger than $N + \frac{1}{4}$ because $\forall N \in \mathbf{R}, (N + \frac{1}{4})^2 \geq N$. If we give the GPT4 with the prompt *"The upper bound is wrong when the input N is less*

```
// GPT4 Output
def find_x(N, e):
  low, high = 0, N
  while low <= high:
    mid = (low + high) / 2
    square = mid * mid
    if square < N and N < (mid +
    e) * (mid + e):
        return mid
    if square < N:
        low = mid + e
    else:
        high = mid - e
  return mid
#
```

```
// Copilot Output
n = input()
n = float(n)
e = 0.0000001
low = 0
high = n
mid = (low + high) / 2
while abs(mid * mid - n) > e:
    if mid * mid > n:
        high = mid
    else:
        low = mid
    mid = (low + high) / 2
print(mid)
#
```

```
// Repair when N less than 1
def sqrt(N):
    x = N+1
    while x * x > N:
        x = (x + N/x) / 2
    return x
```

```
// GPT4 + Constraints
// Spec: x^2 <= N < (x+e)^2
x = N+1
// since (N+1+e)^2 > N
while x*x > N:
    x = x-e
```

Figure 3.1: Wrong implementations of square root algorithm generated by GPT-4 and Copilot. The upper two programs are wrong in the case $N < 1$ due to the wrong upper bound initialization. The third code fails when the variable x goes to the fixed point, while the last code fails in infinite loops.

than 1.", the newly generated code will still fail in several cases, like sqrt(5.1). This is because the variable x will be fixed into a number but not terminate the loop. Similarly, the final code (bottom right), created using traditional program refinement methods, still encounters issues, including infinite loops in some scenarios. This example highlights that while LLMs can generate code close to correctness, they often fall short in critical areas, particularly when applied to traditional program refinement without integrating formal verification systems. Addressing these limitations requires further advancements in guiding LLMs with robust verification methodologies to ensure correctness and reliability.

Developing reliable LLMs for program generation continues to be a significant challenge. Existing methods primarily focus on two aspects: guiding LLMs during input preparation and validating their outputs. Guidance strategies often involve providing LLMs with detailed task-specific prompts to leverage their inherent capabilities effectively. Recent research frequently adopts informal heuristics, such as the chain-of-thought reasoning approach, to enhance LLMs' problem-solving processes [207]. However, rigorous verification of deep learning models as a transparent, white-box process remains feasible only for small, quantized neural networks, which are vastly different from the scale and

complexity of modern LLMs [86].

State-of-the-art verification methods for LLMs often rely on using multiple models to evaluate outputs through majority voting or consensus in natural language [1, 137, 232]. However, prior research [212] demonstrates that hallucinations cannot be entirely eliminated by merely altering prompts. Furthermore, it shows that an ensemble of LLMs essentially behaves like a single model, failing to resolve hallucination issues effectively.

The refinement calculus [140, 11, 28, 180, 193] formalizes the stepwise refinement approach to program construction. In this paradigm, a program's required behavior is first defined through a non-executable specification, which is then transformed into an executable program via a series of correctness-preserving steps. However, this process has traditionally been performed manually, making it both time-consuming and prone to errors.

The reliance on manual intervention not only renders program refinement labor-intensive but also limits its scalability and automation. Integrating LLMs and proof assistants into the refinement process is a natural evolution, making refinement calculus accessible to users without deep expertise in program refinement or theorem proving. Additionally, many refinement steps involve handling sub-derivations for program sub-components [26]. This segmented reasoning necessitates library-based approaches that encapsulate functionality within modular abstractions, effectively managing complexity. By narrowing the search space for applicable refinement laws and pruning candidates for sub-components, this approach enhances scalability and improves the efficiency of program refinement.

In contrast, our proposed tool, LLM4PR, takes a novel approach by explicitly controlling the LLM workflow and integrating external knowledge sources with symbolic reasoning systems. This approach applies constraints to guide the LLM and verifies the generated code through program refinement. Inspired by human problem-solving methods, where tools like calculators and code interpreters are used to tackle tasks beyond immediate capabilities, we conceptualize LLMs as "constraint solvers." Their extensive background knowledge and adaptability offer promising potential for automating program refinement.

Our methodology enables the assertion of constraints to aid debugging and the verification of constraints to ensure the correctness of generated code. This represents a significant leap in applying LLMs to program generation, shifting the focus from merely

reducing errors to achieving reliable and verifiable code. We implement this approach in an automated tool called LLM4PR, which integrates formal program refinement calculus with LLM capabilities to iteratively refine specifications and generate verified code. Our framework defines a formal specification language, a corresponding program language, and a refinement calculus to guide the transformation process.

To improve the efficiency of program refinement, LLM4PR employs a learning strategy that constructs new refinement laws. These laws help compress and streamline the refinement process, reducing its overall depth, as described in Section 3.4.3. We present a top-down algorithm for decomposing high-level specifications into sub-components, followed by a bottom-up algorithm that refines these sub-specifications step by step, detailed in Section 3.5. LLM4PR also incorporates automated theorem provers (ATPs) like Z3 [45] to verify the generated code and validate the application of refinement laws. This combination ensures both correctness and justification for each refinement step.

To evaluate LLM4PR, we tested it on classical program refinement benchmarks and standard LLM benchmarks, including an expanded version of Humaneval with additional test cases [124], to assess the robustness of the generated code. The results, discussed in Section 3.6, demonstrate the effectiveness of LLM4PR in generating reliable and verifiable code.

3.2 Motivating Example

In this section, we use the square root (sqrt) algorithm to demonstrate our intuitions behind LLM4PR. Unlike other program refinement works [140, 180], we generalize the square root algorithm from integers to real numbers, showcasing how LLM4PR guides LLMs and verifies the correctness of the generated code. Additionally, we explain the learning strategy employed to extend refinement laws, enabling the evolution and expansion of the refinement calculus. This approach is designed to simplify the program refinement process and reduce its overall complexity.

3.2.1 Guide the LLM

The specification of the square root example is formulated as follows: given any positive constant N and e, the program C is required to adjust the variable x such that $x^2 \leq N$, while simultaneously ensuring that $(x + e)^2 > N$. Program refinement systematically

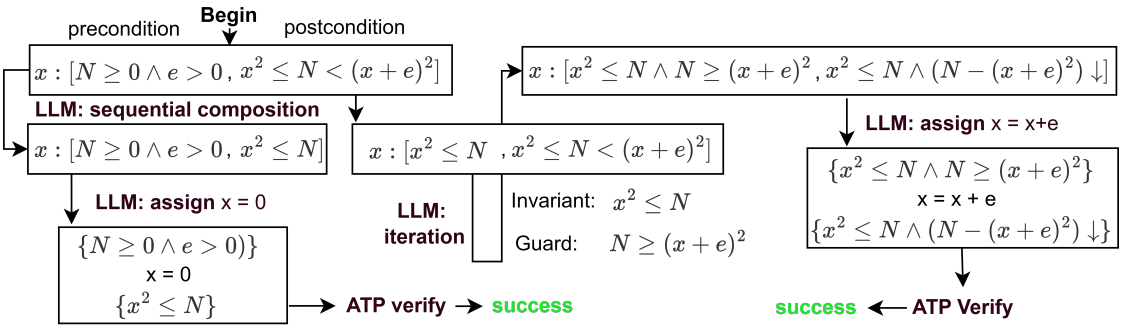

Figure 3.2: The Success Version for Program Refinement on Square Root Algorithm

decomposes the specification into smaller, manageable components, enabling step-by-step program construction. We initiate the process using the basic refinement calculus, which comprises the fundamental laws introduced in the book [140], and demonstrate the refinement process in Figure 3.2.

The first refinement step employs the sequential composition law to segregate the constraints $x^2 \leq N$ and $N < (x+e)^2$ into distinct components. For the initial component, the LLM can deduce a suitable assignment, such as $x = 0$, based on the specification. This assignment can be verified through Hoare logic and supported by automatic theorem provers (ATPs).

LLM4PR subsequently applies the iteration law to the second component, leveraging the specification structure $[\text{Invariant}, \text{Invariant} \wedge \neg\text{Guards}]$ to construct an iterative process. The iteration law decomposes the postcondition into two elements: a guard condition and an invariant. This decomposition establishes an iterative framework that maintains the invariant and modifies the variant until the guard condition no longer holds. The notation $\downarrow$ indicates that the variant strictly decreases during the iteration. LLM4PR incorporates the constraints "Invariant $\wedge$ Guards $\rightarrow$ Invariant $\wedge$ Variant is strictly decreasing" into its prompt, guiding the LLM to produce code, such as $x = x + e$. Finally, LLM4PR verifies whether the generated assignment upholds the invariant and ensures the variant decreases.

It is important to note that this verification establishes only *partial correctness*, confirming that the variant decreases as required. Achieving *total correctness* additionally necessitates verifying that the iteration eventually reaches a state where the guard condition is violated.

3.2.2 Failure Feedback

The program refinement process can be approached in various ways. Figure 3.3 illustrates an alternative approach: initializing x with a large value and decrementing it until the constraints are satisfied. This program starts by assigning x a value that satisfies the invariant $N < (x + e)^2$.

Traditional synthesis techniques struggle to deduce a valid assignment: find x such that $\forall N > 0, e > 0, (x > N + \frac{1}{4} \rightarrow (x + e)^2 > N)$. In contrast, the LLM might propose assignments such as $x = N$ or $x = 1$, but these lack verification or guidance toward a correct solution.

LLM4PR is designed to verify the outputs and provide counterexample feedback to guide the LLM towards a valid assignment. For instance, if the LLM suggests $x = N$, LLM4PR will reject this assignment and offer counterexample feedback until a correct assignment, such as $x = N + 1$, is generated. Once a valid starting point is determined, LLM4PR will apply the iteration law and instruct the LLM to generate code that satisfies the constraints under the new specification.

The LLM might propose code such as $x = x - e$, and LLM4PR will formally derive the corresponding proof obligation:

$$(N < (x+e)^2 \wedge N < x^2) \rightarrow (N < (x'+e)^2 \wedge (N - x'^2 < N - x^2) \wedge x' = x - e) \quad (3.1)$$

However, ATPs will reject this code because the variant $N - x^2$ is not strictly decreasing in some cases.

Interestingly, while the two symmetric approaches to refining the square root algorithm share similarities, the program from Figure 3.2 succeeds, whereas the program in Figure 3.3 encounters an infinite loop. This asymmetry underscores the critical role of formal program verification, even for seemingly straightforward algorithms.

Upon the first failure, the LLM receives feedback and may propose an alternative assignment, such as $x = (x + \frac{N}{x})/2$, inspired by Newton's method. However, ATPs reveal that this approach may fail due to floating-point precision errors, causing the variant to stagnate and fail to reach a fixed point in some cases.

If the LLM fails to generate verified code after several attempts, LLM4PR will backtrack to the last refinement step and explore an alternative refinement law to pursue a different direction.

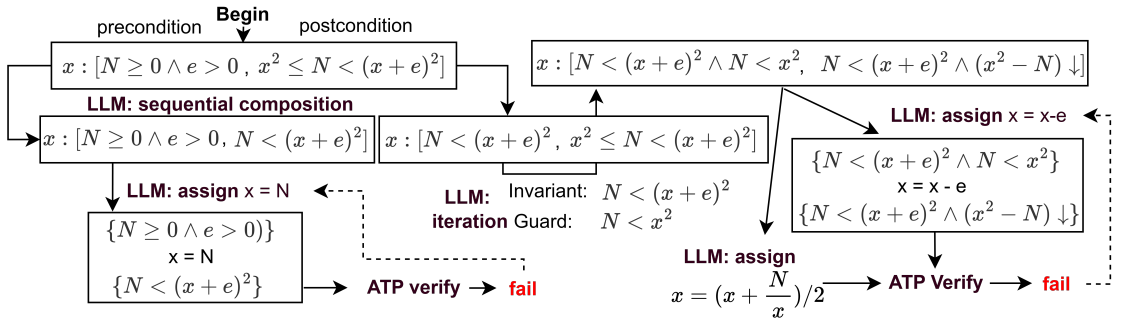

Figure 3.3: A Failed Version for Program Refinement on Square Root Algorithm.

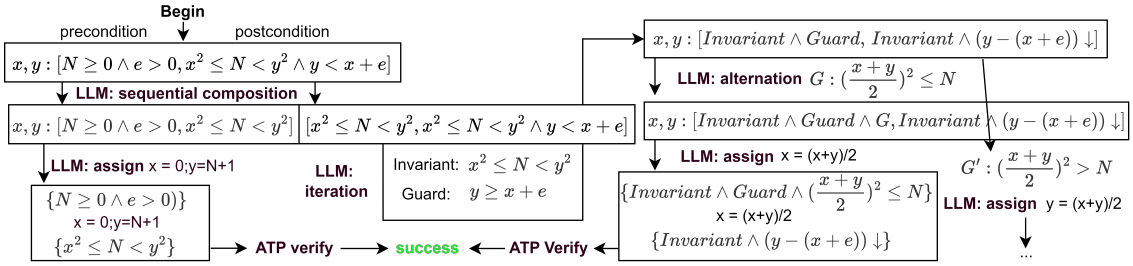

Figure 3.4: A Binary Search Version for Program Refinement on Square Root Algorithm.

3.2.3 Learning Strategies for Extending the Refinement Calculus

The program refinement procedures share notable similarities, beginning with the sequential composition law, followed by the application of the assignment law for initialization and the iteration law for repetitive operations, as depicted in Figure 3.5. Semantically, these procedures first initialize variables to establish the invariant, then preserve the invariant while iteratively updating the variant until the postcondition is satisfied.

The sequential composition law, on one hand, structures the refinement process with $[\text{Invariant}, \text{Invariant} \wedge \text{Guard}]$, setting up the framework for iteration. On the other hand, the *future* iteration law provides valuable guidance for the initial step of sequential composition, aiding in the specification's effective decomposition. When the LLM has access to *future* information regarding the iteration law, it is more likely to split the specification in the intended manner.

To identify and leverage common patterns in program refinement, we propose a learning algorithm that extracts *patterns* from refinement histories and extends the refinement calculus. This algorithm processes a dataset of past refinement procedures, identifying recurring patterns in both laws and specifications. The resulting extended laws encapsulate these patterns, allowing specifications to be refined with fewer steps, thereby reduc-

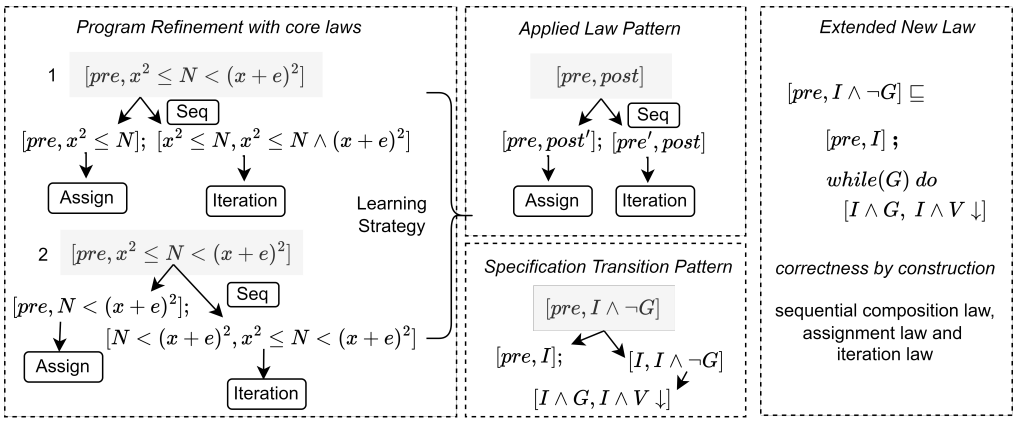

Figure 3.5: This example illustrates the process of learning and extending a refinement law.

ing the search depth and verification effort. Importantly, all new laws are constructed atop the foundational refinement laws, and their correctness is established through the correctness-by-construction principle [21].

In summary, extending refinement laws offers the following benefits:

- Broadens the LLM's scope from one-step refinement to anticipating future refinement steps.
- Reduces the depth of refinement, saving time and resources during interactions with LLMs.
- Simplifies program verification, lowering the reliance on ATPs.

3.3 The Language

We introduce the formal specification language L_{spec}, designed for articulating specifications, alongside the programming language L_{pl}, used in the generated code. To facilitate the program refinement process, we define an *annotated programming language*, which integrates both L_{spec} and L_{pl}. This integration is formally represented as a tuple (L_{spec}, L_{pl}), with each component corresponding to one of the two languages. Given the close interaction between these languages and LLMs, our focus is on crafting languages that are both intuitive and effective for LLM applications.

3.3.1 The Specification Language

Our specification language, L_{spec}, follows the first-order logic (FOL) and the Coq language [14]. LLMs are well-trained in understanding both FOL and Coq. While adhering to the standard syntax and semantics of FOL, we emphasize the following key notations: We utilize common *relation* operators and *function* operators found in SMT, such as $<, =, +, -, *, /, \texttt{Array[Int]}, \texttt{Array[Int:Int]}$.

Syntax. Our specification is formulated using first-order logic (FOL) combined with the theory of arrays. The complete syntax of L_{spec} is presented in Table 3.1, where the key components are defined as follows:

- $\langle Specification \rangle$: Describes the specification to be refined.
- $\langle Definition \rangle$: Specifies the conditions that the variants must satisfy.
- $\langle Params \rangle$: Defines the variants and constants.

For $\langle atom \rangle$, $\langle Expr \rangle_0$ represents the previous value of the expression, while $\langle Name \rangle$ [$\langle atom \rangle$] denotes the array selection operation. Additionally, $\langle Name \rangle [\langle atom \rangle : \langle atom \rangle]$ is used for array slicing. The remainder of the syntax adheres to the standard FOL conventions commonly used in SMT solving.

Semantics. We adhere to the standard semantics of first-order logic (FOL) as defined in Coq, highlighting only the notable elements in Table 3.2. The theory of arrays is implemented using relations and functions, consistent with its treatment in the existing literature [37].

3.3.2 The Program Language

Our programming language is primarily based on the While language, which is designed with simplicity to facilitate ease of understanding and generation by LLMs. The complete syntax is provided in Table 3.3. This imperative language supports various data types, including booleans, natural numbers, integers, floats, characters, and arrays. It extends the basic While language with additional features like Array and Assert statements. Arrays are indexed using natural numbers and support operations for reading, updating, and slicing.

Table 3.1: Specification Language L_{spec} Syntax

$$
\begin{aligned}
\langle \textit{Type} \rangle &::= \texttt{bool} \mid \texttt{nat} \mid \texttt{Z} \mid \texttt{float} \mid \texttt{array}\ \langle \textit{Type} \rangle \\
\langle \textit{Specification} \rangle &::= \texttt{Precondition:}\langle \textit{Definition} \rangle\ \texttt{Postcondition:}\langle \textit{Definition} \rangle \\
\langle \textit{Definition} \rangle &::= \langle \textit{Name} \rangle \langle \textit{Params} \rangle\ \texttt{:=}\ \langle \textit{Expr} \rangle\ \texttt{.} \\
\langle \textit{Params} \rangle &::= \texttt{(}\ \langle \textit{Name} \rangle\ \texttt{:}\ \langle \textit{Type} \rangle\ \texttt{)} \\
\langle \textit{Expr} \rangle &::= \langle \textit{Logit} \rangle \mid \langle \textit{Logit} \rangle \wedge \langle \textit{Expr} \rangle \mid \langle \textit{Logit} \rangle \vee \langle \textit{Expr} \rangle \mid \neg \langle \textit{Expr} \rangle \mid \langle \textit{QExpr} \rangle \\
\langle \textit{QExpr} \rangle &::= \texttt{forall}|\texttt{exists}\ \langle \textit{Params} \rangle\ \langle \textit{Expr} \rangle \\
\langle \textit{Logit} \rangle &::= \langle \textit{Term} \rangle \mid \langle \textit{Term} \rangle < \langle \textit{Logit} \rangle \mid \langle \textit{Term} \rangle <= \langle \textit{Logit} \rangle \mid \langle \textit{Term} \rangle = \langle \textit{Logit} \rangle \\
&\quad \mid \langle \textit{Term} \rangle > \langle \textit{Logit} \rangle \mid \langle \textit{Term} \rangle >= \langle \textit{Logit} \rangle \mid \langle \textit{Term} \rangle <> \langle \textit{Logit} \rangle \\
\langle \textit{Term} \rangle &::= \langle \textit{Factor} \rangle \mid \langle \textit{Factor} \rangle + \langle \textit{Term} \rangle \mid \langle \textit{Factor} \rangle - \langle \textit{Term} \rangle \\
\langle \textit{Factor} \rangle &::= \langle \textit{atom} \rangle \mid \langle \textit{atom} \rangle \star \langle \textit{Factor} \rangle \mid \langle \textit{atom} \rangle\ /\ \langle \textit{Factor} \rangle \\
\langle \textit{atom} \rangle &::= \langle \textit{Number} \rangle \mid \langle \textit{Variable} \rangle \mid \langle \textit{Const} \rangle \mid \texttt{true} \mid \texttt{false} \\
&\quad \mid\ \texttt{-}\ \langle \textit{Expr} \rangle \mid\ \texttt{(}\ \langle \textit{Expr} \rangle\ \texttt{)} \mid \langle \textit{Expr} \rangle_0 \\
&\quad \mid\ \langle \textit{Name} \rangle\ \texttt{[}\langle \textit{atom} \rangle\texttt{]} \mid \langle \textit{Name} \rangle [\langle \textit{atom} \rangle : \langle \textit{atom} \rangle]
\end{aligned}
$$

Table 3.2: Specification Language L_{spec} Semantics

$$
\begin{aligned}
\textit{type } T &\iff \textit{A value set } T\ \ [\![e_T]\!] \in T \\
\textit{variants } v : T &\iff \textit{A value } v \in T\ \ [\![v]\!] \\
\textit{constant } c : T &\iff \textit{A value } c \in T\ \ [\![c]\!] = c \\
\textit{functional operator } f(T_1, T_2, ...) : T &\iff [\![f(a, b, ...)]\!] = f([\![a]\!], [\![b]\!], ...) \\
\textit{relational operator } R(T_1, T_2, ...) : \textit{Bool} &\iff [\![R(a, b, ...)]\!] = R([\![a]\!], [\![b]\!], ...)
\end{aligned}
$$

To manage program size and complexity, we incorporate procedures. Each procedure is defined by a name, a set of parameters, and an associated program body. The formal semantics of the language align with those described in the literature [103].

Finally, we define the *mixed programming language* L_{mix}, which serves as a blend of L_{spec} and L_{pl} for use in the program refinement procedure. Formally, L_{mix} is characterized by a variant of $\langle \textit{Prog} \rangle$, where sections of the program may still consist of specifications, as outlined in Table 3.4. The "intermediate" language is employed during the refinement process, allowing portions of the specifications to be refined into executable code, while other sections remain as specifications. We refer to such constructs informally as "mixed programs."

Table 3.3: Program Language L_{pl} Syntax

$$
\begin{array}{rcl}
\multicolumn{3}{c}{Variable\ v \quad Constant\ c \quad Procedure\ f} \\
Type\ \langle T\rangle & ::= & \texttt{bool} \mid \texttt{nat} \mid \texttt{int} \mid \texttt{float} \mid \texttt{char} \mid \texttt{array}\langle T\rangle \\
Expression\ \langle e\rangle & ::= & c \in \langle T\rangle \mid v \in \langle T\rangle \mid \langle e\rangle \oplus \langle e\rangle \mid not\ \langle e\rangle \\
& \mid & v\,\texttt{[}\langle e\rangle\texttt{]} \mid v[\langle e\rangle : \langle e\rangle] \\
Operator\ \langle\oplus\rangle & ::= & \texttt{and} \mid \texttt{or} \mid \texttt{==} \mid \texttt{<} \mid \texttt{<=} \mid \texttt{>} \mid \texttt{>=} \mid \texttt{!=} \mid + \mid - \mid \texttt{*} \mid / \\
Program\ \langle Prog\rangle & ::= & \texttt{pass} \mid \langle e\rangle = \langle e\rangle \mid \langle Prog\rangle\texttt{;}\ \langle Prog\rangle \mid f\ \texttt{(}\ \overline{v_i}\ \texttt{)} \\
& \mid & \texttt{assert (}\ \langle e\rangle\ \texttt{)} \\
& \mid & \texttt{if (}\langle e\rangle\texttt{)then (}\ \langle Prog\rangle\ \texttt{) else (}\ \langle Prog\rangle\ \texttt{)} \\
& \mid & \texttt{while (}\langle e\rangle\texttt{)do (}\ \langle Prog\rangle\ \texttt{)} \\
Procedure\ \langle Proc\rangle & ::= & \epsilon \mid \texttt{def}\ f\ \ \texttt{(}\overline{v_i}\texttt{)}\ \texttt{(}\ \langle Prog\rangle\ \texttt{)}
\end{array}
$$

Table 3.4: Language L_{mix} Syntax

$$
\begin{array}{rcl}
\langle Mix\rangle & ::= & \langle Spec\rangle \mid \langle Prog\rangle \\
\langle MixProg\rangle & ::= & \texttt{pass} \mid \langle e\rangle = \langle e\rangle \mid \langle Mix\rangle\texttt{;}\ \langle Mix\rangle \mid f\ \texttt{(}\ \overline{v_i}\ \texttt{)} \\
& \mid & \texttt{assert (}\ \langle e\rangle\ \texttt{)} \\
& \mid & \texttt{if (}\langle e\rangle\texttt{): (}\ \langle Mix\rangle\ \texttt{) else: (}\ \langle Mix\rangle\ \texttt{)} \\
& \mid & \texttt{while (}\langle e\rangle\texttt{): (}\ \langle Mix\rangle\ \texttt{)}
\end{array}
$$

3.4 The Refinement Calculus

This section presents our calculus for program refinement, which is grounded in the weakest precondition semantics for programs [49]. Our refinement calculus primarily adopts the notations and methodologies outlined by Morgan in [140]. The refinement process involves a sequential application of refinement laws, transforming an initial specification into an intermediate state comprising a mixture of specifications and programs (mixed programs), and ultimately resulting in pure program code. The process can be depicted as follows:

$$specification \sqsubseteq mixed\ program \sqsubseteq \cdots \sqsubseteq mixed\ program' \sqsubseteq program.$$

3.4.1 Basics

Specification Formal specifications describe what a system should do, not how the system should do it. In detail, a specification contains *variants*, a *precondition*, and a

postcondition, in the form

$$varibales : [precondition,\ postcondition].$$

Variables are the list of program variables; the precondition describes the initial states, and the postcondition describes the final states of the program.

Refinement The refinement of a specification is the relation between two expressions where one can *solve* the other. Informally, a specification is *improved* by weakening its precondition or strengthening its postcondition. Formally, the refinement relation is defined by the weakest preconditions of the related programs [26]. For program S and postcondition P, $wp(S, P)$ represents the weakest precondition where S is guaranteed to terminate in a state satisfying P. Program S_0 is refined by S_1 denoted as $S_0 \sqsubseteq S_1$, iff

$$\forall P, wp(S_0, P) \rightarrow wp(S_1, P) \tag{3.2}$$

which states that S_1 will preserve the total correctness of program S_0. The program refinement can be established in a linear sequence:

$$S_0 \sqsubseteq S_1 \sqsubseteq S_2 \sqsubseteq S_3 ... \sqsubseteq S_n \tag{3.3}$$

which shows the refinement $S_0 \sqsubseteq S_n$ with the transitivity of the refinement relation.

Besides, one can refine sub-components of the programs without affecting the total correctness of the whole program following the congruence of Hoare logic.

$$T_1 \sqsubseteq T_2 \rightarrow P; T_1; Q \sqsubseteq P; T_2; Q \tag{3.4}$$

where T_1, T_2 are sub-components of the program $P; T; Q$, P and Q are the context code of T. The program is successively created using refinement rules that define side conditions preserving the correctness of the program, which is also known as Correctness-by-construction [21].

3.4.2 Core Refinement Laws

This subsection presents the core refinement laws in the literature [140].

Lemma 1 (Strengthen Postcondition Law)**.** *Let pre (precondition) and $post$, $post'$ (post-conditions) be any FOL formula, if $post' \Rrightarrow post$, then $x : [pre, post] \sqsubseteq x : [pre, post']$.*

Lemma 2 (Weaken Precondition Law)**.** *Let pre, pre' (preconditions) and $post$ (postcondition) be any FOL formula, if $pre \Rrightarrow pre'$, then $x : [pre, post] \sqsubseteq x : [pre', post]$.*

The basic refinement calculus is defined as follows:

Skip It is a command where the final state of the program is the same as its initial state. If the precondition entails the postcondition, the specification can be refined by skip.

Lemma 3 (Skip Law)**.** *If $pre \Rrightarrow post,\ then\ x : [pre, post] \sqsubseteq$ **skip**.*

Sequential Composition It refines a single specification to two smaller components.

Lemma 4 (Sequential Composition Law)**.** *Let mid be any formula except for pre or $post$. $x : [pre, post] \sqsubseteq x : [pre, mid]$**;** $x : [mid, post]$.*

Assignment The variant is updated with new expressions. We define $post\langle x := E\rangle$ as a condition where all occurrences of x in $post$ are replaced with E. If the precondition implies the updated postcondition after the assignment, the program can be refined accordingly.

Lemma 5 (Assignment Law)**.** *Let E be any $Expression$, $post\langle x := E\rangle$ assigns every x in $post$ with E. If $pre \Rrightarrow post\langle x := E\rangle,\ then\ x : [pre, post] \sqsubseteq$ **x = E**.*

Alternation It is built with guarded branches.

Lemma 6 (Alternation Law)**.** *Let GG be the disjunctive normal form of the guards $G_0, G_1,$ $, ..., G_i, ..., G_n$, if $pre \Rrightarrow GG,\ then\ x : [pre, post] \sqsubseteq$ **if** $\bigsqcup_i (G_i$ **then** $x : [G_i \wedge pre,\ post])$ where **if** $\bigsqcup_i G_i$ **then** means if G_0 then ... else if G_i then ... .*

Iteration. Iterations like while loops are constructed using loop conditions, invariants, and variants. An invariant, inv, is a formula that remains true throughout the execution of the loop, provided it is true at the start. The variant, V, is a value that changes during each iteration and ensures the termination of the loop by demonstrating progress toward a specific condition.

Lemma 7 (Iteration Law)**.** *Let Inv, the invariant, be any formula; let V, the variant, be any integer-valued expression. Let GG be the disjunctive normal form of the guards $G_0, G_1, ..., G_i, ..., G_n$ then $x : [Inv,\ Inv \wedge \neg GG] \sqsubseteq$ **while** $\bigsqcup_i (G_i$ **do** $x : [Inv \wedge G_i, Inv \wedge (0 \leq V < V_0)])$ where V_0 is the initial value of V,* ***while*** $\bigsqcup_i G_i$ ***do*** *means while G_0 do ... else G_i do ... else G_n do.*

Expand. It expands the variant list by introducing another variant. Note that Lemma 8 is an equality, which means the refinement goes both ways.

Lemma 8 (Expand Law)**.** *Let x be the origin variant and y be another variant and y_0 be the initial value of y, then $x : [pre, post] = x, y : [pre, post \wedge y = y_0]$*

Assertion. [11] It expands the precondition by introducing another condition. Note that Lemma 9 is used for ensuring the termination of the loop in our program refinement.

Lemma 9 (Assertion Law)**.** *Let E be a boolean condition for the variable x, then $x : [pre, post] = assert E; x : [pre \wedge E, post]$*

Procedure. A procedure is declared by a name, some parameters, and a program.

Definition 1 (Procedure)**.** $procedure\ \langle Name \rangle\ (\langle Variable \rangle\ : \langle Type \rangle) \triangleq \langle Prog \rangle$.

Lemma 10 (Procedure Value Specification)**.** *Given a procedure $procedure\ \langle Name \rangle\ (value\ f\ : \langle Type \rangle) \triangleq w, f : [pre, post]$ where w and f are different variables. Let A be the expression of the $\langle Type \rangle$, then $w : [pre\langle f := A \rangle, post\langle f_0 := A_0 \rangle] \sqsubseteq procedure\langle Name \rangle(A)$.*

Lemma 11 (Procedure Result Specification)**.** *Given a procedure $procedure\ \langle Name \rangle\ (result\ f : \langle Type \rangle) \triangleq w, f : [pre, post\langle a := f \rangle]$ where w, a, and f are different variables. Then $w, a : [pre, post] \sqsubseteq procedure\langle Name \rangle(a)$.*

In procedure, substitution by result is complementary to substitution by value since it takes the value out of the procedure rather than into it. The literature [139, 10] has established the correctness of the laws in this section. In particular, define a Hoare-triple-like notation $\{pre\}\{prog\}\{post\}$ that means that starting from a precondition that satisfies

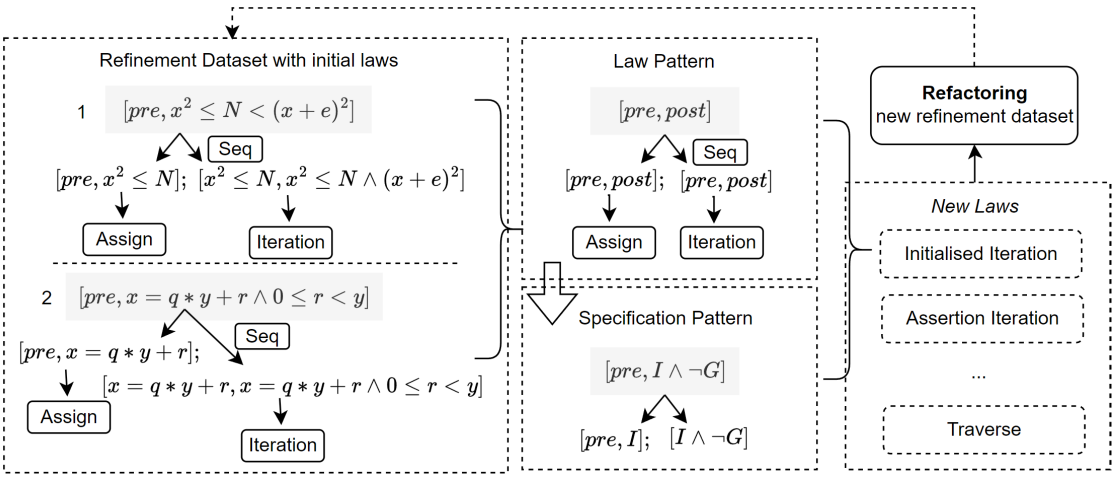

Figure 3.6: Given initial refinement laws and the associated refinement dataset, LLM4PR derives new refinement laws with the learning algorithm to refactor the refinement steps and reduce the depth of program refinement.

pre, if the program $prog$ terminates, then the postcondition satisfies $post$. We represent the result as follows sans proof:

Theorem 1 (Soundness of Core Refinement Laws)**.** *If* $\vec{x} : [pre, post] \sqsubseteq prog$ *is derivable from the laws in Sections 3.4.1 and 3.4.2, then* $\{pre\}prog\{post\}$ *holds.*

3.4.3 Law Learning Strategy

This section introduces our program refinement laws used for interaction with the LLMs. The previous works [22, 54] in program synthesis focus on reducing the search space of the low-level code in a domain-specific language for functional abstraction learning. In contrast, LLM4PR learns high-level refinement laws to guide and verify the LLM since the LLM has great capability for code generation. The algorithm is designed to take a collection of refining processes and extract a set of components from them that can be used to represent the origin refining more compactly. High-level refinement laws enable specifications to be refined in fewer steps, thereby reducing both the depth of the refinement search and the complexity of verification. Therefore, it is useful and efficient to find patterns of the common combination of laws and derive new laws to extend the original calculus. We then formally add them to the refinement calculus, which can be used to solve similar refinement processes in the future.

3.4.3.1 Learning Procedure

LLM4PR begins by taking as input a dataset of refined problems along with the atomic laws defined earlier. These atomic laws are designed to be low-level yet expressive enough to refine the specifications in the dataset. The learning algorithm expands the refinement calculus library by analyzing examples from the dataset, identifying common refinement fragments in refined specifications, and abstracting these fragments into new law primitives.

The output is a library of refinement law patterns and their associated specification patterns. For instance, as shown in Figure 3.6, LLM4PR demonstrates that certain specifications, such as calculating square roots (Figure 3.2) or performing modulo operations (Figure 3.12), can be refined using a combination of the sequential composition law, assignment law, and iteration law.

The refinement process begins by splitting the specification into two parts: the first is refined using the assignment law, while the second is refined using the iteration law. Further analysis of these specifications reveals a pattern in which the postcondition can be expressed as a conjunction of an invariant and a guard condition. Refinement proceeds by initializing variables to satisfy the invariant, followed by constructing the iteration structure.

From this pattern, we abstract and derive the initialized iteration law (Lemma 17). While the base laws can accomplish equivalent refinements, using the advanced laws derived from this learning process significantly shortens the refinement process and reduces verification effort.

Law Pattern One refinement process is structured as a tree, where each node (illustrated in Figure 3.6, left) represents a specification, and each edge corresponds to a refinement law that connects these specifications. In this framework, the refinement laws are predefined and finite in number. Given these constraints, we systematically traverse the tree via its edges, converting the traversal into a sequential representation of the refinement process.

The algorithm then analyzes these sequences to identify common sub-sequences derived from the refinement tree. By examining these sub-sequences, we extract law patterns or recurring refinement strategies.

Specification Pattern Once the law patterns are identified, we analyze their frequency to derive corresponding specification patterns. To manage these patterns efficiently, we construct an E-graph [47]. The E-graph is used to identify equivalences among various expressions, facilitating transitions between high-level abstractions and low-level elements.

To reduce complexity, we first establish a set of rewriting rules at a high level of abstraction, grounded in the semantics of the base laws:

$$\begin{aligned}
&Strengthen - postconditon : [pre, post] \rightarrow [pre, post'] \\
&Weaken - preconditon : [pre, post] \rightarrow [pre', post] \\
&Skip : [pre, post] \rightarrow END \\
&Assign : [pre, post] \rightarrow END \\
&Seq : [pre, post] \rightarrow [pre, mid]; [mid, post] \\
&Alternation : [pre, mid] \rightarrow [pre \wedge G1, post]; [pre \wedge G2, post]; ... \\
&Iteration : [I, I \wedge \neg G] \rightarrow [I \wedge G, I \wedge V \downarrow]
\end{aligned} \tag{3.5}$$

The token *END* means the end of the refinement. The E-graph data structure enables the simultaneous representation and reasoning of multiple equivalent expressions, providing a structured and systematic approach to applying rewriting rules.

The process begins with constructing an initial E-graph, where each node represents a unique specification. Next, we identify nodes that can be merged based on the predefined rewriting rules and merge them. The rewriting rules are then systematically applied to the E-graph, expanding and merging nodes as necessary. After all rules have been applied, we extract the most frequent sub-components from the E-graph.

Following this, the specification tree is expanded to a deeper level, allowing the iterative construction and management of the E-graph from high-level abstractions to low-level details. The goal is not to learn new refinement laws for completeness but to derive frequently-used laws to enhance efficiency. As illustrated in Figure 3.6 (middle), both specifications share the same high-level pattern:

$$[pre, I \wedge \neg G] \rightarrow [Pre, I]; [I, I \wedge \neg G] \rightarrow END; [I \wedge G, I \wedge G'] \tag{3.6}$$

where we derive a new law and formally add it to the refinement calculus.

3.4.3.2 Extended Laws

Skip. The new skip law gives the variant an initial value, utilizing the fact that the initial and final variables have the same value.

Lemma 12 (Initialised Skip Law)**.** *Let x_0 denote the initial value of variant x, if $(x = x_0) \wedge P \Rrightarrow Q$, then the specification $x : [P, Q] \sqsubseteq$ Skip.*

Proof. Directly from the skip law in Lemma 3 as $P \Rrightarrow Q$. □

Seq. We extend a new sequential composition law to divide one specification into two parts flexibly based on the Strengthen-Postcondition Law and Weaken-Precondition Law.

Lemma 13 (Flexible Sequential Composition Law)**.** *Let P, Q, A, B, C, D be some formulate, if $(P \Rrightarrow A) \wedge (B \Rrightarrow C) \wedge (Q \Rrightarrow D)$, then the specification $x : [P, Q] \sqsubseteq x : [A, B]$**;** $x : [C, D]$.*

Proof. First, use the sequential composition law in Lemma 4, $x : [P, Q] \sqsubseteq x : [P, B]; x : [B, Q]$. Then refine the two parts with the weaken-precondition law in Lemma 2, $x : [P, B] \sqsubseteq x : [A, B]; x : [B, Q] \sqsubseteq x : [C, Q]$. Finally, refine the second part with the strengthen-postcondition law in Lemma 1, $x : [C, Q] \sqsubseteq x : [C, D]$. □

Assign. We extend two assignment laws. The initialized assignment law utilizes the initial values of the variants to simplify the further proof for $pre \Rrightarrow post\langle x := E\rangle$. The following assignment law allows any assignment in its second half, provided the changed variants.

Lemma 14 (Initialized Assignment Law)**.** *Let x_0, y_0 denote the initial value of variant x, y, E be any $Expr$ in the programming language, $post\langle x := E\rangle$ replaces every x in the formula post with E. If $(x = x_0) \wedge (y = y_0) \wedge\ pre \Rrightarrow post\langle x := E\rangle$, $then\ x, y : [pre, post] \sqsubseteq$ **x = E**.*

Proof. Use the assignment law in Lemma 5 as $pre \Rrightarrow post\langle x := E\rangle$. □

Lemma 15 (Following Assignment Law)**.** *Let E be any $Expr$ in the programming language, $post\langle x := E\rangle$ replaces every x in the formula post with E. $x : [pre, post] \sqsubseteq x : [pre, post\langle x := E\rangle]$ **; x = E**.*

Proof. First use the sequential composition law, $x : [pre, post] \sqsubseteq x : [pre, post\langle x := E\rangle]$**;** $x : [post\langle x := E\rangle, post]$. Then, refine the second part using the assignment law, $x : [post\langle x := E\rangle, post] \sqsubseteq$ **x = E**. □

Alternate. The if-else alternation law simplifies the original formulation by excluding the need for explicit proof of the guard condition.

Lemma 16 (If-else Alternation Law)**.** *Let P, Q, and G be some formulae, then the specification* $x : [P,\ Q] \sqsubseteq if\ (G)\ (x : [P \wedge G,\ Q])\ else\ (x : [P \wedge \neg G,\ Q])$.

Proof. As $Pre \Rrightarrow G \vee \neg G$ based on the law of excluded middle, the lemma can be directly implied from the alternation law in Lemma 6. □

Iterate. We extend the origin iterative law to float numbers, which need to find an upper bound to guarantee the loop termination in finite time. The newly introduced initialized iteration law begins by assigning an initial value that satisfies the invariant. The second specification ensures that the invariant is preserved while the variant V changes during iteration, continuing until the negation of the guard condition is satisfied.

In practice, leveraging the convergence of monotonic sequences of real numbers, we replace the existing condition with the monotonic and bounded condition specified in Lemma 18. To prevent infinite loops, we incorporate an assertion to verify that the variant V decreases by at least the error bound determined by the precision of the floating-point representation.

Lemma 17 (Initialised Iteration Law)**.** *Let P, I, and G be some formulae, V be any variant expression, and i and M are positive integers, then the specification* $x : [P, I \wedge \neg G] \sqsubseteq x : [P, I]$ **;** $while(G)\ do\ (x : [I \wedge G,\ I \wedge (\exists i < M,\ V_i \rightarrow \neg G))]$.

Proof. First, applying the sequential composition law from Lemma 4, we derive:

$$x : [P, I \wedge \neg G] \sqsubseteq x : [P, I]; x : [I, I \wedge \neg G].$$

Next, we refine the second part using the iteration law from Lemma 7. It is important to note that, for scalability, we replace the condition for integer-valued variants with a general variant expression. To ensure termination of the iteration, there must exist a state of the variant that negates the guard condition after a finite number of iterations. □

Lemma 18 (Assertion Iteration Law)**.** *Let P, I, and G be some formulae, V be any variant expression, then the specification* $x : [P, I \wedge \neg G] \sqsubseteq x : [P, I]$ **;** $while(G)\ do\ (x : [I \wedge G,\ I \wedge V < V_0]; assert\ V \neq V_0)$.

Proof. First, follow the initialized Iteration Law. Then, note that the float precision error is e, then we have $\exists i = \lceil \frac{V_0}{e} \rceil < M, V < 0 \rightarrow \neg G$. □

Traverse. We introduce a traverse law to address problems involving arrays. The formula P incorporates the variants l and i, which may represent equations that recursively define a sequence. The subsequent refinement must ensure that the invariant $P(l, i)$ is maintained and that progress is made toward $P(l, i+1)$ in accordance with the principles of induction.

Lemma 19 (Traverse Law)**.** *Let l be the list of type T, natural numbers m and n denote the range, pre and P be some formula,* $l : [pre, \forall i : nat \wedge m \leq i < n \rightarrow P(l, i)] \sqsubseteq l, i : [pre, l[m]];\ i = m$ **;** $while(i < n)\ do\ (l, i : [P(l, i), P(l, i+1)]$**;** $i = i + 1)$.

Proof. First, applying the Wxpand law and the sequential composition law from Lemma 4, we derive:

$$l, i : [pre, l[i] \wedge i = m];\ l, i : [l[i] \wedge i = m, l[i] \wedge i = n].$$

Next, refining the second part using the initialized assignment law (Lemma 14) and the iteration law (Lemma 7), we obtain:

$$i = m \textbf{ ; } while(i < n)\ do\ (l, i : [P(l, i), P(l, i) \wedge 0 \leq n - i < n - i_0]).$$

Finally, applying the following assignment law from Lemma 15 to the specification,

$$[P(l, i), P(l, i+1) \wedge 0 \leq n - (i+1) < n - i] \textbf{; } i = i + 1,$$

This can be further simplified to match the target specification. □

The new laws are designed to be directly used by LLMs. The refinement calculus is transformed into background knowledge embedded within prompts, guiding the LLM in applying these laws effectively. The detailed instructions provided in the refinement laws enhance both interaction with the LLM and automated theorem prover (ATP) verification. These new laws are extensions of the core refinement laws, and their correctness can be formally derived from the core principles. A summary of these refinement laws is presented in Figure 3.7.

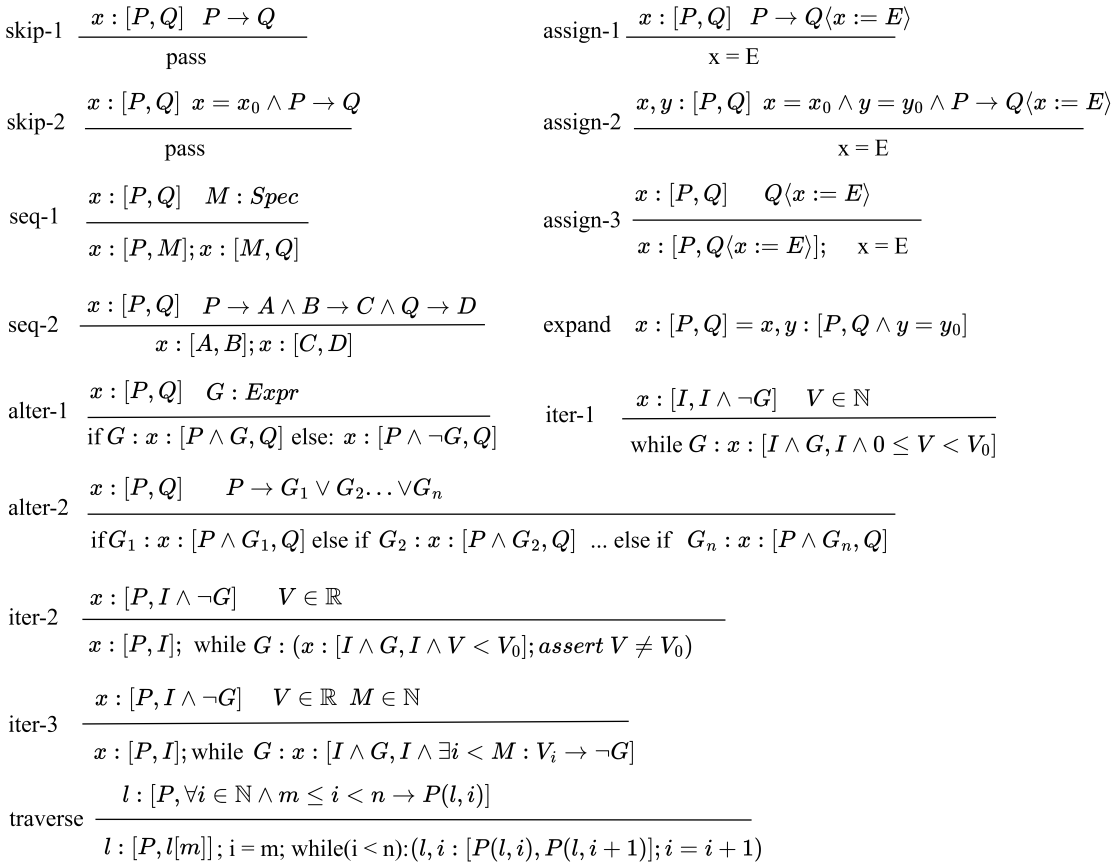

Figure 3.7: Illustration of the Refinement laws.

Theorem 2 (Soundness of Derived Refinement Laws)**.** *If* $\vec{x} : [pre, post] \sqsubseteq prog$ *is derivable from the laws in Sections 3.4.1 to 3.4.3, then* $\{pre\}prog\{post\}$ *holds.*

Proof. Immediate from the proof of individual laws. □

3.5 Interaction with LLM and ATPs

This section presents our approach, integrating the program refinement calculus with LLMs and ATPs.

3.5.1 Overview

Figure 3.8 illustrates an overview of our approach. In general, the formal specification, written in L_{spec}, is first transformed into an abstract syntax tree (AST). LLM4PR then extracts the conditions from the specification to input into the LLM.

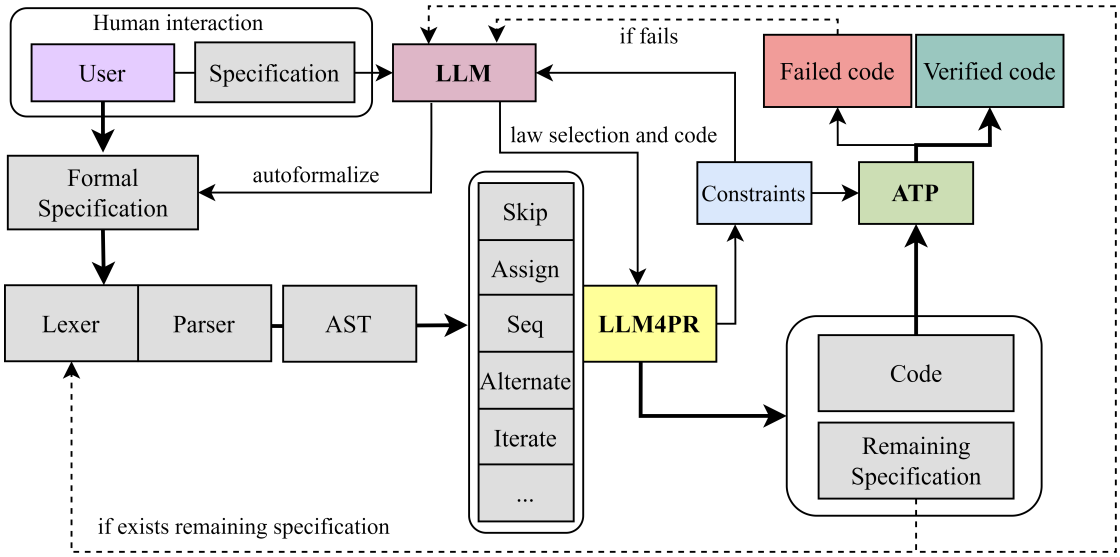

Figure 3.8: Overview of LLM4PR with the integration of LLMs and program refinement.

The LLM selects an appropriate refinement law to refine the specification based on the description and constraints of the formal specification. It subsequently generates the associated code that adheres to the selected law. LLM4PR, in turn, generates the proviso condition required by the refinement law and constructs verification scripts to validate the code produced by the LLM.

The ATPs attempt to automatically verify the generated scripts, providing either a success message or an error message as output. If the ATP verification fails, the LLM regenerates the code; if it succeeds, LLM4PR saves the verified code and produces a new specification for the next refinement step.

If multiple failures occur during verification, LLM4PR backtracks to the previous refinement step and specification, interacting with the LLM to select an alternative refinement law and generate the corresponding code.

Table 3.5 shows an example schema of LLM4PR's actions based on the refinement calculus. The refinement process can be visualized as a specification tree, where the nodes represent specifications (each containing a precondition and postcondition), and the refinement laws define the links between the nodes. Each node in the tree includes its associated specification and the possible refinement paths, along with the generated code for each path.

Table 3.5: LLM4PR will generate the specifications and conditions for further verification in ATP.

Law	**GPT4**	**LLM4PR**
Skip	-	verify $P \Rrightarrow Q$
Sequence	M	new spec $[P, M]; [M, Q]$
Assignment	$x = Expr$	verify $P \Rrightarrow Q\langle x := Expr\rangle$
Alternation	G	new spec *if* (G) $(x : [P \wedge G, Q])$ else $(x : [P \wedge \neg G, Q])$
Iteration	I, G	new spec $x : [P, I]$; while(G) do($x : [I \wedge G, I \wedge (\exists i < M,\ V_i \rightarrow \neg G)]$)
Traverse	P	new spec $l : [pre, l[m]]$; i = m; while(i < n) do $(l, i : [P(l, i), P(l, i+1)];\ i = i + 1)$

3.5.2 Complex Formal Specification

The process involves breaking down complex specifications into smaller, manageable sub-specifications and refining them bottom-up to build the complex specification.

Specification Formalization The initial input to LLM4PR is a formal specification, which requires the user to formalize their requirements. While LLMs can automatically translate informal descriptions into the formal specification language L_{spec}, users must verify the correctness of this transformation. Ensuring the accuracy of this step is crucial for maintaining the integrity of the refinement process. For the formal methods community, this verification step should not present a significant challenge and is essential—without it, the notion of correctness cannot be upheld.

Top-down Specification Decomposition We assume that all formal specifications verified by the user are accurate and align with their requirements. The specification decomposition algorithm begins with a high-level specification encompassing the entire system or software under development. Some functions may already have been refined and stored in the library, while others require new refinement.

Each specification is decomposed into smaller sub-specifications, iteratively breaking them down until all components are either refined or represented by atomic elements in the defined language. To match sub-specifications with those already stored in the library, we retrieve similar specifications.

Weaker preconditions allow for handling a broader range of potential inputs during implementation but reduce the approach's flexibility. Initially, the LLM retrieves pos-

sible equivalent specifications from the library and verifies their correctness using the logic laws of Lemma 1 and Lemma 2. Stronger postconditions, on the other hand, restrict implementation freedom, requiring closer alignment with the specified outputs. Retrieved specifications replace their corresponding sub-specifications where applicable. Finally, LLM4PR shares all specifications, including sub-specifications, with the user for validation.

Bottom-up Refinement For sub-specifications that are not yet refined, LLM4PR applies the procedure outlined in Definition 1 to formalize and refine them from the ground up. Each sub-specification is independently refined and validated, ensuring correctness at the module level before integration into the broader system. This bottom-up approach emphasizes thorough refinement and validation of each component, enhancing overall quality. The refinement steps, along with associated programs, are stored in the refinement library. These stored refinements can be reused for new specifications sharing the same preconditions and postconditions.

By combining top-down decomposition and bottom-up refinement, LLM4PR minimizes redundant specification efforts while ensuring consistency and reliability through the reuse of validated specifications.

3.5.3 Interaction with LLMs

Refinement-augmented LLMs We enhance the performance of LLMs in program refinement by embedding the refinement calculus as background knowledge. The LLM utilizes refinement laws through retrieval-augmented techniques to improve its effectiveness. Furthermore, we tailor the LLM specifically for program refinement tasks by designing prompts based on the formal specification language L_{spec} and the program language L_{pl} introduced earlier. To align the LLM with these tasks, we fine-tuned it using examples from Morgan's book [140], ensuring its outputs adhere to the principles of formal program refinement.

Dynamic Guidance with Prompts A *prompt* serves as an instruction guiding the LLM's output. Traditional static prompts, such as *Program Refinement for the following specification*, are replaced with dynamic, task-specific prompts. We treat the LLM as

a constraint solver, constructing prompts that include logical formulae representing the constraints of the specifications.

These formulae detail the requirements the LLM's output must satisfy, ensuring precise alignment with task-specific needs. Based on these prompts, the LLM selects the appropriate refinement laws and generates the corresponding code. Since each refinement step has its own generated specification, historical refinement steps need not be retained due to the congruence principles of *Hoare Logic*.

Prompt Engineering Prompt engineering involves the careful design of queries or inputs to optimize responses from an AI model. This practice is particularly relevant in machine learning and AI-driven interactive systems [19].

Our prompts are straightforward yet comprehensive, containing all relevant details about the refinement rules and specifications:

> Given the refinement rule ... The previous code ... is not correct since ... Provide a correct code satisfying the specification [pre, post].

3.5.4 Interaction with ATPs

Passively Verify. Once the LLM selects a refinement law and generates the associated code, LLM4PR uses ATPs to verify whether the code satisfies the constraints dictated by the chosen refinement law.

If the ATP confirms that the constraints are satisfied, LLM4PR applies the refinement law to the current specification, generating a new formal specification along with the verified code. If the verification fails, the ATP provides failure messages and potential counterexamples. The LLM then uses this feedback to generate alternative code. The retry process has a predefined limit.

In cases of repeated failure, LLM4PR reverts to the last successful refinement step and the associated valid specification. The LLM receives detailed feedback on the failures, including counterexamples and the last valid specification, to guide further attempts.

Modular Verification. Figure 3.9 illustrates the interaction between LLM4PR and the program refinement library. LLM4PR implements modular verification by dividing

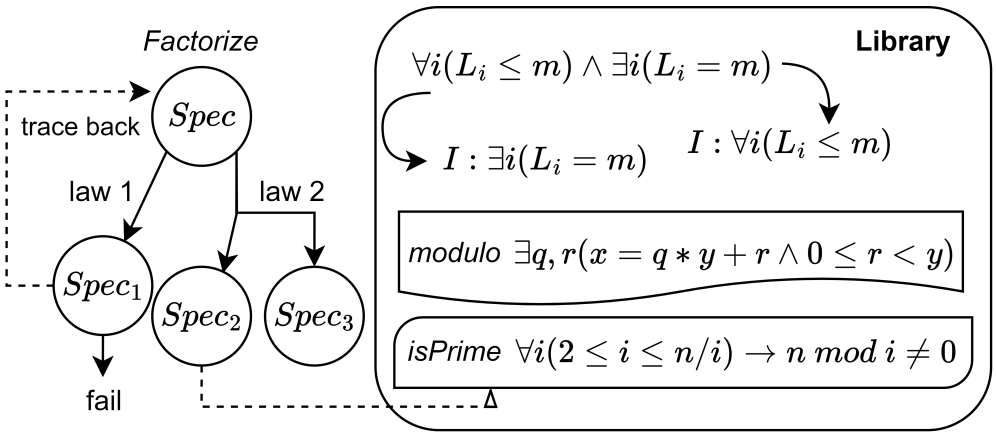

Figure 3.9: The specification tree and the program refinement library.

the refinement process into smaller steps and splitting specifications into independent modules. Each module, with its unique constraints, is verified separately by ATPs. This modular approach localizes errors to specific parts of the system, facilitating efficient error identification. Additionally, it allows for parallel verification, reducing overall verification time.

If a module fails verification, the ATP provides targeted feedback to the LLM, including specific reasons for failure and counterexamples. This focused feedback enables the LLM to refine or correct the problematic module without affecting other parts of the system. If all modules pass verification, LLM4PR integrates them into a complete and verified refinement step, ensuring the correctness of the overall program refinement process.

3.6 Evaluation

In this section, we first conduct a quantitative analysis of the most popular benchmarks, comparing them against state-of-the-art LLMs and a program refinement baseline.

3.6.1 Research Questions

We evaluate LLM4PR to address the following research questions:

- Can LLM4PR generate more robust programs compared to the baselines?
- Can the learning algorithm, together with the extended refinement calculus, reduce the time and depth required for the refinement process?

- Can the top-down splitting and bottom-up refining methodology for constructing a library of program refinement enhance LLM4PR's ability to solve complex problems?

3.6.2 Baselines

GPT-4. Generative Pre-trained Transformer 4 (GPT-4) [155] is a multi-modal large language model developed by OpenAI, representing the fourth generation of its GPT series. As a transformer-based model, GPT-4 adopts a paradigm involving pre-training on both public datasets and data licensed from third-party providers, focusing on predicting the next token. Following pre-training, the model undergoes fine-tuning with reinforcement learning based on human feedback.

CorC. CorC [177, 102] is an integrated development environment (IDE) for constructing programs in a simple while language, adhering to the Correctness-by-Construction (CbC) paradigm. Starting from a specification, this open-source tool assists developers in refining programs through a series of refinement steps, with the correctness of each step verified using the theorem prover KeY [77].

3.6.3 Benchmarks

We evaluate LLM4PR using the example programs provided in the baseline [177] and the HumanEval benchmarks, which are widely used in code generation evaluations [155, 151, 203]. Additionally, to assess the correctness and robustness of the generated code, we incorporate the *EvalPlus* dataset [124], which includes the same examples but with, on average, over 80 times more test cases than the original.

For formal specification evaluation, we use the Coq version of the dataset [16], and we manually review all specifications in the HumanEval dataset. This manual validation is essential for establishing correctness criteria, which is a necessary step in program verification and not an obstacle to the evaluation process.

3.6.4 Implementation

Our approach is implemented using Coq [14], CoqHammer [43], and GPT-4 by OpenAI [155]. Automatic verification is facilitated through CoqHammer, an open-source automated reasoning tool. In line with CoqHammer's design, we integrate four automated theorem provers: `cvc`[15], `vampire`[104], `Eprover`[183], and `Z3`[45], to enhance the automation of verification tasks.

Informal and formal specifications are input into GPT-4 to generate code snippets, which are then tested against corresponding test cases. We employ GPT-4, the state-of-the-art LLM, as the foundation of LLM4PR. To customize GPT-4 for our approach, we use instruction tuning [229] with classic refinement examples [140], enabling the model to learn extensions of the refinement calculus.

3.6.5 Experiment Results

The following sections give the results of the experiment.

3.6.5.1 Robustness of Code Generation

Table 3.6 presents the evaluation results for the HumanEval benchmarks. We include the latest powerful LLMs like LLama3 [175], GPT-3.5, and GPT-4 as baselines. The results for these baselines are taken from prior work [124].

To ensure fairness, we extend GPT-4's evaluation by incorporating formal specifications alongside natural language descriptions. When using only natural language descriptions, GPT-4 demonstrates the best overall performance among all LLMs. However, all LLMs experience a performance decline from HumanEval to EvalPlus, as EvalPlus includes more challenging test cases, revealing bugs in the generated code that fail the additional tests. In contrast, LLM4PR maintains consistent performance between HumanEval and EvalPlus due to its use of verified code, ensuring correctness. Theoretically, the code generated by LLM4PR can be regarded as canonical solutions, independent of the number of test cases.

We conducted a further error analysis comparing GPT-4 and LLM4PR. GPT-4 often exhibits *carelessness*, overlooking edge cases such as negative numbers or zero. This observation aligns with the experimental finding that incorporating formal specifications enhances GPT-4's performance, as these specifications provide useful constraints.

Table 3.6: A comparison of LLM4PR and LLMs on the HumanEval and EvalPlus benchmarks.

Model	Llama3	GPT-3.5	Claude-3	GPT4		LLM4PR
Input Specification	NL	NL	NL	NL	NL+ FS	FS
HumanEval Passed	125	126	136	145	148	**150**
EvalPlus Passed	116	116	126	128	142	**150**

LLM4PR's failures are primarily attributed to specifications involving complex data structures or unsupported functions. Overall, LLM4PR outperforms the LLMs, generating more robust and reliable code.

3.6.5.2 Efficiency of Program Refinement and Verification

Table 3.7 presents the evaluation results for the example programs in the CorC baseline [177]. Since the CorC baseline is not automated, we assume that the user has completed all the necessary refinement steps manually.

Across all evaluated algorithms, LLM4PR demonstrates a general reduction in both the number of refinement steps and the proof times compared to CorC and the Initial variants, which rely solely on the core refinement laws. The observed variability in refinement steps and proof times among different algorithms reflects differences in their complexity and optimization challenges.

Algorithms with inherently complex structures, such as pattern matching and logarithmic approximation, appear to benefit more significantly from the inclusion of advanced refinement laws and optimizations. Notably, the experiment reveals a significant difference in proof time for complex algorithms, with LLM4PR requiring substantially less time than both CorC and the Initial variant.

3.6.5.3 Capability of LLM4PR

Table 3.8 compares two approaches: one utilizing LLM4PR and the other relying solely on a basic top-down splitting and bottom-up refinement algorithm without a program-refined library. The evaluation was based on 50 examples selected from the HumanEval benchmarks, specifically those containing more than two sub-questions. The refinement process was conducted with a maximum allowable duration of 600 seconds.

Using LLM4PR significantly reduced the average number of refinement steps from

Table 3.7: Our LLM4PR and baseline *CorC* comparison on several program refinement problems.

Metrics	# Refinement Steps			Proof Time(s)		
Approach	CorC	Initial	LLM4PR	CorC	Initial	LLM4PR
Linear Search	5	5	4	0.4	0.2	0.1
Max Element	9	10	5	1.2	1.0	0.2
Pattern Matching	14	16	8	54.9	35.8	24.5
Exponentiation	7	7	5	15.2	14.4	10.4
Log Approximation	5	5	4	42.7	22.9	20.1
Dutch Flag Sort	8	9	5	5.7	4.3	4.1
Factorial	5	5	3	3.6	1.5	0.4

21.4 to 5.6 and shortened the refinement time from approximately 514 seconds to 275 seconds. Additionally, the refined programs generated with LLM4PR were more compact, demonstrating that the bottom-up approach results in smaller, well-organized library programs.

The modular verification enabled by LLM4PR 's library further minimized proof times and decreased the LLM's fallback rate. This highlights LLM4PR 's effectiveness in guiding the LLM to produce accurate code that adheres to the given constraints while refining specifications.

Overall, LLM4PR greatly enhances the program refinement process by streamlining the workflow, reducing complexity, and increasing both the reliability and success rates of program generation.

Table 3.8: A comparison of LLM4PR and its variant without the program refinement library is performed on the EvalPlus benchmarks.

Model	No Library	LLM4PR
#Refinement step	21.4	5.6
Refinement time(s)	∼514	∼275
Proof time(s)	∼215	∼87
Fall-Back rate(%)	26.87	9.45
# Program size	41.3	14.1
Pass rate(%)	52	82

```
// pre:  (N:float)(e: float) := N >= 0 /\ e > 0
// post: (x:float)(y: float) := x*x <= N < y*y /\ y <= x+e
# LLM selects Sequential Composition Law: Part 1
// pre_1:= N >= 0 /\ e > 0
// post_1:= x*x <= N < y*y
# LLM selects Assignment law
x = 0
y = N+1
# verify pre_1 -> post_1(x := 0, y := N+1)
# Part 2
// pre_2:= x*x <= N < y*y
// post_2:= x*x <= N < y*y /\ y <= x+e
# LLM selects Iteration law: I(pre_2) G(~(y <= x+e))
while y > x+e:
  if y > x+e:
    // pre_2_1:= pre_2 /\ x+e < y
    // post_2_1:= pre_2 /\ (...)
    # LLM selects Alternation law G((x+y)/2*(x+y)/2 > N)
    if (x+y)/2*(x+y)/2 > N:
      // pre_2_1_1:= pre_2_1 /\ (x+y)/2*(x+y)/2 > N
      // post_2_1_1:= post_2_1 /\ (...)
      y = (x+y)/2
      # verify pre_2_1_1 -> post_2_1_1(y := (x+y)/2)
    else:
      // pre_2_1_2:=  pre_2_1 /\ (x+y)/2*(x+y)/2 <= N
      // post_2_1_2:= post_2_1 /\ (...)
      x = (x+y)/2
      # verify pre_2_1_2 -> post_2_1_2(x := (x+y)/2)
```

Figure 3.10: Program Refinement Code Example of the Square Root Algorithm

3.7 Case Study

In this section, we show three examples of the program refinement code.

3.7.1 Square Root Algorithm

We demonstrate how LLM4PR addresses the motivating example from Section 3.1 in Figure 3.10. The verification statement serves as the proviso condition required to apply the refinement law. For clarity, we omit the iteration termination check condition in `(...)`.

In detail, the LLM sequentially splits the original specification into two parts. The first specification defines x and y such that $x^2 \leq N < y^2$, which can be implemented using assignment. Importantly, the assignment for y must satisfy the postcondition constraint $N < y^2$, avoiding potential LLM-generated bugs like assigning $y = N$ as shown in Figure 3.1.

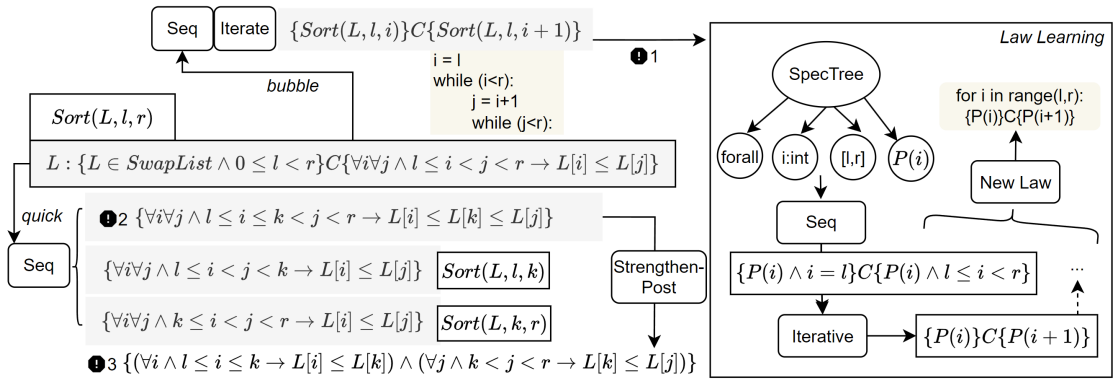

Figure 3.11: Bubble Sort and Quick Sort with Program Refinement.

The second specification preserves the invariant $x^2 \leq N < y^2$ and adjusts the variants x and y iteratively until the condition $x + e \geq y$ is met. This iteration can be implemented with a loop. The invariant, guard condition, and variant are derived directly from the specification. The LLM reduces the gap between x and y by assigning either x or y to the mean of x and y. Additionally, alternation introduces constraints that strengthen the precondition, simplifying the derivation of the postcondition.

Compared to LLM-generated code, LLM4PR ensures that each refinement step is verified, as each step is paired with its corresponding specification. The LLM is utilized to select the refinement law and automatically generate associated code based on the constraints it generates. These constraints are built automatically according to the chosen law and the generated code in LLM4PR. When a refinement law is applied, the new specification is formally generated in alignment with the refinement laws.

3.7.2 Sorting Algorithm

We utilize the bubble sort and quick sort algorithms as representative examples to demonstrate the extensibility of our refinement framework. Following the approach in [26], we first build the array type, *SwapList*, which is characterized by its sole ability to perform swap operations:

$$swap : SwapList \rightarrow num \rightarrow num \rightarrow SwapList \tag{3.7}$$

For the sorting problem illustrated in Figure 3.11, the specification comprises a *SwapList* and a postcondition ensuring that the array elements are arranged in ascending order. The refinement process starts by initializing the variable $i = l$, establishing the

invariant that the sublist $L[l : i]$ is sorted when $i = l$. Subsequently, the refinement expands this sorted sublist incrementally from $L[l : i]$ to $L[l : i + 1]$, with a corresponding decrease in the variant $r - i$. A similar refinement strategy applies to the variable j. Both i and j iterate within bounds, progressing from the left to the right under specified constraints, reflecting the iterative nature of these integer variables.

(1) Law Learning. Such array-based iterative structures are commonplace in the dataset. The combination of the core refinement laws serves as a basis for deriving new advanced refinement laws based on the correctness-by-constructions [21]. These insights lead to the formulation of a new refinement law (we called a *traverse* law) that merges Seq and iteration laws, thereby streamlining the origin refinement process. Intuitively, LLM4PR will learn the patterns of law sequence in the refinement process that have been built and conclude the combination of refinement laws that appear frequently to build a new law for future refinement. The traverse law is derived for traversing the elements in the array from one index to another index. It also simplifies the proof obligation just to maintain the invariant $P(i)$ without loop termination checking.

(2) Recursion. Another refinement way is to break down the original problem into smaller, similar sub-problems, a strategy widely recognized as recursion. In this context, the sequence is divided into two parts: the elements in the first part are less than or equal to $L[k]$, while those in the second part are greater. Each part is then sorted independently.

The base case is the sorting of a single-element sequence. The associated proof obligation requires demonstrating that the three newly introduced postconditions collectively imply the original postcondition. This can be established through structural induction.

$$(\forall i \forall j \wedge l \leq i \leq k < j < r, L[i] \leq L[k] \leq L[j]) \wedge (\forall i \forall j \wedge l \leq i < j < k, L[i] \leq L[j]) \wedge (\forall i \forall j \wedge k \leq i < j < r, L[i] \leq L[j]) \rightarrow (\forall i \forall j \wedge l \leq i < j < r, L[i] \leq L[j]) \tag{3.8}$$

(3) Law Matching. Further refinement seeks to disentangle conditions related to i, j and prove that the revised conditions can deduce the original requirements. The previously learned traverse laws can **not** apply since the invariants $L[i] \leq L[k]$ and $L[k] \leq L[j]$ are disrupted when $L[k]$ changed, but a general iteration law can match the new specification that is in a conjunction normal form. LLM4PR starts with initializing $k = l$ to satisfy the

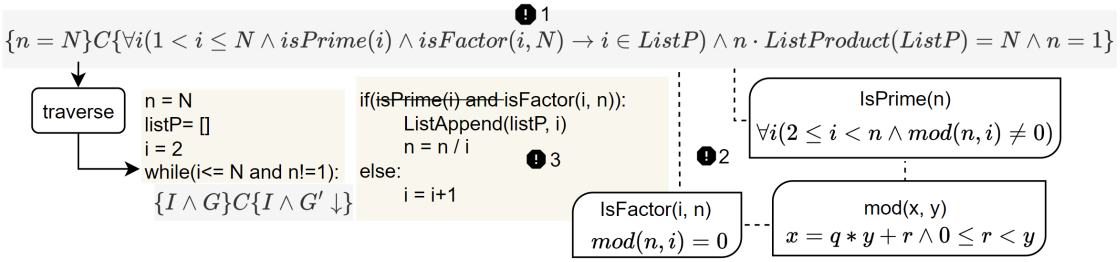

Figure 3.12: Prime Factorization with Program Refinement.

first condition and then iterating j to ensure $L[k] \leq L[j]$ while the invariant $L[i] \leq L[k]$ is maintained.

3.7.3 Prime Factorization Algorithm

Programmers widely use functional abstractions to handle complexity effectively. In program refinement, the specification naturally defines the inputs and outputs of a functional abstraction. Based on this, we design a top-down approach to decompose the specification and a complementary bottom-up strategy to refine and synthesize functional abstractions that encapsulate standard operations.

(1) Formal Specification. We presume that all user-verified formal specifications are correct and aligned with their requirements. LLM4PR engages users to help formalize high-level specifications. Some functions specified may already have been refined and stored in the library from prior use, while others may require new refinements. For instance, as illustrated in Figure 3.12, the requirement is specified as follows:

Prime factorization of any given number is to break down the number into its factors until all of its factors are prime numbers.

The formalized specification of prime factorization consists of *IsPrime*, *IsFactor*, and *ListProduct*. We assume that only *ListProduct*, which represents the products of all the elements in the list, has been refined and stored in the library. Then, LLM4PR will use the top-down algorithm to split further and formalize the specification of *IsPrime* and *IsFactor*. Each sub-specification has its own specification, and they will be split into smaller specifications like *mod* until all elements defined in their specifications have been refined or atomic elements in the language. Finally, LLM4PR will share all the specifications with the user to ensure the correctness of the formal specification.

(2) Functional Abstraction. The above top-down algorithm formalizes and splits the high-level specification into a unilateral connected digraph of sub-specifications. Then, a bottom-up algorithm will refine the remaining sub-specifications, synthesize the programs, and store them in the library. In this example Figure 3.12, LLM4PR will first refine the *mod* with the iteration law where the invariant is $x = q \cdot y + r$ and the guard condition is $r \geq y$. The LLM generated code is $q = q + 1; r = r - y$ and LLM4PR verifies as follows:

$$x = (q + 1) \cdot y + r - y \rightarrow x = q \cdot y + r \tag{3.9}$$

(3) Refinement and Verification. For prime factorization, LLM4PR will use the *traverse* law to initialize the variables with the invariant $P(i')$:

$$\forall i(1 < i \leq i' \wedge isPrime(i) \wedge isFactor(i, N) \rightarrow i \in ListP) \wedge n \cdot Product(ListP) = N \tag{3.10}$$

It means that for all i from 2 to i', ***ListP*** contains all the elements that are prime numbers and factors of the N. LLM4PR first initializes the n with N satisfying the precondition $n = N$ and the ***ListP*** with empty array satisfying the condition $n \cdot Product(ListP) = N$ since $n = N$. The associated verification of the initialization is also conducted in a sequence, following the value of prior variables. This modular verification in program refinement reduces the workload of the formal verification system and enables the LLM4PR to prune the proof for different subprograms. The proof of the invariant can also reuse the modularized proof of ***IsPrime*** and ***IsFactor*** without considering the details of these two functions. Another point lies in the optimization of the conditions. In the if-condition of the program, the LLM will generate the code ***IsFactor(i,n)*** since

$$P(i') \wedge isFactor(i', n) \rightarrow isPrime(i') \tag{3.11}$$

Intuitively, if i' is not a prime, k exists that is less than i' and is a factor of i'. Since i' is a factor of n and n is a factor of N, k is also a factor of N, which contradicts the invariant that all factors of N that are less than i have been considered. It is a non-trivial resolution for the prime factorization algorithm to reduce the time and proof of ***IsPrime***.

3.8 Threats to Validity

First, if a refinement lacks proof of loop termination in the iterative law, we still consider it *partially correct*. To address this limitation, we provide additional iteration

laws in Section 3.4, which avoid relying on termination conditions. This approach acknowledges that proving termination is a complex and generally undecidable problem.

Second, LLM4PR is designed to guide the LLM in generating robust code rather than building specifications. It still requires human input to provide formalized specifications, similar to other formal methods tools.

Third, the current set of data structures and learned refinement laws is relatively straightforward, as LLM4PR's capabilities depend on the power of LLMs and automated theorem provers (ATPs). To mitigate this limitation, users can actively interact with the LLM during the program refinement process by selecting refinement laws, constructing proofs, and verifying them.

Chapter 4

Program Documentation

4.1 Introduction

Code documentation and summarization are important to facilitate code comprehension, which in turn supports various programming tasks such as code review [178], refactoring [129, 63], and code education [42, 185]. Recent years have seen that neural network driven code documentation and summarization techniques are emerging [7, 109, 3, 206, 89, 6, 25, 202, 227, 215, 237, 184, 222, 164, 115, 68]. Many researchers regard code summarization as a machine translation problem (e.g., translating English to German). Taking the program code tokens as the source and labeled comments as the target, researchers have proposed various language models to translate or summarize the former to the latter. The models are rapidly evolving, with the neuron size growing from millions to billions, gravitating to the state-of-the-art ChatGPT [154]. Recent works also utilize retrieval-augmented techniques(RAG) to boost the generator's performance, using similar examples. While achieving good performance in generating comments, existing techniques are designed with a close-world assumption, i.e., the comments are only derived from the code body of the target function.

However, unlike English-to-German translation, where an English word is usually informative to infer a German counterpart, there is usually an *information gap* between the target code and its comments. The retrieval selector and generator in the baselines consider only the target method body and ignore the rich structural information of code entities like method invocations, declarations, and co-locations. Compared to translating the code into the comments in a *general* way, a more precise summa-

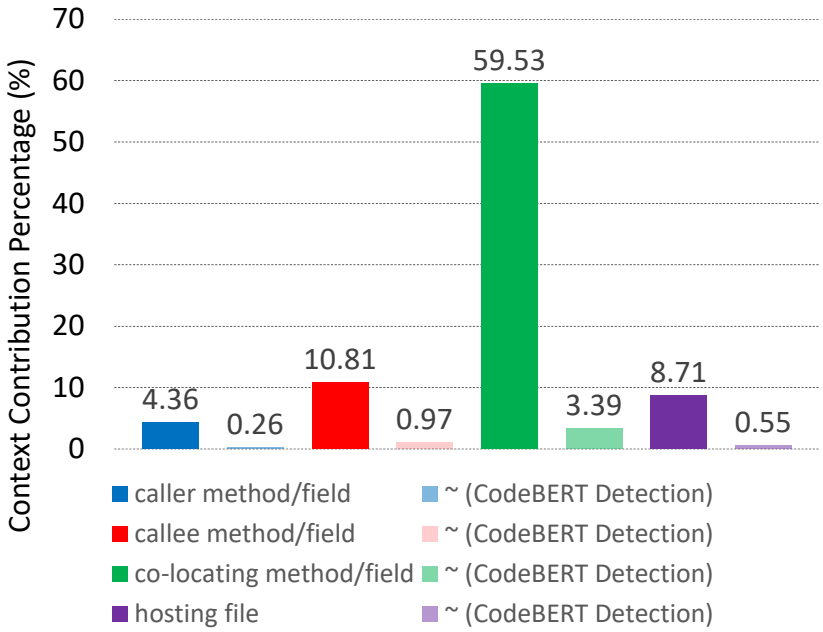

Figure 4.1: The lexical contribution of different context types to the comment of a target method.

rization requires the solution to be *project-specific*. For example, in the *bazel* project, a method as `getSize(){return size;}` has the comment as "`Returns the uncompressed size of the entry data, or -1 if not known`". The concepts such as "compression", "entry", and "data" are hard to derive just by observing such a method body.

In practice, the programmers infer the comments with an open-world assumption: they can be derived from any helpful source (e.g., co-located methods, caller/callee methods, or documentation) as long as they help understand the code and accomplish the tasks. Besides, as [131] points out, the most prevalent prediction errors of current models are caused by missing information, which inspires us to include any helpful information to predict the code comment.

Our empirical studies on 100 popular Java projects from GitHub (with top number of stars) show that (1) there is a large lexical gap between the comments and the method bodies and (2) the representative deep language models serve as a limited remedy. Figure 4.1 further shows the linguistic contribution of different types of context to the comment of a target Java method. Each color represents a type of context. For one color, each left bar (deeper color) indicates the ratio of nontrivial words in the comments, which are missing

in the target method body and present in a specific context. Each right bar (shallower color) indicates the ratio of the above nontrivial words that CodeBERT can recommend. For example, given a Java method, on average, 59.53% nontrivial words in its comment can be found in one of the methods co-located in the same Java file but do not appear in the code body. Nevertheless, a CodeBERT-based comment generator [57] can only recommend 3.39% of the missing words in the method body since the co-location file information is missing. The numbers are 4.36%, 10.81%, and 8.71% for the context type of caller method/field, callee method/field, and hosting file (i.e., class name, class comments, etc.) while existing state-of-the-art techniques such as CodeBERT [57] serves as a very limited remedy. Generally, many nontrivial words in comments usually include project-specific domain knowledge, which indicates that they are important in a specific project but *rare* words in the general projects. Therefore, it is difficult for a language-model-based comment generator to synthesize project-specific concepts just by training on general projects.

While context is known as an essential remedy, it is still an open question of how to (1) define context scope, (2) retrieve helpful context, and (3) fuse the retrieved context with the target method to facilitate the comment generator. The state-of-the-art approaches extract the contextual information from code in the same file [81], code in the same project [13], and similar code pieces [225, 126]. The latest approaches [145, 68] mainly utilize sentence transformers to embed the code body to a vector and search for similar code examples. The retrieved code and comments will be the context and integrated with the target code to predict the target comment. In detail, given a pre-defined scope (e.g., a retrieval database, many files, or projects), some similar methods are selected and fed into an encoder to have their embeddings. Finally, a new embedding consisting of original and retrieved-context embeddings is further fed into a transformer model to generate comments.

Although these approaches can be effective in a way, they still suffer from the technical challenges of selecting and aggregating useful contexts.

- **Context Selection**: First, the useful contextual information does not necessarily appear in the training set or the retrieval database, especially when the programmer is working on a new project. Such structural awareness capability is also a limitation in the GPT-4 model. Second, the contextual information of a method can include both

practical and irrelevant contexts. Third, although many retrieval-augmented methods use comments from similar code as a template, similar code does not necessarily imply similar comments.

- **Context Aggregation**: Given a piece of useful context code, effectively extracting and integrating the contextual information to generate the comment is still a nontrivial task. The state-of-the-art usually sets a fixed number of retrieved contexts or a threshold for similarity score. Then, it aggregates all the context and target code into one encoder embedding for the decoder to generate the comment. It is unclear how much contextual information contributes to the boosted performance of the comment generator. It remains a question whether the performance of the comment generator will continue to increase if more contextual information is provided. Since the signals of both context and the target code are mixed, it is unclear whether the aggregated embedding is (1) informative enough to derive the precise comment and (2) discriminative enough to avoid noisy signals.

We propose a novel approach, CProSum (**C**ontextual **Pro**mpt Based **Sum**marization), to address both the problems of context selection and aggregation. Our rationale lies in the fact that a context evaluator can be trained to evaluate the structural context information and predict the score of how much the context will help the comment generator. The comment generator can be trained interactively with the selected context as prompts for generating the comment and computing the score feedback to the evaluator. CProSum has the following advantages: First, most structural contexts, such as classes, fields, and callees in the knowledge graph, usually exist in real-world scenarios and can be easily accessed. Second, structural context code, like co-location methods, may have different code bodies but usually share similar commenting styles with the target code as they typically come from the same programmer or team. Third, the context evaluator can score each context to clearly indicate how much it will contribute to the performance boost and guide the user in balancing choosing shorter input tokens or achieving better performance. Finally, the prompt and attention mechanism can help the model extract helpful information and remove the noise based on the score and structural type.

Technically, given a target code, we define its contextual scope by tracking its diverse code relations to capture its calling methods, called methods, co-locating methods, declaring class, etc. We regard the existing neighbors within three hops in the code knowledge

graph as the structure context. Then, we learn a context evaluator and the comment generator interactively. On the one hand, the context evaluator is trained to predict a score based on how much the context will boost the comment generator's performance. On the other hand, after ranking the scores predicted by the context evaluator, the comment generator is trained to utilize that useful context as a prompt to generate new comments. Based on the new comments and gold comments, new scores will be calculated and sent back to the context evaluator. The two models are trained iteratively and interactively until the loss of both models converges. Note that our solution is non-intrusive. Our comment generator follows an encoder-decoder architecture and can be adopted by many existing language models [57, 74, 3, 109]).

To evaluate our approach, we construct a graph dataset of the top 100 Java projects by stars, using defined code relations as contextual information. We implement our graph-based solution based on transformer [90], which outperforms eight baselines in the comment generation tasks. Besides, we find that equipping the state-of-the-art transformer-based comment generator with CProSum framework can well improve their performance by, on average, 21.11% on BLEU4, 18.14% on METEOR, and 14.03% on ROUGE-L. Our quantitative and qualitative analysis further shows that CProSum can effectively extract and utilize the context information.

In summary, this work makes the following contributions:

- We design, CProSum, a context-aware comment generator with a context evaluator, where the latter learns to score the context of the target code, and the former utilizes the useful context as a prompt to generate more precise and adaptive comments.
- We construct a *graph dataset* regarding comprehensive project information from 100 popular Java open-source projects, including 7.4M nodes and 8.8M edges, facilitating the follow-up structural context-augmented approaches in the SE and AI community.
- We show that CProSum can significantly improve the performance of code summarization. The performance boost is effective across different transformer-based model architectures and code samples under different distributions.

4.2 Overview

4.2.1 Motivating Example

Figure 4.2 shows a code example extracted from the Spring-framework project [194]. The pink region is the comment of the code. The target method shares a similar comment with the second code retrieved by our tool CProSum, although they have different code tokens. The third method retrieved by baselines [145] and [68] with the highest score 79.1% has a different code comment. The fourth comment generated by GPT4 has many details to explain the method step by step but lacks the key point that the sort is via getPatternComparator. However, our retrieved examples will contain this method invocation information as they have similar structural information in the knowledge graph. This will further help the GPT4 model to include the method invocation PathMatcher#getPatternComparator(String) that is missing in both the code body and the retrieved example by baselines. In this example, the target method `getMatchingCondition()` checks if any of the patterns match the given message's destination and returns a new instance containing the matching patterns. The patterns sorted by their specificity using `{@link PathMatcher#getPatternComparator(String)}`. In the Spring-framework project, there are 33 methods called `getMatchingCondition()`, and there are no regulars among the code tokens similarity and comment similarity. For example, semantic similarity-based methods like [145] and [68] utilize the sentence transformer model st-codesearch-distilroberta-base model to score the code tokens. Due to the limitations of transformer modes, short and token-similar codes will have a higher score like the third code in Figure 4.2. However, they may have a contrasting code comment that would mislead the comment generator.

Besides, the standalone target code body lacks the structural information of method invocations, imports, and types. Figure 4.3 shows that the target method `DestinationPatternsMessageCondition getMatchingCondition()` have similar graph structure with our retrieved method `PatternRequestCondition getMatchingCondition()`. In other words, the retrieved method by CProSum and the target method in Figure 4.2 share a similar graph structure in the knowledge graph, although they differ on code tokens. The two methods are called by their neighbors (but do not share the same), with code names like matchSortPatterns, compareNumberOfMatchingPatterns,

Target Method

Documentation: Check if any of the patterns match the given Message destination and return an instance that is guaranteed to contain matching patterns, sorted via {@link org.springframework.util.PathMatcher#getPatternComparator(String)}.

```java
public DestinationPatternsMessageCondition getMatchingCondition(Message<?> message) {
    Object destination = message.getHeaders().get(LOOKUP_DESTINATION_HEADER);
    if (destination == null) {
        return null;
    }
    if (this.patterns.isEmpty()) {
        return this;
    }
    List<String> matches = null;
    for (String pattern : this.patterns) {
        if (pattern.equals(destination) || matchPattern(pattern, destination)) {
            if (matches == null) {
                matches = new ArrayList<>();
            }
            matches.add(pattern);
        }
    }
    if (CollectionUtils.isEmpty(matches)) {
        return null;
    }
    matches.sort(getPatternComparator(destination));
    return new DestinationPatternsMessageCondition(new LinkedHashSet<>(matches), this.routeMatcher);
}
```

Retrieved by Our Method

Documentation: *Checks if any of the patterns match* the given request and *returns an instance that is guaranteed to contain matching patterns, sorted via {@link PathMatcher#getPatternComparator(String)}.*

```java
public PatternsRequestCondition getMatchingCondition(HttpServletRequest request) {
    String lookupPath = UrlPathHelper.getResolvedLookupPath(request);
    List<String> matches = getMatchingPatterns(lookupPath);
    return !matches.isEmpty() ? new PatternsRequestCondition(new LinkedHashSet<>(matches), this) : null;
}
```

Retrieved by Baseline

Documentation: *Check if* this condition matches the given *Message* and returns a potentially new condition with content tailored to the current message. For example a condition with destination patterns might return a new condition with sorted, *matching patterns* only.

```java
T getMatchingCondition(Message<?> message)
```

GPT4 Prediction

Determines the matching condition for a given message based on its destination. This method first checks if the destination of the message is specified. If not, or if there are no patterns it returns {@code null} or {@code this} respectively, indicating no specific matching condition. For messages with a specified destination, the method iterates through all configured patterns to find matches. A match is found either if a pattern exactly equals the message destination or if it matches according to a custom matching logic. All matching patterns are collected, sorted based on their relevance or specificity, and then used to construct a new {@link DestinationPatternsMessageCondition} instance, which is returned as the matching condition.

GPT4 + Our Retrieved Examples

Checks if any of the patterns match the given message's destination and returns a new {@link DestinationPatternsMessageCondition} *instance containing the matching patterns*, if any, *sorted by their specificity using {@link PathMatcher#getPatternComparator(String)}.* If no destination is present in the message or no patterns match, the method returns {@code null} or {@code this} respectively.

Figure 4.2: An example extracted from the Spring-framework project.

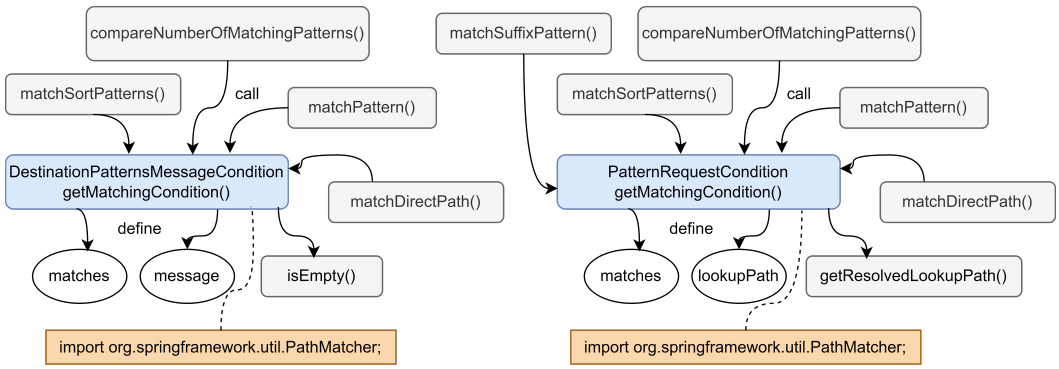

Figure 4.3: An example shows how our knowledge graph structure-based approach outperforms the traditional retrieval method.

matchPattern, and matchDirectPath. They share the same 13 imports, like PathMatcher. The graph structure indicates similar functionality, although the code tokens may differ. The same code comment tokens, such as "check if any of the pattern," "returns an instance," and "sorted via PathMatcher," shed light on the prediction of the target method comment. Their neighbors, such as `matchDirectPath`, `matchPattern`, `matchSortPatterns`, are quite similar, although they are not the same nodes in the knowledge graph. The two methods share 13 imports, including the key method invocation `PathMatcher`. The structural information and context information regarding the target method of these graphs will help predict the comment.

Generally, we have the following observations:

- **Large Language Models** Compared to the ground truth, the classical LLMs generate detailed but less concise comments that largely borrow superficial words like return type and method name. However, 33 methods named `getMatchingCondition()` in the spring-framework project with quite different comments. We need to describe each function and its characteristics.

- **Retrieval-augmented Methods** Retrieval-based methods first try to find the most similar code representation in the pre-defined database. Even if some related code (in the spring-framework project) of the target code appears in the retrieval database, the retrieval still lacks structural information beyond similar code body tokens.

- **Usefulness of CProSum** CProSum find a more structurally similar code from the code knowledge graph where the similar template of the code comment and the relevant contextual information is present in the retrieved comment. Besides, the comment generator

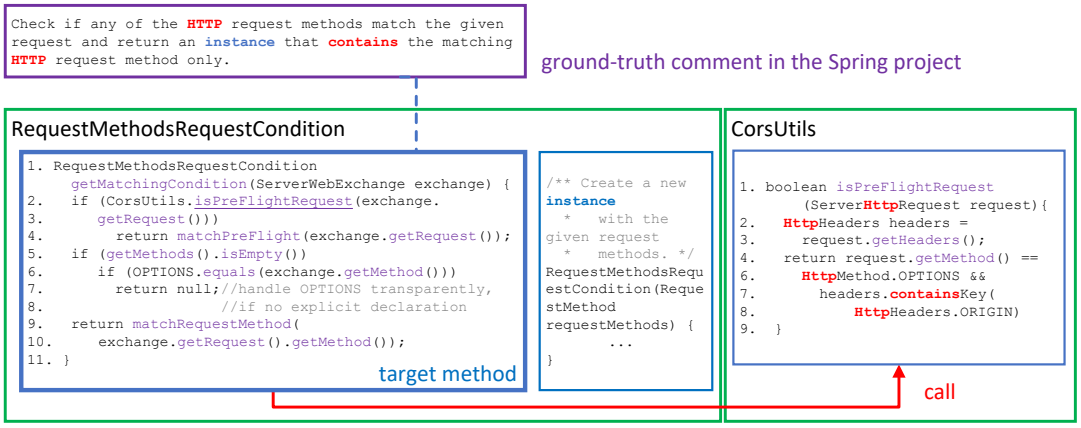

Figure 4.4: An example for context-based comment generation, extracted from the Spring framework.

of CProSum will also include the graph context information shown in Figure 4.3 to enhance the generator with more related hints.

4.2.2 Our Solution

Given the target code in Figure 4.4, CProSum takes three steps for its comment generation.

Step 1. Structure Extraction CProSum first extracts the related neighbors with different code types. In this example, the target method has six method invocations, 14 co-located methods, and one declaring class. In real-world scenarios, some co-located and caller methods may be unavailable. To enlarge the structural context, we also include the method invocation of the existing caller and callee methods, where a caller is a function that calls the target method, a callee is a function that was called, and a co-located method is a function that shares the same class with the target method. We regard some existing neighbors within three hops in the knowledge graph as the structure context.

Step 2. Structure Evaluation Then, CProSum search in the code knowledge graph to find some potential of structurally similar entities with the structural evaluator model. The evaluator will generate the graph embedding of the target code and scores all the nodes in a static analysis order that first checks the methods in the same public class, then the methods in the same package, and the same project, and finally goes through the nodes in the knowledge graph. We then rank all the scores and use the top-k nodes as prompts for

the comment generator. To improve performance, the user can stop the search procedure early or set a threshold for fast searching.

Step 3. Context Fusion Finally, CProSum fuses the target code tokens with the selected examples and the graph structure, including code type and scores, into a context-aware comment generator model. The generator can take any backbone model such as CodeBERT[57] or Code T5[204].

4.3 Approach

4.3.1 Embedding-based Representation

Embedding-based representation is a method of encapsulating the semantic and syntactic properties of software specifications into high-dimensional numerical vectors. It can handle more complex and nuanced specifications that might be difficult to express formally. Embedding-based representation is useful for machine learning applications, where the ability to approximate and generalize from examples can enhance system performance. This approach is derived from techniques commonly used in natural language processing (NLP) and machine learning, where embeddings have proven highly effective in capturing the nuanced meanings of words and sentences.

Vector Space Modeling Code tokens are converted into high-dimension vectors to capture the semantic meaning. Techniques such as word embeddings (e.g., Word2Vec [136], GloVe [163]) or sentence embeddings (e.g., BERT [48], Sentence-BERT [173]) are typically employed. Similar code tokens will be closer together in this space while differing ones will be farther apart.

Feature Extraction Deep learning models like Convolutional Neural Networks (CNNs) or Transformers can be used to process these vectors, extracting and learning complex patterns that represent the underlying features of the code tokens. This allows the model to understand and quantify relationships, dependencies, and patterns within the code that might not be readily apparent.

The advantages of Embedding-Based Representation are:

- Flexibility: Capable of handling a wide range of code, including those that are less structured or too complex for traditional formal-based methods.
- Scalability: Easily scales with the addition of new data, making it suitable for dynamic environments where specifications are frequently updated or expanded.
- Integration: Seamlessly integrates with other AI components, supporting systems where machine learning models need to interact with or adapt to changing code distributions.

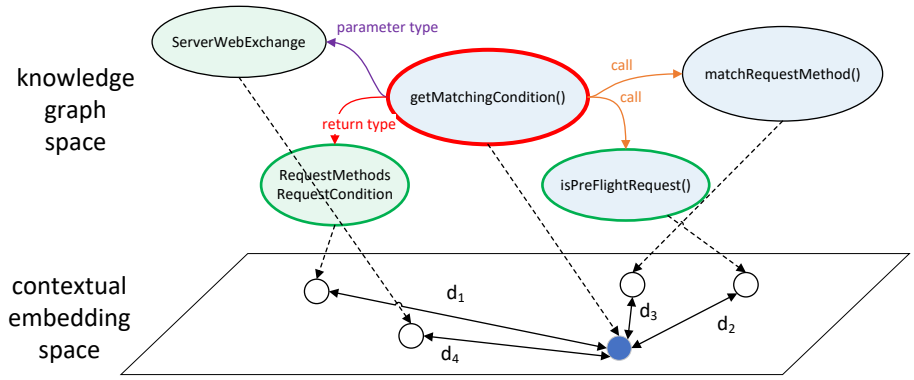

Figure 4.5: The contextual embedding space before learning

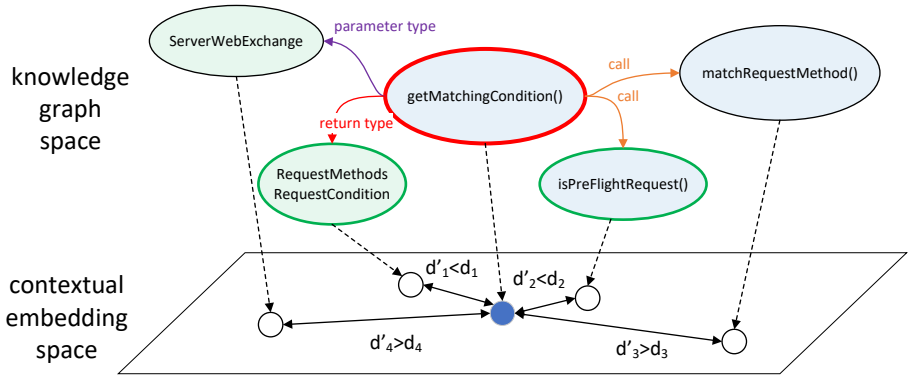

Figure 4.6: The contextual embedding space after learning

Knowledge Graph Enhanced Representation We explore achieving a more informative embedding representation with the knowledge graph. To illustrate our idea, we

show how the embedding of the program entities (e.g., method, class, etc.) in the embedding space can change in Figure 4.5 and Figure 4.6. We represent the target method for comment generation as an ellipse with a red border; the program entities are contributable to generating comments as one with a green border and not as contributable as one with a black border. The neural network encoder is guided to project a program entity in a *knowledge graph space* into a *contextual embedding space* so that the program entities that contribute more to their comments are closer. In Figure 4.6, the embedding distance between the target method and comment-contributable/incontributable entities (projected in the contextual embedding space) are learned from random distance to the distance ordered by comment contribution. We design a contextual fusion solution to make comment-contributable entities closer.

Table 4.1: Meta-path schemas defined on the code knowledge graph

Context Meta-path Type	**Context Semantics**
<method, call, method>	a method calls another method
<method, called-by, method>	a method is called by another method
<method, has-return-type, class>	a method has the return type of a class
<class, as-return-type, method>	a class serves as the return type of a method
<method, has-return-type, interface>	a method has the return type of an interface
<interface, as-return-type, method>	an interface serves as the return type of a method
<method, has-parameter-type, class>	a method has a parameter type of a class
<class, as-parameter-type, method>	a class serves as a parameter type of a method
<method, has-parameter-type, interface>	a method has a parameter type of an interface
<interface, as-parameter-type, method>	an interface serves as a parameter type of a method
<method, use, field>	a method uses a field
<field, used-by, method>	a field is used by a method
<class, declare, method>	a class declares a method
<method, declared-by, class>	a method is declared by a class
<class, declare, field>	a class declares a field
<field, declared-by, class>	a field is declared by a class
<class, extend, class>	a class extends another class
<class, extended-by, class>	a class is extended by another class
<class, implement, interface>	a class implements an interface
<interface, implemented-by, class>	an interface is implemented by a class
<method, declared-by, class, declare, method>	two methods are co-declared by a class
<method, declared-by, class, use, field> (i.e., <method, co-locate, field>)	a method and a field are co-declared by a class

4.3.2 Code Knowledge Graph

Figure 4.7 shows the schema we use to extract the knowledge graph from a code project. Each node represents a type of program entity, and each edge represents a relation between two program entity types. Each relation implies a tuple $\langle subject, relation, object \rangle$. The overall meta-path schemas defined on the code knowledge graph are shown in Table 4.1. For example, $\langle method_1, call, method_2 \rangle$ indicates a call relation between two methods in the code project. Moreover, for each relation type (e.g., call), we define its inverse-relation

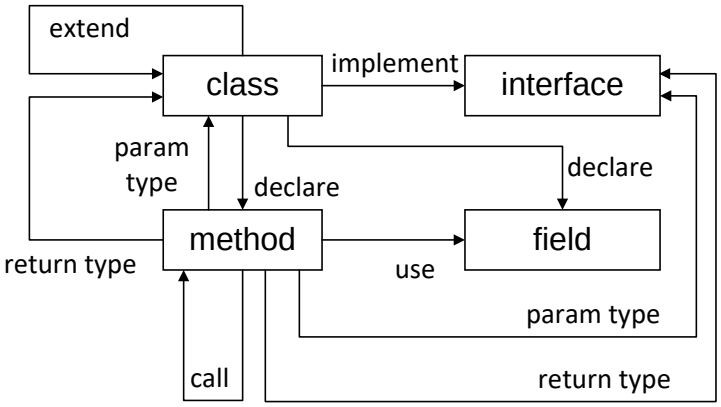

Figure 4.7: Schema for contextual knowledge graph.

type (e.g., called-by) on the graph. By this means, we can transform a whole project into a *code knowledge graph*. Figure 4.8 shows an example of the partial knowledge graph. The green nodes represent classes, the blue nodes represent methods, the yellow nodes represent fields, and the edges with different colors represent different program relations. The graph is centered at the target method and expands along the code relations defined in Figure 4.7, such as *declare*, *call*, etc.

Formally, we have $G_{kg} = \langle N_1, N_2, ..., N_k, E_1, E_2, ..., E_l \rangle$ where each N_i represents the set of nodes following a node type (e.g., method type) and each $E_m = N_i \times N_j$ (i can be equal to j) represents the set of edges following a relation type between two sets of nodes (e.g., call relation type). Given a target method m (or a node of method type on the code knowledge graph), we define its context as a set of reachable nodes in G_{kg} following a predefined meta-path schema.

Contextual Meta-path Schema. Specifically, a contextual meta-path schema is a set of sequences of node/edge types $\mathcal{S} = \{\mathcal{MP} | \mathcal{MP} = \langle \mathcal{N}_{t1}, \mathcal{E}_{t2}, ..., \mathcal{N}_{tk} \rangle\}$. Note that each $\mathcal{N}_{ti}$ or $\mathcal{E}_{ti}$ is a node type or edge type defined in Figure 4.7. We say a path on G_{kg}, $p = \langle n_1, e_2, n_3, ..., n_k \rangle$, conforms to a meta-path schema $\mathcal{MP} = \langle \mathcal{N}_{t1}, \mathcal{E}_{t2}, \mathcal{N}_{t3}, ..., \mathcal{N}_{tk} \rangle$ if $\forall i (i = 1, ..., k)$, n_i is an instance of type N_{ti} or e_i is an instance of type E_{ti}, denoted as $p \sim \mathcal{MP}$. For example in Figure 4.8, the path p = ⟨ `getMatchingCondition()`, *call*, `isPreFlightRequest()` ⟩ is an instance of the meta-path schema $\mathcal{MP}$ = ⟨ method, call, method ⟩.

Furthermore, we denote $p[0]$ as the first element in p and $p[-1]$ as the last element in p. We define a path $p = \langle n_1, e_2, n_3, ..., n_k \rangle$ as a *contextual path* of a node n if $p[0] = n$

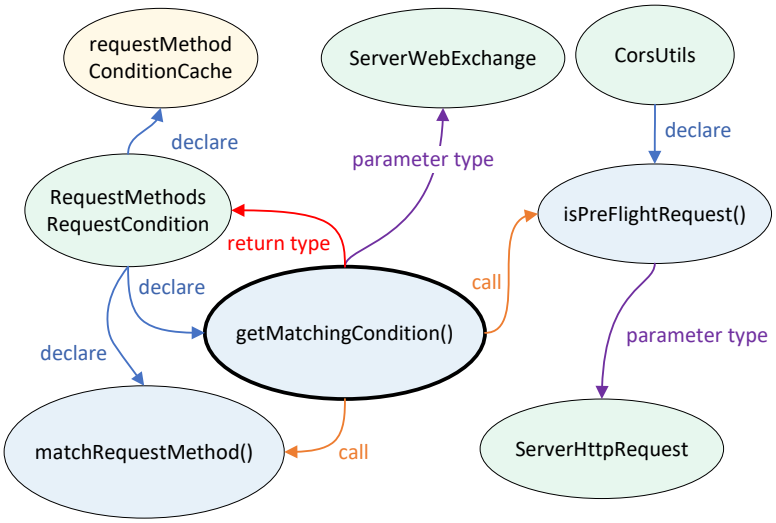

Figure 4.8: The code knowledge graph example for Figure 4.4.

and $\exists \mathcal{MP} \in \mathcal{S}$ so that $p \sim \mathcal{MP}$. For example in Figure 4.8, taking the method `getMatchingCondition()` as the target node, the path p = ⟨ `getMatchingCondition()`, *call*, `isPreFlightRequest()` ⟩ is its contextual path because (1) p starts at `getMatchingCondition()` and (2) $p \sim \mathcal{MP}$ = ⟨ method, call, method ⟩.

Thus, we define the *context* of a node n as $\mathcal{C}(n) = \{n_c | \exists \mathcal{MP} \in \mathcal{S}$ and p, $p \sim \mathcal{MP}, p[0] = n$, and $p[-1] = n_c\}$. For convenience, we use $\mathcal{C}_{\mathcal{MP}}(n) = \{n_c | p \sim \mathcal{MP}, p[0] = n$, and $p[-1] = n_c\}$ to denote the context of n under the meta-path schema $\mathcal{MP}$.

4.3.3 Context Sampling

There can be many contextual nodes of an anchor node, which makes it challenging to feed all the contextual nodes into a training batch. Thus, we sample contextual nodes regarding the context and embedding diversity during the training. Specifically, given the set of contextual node $\mathcal{C} = \{c_1, c_2, ..., c_n\}$, for each $c_i \in \mathcal{C}$, we estimate its context and embedding diversity.

We estimate its context diversity as Equation 4.1, where $Type_c(\mathcal{C})$ represents the set of contextual types in $\mathcal{C}$, and the function $type(c_i)$ returns the set of contextual nodes sharing the same type with c_i. As a result, $div_c(c_i)$ represents the probability of sample c_i regarding the contextual diversity.

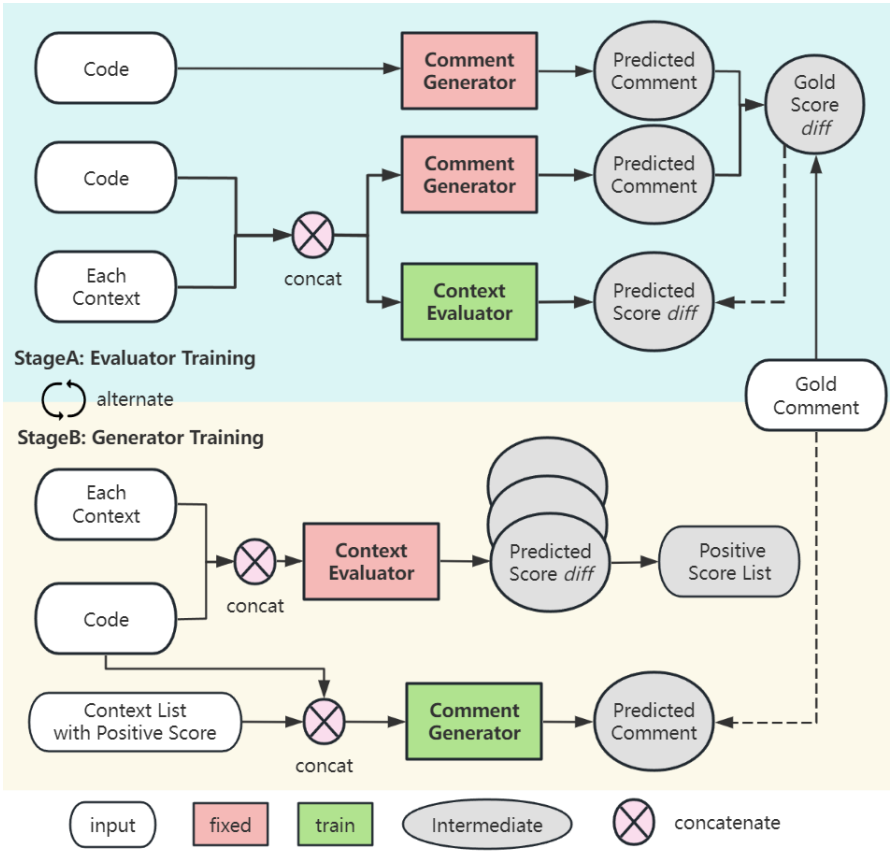

Figure 4.9: Model training architecture of CProSum.

$$div_c(c_i) = \frac{1}{|Type_c(\mathcal{C})|} \cdot \frac{1}{|type(c_i)|} \tag{4.1}$$

We estimate its embedding diversity as Equation 4.2. Given the embedding of the target method as e_m, and that of the contextual node c as e_c, $Range(\mathcal{C}, k)$ is the set of k ranges evenly distributed between the L2 distance $||e_c, e_m||$. In addition, the function $range(c_i)$ returns the contextual nodes sharing the same range with c_i. As a result, $div_e(c_i)$ represents the probability of sample c_i regarding the embedding diversity.

$$div_e(c_i) = \frac{1}{|Range(\mathcal{C}, k)|} \cdot \frac{1}{|range(c_i)|} \tag{4.2}$$

Finally, we let the probability to sample c_i by $p(c_i) = \frac{1}{2}(div_c(c_i + div_e(c_i))$. Note that, it is guaranteed that $\sum_{i=1}^{n} p(c_i) = 1$.

4.3.4 Context Evaluation

Figure 4.9 shows our model training architecture, which includes one context evaluator and one context-aware comment generator. StageA fixes the generator (in red) and trains the evaluator (in green). For each context, the generator predicts the comment with or without the context and calculates the score difference between the two predicted comments. The context evaluator will score the context and use the gold score difference to compute the loss. StageB fixes the evaluator and trains the generator. The evaluator first scores each context and gives a ranked positive score list to select the useful context. The generator will input the target code component and useful context list as a prompt and predict the comment. The solid line passes the messages, and the dotted line computes the loss. Stages A and B are alternated until the two models converge.

The context evaluator g is designed as a sentence transformer model to estimate the potential of the performance boosting for a context to a target code. The input is a $code_{tar}$ and a prompt sequence $prompt$ including a list of $type_{con}, code_{con}, graph_{con})$ where $code_{tar}$ is the target code, $type_{con}$ is the context type, $code_{con}$ is the context code, and $graph_{con}$ is the graph structure including project, package, class, caller, callee and field information. We traverse the graph in fixed order into a sequence and train the evaluator to learn the graph information using the target method. Compared to siamese model architecture [33], this design is equipped with abundant attention neurons, which allows models to capture more enriched and detailed relations between the context and the target code. The output is a value normalized between 0 and 1, which indicates the similarity score when concatenating with the selected context. The gold score depends on the measurement method used in the code comment. In this work, we follow the previous work [145][68] using the smoothing BLEU4 as the golden label for the evaluator. With a trainable context evaluator g, a fixed comment generator $\mathcal{F}$ and a score calculator S, we design the loss function as:

$$\begin{aligned} Loss_{eval} = |g(code_{tar}, prompt) - \\ (S(\mathcal{F}(code_{tar}, prompt)) - S(\mathcal{F}(code_{tar})))| \end{aligned} \tag{4.3}$$

Equation 4.3 evaluates the difference between (1) the estimated boost potential and (2) the actual score difference between the new predicted comment with prompt and the original predicted comment without prompt by the comment generator.

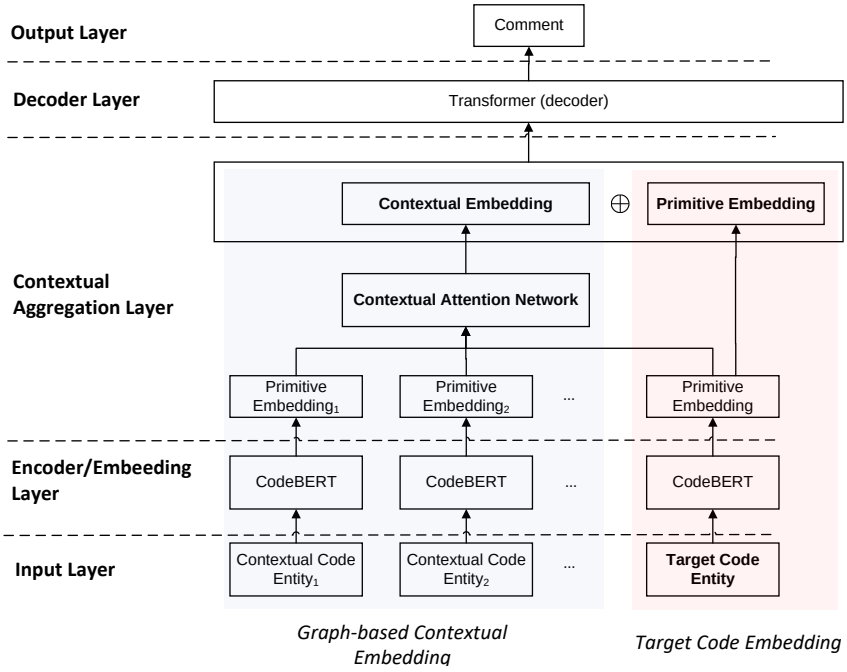

Figure 4.10: A comment generator architecture.

4.3.5 Context Fusion

Figure 4.10 shows the architecture of the generator to fuse the target method and its context, consisting of the input layer, embedding layer, encoder layer, contextual aggregation layer, decoder layer, and output layer. The embedding layer has the alternative solution as CodeBERT [57] or GraphCodeBERT [74]. The decoder layer has alternative solutions such as GRU [36], LSTM [84], or Transformer [198]. We design the contextual aggregation layer to discriminate and fuse relevant context with the target method. This layer takes the primitive embedding e_p from the encoder layer and all its contextual node entities in the project as input and generates a contextual embedding e_c as output. The *Contextual Attention Network* is designed to guide the embedding layer (e.g., the CodeBERT [57] or GraphCodeBERT [74] layer) to generate embedding of both the contextual code entities and target method, so that the comment-contributable contextual entities have close embedding with that of the target code. Then, we combine the generated primitive and contextual embedding into $e = e_p \oplus e_c$ (or, $e = p(e_p, e_c)$) and feed e to the decoder to derive the code comments. Practitioners can choose any encoder and decoder solutions in practice.

Graph-based Aggregation Given a target method m, based on the context definition (see Section 4.3.2), we can have the set of context under different meta-path schemas, i.e., $\mathcal{C}$ = $\{\mathcal{C}_{\mathcal{MP}_1}(m), \mathcal{C}_{\mathcal{MP}_2}(m), ..., \mathcal{C}_{\mathcal{MP}_k}(m)\}$ (where k is the number of meta-path schemas). In Figure 4.8, the method `getMatchingCondition()` has the following contexts:

- **call**: { ⟨`getMatchingCondition()`, call, `isPreFlightRequest()`⟩, ⟨`getMatchingCondition()`, call, `matchRequestMethod()`⟩ }
- **co-location**: { ⟨`getMatchingCondition()`, co-locate, `matchRequestMethod()`⟩ }
- **return-type**: { ⟨`getMatchingCondition()`, has-return-type, `RequestMethodsRequestCondition`⟩ }
- **param-type**: { ⟨`getMatchingCondition()`, has-parameter-type, `ServerWebExchange`⟩ }

$$\overline{e_m} = \sum_{m_i \in \mathcal{C}_{\mathcal{MP}}(m)} \frac{g_{\mathcal{MP}}(e_m, e_{m_i})}{|\mathcal{C}_{\mathcal{MP}}(m)|} \cdot e_{m_i} \tag{4.4}$$

For each meta-path schema $\mathcal{MP}$, we follow the intuition of PageRank algorithm [158] to calculate its embedding aggregation as Equation 4.4. $\mathcal{C}_{\mathcal{MP}}(m)$ is the set of neighbours under the context meta-path schema $\mathcal{MP}$. For every neighbor e_{m_i} under $\mathcal{MP}$, we learn its attention with the embedding of the target method m, i.e., $g_{\mathcal{MP}}(e_m, e_{m_i})$. For example, let $\mathcal{MP}$ be the call context of the target method m = `getMatchingCondition()`, $\mathcal{C}_{\mathcal{MP}}(m)$ = {`isPreFlightRequest()`, `matchRequestMethod()`}. Therefore, $|\mathcal{C}_{\mathcal{MP}}(m)| = 2$. For each neighbor in the call context, e.g., `isPreFlightRequest()`, we denote its primitive embedding as e_{m_1} and that of the target method as e_m. An attention function $g_{\mathcal{MP}}(e_m, e_{m_1})$ is designed to represent how important `isPreFlightRequest()` can help to generate the comment of the target method. Then, we multiply e_{m_1} the embedding of `isPreFlightRequest()` with $g_{\mathcal{MP}}(e_m, e_{m_1})$ as the contextual contribution of `isPreFlightRequest()`. By this means, we can learn a discriminative model to distinguish more essential neighbors contributing to the target method.

$$\overline{e_m} = \sum_{\mathcal{MP} \in \mathcal{C}} \frac{h_{\mathcal{MP}}(e_m)}{|\mathcal{C}|} \cdot \sum_{m_i \in \mathcal{C}_{\mathcal{MP}}(m)} \frac{g_{\mathcal{MP}}(e_m, e_{m_i})}{|\mathcal{C}_{\mathcal{MP}}(m)|} \cdot e_{m_i} \tag{4.5}$$

Similarly, given many types of meta-path schema, we design Equation 4.5 to calculate the aggregation. In addition to the attention between a neighbor and the target method m, we also introduce an attention function to learn the importance of a context type (i.e., meta-path schema) to m, i.e., $h_{\mathcal{MP}}(e_m)$. For example, the target method has context type set $\mathcal{C}$ = {call, co-location, return-type, param-type}. For a context type, e.g., call, we learn an attention function $h_{\mathcal{MP}}(e_m)$ to evaluate how important a context type is to the target method. By this means, we can also discriminate context types for every target method.

$$g_{\mathcal{MP}}(e_m, e_i) = S(e_i^T \times W_{\mathcal{MP}} \times e_m) \tag{4.6}$$

$$h_{\mathcal{MP}}(e_m) = S(e_m^T \times W'_{\mathcal{MP}}) \tag{4.7}$$

We use Equation 4.6 and Equation 4.7 for the two learnable attention functions $g_{\mathcal{MP}}(.)$ and $h_{\mathcal{MP}}(.)$, respectively. In Equation 4.6, we use $W_{\mathcal{MP}} \in R^{d \times d}$ to learn the importance of the contextual neighbor to the target method, and $S(.)$ represents the sigmoid function. In Equation 4.7, we use $W'_{\mathcal{MP}} \in R^{d \times 1}$ to learn the importance of a context type to the target method. Note that, $W_{\mathcal{MP}}$ and $W'_{\mathcal{MP}}$ differ regarding different $\mathcal{MP}$.

Overall, given the original embedding e_m and learned contextual embedding $\overline{e_m}$, we fuse the final embedding as Equation 4.8 where α and β are learnable parameters.

$$e'_m = \alpha \cdot e_m + \beta \cdot \overline{e_m} \tag{4.8}$$

Here, e'_m is the representation encoding both the target method and its context information, which is further fed to the decoder to generate the comment. Note that, despite that, we consider only one-layer contextual neighbor in Equation 4.5; the primitive embedding will be updated during the training, which allows the contextual information to propagate for multiple hops.

4.4 Experiment

We have the following research questions for CProSum:

- **RQ1 (Overall Performance):** Whether CProSum has a better performance to generate comments compared to the state-of-the-art comment generators?

Table 4.2: Overview of the chosen baselines

Approach	Input Representation	Base Model
RoBERTa [127]	Code token	BERT[48]
CodeBERT [57]	Code token	RoBERTa
GraphCodeBERT [74]	Code token + data flow	RoBERTa
Code T5 [204]	Code token	T5[171]
UniXCoder [73]	Code token	XLM[105]
REDCODER [161]	Code token + retrieval	PLBART[2]
RAG [68] [145]	Code token + retrieval	GPT3.5[152]
GPT-4 Turbo [154]	Code token	GPT4

- **RQ2 (Evaluator Performance):** Whether the evaluator can score and select useful code examples?
- **RQ3 (Generator Performance):** Whether comment generators can effectively utilize the graph context of the target method and the retrieved code examples?
- **RQ4 (Ablation Study):** How does the performance of CProSum vary with project information loss?

4.4.1 Experiment Setup

4.4.1.1 Baselines

In this study, we chose several baselines, including two retrieval-augmented methods, two large language models (GPT), and five different architectures of language models based on transformers. Both [68] and [145] use the CodeX[151] for the generator and incorporate the retrieved examples with the code as prompts to the large language model. However, as CodeX API is deprecated, we replace it with GPT3.5 Turbo[152]. Due to the high cost, we randomly sampled 3K examples in our test dataset to test the GPT3.5 and GPT4 models. Except for two GPT models, all other baselines have been fine-tuned on our training dataset.

Those encoder-decoder solutions are selected because (1) they are shown to outperform similar solutions and (2) they are popular and impactful (e.g., we choose CodeBERT [57] and GraphCodeBERT [74] for their high impact), (3) they are diverse regarding input structure and base model architecture. Table 4.2 shows more details of our selected baselines. We use the default hyper-parameters suggested in the literature.

4.4.1.2 Measurement

For context evaluator, we follow the existing literature [88, 57, 74] and use Mean Average Precision (mAP), Mean Reciprocal Rank (MRR), Normalized Discounted Cumulative Gain (NDCG) and R-precision (RPrec) to evaluate the performance of evaluator. For comment generator, we follow the existing literature [7, 109, 3, 206, 89, 6, 25], and use Smoothing BLEU4, METEOR, and ROUGE-L to evaluate the performance of code summarization.

4.4.1.3 RQ1 Design (Overall Performance)

To answer RQ1, we train CProSum on top of CodeT5 and compare its performance against the baselines. In this experiment, we split the training, valid, and test datasets by projects to build the project-split dataset. All the models will be fine-tuned on 60 training projects, validated on 10 projects, and tested on unseen 30 projects. For REDCODER, we retrieve similar code from its own extra database extracted from GitHub and StackOverflow. For RAG, we retrieve similar code from the training dataset. CProSum will include the graph structural information and retrieved code examples from the code knowledge graph that usually exists in real-world scenarios.

4.4.1.4 RQ2 Design (Evaluator Performance)

To test how well the evaluator can select useful context compared to traditional retrieved methods, we use the smoothing BLEU4 as the golden metric to rank the candidates and regard the ranking as golden labels for the evaluators. We then compare the baseline retriever with our evaluator on several metrics for selecting the most helpful code examples for comment generation.

4.4.1.5 RQ3 Design (Generator Performance)

To test how well our graph structure of the target method will enhance the generator, we equip all baselines with semantic-based retrieved information as prompts. CProSum will include both the graph structural information and retrieved code examples from the code knowledge graph.

Besides, to test how well the prompt mechanism can boost the generator's performance compared to the traditional fine-tuning method, we have a new dataset split setting (function-split) where we add the context information of test samples to the training

dataset. In detail, we split the dataset based on functions and randomly select 161K (∼70%) code-comment pairs as the training set, 40K (∼20%) code-comment pairs as the testing set, and the remaining for the validation. In a function-split dataset, the traditional language model can be fine-tuned by the possible context of all the samples. In this scenario, both retrieval methods and fine-tuning approaches can make use of the contextual information present in the training dataset. RAG has been excluded from this setting due to the non-public and high-cost nature of GPT-3.5.

4.4.1.6 RQ4 Design (Ablation Study)

As the effectiveness of the prompt mechanism has been widely accepted [125], we design an ablation study mainly for the possibility of information loss. We choose the project-split dataset and CodeT5 as the base comment generator. The setting will randomly mask the node in the knowledge graph to simulate the situation when the code project is incomplete. CProSum will try to include the most similar retrieved code examples from the code knowledge graph, but we will set a threshold for similarity score as we do not want to introduce much noise as context.

4.4.1.7 Training Configuration

We train the models on a GPU workstation with 4 GeForce RTX 2070 SUPER, Intel(R) Core(TM) i9-10900X CPU @ 3.70GHz, and 64G memory.

4.4.2 Experiment Results

4.4.2.1 Results (RQ1): Overall Performance

Table 4.3 shows the performance of CProSum, comparing to the 8 baselines. Overall, CProSum leads a non-trivial performance on all metrics BLEU4, METEOR, and ROUGE-L. Generally, borrowing contextual information does not necessarily improve performance. The retrieval-augmented approach REDCODER shows a great increase in performance, but it remains a question whether the external database contains some unseen project context as it crawls from GitHub and StackOverflow. In contrast, CProSum will utilize the structural context in the code knowledge graph, such as class, callees, or part of co-location methods that usually already exist when predicting the comment of the target code. The relation between the structural context and the target code is more reasonable

Table 4.3: Overall performance of different comment generators on the project-split dataset. The last three LLMs are not fine tuned on the dataset due to high cost.

Comment Generator	**BLEU4**	**METEOR**	**ROUGE-L**
RoBERTa [127]	10.47	18.82	24.72
CodeBERT-base [57]	10.74	19.12	25.46
GraphCodeBERT [74]	10.67	18.38	24.72
UniCoder [73]	10.46	18.14	23.66
CodeT5-base [204]	12.92	21.51	26.13
REDCODER [161]	18.54	27.25	29.54
CProSum (Ours)	**36.78**	**47.51**	**52.67**
RAG-GPT3.5 [145]	9.31	28.62	28.27
GPT-4 turbo	9.57	25.48	14.41
GPT-4 + Our evaluator	18.84	31.70	16.61

and explainable than the relation between the retrieved similar code and the target code. Finally, RAG-GPT3.5 using in [145] and [68] utilizing LLMs shows more boosting on METEOR but less on BLEU4 score. Based on our observations, this phenomenon can be attributed to LLMs' tendency to generate detailed comments, which may lower precision but enhance recall. However, our evaluator can select useful code examples to enhance the performance of GPT4.

4.4.2.2 Results (RQ2): Evaluator Performance

Table 4.4: Overall performance of different selectors and evaluators on the dataset.

Setting	MAP	RPrec	MRR	NDCG	Time(s)
RAG @1	36.51	28.65	36.51	40.22	0.013
RAG @3	28.91	27.79	49.36	36.95	
RAG @5	24.08	25.92	54.01	33.67	
Graph @1	38.80	29.54	38.80	43.20	0.014
Graph @3	32.17	30.57	53.26	41.19	
Graph @5	27.61	29.48	58.31	38.17	
CProSum @1	**40.27**	**31.50**	40.27	**44.33**	0.019
CProSum @3	32.42	30.95	53.99	41.01	
CProSum @5	27.11	28.82	**58.72**	37.33	

Table 4.4 shows the performance of our evaluator and baseline's selectors on the dataset. Our *Graph* setting only contains the project, package, and class information of the target method, and CProSum setting combines all the graph structures, including the caller, callee, and field. We test them on three settings with only one golden example, the top 3 golden examples, and the top 5 golden examples based on the smoothing BLEU4 score

of the target comment and similar code comment. The time for retrieval of each sample is computed over the 220K dataset. Generally, our evaluator outperforms the baseline and shows that our retrieved examples will contain more informative relations with the target method. The *Graph* setting shows that only using simple project, package, and class information in the code knowledge graph will enhance the model performance. Finally, when the number of golden examples increases, our method will have better performance compared to baselines as our graph representation will be less affected by code tokens and select more structurally similar code.

4.4.2.3 Results (RQ3): Generator Performance

Table 4.5: Boosting Performance of CProSum's comment evaluator with various comment generators on project-split dataset

Model	Scale	Parameter	BLEU4			METEOR			ROUGE-L		
			Origin	CProSum	bst (%)	Origin	CProSum	bst (%)	Origin	CProSum	bst (%)
CodeBERT	small	84M	25.12	33.23	32.29	34.02	43.42	27.63	40.09	49.31	23.00
CodeT5		60M	27.16	36.33	33.76	36.98	47.52	28.50	42.87	52.05	21.41
RoBERTa	base	173M	25.34	34.10	34.57	33.94	43.90	29.35	39.91	49.37	23.70
CodeBERT		173M	25.41	33.61	32.27	34.45	44.29	28.56	41.01	49.59	20.92
GraphCode		173M	25.44	33.35	31.09	34.66	44.54	28.51	40.64	49.99	23.01
CodeT5		223M	27.92	36.78	31.73	38.36	47.51	23.85	43.77	52.67	20.33
CodeT5	large	738M	30.44	36.89	21.19	41.76	47.88	14.66	45.04	52.49	16.54

Table 4.5 shows the generalizability of the boosting performance of CProSum for the project-split samples. In general, the improvement is significant, which indicates that the structural context incorporates abundant domain knowledge and templates that might be largely missed by a comment generator only prompt with the semantic-retrieved code comment. It shows that our proposed graph structure can effectively boost state-of-the-art neural network models when the test project is not seen in the training procedure.

Table 4.6 shows the performance of CProSum with CodeT5-base as comment generator, comparing to the 8 baselines. Overall, CProSum leads a non-trivial performance on all metrics, smoothing BLEU4, METEOR, and ROUGE-L. Traditional language models such as CodeBERT achieve comparable performance to retrieval-augmented methods like REDCODER. This is because the relevant context tends to be present in the training dataset, allowing language models to utilize the memory in the fine-tuning stage on these test examples. Further experiments on Table 4.7 where different comment generators are trained with our graph structure and examples show that the structural information of the target method can further boost the performance of different traditional transformer-based comment generators. The result also shows CProSum model can explicitly evaluate the

Table 4.6: The performance of comment generators on function-split dataset

Comment Generator	**BLEU4**	**METEOR**	**ROUGE-L**
RoBERTa [127]	43.56	51.66	57.11
CodeBERT-base [57]	44.71	52.59	57.63
GraphCodeBERT [74]	44.74	53.07	58.16
UniCoder [73]	43.48	51.65	57.11
CodeT5-base[204]	45.17	54.70	58.63
REDCODER [161]	45.24	53.98	58.58
CProSum (Ours)	**53.56**	**62.93**	**65.90**

context type, code, comment and combine them to predict more concise and informative comments.

Table 4.7: Boosting Performance of CProSum's comment evaluator with various comment generators on function-split dataset

Model+Retrieval	**Scale**	**Parameter**	**BLEU4**			**METEOR**			**ROUGE-L**		
			Origin	**CProSum**	**bst (%)**	**Origin**	**CProSum**	**bst (%)**	**Origin**	**CProSum**	**bst (%)**
CodeBERT	small	84M	40.83	51.13	25.23	48.84	59.90	22.65	54.82	64.00	16.75
CodeT5		60M	36.35	48.54	33.54	45.96	58.17	26.57	51.74	62.11	20.04
RoBERTa	base	173M	43.56	52.21	19.86	51.66	60.49	17.09	57.11	64.58	13.08
CodeBERT		173M	44.71	52.74	17.96	52.59	61.07	16.12	57.63	65.09	12.94
GraphCode		173M	44.74	52.59	17.55	53.07	60.79	14.55	58.16	65.02	11.80
CodeT5		223M	45.53	53.56	17.64	54.19	62.93	16.13	58.48	65.90	12.69
CodeT5	large	738M	46.57	54.04	16.04	55.44	63.13	13.87	59.87	66.39	10.89

4.4.2.4 Results (RQ4): Ablation Study

Table 4.8 shows the results of our ablation study on the comment generator. Generally, we can see that with more code nodes being masked, the comment generator's performance will decrease as it may find it hard to retrieve a similar example. Besides, when the node in the code knowledge graph disappears, it will also influence the graph structure of the target method, which will further hurt the performance of the comment generator. Experiments show that 20% mask of nodes makes a small difference, while about half mask of all nodes will greatly decrease the performance of our structural retrieval method. CProSum can sometimes restrict itself from selecting any contextual entities if all the relevance scores are evaluated to be low.

Table 4.8: Ablation study on the information loss of code knowledge graph

Model	**BLEU4**	**METEOR**	**ROUGE-L**
CProSum	36.78	47.51	52.67
20% mask	35.89	47.01	51.22
40% mask	26.55	39.54	47.28
60% mask	19.45	31.20	40.25
80% mask	14.23	23.36	30.12
REDCODER	18.54	27.25	29.54
CodeT5	12.92	21.51	26.13

Chapter 5

Program Evolution

5.1 Introduction

Language Models (LM) has achieved notable success in recent years in code generation tasks. LM-based approaches, such as CodeBERT [57], GraphCodeBERT [74], CodeT5 [204], Copilot [70], and ChatGPT [154], have become predominant in the realm of code generation, adeptly translating user descriptions and contextual code into executable code snippets. Despite the prowess in generating new code, empirical studies suggest that editing existing code is a more frequent activity among developers [107, 101, 143], with edits comprising about 70% of commit activities in numerous open-source projects [147].

Several transformer-based methodologies have been developed to adapt code generation technologies to code editing tasks to address this. Notable examples include GRACE [75], CCT5 [118], CoditT5 [226], and MODIT [29]. These approaches vary in how they represent edits within deep learning models but commonly treat the code editing process as a translation task. This involves (1) accepting inputs of known relevant prior edits (along with their contexts) and specific code regions (with their contexts) where changes are anticipated and (2) generating the edited code as output. We show a model architecture presented in Figure 5.1 to showcase a typical state-of-the-art solution where optional edit descriptions, prior edits with their contexts, and the targeted code regions are processed by a language model to produce the edited code. This representation encapsulates the contemporary approach to integrating deep learning techniques into practical code editing applications.

While the solutions above have established a crucial foundation for code editing tasks,

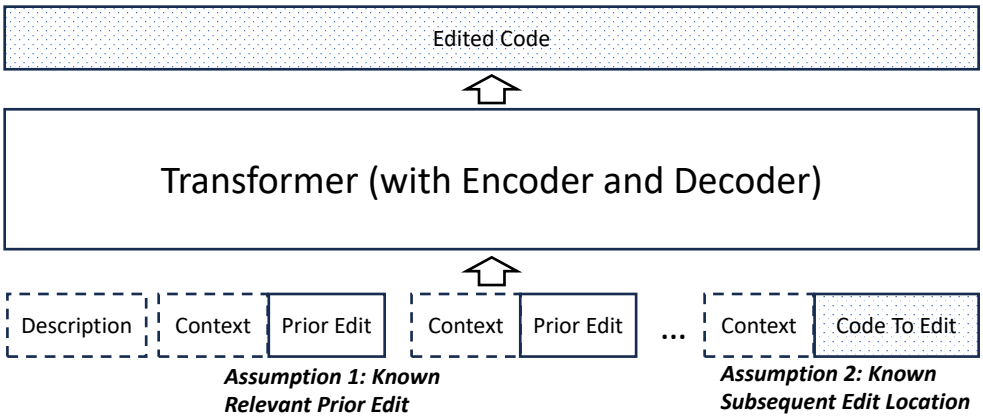

Figure 5.1: The Code Editing Framework in [29] [75] [112].

they still fail to mirror the complexities encountered in real-world scenarios. Several assumptions prevalent in current models may not hold in practice, leading to gaps in effectiveness:

- Relevance of Prior Edits. Traditional models often presume that all prior edits related to a target edit are relevant. This assumption might not always be valid in actual development environments. Feeding models with irrelevant prior edits can introduce noise, thereby degrading the accuracy of the proposed modifications.
- The Next Edit Location. Identifying potential locations for subsequent edits is challenging due to the ripple effects that a single change can trigger across a project [195]. This dynamic makes it difficult to predict where edits will be necessary without a comprehensive understanding of the entire codebase.
- Interaction and Connection between Different Edits. Code edits often interact with one another, influencing each other through syntactic dependencies and semantic relationships. Current transformer-based models, however, generally lack the sophisticated design required to capture and interpret these complex interactions effectively.

Addressing these issues requires enhancing current models to reflect the nuanced dynamics of software development better, thereby improving the practical utility of automated code editing tools. We introduce CoEdPilot, an LM-based solution crafted to tackle the outlined challenges in code editing. CoEdPilot is meticulously designed to monitor the ripple effects of edits, accurately infer the relevance of prior edits, and explicitly capture interactions between edits. Our approach integrates a suite of neural transformers [198] to function cohesively.

Upon the occurrence of an edit-triggering event (e.g., an edit e complemented by an optional edit description prp), CoEdPilot activates its components sequentially:

- *Two-stage edit location:* Initially, an *Edit-propagating File Locator* scans the entire project to identify a coarse-grained set of files, $\mathcal{F}$, where changes are likely to occur. Using the identified files $\mathcal{F}$, a *Edit-propagating Line Locator* applies a sliding window approach to determine the edit type for each line of code within these files. The result is a set of labeled lines of code, $\mathcal{L}_e = l_e = (l, t) \mid l \in \mathcal{L}, t \in \textit{insert}, \textit{replace}$, where $\mathcal{L}$ denotes the set of all code lines in the project. $\mathcal{L}_e$ encompasses all lines that need to be updated with new code lines.

- *Edit Code Generation:* For each identified edit location $\mathcal{L}_e$, the *Edit-content Generator* generates the corresponding edit content $e_t = (l, t)$ using the editing description prp and a curated set of relevant prior edits. The prior edits $\mathcal{P} = \{e = (l, t, c_a, c_b)\}$ are selected to provide context, where l denotes the line of code being modified, t specifies the edit type, c_a is the updated code content, and c_b is the original code content. Feedback on c_a and c_b is dynamically incorporated to iteratively adjust the generated edit content, ensuring alignment with the user's requirements and preferences during the editing session.

- *Edit dependency analysis:* The *Edit-dependency Analyzer* examines prior edits to identify those most relevant to the current task, ensuring both syntactic and semantic alignment. By filtering for edits that are contextually appropriate, this component improves the accuracy and relevance of the generated edits.

We trained our neural models on a dataset of over 180,000 commits crawled from 471 open-source projects. Our experiments yielded the following findings:

CoEdPilot identifies edit locations with an accuracy ranging from 70.8% to 85.3%. For each identified edit location, CoEdPilot achieves a top-1 exact match rate of 41.8% and a BLEU score of 60.7. Furthermore, an ablation study demonstrated that CoEdPilot improves the performance of baselines by an average of 8.57% in exact match rate and 18.08 in BLEU score. In a user study involving 18 participants tasked with three types of editing activities—feature enhancement, refactoring, and bug fixing—the results showed:

Table 5.1: The illustration of code edits in the file src/testing/benchmark.go

Hunk	Before Edit	After Edit

H1 (insert)	`type benchContext struct {`
	
	`  maxLen int // The`
	`  largest recorded`
	`  benchmark name.`
	`}`
		`type benchContext struct {`
		`  match *matcher`
		
		`  maxLen int // The`
		`  largest recorded`
		`  benchmark name.`
		`}`

H2 (insert)	`func runBenchmarksInternal`
	`  (...) bool {`
	`  // ... other code ...`
	`  ctx := &benchContext{`
	
	`  extLen: len(`
	`  benchmarkName("",`
	`  maxprocs)),`
	`  }`
	`  // ... other code ...`
	`}`
		`func runBenchmarksInternal`
		`  (...) bool {`
		`  // ... other code ...`
		`  ctx := &benchContext{`
		`  match: newMatcher(`
		`  matchString, *`
		`  matchBenchmarks, "-test`
		`  .bench"),`
		`  extLen: len(`
		`  benchmarkName("",`
		`  maxprocs)),`
		`  }`
		`  // ... other code ...`
		`}`

H3 (replace)	`func (b *B) runBench(...)`
	`  bool {`
	`  // ... other code ...`
	`  if b.level > 0 {`
	`  name = b.name + "/" +`
	`  name`
	
	`  }`
	`  // ... other code ...`
	`}`
		`func (b *B) runBench(...)`
		`  bool {`
		`  // ... other code ...`
		`  benchName, ok := b.name,`
		`  true`
		`  if b.context != nil {`
		`  benchName, ok = b.`
		`  context.match.fullName`
		`  (&b.common, name)`
		`  }`
		`  if !ok {`
		`  return true`
		`  }`
		`  // ... other code ...`
		`}`

Compared to the baseline Copilot, CoEdPilot effectively supports users by leveraging project-wide awareness and capturing the interaction dynamics between relevant edits.

Our key contributions are summarized as follows:

- We present CoEdPilot, a framework designed to improve edit generation models by anticipating related prior edits, pinpointing future edit locations, and modeling the interactive relationships between edits.

Table 5.2: The illustration of code edits in the file src/testing/testing.go

Hunk	Before Edit	After Edit

H4 (insert)	`type testContext struct {`
	
	`  mu sync.Mutex`
	`  // ... other code ...`
	`}`
		`type testContext struct {`
		`  match *matcher`
		
		`  mu sync.Mutex`
		`  // ... other code ...`
		`}`

H5 (replace)	`func (t *T) run(...) bool {`
	`  testName := name`
	`  if t.level > 0 {`
	`  testName = t.name +`
	`  "/" + name`
	
	`  }`
	`  // ... other code ...`
	`}`
		`func (t *T) run(...) bool {`
		`  testName, ok := t.`
		`  context.match.fullName`
		`  (&t.common, name)`
		`  if !ok {`
		`  return true`
		`  }`
		`  // ... other code ...`
		`}`

H6 (replace)	`func newTestContext(`
	`  maxParallel int) *`
	`  testContext {`
	`  return &testContext{`
	
	`  startParallel: make(`
	`  chan bool),`
	`  maxParallel:`
	`  maxParallel,`
	`  running: 1, //`
	`  Set the count to 1 for`
	`  the main (sequential)`
	`  test.`
	`  }`
	`}`
		`func newTestContext(`
		`  maxParallel int, m *`
		`  matcher) *testContext {`
		`  return &testContext{`
		`  match: m,`
		`  startParallel: make(`
		`  chan bool),`
		`  maxParallel:`
		`  maxParallel,`
		`  running: 1, //`
		`  Set the count to 1 for`
		`  the main (sequential)`
		`  test.`
		`  }`
		`}`

- CoEdPilot is designed as a modular framework, enabling seamless integration with any edit-content generator within the community.
- We developed CoEdPilot as a VS Code plugin leveraging cloud infrastructure, providing programmers with a practical tool for experimentation.
- We conducted comprehensive evaluations, including simulations, model-wise analysis, and a user study, demonstrating the effectiveness of individual model components, their integration as a system, and the UI design in a real-world tool.

5.2 Overview

Table 5.1 and Table 5.2 showcase a simplified code-editing example from commit `00a2`. Below, we provide a comprehensive summary of the programmer's editing intentions for this commit:

Intuition Design. The modified function is responsible for selecting test cases and benchmarks in the `golang/go` project. This project features the `testing` package, which orchestrates test case execution and evaluates benchmarks to measure runtime performance, memory allocation, and locking efficiency in Go programs.

The file `src/testing/testing.go` automates the selection of specific test cases, while `src/testing/benchmark.go` handles benchmarks. In the previous implementation, the selection process relied on keyword-based matching. Specifically, test and benchmark names were matched against a string (as illustrated in the *Before Edit* sections of H3 in Table 5.1 and H5 in Table 5.2).

Editing Intent. The programmer aimed to improve flexibility and precision in test and benchmark selection by implementing a regular-expression-based matching system. This new approach replaces the earlier keyword-based system, enabling more accurate identification of relevant test cases and benchmarks.

Editing Implementation. To achieve this goal, the following changes were made to `benchmark.go` and `testing.go`:

- **H1** (Table 5.1): Introduced a pointer variable `matcher` in the type `benchContext`.
- **H2** (Table 5.1): Added a `matcher` parameter during the initialization of a `benchContext` instance.
- **H3** (Table 5.1): Replaced the keyword-based matching logic with regular-expression-based matching.
- **H4** (Table 5.2): Introduced a pointer variable `matcher` in the type `testContext`.
- **H5** (Table 5.2): Substituted the keyword-based matching logic with a regular-expression-based approach.
- **H6** (Table 5.2): Added a `matcher` parameter during the initialization of a `testContext` instance.

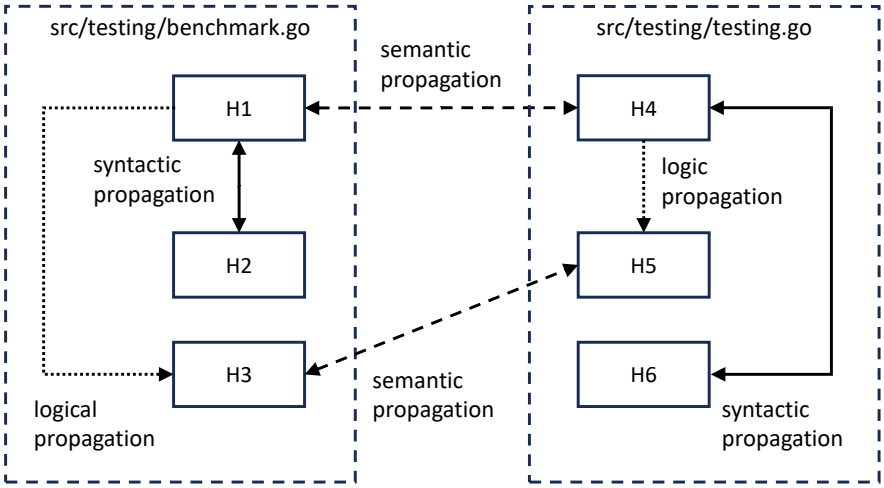

Figure 5.2: Illustration of Edit Propagation for the examples in Table 5.1 and Table 5.2.

Although these edits are relatively simple, they are interconnected and demonstrate different types of edit propagation, as shown in Figure 5.2. Following the notation in Table 5.1 and Table 5.2, we denote hunks as H_i ($i = 1, \ldots, 6$).

Types of Edit Propagation. Below are the observed categories of edit propagation in this example:

- **Syntactic Propagation:** This occurs when an edit e_i introduces a syntax-related dependency, requiring a follow-up edit e_j to resolve a compilation error. For instance, in Figure 5.2, an edit in H1 leads to a compilation error in H2 due to an uninitialized parameter. This highlights the mutual dependency between syntax-related changes.

- **Semantic Propagation:** Semantic propagation reflects edits e_i and e_j being applied to related functionalities. This relationship propagates changes across similar sections of the codebase. For example, in Figure 5.2, edits like (H1, H4) and (H3, H5) exhibit semantic propagation.

- **Logical Propagation:** Logical propagation indicates that an edit e_i serves as a prerequisite for another edit e_j. In Figure 5.2, H1 introduces the `matcher` variable, enabling the matching logic to be enhanced in H3, even though H1 does not directly cause an issue at H3.

From these observations, it is evident that:

- Edits interact in diverse ways, each influencing the project differently.

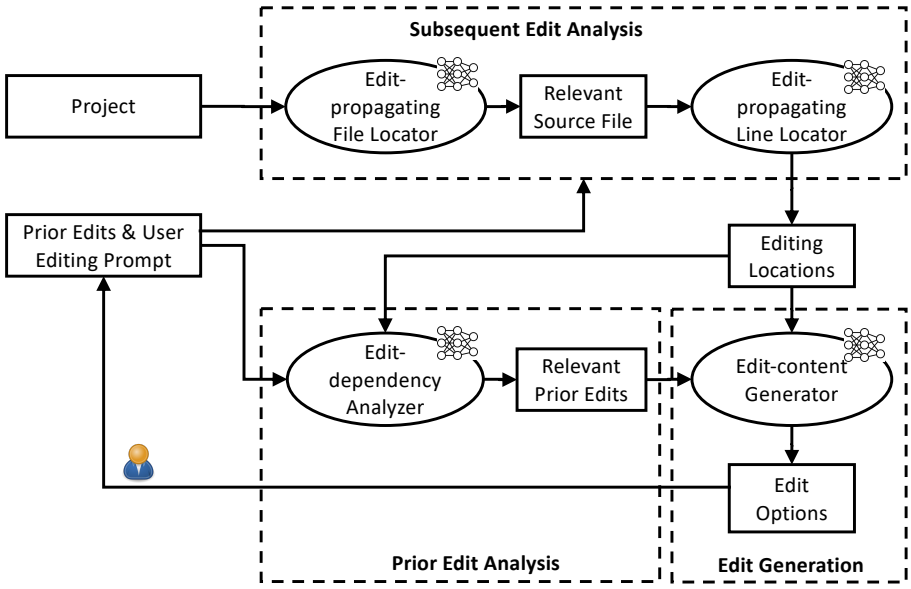

Figure 5.3: CoEdPilot includes prior edit retrieval, subsequent edit analysis, and edit generation.

- Only a select number of prior edits are relevant and sufficiently informative to contribute effectively to subsequent edits.
- Edits have the potential to propagate across any files within the project, underscoring the broad scope of their impact.

While state-of-the-art solutions like MODIT [29], and CoditT5 [226] have made significant advancements and established a solid foundation (see their model architectures summarized in Figure 5.1), they still fall short of effectively addressing the edit recommendation challenges encountered in real-world practice.

5.3 Approach

Figure 5.3 shows the framework of CoEdPilot. The CoEdPilot architecture is designed to facilitate code editing by analyzing and generating code edits based on a set of prior modifications and an optional edit prompt. The output is generated as a list of potential editing locations along with their corresponding edit options. The architecture comprises three main components:

- **Subsequent Edit Analysis** This component processes a set of selected prior code edits along with an optional editing prompt to estimate future edits within the project. The

analysis is conducted in two stages. The first stage utilizes *Edit-propagating File Locator* to identify the relevant source files where edits may occur, providing a coarse-grained overview. The second stage employs a fine-grained detector, *Edit-propagating Line Locator*, to predict the type of edits (e.g., insertions or replacements) needed for each line in these files.

- **Prior Edit Analysis** This module analyzes the identified editing locations to select the most relevant previous edits using *Edit-dependency Analyzer*. It evaluates these edits based on their ability to influence future edits in terms of syntax, semantics, and logic.

- **Edit Generation** This component generates concrete editing options for each identified location, specifically for edits classified as *insert* or *replace*. The process includes user interaction where: (1) The user can accept a recommended edit as is. (2) The user may modify a suggested edit based on the recommendation. (3) The user inputs their own edit if the recommendations do not suffice. Once an edit is confirmed, it is recorded as a new prior edit, which triggers a new cycle of edit generation and recommendations.

This structured approach allows for dynamic and iterative enhancements to code, leveraging both past edits and real-time user input to refine and optimize code continuously.

5.3.1 Subsequent Edit Analysis

Problem Statement. We tackle the challenge of identifying subsequent edits in a software project by framing it as a task of edit propagation triggered by an initial edit and an optional user prompt.

The problem is defined as follows: Given a project P, represented as a set of files, a user's editing prompt prp, and the latest edit $e = (c_b, c_a)$, where c_b denotes the code before the edit and c_a represents the code after the edit, our goal is to identify a subset of files $F \subset P$. For each file $f \in F$, subsequent edits are specified by annotating each line of code with an editing type: *keep*, *insert*, or *replace*.

Challenge. As discussed earlier, the interactions between edits involve complex syntactic dependencies and semantic relevancies. To analyze these syntactic dependencies, it is standard practice to parse the entire compilable project. This process involves constructing a program dependency graph [59], which helps in tracking data, control, and call dependen-

cies. However, constructing such graphs for large projects can be notably time-consuming and computationally intensive.

Moreover, the tools used for constructing syntactic graphs [197, 5] and analyzing semantic relevance [100, 53, 4] typically depend on the programming language, which limits their versatility. To overcome these challenges, we employ neural models that estimate both syntactic dependencies and semantic relevances. These models offer advantages in terms of runtime efficiency and language independence, providing a more scalable and flexible approach to understanding the interactions between code edits.

In this study, we implement a two-tiered approach to localization, designed to pinpoint the areas requiring edits within a codebase efficiently. The first stage involves file localization, which is performed in a coarse-grained manner. Following this, we refine our focus in the second stage with line of code localization. This fine-grained analysis precisely determines the specific lines within the identified files that need editing. This methodical approach ensures a systematic and accurate identification of edit locations, enhancing the effectiveness of subsequent code modifications.

5.3.1.1 Propagation File Localization

To achieve this, we select a subset $F' \subset F$, where $F' = \{f \mid sub_{edt}(f, e) > th_{sub}, f \in F\}$. Here, $sub_{edt}(\cdot, \cdot)$ denotes a likelihood estimation function for determining the probability that a file f will be co-edited given the input edit e. The threshold th_{sub} is used to quantify this likelihood.

The propagation likelihood is evaluated based on two primary factors: (1) the estimated dependency of the file f on the input edit e; and (2) the semantic similarity between code fragments in f and the edit e.

Namely, we design Equation 5.1 as follows.

$$sub_{edt}(f, e) = \alpha_1 \cdot dep(e, f) + \alpha_2 \cdot sem(e, f) + \epsilon \tag{5.1}$$

In Equation 5.1, we let each coefficient $\alpha_i > 0$. We quantize each factor (estimated dependency $dep(e, f)$ and semantic similarity $sem(e, f)$) as a score between 0 and 1 as follows.

Estimated Dependency To determine whether a source file f is influenced by a given edit $e = (c_b, c_a)$, we design a dependency inference function $dep(e, f)$ that quantifies the

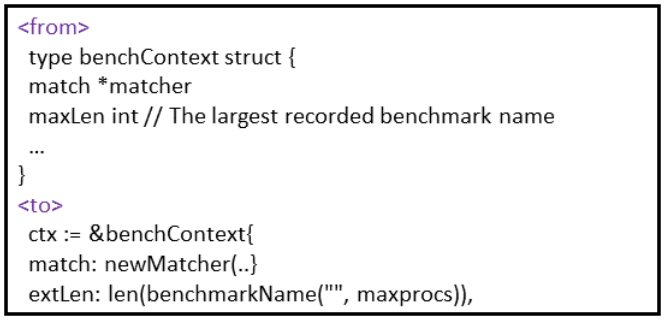

Figure 5.4: Illustration of the transformer-based model to learn the dependency of the code edits.

likelihood of such a dependency. This function leverages transformer-based models, such as CodeT5 and CodeBERT, to capture relationships within the source code. The design of our approach is guided by the framework proposed in GRACE [75].

In our approach, we utilize the tags `<from>` and `<to>` as separators between two segments of source code, serving a crucial role in instruction tuning. These tags help delineate the beginning and end of code snippets, facilitating clearer parsing and processing by our model. Following this segmentation, we introduce a dense neural layer equipped with two output neurons. These neurons are activated using a sigmoid function designed to assess code dependencies in both directions:

1. Determining whether the former code snippet depends on the latter, and

2. Assessing if the latter code snippet depends on the former.

Given a pair of source code c_1, c_2, their labeled dependencies are y_1 and y_2 ($y_1 = 1$ or 0 indicates whether c_1 depends on c_2, and $y_2 = 1$ or 0 indicates whether c_2 depends on c_1), and their estimated dependencies are $\hat{y_1}$ and $\hat{y_2}$. We design the loss function as shown in Equation 5.2:

$$\begin{aligned} loss(c_1, c_2) = -(y_1 \times log(\hat{y_1}) + (1 - y_1) \times log(1 - \hat{y_1}) + \\ y_2 \times log(\hat{y_2}) + (1 - y_2) \times log(1 - \hat{y_2})) \end{aligned} \tag{5.2}$$

We leverage the dependency analyzer developed by Jin et al. [95, 96] to construct our training dataset. Due to input length constraints, a file f is divided into k smaller segments, denoted as $seg_1, \ldots, seg_k$. The code before the most recent edit, c_b, is selected as the target code c_{tar}. We then estimate the likelihood of a dependency between c_{tar} and each segment.

The second output neuron, $\hat{y_2}$, represents the likelihood of the latter code depending on the former. We define the dependency function as:

$$dep(e, f) = \max(\hat{y_2}(c_{tar}, seg_i)).$$

This formulation uses a one-directional dependency to infer edit propagation. The $\max(\cdot)$ function is chosen to prioritize recall over precision at this stage. By substituting the dependency analyzer tool [95, 96] with a neural network, we significantly reduce the runtime required for analyzing a pair of code snippets, decreasing it from approximately 70 seconds to 0.01 seconds.

Semantic Similarity and Prompt Relevance We leverage neural embeddings to universally capture the semantic similarity between different pieces of source code. The underlying rationale is our belief that pretrained neural networks, such as CodeT5 and CodeBERT, are adept at capturing both syntactic and semantic similarities. Given these models' capabilities, we continue to factor in the constraints posed by the maximum input length permissible by transformer architectures. This consideration ensures that while we utilize these models' advanced capabilities to analyze code, we also efficiently manage the data input to fit within the operational parameters of the neural networks, thereby maximizing the effectiveness and accuracy of our semantic similarity assessments. We split a source file f into k segments as $seg_1, ..., seg_k$, $c_{tar} = c_b$ where c_b is the code before the edit, and $emd(.)$ as the representation of a piece of code or a prompt extracted from the transformer, we can have:

$$sem(e, f) = max(cos(emd(c_{tar}), emd(seg_i))) \tag{5.3}$$

5.3.1.2 Propagation Line Localization

Given the identified source files that exhibit potential for edit propagation, we implement a sliding window technique across each file to systematically analyze the editing requirements for each line of source code. This method involves sequentially moving a window across the file's text, allowing us to scrutinize every line individually. By doing so, we can precisely categorize the type of edits needed—whether a line should be kept, inserted, or replaced. As illustrated in Figure 5.5, we fine-tune a base transformer model to solve a Masked Language Modeling task [48], incorporating instruction tuning

techniques [179]. The transformer's input is composed of three main components: the target code within the window, the user-provided prompt, and the relevant prior edits (see Section 5.3.2 for details). To guide the model in understanding the structure of the input, we include specific instruction tags such as `code-window`, `prompt`, `prior-edits`, and `edit` as separators.

Additionally, each line of code is annotated with an operator that specifies its intended action:

- *keep*: This operator indicates that the line should remain unchanged, represented by `<K>`.
- *insert*: This operator signifies that new code should be inserted after the current line, represented by `<I>`.
- *replace*: This operator denotes that the line should be replaced, either with an empty line (deletion) or with modified content (update), represented by `<R>`.

In our approach, the edit operators are represented in the input using a special token, `<MASK>`. To train the model to identify and replace these masked tokens accurately, we engage in a Masked Language Modeling (MLM) task specifically focused on the edit operators. The training prompts for this task are derived from commit messages found in the code's commit history, providing contextually rich cues for learning.

5.3.2 Prior Edit Analysis

Problem Statement. Given a set of prior edits $E_p = \{e_{p_1}, \ldots, e_{p_k}\}$, where each $e_i = (c_{b_i}, c_{a_i})$, and a target code $c_{b_{tar}}$, we measure the likelihood of influence of e_{p_i} on $c_{b_{tar}}$ as a value between 0 and 1. To achieve this, we define an estimation function $rel(.,.) : E_p \times C \rightarrow (0, 1)$, where C represents the set of code fragments, and $rel(e_i, c_{b_{tar}}) \in (0, 1)$ quantifies the degree of relevance.

We quantize the relevance of prior edits by their syntactic dependency and semantic similarity by Equation 5.4:

$$rel(e_{p_i}, c_{b_{tar}}) = FCN(dep(e_{p_i}, c_{b_{tar}}), sem(e_{p_i}, c_{b_{tar}}), loc_{sim}(e_{p_i}, c_{b_{tar}})) \tag{5.4}$$

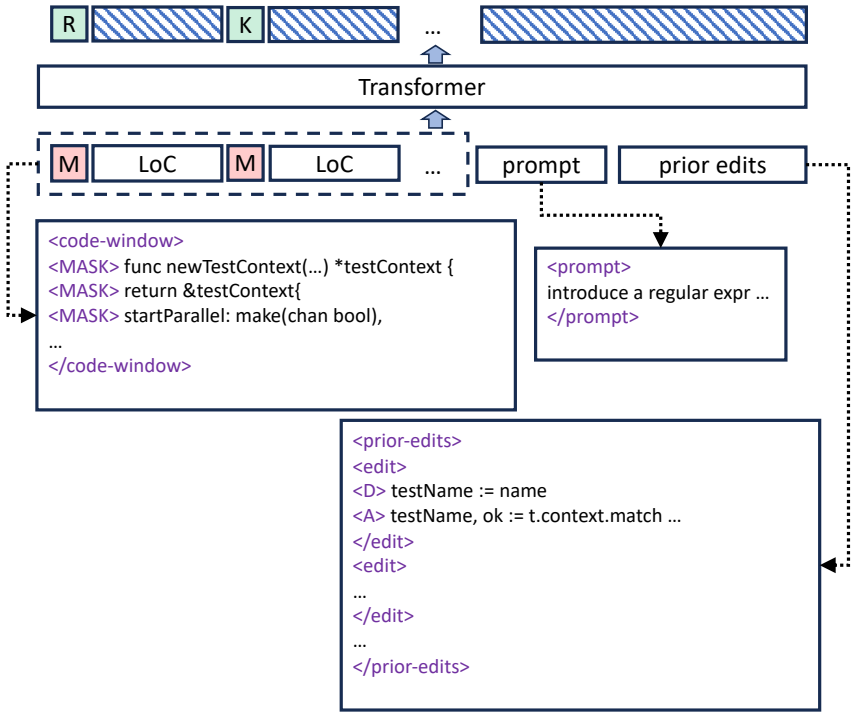

Figure 5.5: Architecture of the Edit Location Prediction.

Further, in Equation 5.4, FCN is a multi-layer fully connected network, the dependency estimation function $dep(.,.)$ for estimating the dependency from $c_{c_{tar}}$ to the code before the edit of e_{p_i} and the semantic relevance function $sem(.,.)$ is defined in Section 5.3.1.1. Function loc_{sim} evaluates the proximity between e_{p_i} and $c_{b_{tar}}$ as:

$$loc_{sim}(e_{p_i}, c_{b_{tar}}) = \begin{cases} 1 - \frac{|loc(e_{p_i}) - loc(c_{b_{tar}})|}{k} & \text{if } ld(e_{p_i}, c_{b_{tar}}) < k \\ 0 & otherwise \end{cases} \tag{5.5}$$

In Equation 5.5, we use a sliding window of size k to define whether the location difference of e_{p_i} and $c_{b_{tar}}$ is small (i.e., $ld(e_{p_i}, c_{b_{tar}}) < k$). If it is, we estimate the proximity as Equation 5.5. Otherwise, the function $loc_{sim}(.,.)$ is 0. Finally, we define a threshold th_{pri} to identify the set of relevant prior edits $E_{rel} = \{e_p | rel(e_p, c_{b_{tar}}) > th_{pri}\}$.

5.3.3 Edit Generation

Figure 5.6 depicts the model architecture for generating edit content at a *single* edit location, utilizing selected prior edits. Like the approach used for locating edit lines, the edit generation model processes three key inputs: a code window surrounding the edit, the user's prompt, and relevant prior edits. To enhance structural understanding, the prompt

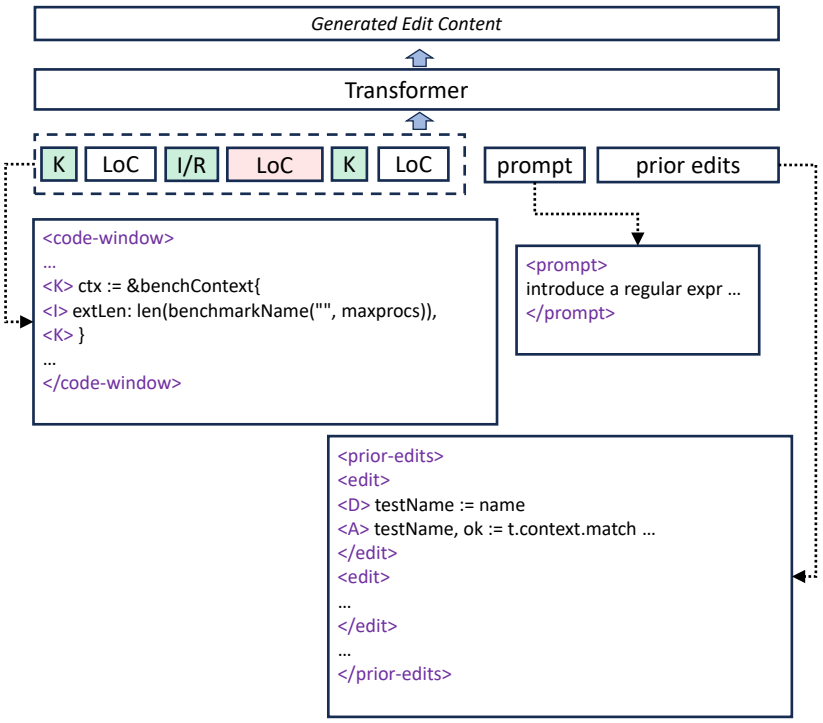

Figure 5.6: Architecture of Our Edit Generator.

and prior edits are tagged consistently to highlight key elements.

Unlike other inputs, the code window represents a *hunk*, which includes consecutive lines of the same edit type (*replace* or *insert*) along with a few surrounding lines of the *keep* type as contextual information. The types of edition are:

- Lines of type *keep* are tagged with `<K>`.
- Lines of type *insert* are tagged with `<I>`.
- Lines of type *replace* are tagged with `<R>`.

The model predicts the edit content for the specified edit location as output. Training is conducted using the classical cross-entropy loss [41].

At runtime, we employ Beam Search [64] to generate k ranked edit options based on their confidence scores. Lastly, users can either accept or modify the suggested edits. Any new edit provided by the user is stored as a prior edit, serving as feedback to enhance the subsequent editing process.

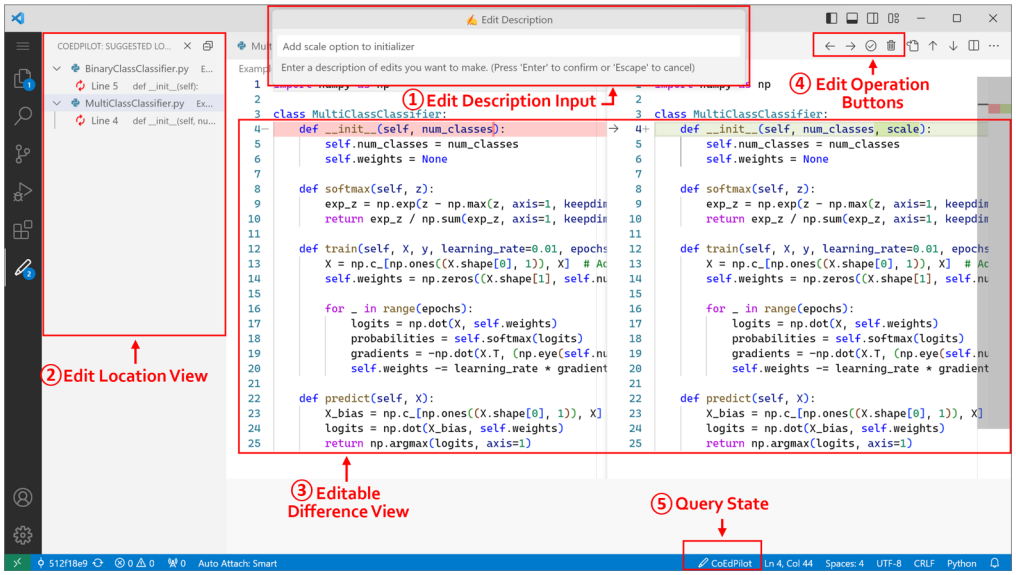

Figure 5.7: We implemented CoEdPilot as a Visual Studio Code extension.

5.3.4 Model Training

Overall, we trained three neural models, i.e., an *Edit-dependency Analyzer* (see Section 5.3.1.1), an *Edit-propagating Line Locator* (see Section 5.3.1.2), and an *Edit-content Generator* (see Section 5.3.3).

We initiate the training of our *Edit-dependency Analyzer* by leveraging the dependency analysis tool developed by Jin et al. [95, 96], applied across various open-source projects. This tool assists in gathering data on the dependencies inherent within source code, forming the foundational dataset for training. Importantly, our neural dependency analyzer is designed to predict code dependencies irrespective of programming language, enhancing its versatility and applicability. Subsequently, we conduct interactive training for the *Edit-propagating Line Locator* and *Edit-content Generator*. Our training dataset comprises commits from open-source projects. Each commit is broken down into individual edits, or hunks, which we use to train our models. This training involves estimating the edits in a randomized sequence, both within and across files. Additionally, when dealing with a set of prior edits, we transform their relevance into a probabilistic distribution X.

5.4 Tool Design

Figure 5.7 presents a screenshot of our CoEdPilot tool, integrated as an extension for Visual Studio Code, showcasing functions developed according to our proposed approach.

The key features and graphical user interface (GUI) are described as follows:

- **Triggering the Edit Recommendation**: Users can activate the edit recommendation feature within the CoEdPilot tool through a keyboard shortcut or by right-clicking in the editor. This action opens an *Edit Description Input* ①, allowing users to provide an optional description of the desired edit.
- **Subsequent Edit Recommendation**: Afterward, CoEdPilot displays an *Edit Location View* ②, where edit locations are organized hierarchically, with edit files as parent nodes and corresponding edit lines as child nodes. Users can interact directly with the visualization by selecting a child node. Lines marked for insertion are highlighted in green to indicate additions, while lines suggested for replacement are highlighted in red to indicate modifications.
- **Edit Option Recommendation**: Users can request detailed edit options for each location, displayed in the *Editable Difference View* ③ in Figure 5.7. This view simulates how the code appears before and after the edit. Using the *Edit Operation Button* ④, users can *browse*, *accept*, or *ignore* the edit options. Accepted edits, along with any subsequent modifications, are recorded as prior edits to improve future recommendations.
- **Cloud Service**: For deployment, the CoEdPilot tool adopts a cloud-based design, similar to Copilot, to manage interactions between the client and server. Users can monitor the network connection status through the *Query State* ⑤, as shown in Figure 5.7.

5.5 Experiment

5.5.1 Research Questions

We have five research questions:

- **RQ1 (Locating Propagating Files)**: Can CoEdPilot accurately identify the source files where edits propagate?
- **RQ2 (Locating Propagating Lines)**: Given the identified source files, can CoEdPilot locate the specific lines of code affected by edit propagation?

- **RQ3 (Edit Generation)**: How effectively can CoEdPilot generate edit options for a given edit location?
- **RQ4 (Prior Edit Relevance)**: Can CoEdPilot accurately select relevant prior edits to assist in the editing process?
- **RQ5 (Performance Boost for State-of-the-art Solutions)**: Does the CoEdPilot framework enhance the performance of state-of-the-art edit generation models?

5.5.2 Benchmark Construction

To assess the performance of CoEdPilot, we established a comprehensive benchmark comprising five programming languages (i.e., JavaScript, Java, Go, Python, and TypeScript) sourced from 471 open-source projects. For our dataset, we selected projects based on popularity, specifically choosing the top 100 projects on GitHub by star count. We excluded projects primarily intended for educational purposes, such as tutorials, and those with non-English commit messages for each programming language. Our selection criteria for commits within these projects are designed to optimize the relevance and manageability of the data for our model. These criteria include:

- Each commit must contain at least three hunks;
- Each commit must include hunks where the number of changed lines is fewer than 15, in alignment with our model's input length limitations;
- Commit messages must be in English and contain more than five tokens to ensure sufficient descriptiveness;
- Commits should not include automatically generated source files (e.g., Java files marked with `@auto`) or non-source files such as `.bak`, `.log`, and `.pyc` files.

As detailed in Table 5.3, these stringent selection criteria resulted in an average commit filter rate of 6.89%. We then trained our dependency analyzer on a subset of 49 projects representing various programming languages. This training involved 77,000 positive pairs and 24,000 randomly sampled negative pairs, ensuring a robust dataset for developing an effective analyzer.

Table 5.3: Benchmark of CoEdPilot including 471 projects with five programming languages.

Language	Model	Train	Valid	Test	#Proj	#Com	#File	#Hunk
JavaScript	File location	22K	3K	6K	93	34K	34K	658K
	Line location	382K	54K	109K				
	Edit generation	460K	65K	130K				
Java	File location	68K	10K	20K	89	24K	72K	556K
	Line location	335K	47K	95K				
	Edit generation	389K	55K	111K				
Go	File location	46K	7K	14K	98	50K	88K	1174K
	Line location	695K	99K	198K				
	Edit generation	822K	117K	234K				
Python	File location	60K	9K	17K	91	33K	42K	555K
	Line location	327K	46K	93K				
	edit generation	389K	55K	111K				
TypeScript	File location	65K	9K	17K	100	39K	76K	817K
	Line location	480K	68K	137K				
	Edit generation	572K	81K	163K				

5.5.3 Experiment Setup

5.5.3.1 RQ1 (Propagating-file Location)

We construct our training samples by extracting commits that consist of k hunks, collectively referred to as a set H, dispersed across m source files. For each training sample, we designate one hunk, $h \in H$, as the target hunk. The files containing the remaining hunks are considered ground-truth positive files, numbered as m'. To assess the robustness of our model, we also select n files (where $n > m$) that are not part of the commit, labeling these as harmful files to introduce potential false positives into the training data. The effectiveness of CoEdPilot in identifying relevant files is evaluated through its precision and recall metrics. If CoEdPilot reports g files as positive and h out of g files are truly positive, we measure the precision of file location as $\frac{h}{g}$ and the recall of the file location as $\frac{h}{m}$.

5.5.3.2 RQ2 (Propagating-line Location)

We structure our training and evaluation process by parsing a commit consisting of k hunks, collectively labeled as H, dispersed across m files. This setup allows us to generate k distinct training samples by employing the following procedure iteratively:

- In each iteration, we select one hunk, $h \in H$, to serve as the target edit to be

predicted.

- We identify relevant prior edits using $H \setminus h$ with CoEdPilot.
- We apply a sliding window of size s across the m files enabling CoEdPilot to identify the target hunk h.
- This process is repeated k times, each time selecting a different hunk from H as the target edit.

5.5.3.3 RQ3 (Edit Generation)

Given a commit with a set of hunks H, one hunk h is selected as the target edit. The remaining hunks form the set of prior edits, defined as $H' = H \setminus \{h\}$. We use beam search to generate the top-1, top-3, top-5, and top-10 edit options for each edit location.

For evaluation, we measure performance using:

1. **Exact Match Rate (EMR):** This metric evaluates the accuracy of the entire commit (i.e., the edit session). If the generated edit content matches the ground truth edit in m out of k cases, the EMR is calculated as $\frac{m}{k}$.
2. **BLEU4 Score:** Following [159], we compute the BLEU4 score of the generated edit content. Specifically, we determine the highest BLEU4 score across all k predictions for each configuration.

5.5.3.4 RQ4 (Prior Edit Prediction)

We evaluate the impact of using selective prior edits (identified by our *Edit-dependency Analyzer*) versus random prior edits in training the edit locating models and the edit generation models. The performance is compared based on the metrics outlined in Section 5.5.3.2 and Section 5.5.3.3.

5.5.3.5 RQ5 (Performance Boost)

The experiment is designed to evaluate the performance boost provided by CoEdPilot when integrated with state-of-the-art solutions. We select GRACE [75], CCT5 [118], and CoditT5 [226] as baselines to observe the enhancement effect of CoEdPilot. CoPilot [70] is excluded from the comparison due to the unavailability of its programming API at the time of this study.

- **Rough Edit Location:** We provide the baseline models with rough edit locations, represented as general hunk areas, to evaluate their performance in generating edits.
- **Precise Edit Location:** We integrate the baseline models with our edit location model, supplying them with specific line-level edit locations, to assess their performance under more precise guidance.

5.5.4 Experiment Results

5.5.4.1 RQ1 and RQ2 (Propagating-file Location & Line)

Table 5.4 summarizes the performance of CoEdPilot in detecting edit locations at different granularities (i.e., file-level and line-level). At the file level, the tool achieves an average precision of 79.52% and a recall of 72.93%. For line-level detection, it attains an average precision of 86.97% and a recall of 84.82%. The effectiveness of CoEdPilot is exemplified in identifying specific edit patterns, such as those seen in the commit `4bf1c` in Golang/Go project. Additionally, the average runtime overhead for inferring edits in a file is recorded at 1.6 seconds, demonstrating the tool's operational efficiency.

Challenges and Future Directions We discuss and explore all the commits and summarize the reasons for wrong prediction samples as follows:

Reason 1: Noisy Samples in the Training Dataset. Despite rigorous filtering, the quality of the training dataset significantly impacts the performance, particularly in the localization of subsequent edits. Noisy data continues to pose challenges. Despite applying several criteria to filter out commits, we observed that noisy training samples can still introduce adverse effects. One notable issue is that some programmers include *irrelevant* file changes (and edits) within a single commit, complicating CoEdPilot's ability to identify edit locations accurately. Additionally, we found that a significant number of edits pertain to code comments and documentation (e.g., commit `3f442` in the `golang/go` project), which may not be effectively captured by CoEdPilot.

These extraneous changes complicate CoEdPilot 's ability to accurately identify and report edit locations. Cleaning the dataset to enhance edit relevance involves considerable effort and is both iterative and interactive, requiring a blend of human oversight and automated processes. This aspect of data refinement is earmarked for future development.

Table 5.4: The accuracy of propagating-file & line location

Programming Language	File Location		Line Location		
	Precision (%)	Recall (%)	Accuracy (%)	Precision (%)	Recall (%)
JavaScript	81.52	71.21	94.89	86.62	83.88
Python	70.84	73.40	94.48	85.03	82.64
Java	85.28	75.67	95.37	87.99	85.99
Go	80.10	72.12	95.79	88.99	87.32
TypeScript	79.84	72.25	95.23	86.21	84.25
Average	79.52	72.93	95.15	86.97	84.82

Reason 2: Informativeness of Edit Inference. Another issue stems from the unidirectional nature of certain code edits. For instance, the addition of a method call typically necessitates the inclusion of a corresponding library import, but the reverse may not always be true. This non-causal relationship between edits presents a significant challenge for inference mechanisms. While the current language model captures interactions between edits, integrating additional information sources, such as test cases, could potentially enhance the accuracy and robustness of the inferences.

These insights underline the complexity of edit pattern recognition and suggest avenues for enhancing the efficacy of CoEdPilot through targeted dataset refinement and enriched inference mechanisms in future iterations.

5.5.4.2 RQ3 and RQ4 (Edit Generation & Prior Edit Prediction)

Table 5.5 outlines the overall performance of edit generation using Top-k candidates. Furthermore, Table 5.6 emphasizes the impact of prior edits in improving the accuracy of locating subsequent edits and generating corresponding content. The results demonstrate that (1) CoEdPilot achieves strong performance in generating edit options. (2) Selective prior edits significantly enhance the performance of the system. In contrast, random prior edits disrupt the patterns, introducing additional noise and confusion during the recommendation process. Moreover, we observe that the reasons for mispredictions in edit options align closely with those discussed in Section 5.5.4.1.

5.5.4.3 RQ5 (Performance Boost)

Table 5.7 compares the performance of CoEdPilot against three baseline methods—GRACE, CCT5, and CoditT5—for generating the top-1 edit option. As detailed in Section 5.5.3.5, our experimental setup involves providing fine-tuned baseline models with hunk-level loca-

Table 5.5: The performance of edit generation

Programming Language	Metric	Top-1	Top-3	Top-5	Top-10
Javascript	**BLEU4**	60.70	69.71	71.37	73.02
	EMR(%)	41.83	47.50	49.31	50.99
Python	**BLEU4**	57.59	65.65	67.47	69.11
	EMR(%)	33.48	38.52	40.41	42.09%
Java	**BLEU4**	60.54	68.35	70.11	71.73
	EMR(%)	40.69	46.87	48.78	50.51
Go	**BLEU4**	65.37	71.96	73.47	74.98
	EMR(%)	48.94	55.09	57.18	59.16%
Typescript	**BLEU4**	61.75	70.31	71.99	73.68
	EMR(%)	41.58	46.86	48.57	50.65

Table 5.6: The Relevance Score for the Edit Location & Generation of Prior Edits

Prior Edit Relevance	Edit-propagating line locator			Edit-content generator	
	Accuracy (%)	Precision (%)	Recall (%)	EMR (%)	BLEU4
Selective Prior Edits	94.89	86.62	83.88	41.83	60.70
Random Prior Edits	91.86	81.73	72.37	18.87	46.56

tions—that is, the specific lines involved in a hunk—to predict the necessary code edits. It is evident from our findings that there is a significant performance gap between CoEdPilot and these baseline models. The primary reason for this disparity is that CoEdPilot's edit locator component significantly enhances the precision with which the edit generator can apply modifications, allowing for more accurate and contextually appropriate changes.

Expanding on the capabilities of CoEdPilot, we experimented with substituting our native edit generation model with those from fine-tuned baselines and observed a substantial improvement in the performance metrics of both GRACE and CCT5 models. This improvement underscores CoEdPilot's versatility and effectiveness as an integrative framework that boosts existing technologies. It is important to note that while CoditT5 also has the capability to predict edit locations similar to CoEdPilot, we did not pair it with our locator. The reason for this is our model's ability to utilize a greater input length than CoditT5, which is particularly advantageous given the limitations of existing models like CodeT5.

This strategy not only showcases the robustness of CoEdPilot but also highlights its potential to serve as a foundational framework that can be further enhanced by integrating

Table 5.7: CoEdPilot Enhance the Performance

Approach	**EMR(%)**	**BLEU4**
CoEdPilot (Line Locator + Edit Generator)	**29.96**	**78.58**
CoditT5	7.42	69.01
GRACE	2.73	38.36
CCT5	14.19	75.37
GRACE + Line Locator	18.61	71.61
CCT5 + Line Locator	15.45	78.27

Table 5.8: Runtime Estimation of CoEdPilot

Step	**File locator** (s / file)	**Line locator** (s / file)	**Edit-content generator** (s / location)
Prepare Input	0.0064	0.3976	0.0683
Model Inference	0.1008	0.0878	0.3972
Total	0.1072	0.4854	0.4655

diverse model components, thereby extending its utility and applicability in real-world coding environments.

5.6 User Study

To further assess how programmers can utilize CoEdPilot in practical settings, we have devised a user study to evaluate its features.

Baseline. To determine the efficacy of CoEdPilot in supporting real-world code editing tasks, we have selected Copilot [70] as the baseline comparison due to its widespread use and proven effectiveness in code generation. We decided not to include a full manual editing mode in this study for the following reasons: (1) Copilot is enhanced by the advanced GPT-3.5 Turbo technology, which has been empirically demonstrated to increase programming productivity by 27% to 57% [148], and (2) constraints related to budget and logistical overhead.

Participant. We recruited 18 participants from three universities across China and Singapore, comprising both undergraduate and graduate students. A preliminary assessment, including a test based on their programming experience, was conducted to form a demographic profile, detailed in [40]. Participants were then evenly divided into two groups according to their experience levels. The experimental group utilized CoEdPilot, while the control group worked with Copilot during the study.

Code Edit Tasks. We created simplified versions of three real-world code commits to

reduce comprehension complexity and help participants focus solely on the editing tasks. The tasks were designed to encompass a variety of common programming activities:

- **Bug Fix (Task 1)**: Participants were presented with a bug where `range(arr)` was incorrectly used instead of `range(len(arr))` in multiple instances throughout the project. They were tasked with identifying and correcting all occurrences of this error.
- **Refactoring (Task 2)**: Participants were instructed to refactor the code by extracting three duplicate code blocks into a reusable function.
- **Feature Enhancement (Task 3)**: Participants were asked to add a *scale* capability for normalizing input vectors in an existing class of classifiers. This task required making multiple interdependent edits across the codebase.

Study Setup. We initiated the study with a comprehensive warm-up session that included a tutorial for both CoEdPilot and Copilot, complemented by a practice task to ensure participants were comfortable with the tools. Each participant was allotted 30 minutes to complete each designated task. To facilitate the validation of their edits, we prepared specific test cases for each task. These test cases were meticulously designed to ensure that participants could verify the correctness of their edits.

Throughout the study, participants were instructed to record their sessions using a video recorder, enabling us to perform a detailed analysis of their coding process. Performance metrics were evaluated based on two main criteria: (1) The ability to complete the tasks, as evidenced by passing all designated test cases and (2) The efficiency with which participants completed the tasks, measuring how quickly and effectively they reached solutions.

Results. Table 5.9 presents the performance of participants in completing the code-editing tasks, leading to the following observations:

- **Task 1**: On average, CG completes Task 1 slightly faster than EG, though the difference is not statistically significant (Wilcoxon Signed Rank test p-value = 0.33, effect size = -0.08).
- **Task 2**: EG demonstrates faster completion times than CG for Task 2, but this improvement lacks statistical significance (p-value = 0.07, which is greater than 0.05; effect size = 0.60).

Table 5.9: The overall performance in seconds of Experimental Group (EG) and Control Group (CG).

EG	Task1	Task2	Task3	CG	Task1	Task2	Task3
P1	221	515	1196	**P10**	339	696	1287
P2	897	389	279	**P11**	360	776	1563
P3	366	487	216	**P12**	480	483	545
P4	160	529	963	**P13**	522	724	1770
P5	230	301	756	**P14**	277	395	838
P6	364	473	617	**P15**	181	446	930
P7	329	688	588	**P16**	337	720	825
P8	840	780	1020	**P17**	151	666	1515
P9	290	638	1050	**P18**	266	722	1563
Average	410.78	533.33	742.78	**Average**	323.67	625.33	1070.33

- **Task 3**: Conversely, EG significantly outperforms CG in Task 3, with the difference in completion time being statistically significant (p-value = 0.003, which is less than 0.05; effect size = 0.96).

Why CG Outperforms EG in Task 1? In Task 1, aimed at fixing duplicated bugs, we observed that CoEdPilot users (EG group) initially struggled with the tool's learning curve, particularly in utilizing features for predicting edit locations and content. Additionally, participants such as P2 and P8 exhibited hesitancy in trusting the tool's recommendations despite their accuracy. This skepticism led them to spend more time verifying the suggested edits, slowing their overall performance. This scenario is typical for users adapting to any new tool, whether in a user study or an operational setting. In contrast, some participants in the Copilot group (CG), such as P17, efficiently utilized simple keyword searches to locate necessary edits across the project due to the straightforward edit pattern in Task 1.

Why EG Outperforms CG in Task 2, Albeit Without Statistical Significance? In Task 2, which involved refactoring by method extraction, CoEdPilot users grew more adept at navigating the tool, effectively switching between functions like location prediction, edit generation, and edit option selection. This proficiency allowed them to identify cross-file code duplications more efficiently, reducing the effort required for creating new functions. Although the EG group began to outperform the CG group, the statistical significance (a p-value of 0.07 approaching the critical threshold of 0.05) was not strong enough to conclusively demonstrate a difference due to the limited sample size.

How EG outperforms CG in Task 3? Task 3, which involved enhancing model training with a *scale* function, proved to be the most challenging. The required edits, such as

inserting a `scale` parameter and associated decision logic, could not be easily identified through keyword searches. The EG group generally outperformed the CG in handling these complex editing patterns. However, performance varied significantly among participants: some completed the task in less than five minutes, while others took considerably longer. Analysis of tool logs and video recordings revealed that some participants misinterpreted edit content, leading to erroneous code. Such errors propagated further confusing edits, which were only identified after testing the edits. Human error in interaction-heavy tools remains a persistent challenge. We aim to address these issues in future iterations of CoEdPilot. The EG group accepted approximately 69.3% of the recommended edit options, modifying 31.6% of these suggestions.

5.7 Threats to Validity

This section outlines potential factors that might compromise the validity of our work:
Internal Validity: There exists a potential bias in the study's internal validity due to different levels of familiarity with the tools used. The experimental group may experience a steeper learning curve than the control group, which is already familiar with CoPilot. This disparity in learning experiences might cause the differences observed in the test results, attributing them more to learning effects than to the true performance of the extension.
External Validity: Our study's code editing tasks were simplified versions of real-world code commits, complete with detailed instructions. This setup may not accurately reflect the complexity and unpredictability of typical coding scenarios developers encounter. Additionally, the study's focus solely on Python programming tasks may limit the applicability of our findings across other programming languages, potentially skewing the perceived effectiveness of the plugin.
Statistical Validity: Due to constraints on time and resources, the study was conducted with a relatively small sample size of 18 participants. This limited participant pool may not provide enough statistical power to reliably detect significant differences in the effectiveness of the extension. As such, the generalizability and robustness of our findings could be questioned, suggesting a need for cautious interpretation of the study results.

Chapter 6

Program Adaptation

6.1 Introduction

Deep language models have progressively scaled in size to enhance their performance. (Ultra-)large language models, such as GPT-3 [153], CodeX [34], and InstructGPT [156], have demonstrated remarkable effectiveness. However, these models typically comprise millions of neurons, resulting in significant training and maintenance costs, making them less accessible for individual users.

In contrast, relatively smaller models, such as CodeBERT [57] and CodeT5 [204], are more cost-effective and easier to maintain. Nevertheless, these smaller models often face challenges in handling large and diverse training datasets, leading to compromises in performance across varied training samples. Researchers usually prepare a diversified corpus regarding different projects, topics, and programming languages to achieve a more generalizable performance for software engineering tasks.

For example, the CodeSearchNet[88] dataset used to train CodeBert and CodeT5 consists of 6 program languages, more than 100 projects, and multiple programming topics. The diversity is expected to train a model that can generalize well in as many unseen code samples as possible. However, the multiple training samples can also lead to a "conflicting" effect when the model is training. Our study quantifies the conflicting effect of a pair of training samples s_1 and s_2 by Equation 6.1.

$$k_{conf} = \frac{\partial loss(s_1)}{\partial \theta} \times \frac{\partial loss(s_2)}{\partial \theta} \tag{6.1}$$

In Equation 6.1, $loss(.)$ is the function to measure the loss of a training sample and θ represents all the trainable parameters of a deep neural network. Intuitively, k_{conf} being

positive indicates θ fitting s_i can facilitate the fitting of s_j and vice versa. In contrast, k_{conf} being negative indicates θ fitting s_i has the cost of fitting s_j and vice versa. Our empirical study on the popular CodeSearchNet datasets and Funcom[110] dataset shows that 78% training samples have a *conflicting* effect with each other. Overall, we observe that conflicting effects are prevalent regardless of the types of models and training corpora.

We take the code summarization task as an example. To achieve an adaptive model, we propose Adacom, a real-time model adaptation solution designed to enhance the performance of deep neural networks. Adacom takes as input a target code c, a comment generator g, and a dataset D. It adapts g in real time to produce an improved model g', enabling better performance in generating comments for c.

Our approach is based on the rationale that there exists a subset of samples $D_s \subset D$ that are particularly useful for the comment generator to learn how to summarize c. Technically, the process involves the following steps:

1. **Building an Influence Graph:** We construct an influence graph G_{inf} over the dataset D, where each node represents a sample, and edges indicate whether the relationship between two samples is helpful or harmful for training.

2. **Identifying Semantic Helpful Samples:** For a given target code c, we employ a model-representation-based metric to identify training samples in D that are potentially semantically helpful to c.

3. **Enriching the Helpful Sample Set:** Using the influence graph G_{inf}, we expand the set of helpful samples D_s. This set is then used to fine-tune the model in a lightweight manner, allowing the model to:

    - *Learn* from helpful samples, and
    - *Unlearn* harmful samples.

4. **On-the-Fly Retraining:** If certain training samples are identified as having a positive influence or being semantically helpful for c, the model g is retrained on the fly to produce g'. This retraining reinforces helpful samples and eliminates the influence of harmful samples.

Adacom thus enables a dynamic, targeted adaptation process to improve comment generation for specific target code snippets.

Above all, we design Adacom to achieve three key research goals:

- **Compromise Detection:** Adacom aims to detect whether a language model may exhibit compromised performance on a given test sample.
- **Sample Searching:** Given a test sample and a dataset, Adacom identifies a subset of samples within the dataset that can help mitigate model bias and enhance specialization for the test sample.
- **On-the-Fly Model Tuning:** Leveraging a test sample and a few labeled samples, Adacom dynamically adapts the model in real-time to improve its performance on the test sample.

The solution provided by Adacom is orthogonal to many deep-learning-based methods, including advancements in model architectures [126], model training [74], and representation learning [113]. Additionally, Adacom is delivered as an assistant framework with abstracted APIs, enabling support for various models based on the encoder-decoder architecture.

We evaluate Adacom by testing its ability to enhance seven comment generators across four datasets. The experimental results demonstrate that:

1. Adacom significantly improves comment generation performance, with average increases of 14.9% in BLEU4, 12.2% in METEOR, and 7.4% in ROUGE-L scores.
2. The adaptation process for an individual code sample incurs minimal runtime overhead.
3. Adacom generalizes effectively to out-of-distribution code samples, showcasing its robustness.

In summary, this work makes the following contributions:

- **Empirical Observation**: We report the conflicting effect on the training datasets over the state-of-the-art deep language models, which shed light on one of the potential performance bottlenecks on the comment generators.
- **Technical Design:** We propose the Adacom solution, generalizable for any deep language models with encoder-decoder architectures, to boost their performance on individual code samples on the fly.

Table 6.1: Motivating Example: Compromise Problem in Neural Network Models with Conflicting Training Effects on Test Predictions

Sample	Target Test Sample, c_t	Harmful Training Sample, c_{harm}	Helpful Training Sample, c_{help}
Code	`public static java.sql.Timestamp internalToTimestamp (long v) { return new java.sql.Timestamp (v - LOCAL_TZ.getOffset(v)); }`	`private Date longToDate (long val, int sqlDataType) { switch (sqlDataType) { case Types.DATE: return new java.sql.Date(val); case Types.TIME: return new java.sql.Time(val); case Types.TIMESTAMP: return new java.sql.Timestamp(val); } }`	`public static java.sql.Date internalToDate( int v) { final long t = v * MILLIS_PER_DAY; return new java.sql.Date(t - LOCAL_TZ.getOffset(t)); }`
Ground Truth	Converts the internal representation of a SQL TIMESTAMP (long) to the Java type used for UDF	Parse the long-valued timestamp into the appropriate SQL date type	Converts the internal representation of a SQL DATE (int) to the Java type used for UDF parameters (@link java.sql.Date)
Origin	Converts the internal { @ link long } (local to a { @ link TIMESTAMP }) representation		
After Harmful	Parse the long-valued TIMESTAMP into a { @ link TIMESTAMP } representation		
After Helpful	Converts the internal representation of a SQL TIMESTAMP (long) to the java type used for UDF		

- **Experimental Evaluation**: We conduct extensive experiments on seven models over four datasets, showing that our Adacom solution can significantly boost the performance of comment generation under diverse scenarios.

6.2 Overview

Given a piece of source code c, we first track the most influential training samples S_{inf}, which should be the most useful for contributing to g's performance on c. Next, we estimate the potential boosting performance of g on c by (1) how well g can fit S_{inf} and (2) how consistent the training samples in S_{inf} can facilitate g's performance on c. Then, a confidence score is derived as the measurement. Intuitively, the better performance of g can fit S_{inf}, and the more consistent S_{inf} can facilitate g's performance on c, the higher the confidence score. Once the confidence score exceeds a pre-defined threshold, we fine-tune g on S_{inf} with a limited number of learning steps to g' to boost the comment generation performance of c.

Table 6.1 shows an example of how Adacom applies on-the-fly adaptation to adap-

tively switch the compromise of a CodeBERT-based comment generator to improve its performance on a specific code sample. In Table 6.1, three code samples are highlighted: the second column represents the target code c_t for generating comments, the third column contains the harmful training sample c_{harm} that negatively impacts predictions for the target code, and the last column presents the helpful training sample c_{help} that improves predictions for the target code.

The second row provides the code text, while the third row lists the ground truth comments. The fourth row shows the comments generated by the model after fine-tuning without any additional training. The fifth row displays the comments generated after training on harmful samples, and the last row demonstrates the output after training on helpful samples. Overall, the results indicate that training with helpful samples leads to improved performance in generating comments for c_t. Next, we illustrate how Adacom detects the model compromise and adapts the model for a more precise code summarization performance.

Step 1. Influence Construction. In this example, Adacom begins by determining the influential relationships for the training samples. For simplicity, we highlight only the influence between c_{harm} and c_{help}, as shown in Table 6.2. The calculated influence score for c_{harm} and c_{help} is -0.76, suggesting that the deep learning model has adjusted its behavior to accommodate this pair. Notably, c_{harm} and c_{help} exhibit a similar get-by-return code structure, but their associated comments differ significantly in tone and style, reflecting project-specific conventions. Details on the influence score computation can be found in Section 6.4. At this point, we identify and store all "contradictory" pairs within the training dataset for further analysis.

Step 2. Training Contribution Construction. Next, to find the helpful training samples, we estimate the potential contribution of the training samples to the target code c_t. We assume that the c_{help} is one of the semantically relevant training samples. Intuitively, c_t and c_{help} share a similar representation within the deep comment generator, providing a strong indication for selecting c_{help}. Using the influence score, we further expand c_{help} into a set of related code samples. The methodology for measuring training contributions will be detailed in Section 6.4.3. In some cases, Adacom may not identify any contributing training samples. In the example shown in Table 6.1, the set c_{help} is identified as the contributing sample. At this stage, we assemble the set of potentially helpful samples from the training dataset for further analysis.

Table 6.2: We show the influence scores and the corresponding estimated training contribution of the examples shown in Table 6.1.

Sample	Estimated Influence		Estimated Contribution	
	c_{harm}	c_{help}	c_{harm}	c_{help}
c_t	/	/	0.23	0.67
c_{harm}	1	-0.76	/	/
c_{help}	-0.76	1	/	/

Step 3. On-the-fly Retraining. Finally, helpful code samples such as c_{help} are utilized to retrain the comment generator, transforming g into an updated version, g'. While both c_{harm} and c_{help} are effectively modeled by g, retraining introduces observable changes in the generated comments, particularly in the last three rows. Before retraining, the original comment generator produces a broader and less accurate comment. After incorporating the helpful sample, however, the updated generator g' generates a more specific and precise comment for c_t—without relying on the ground-truth comment.

In essence, Adacom is designed to identify potential compromises and adapt the model for targeted scenarios. It is important to note that both g and g' involve trade-offs. The original generator g aims to balance fitting c_{harm} and c_{help}, striving for a globally optimized solution. In contrast, the retrained generator g' prioritizes fitting c_{help} over c_{harm}, adopting a more localized optimization approach to improve the comments for c_t. As demonstrated in our experiments (see Section 6.5), this targeted adaptation enhances the performance of state-of-the-art comment generators with minimal runtime overhead.

6.3 Problem Formulation

In this work, we define the problem formulation as follows. Given a comment generator g, the training dataset $D = \{d_i = \langle c_i, com_i \rangle\}$, one target code sample c_t, and its ground truth comment com_t, without knowing com_t, we need to find a subset selection solution $W = \{w_i \in [0, 1]^1\}$ on D to minimize $\mathcal{L}(g(c_t), com_t)$ where:

$$g' = \arg\min \sum_{i=1}^{N} w_i \cdot \mathcal{L}(g(c_i), com_i) \tag{6.2}$$

In Equation 6.2, we assume that g' is trained from g, $w_i = 1$ indicates that the sample d_i remains in the training set; in contrast, $w_i = 0$ indicates that the sample d_i is removed from the training set. As for the motivating example in Section 6.2, if we only retrain the

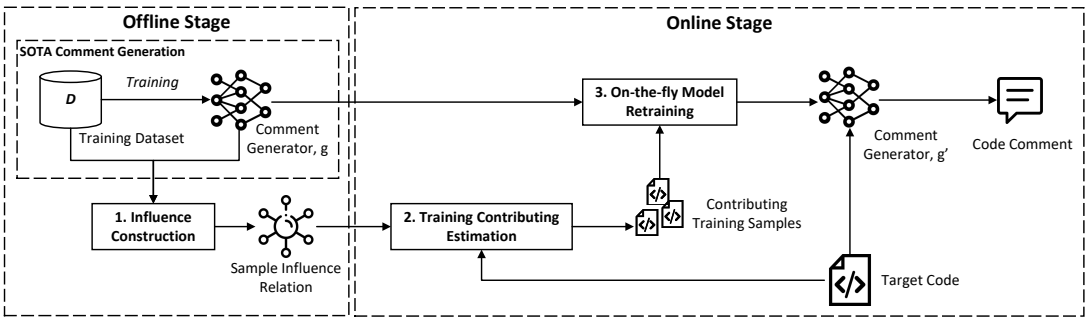

Figure 6.1: Adacom framework.

comment generator on the subset $\{c_{help}\}$, we set the weight w_i for c_{help} to be one and all the other weights to be 0. Generally speaking, the problem is challenging because

- **Information Insufficiency:** Given that com_t is unknown, we cannot define a precise loss function $\mathcal{L}(g(c_t), com_t)$. Therefore, an estimation of com_t is needed;
- **Combinatorial Explosion:** The training dataset is usually huge even if we have the estimated comment com_t^*. Thus, it would be computationally expensive to find the global optima W to minimize $\mathcal{L}(g(c_t), com_t^*)$.

We address the information insufficiency problem by estimating the target sample's unknown loss using similar training samples in D. The technical challenge lies in how we systematically consider the similarity of both the code and the model interpretation of the code for more precise estimation. Further, we address the combinatorial explosion problem by constructing the influence graph on the training dataset D, which allows us to prune many less influential and negatively influential training samples.

6.4 Approach

Figure 6.1 illustrates the framework of Adacom, which is divided into two stages: an offline stage and an online stage. We assume that a state-of-the-art method has been used to train the comment generator g on a dataset D.

During the offline stage, the comment generator g is trained using the dataset D. Based on D and g, we define the concept of influence estimation and construct an influence graph that captures the pairwise mutual influence among training samples in D. This Influence

Construction process identifies helpful and harmful samples, while cached representations and influence scores facilitate efficient retrieval during the online stage.

In the online stage, the system retrieves semantically helpful training samples based on the target code and uses them to retrain the comment generator, transforming g into an updated version, g'. This lightweight retraining process adapts the model to generate new and improved code comments. Specifically, training contributions are estimated to identify samples with potential semantic relevance to the target code, enabling the generation of more accurate and precise comments.

6.4.1 Influence Construction

We first introduce the definition of the influence score and then utilize the training contribution to estimate the influence score of the training samples to the test sample.

Influence Definition We denote a comment generator as g, its training dataset as $D = \{d_1, d_2, ..., d_n\}$ where $d_i = (c_i, com_i)$, c_i is the code and com_i is its comment. Let $\mathcal{L}(g(c_i), com_i)$ be the loss of d_i, and space of all the possible generators be $\mathcal{G}$. The state-of-the-art approaches search for $g^* = \arg\min_{g \in \mathcal{G}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(g(c_i), com_i)$. Given two samples d_1 and d_2 where $d_1, d_2 \in D$, we define the influence of d_2 on d_1, $inf(d_1, d_2, \epsilon)$, as follows:

$$inf(d_1, d_2, \epsilon) = \mathcal{L}(g^*(c_1), com_1) - \mathcal{L}(\hat{g}(c_1), com_1) \tag{6.3}$$

$$g^* = \arg\min_{g \in \mathcal{G}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(g(c_i), com_i) \tag{6.4}$$

$$\hat{g} = \arg\min_{g \in \mathcal{G}} \frac{\sum_{d_i \in D \setminus \{d_2\}} \mathcal{L}(g(c_i), com_i) + (1+\epsilon)\mathcal{L}(g(c_2), com_2)}{n} \tag{6.5}$$

In Equation 6.3, we define the influence of d_2 on d_1 by the difference of the performance of two trained comment generators g^* and $\hat{g}$ on the loss of d_1. g^* is the trained generator on the dataset D as showed in Equation 6.4; in contrast, $\hat{g}$ is the trained generator on the dataset D where the training weight of d_2 is readjusted by ϵ ($\epsilon > 0$). Intuitively, $inf(d_1, d_2, \epsilon)$ evaluates whether the model can better fit if we upweight d_2 by ϵ during the model training. Further, we call the ϵ the *empirical influence margin*. Intuitively, the larger positive $inf(d_1, d_2, \epsilon)$, the more influential d_2 is on d_1.

Similarly, the influence of d_1 on d_2 can be denoted as $inf(d_2, d_1, \epsilon)$. Therefore, we define the mutual influence between d_1 and d_2 as $mul\_inf(d_1, d_2, \epsilon)$ as

$$mul\_inf(d_1, d_2, \epsilon) = inf(d_1, d_2, \epsilon) + inf(d_2, d_1, \epsilon) \tag{6.6}$$

We can see that Equation 6.6 is symmetric. Intuitively, a large positive $mul\_inf(d_1, d_2, \epsilon)$ indicates that d_1 and d_2 can "help" each other during the model training; a large negative $mul\_inf(d_1, d_2, \epsilon)$ indicates that d_1 and d_2 can "harm" each other during the model training; and small $|mul\_inf(d_1, d_2, \epsilon)|$ indicates that d_1 and d_2 are independent of each other.

While the above definition is intuitively measurable, it is challenging to make it practical because

- **Computationally Cost**: The above definition requires exhaustively retraining the model for every pair of training samples. While the training dataset is vast, the retraining cost is hardly acceptable.
- **Empirical Hyperparameter**: The empirical influence margin usually needs to be decided through trial and error, which incurs additional engineering overhead.

6.4.2 Estimated Influence

We propose a gradient-based method for efficiently estimating the *model-dependent* mutual influence between any pair of training samples.

Influence Estimation Consider a comment generator g with a training dataset $D = \{d_1, d_2, \ldots, d_n\}$, where each sample $d_i = (c_i, \text{com}_i)$ comprises a code snippet c_i and its corresponding comment com_i. Let $\mathcal{L}(g(c_i), \text{com}_i)$ denote the loss for sample d_i, and let θ represent the parameters of the trained comment generator g^*. The empirical mutual influence mul_inf_e between d_1 and d_2 is defined as:

$$mul\_inf_e(d_1, d_2) = \frac{grad(d_1) \cdot grad(d_2)}{|grad(d_1)| \cdot |grad(d_2)|} \tag{6.7}$$

$$grad(d_i) = \frac{\partial \mathcal{L}(g^*(c_i), com_i)}{\partial \theta} \tag{6.8}$$

Equation 6.7 is model-dependent compared with the definition in Equation 6.6. In other words, we now empirically investigate how likely it is that we can reduce the loss of

d_1 by reducing the loss on d_2 (and vice versa) on the comment generator $g*$. Nevertheless, Equation 6.7 has the following benefits:

- **Symmetry**: Similar to Equation 6.6, the gradient-based calculation is symmetrical, i.e., $mul\_inf_e(d_1,d_2)$ = $mul\_inf_e(d_2,d_1)$. The property is helpful for many follow-up calculations, such as clustering (see Section 6.4.2).
- **Efficiency**: Different from Equation 6.6, $mul\_inf_e(d_1,d_2)$ does not require retraining the model. The complexity is only relevant to the size of the model. We can even empirically select partial model layers to improve efficiency.
- **Bound**: $mul\_inf_e(d_1,d_2)$ is bounded within (-1, 1), which is convenient for the follow-up measurement and calculation. Specifically, $mul\_inf_e(d_1,d_2)$ being close to 1 indicates that d_1 and d_2 are mutually helpful, that being close to -1 indicates that d_1 and d_2 are mutually harmful.

Influence-based Graph Taking the mutual influence between every pair of the training samples, We further build the influence graph by clustering all the training samples by the Birch hierarchical clustering algorithm [230]. We use complete linkage for the clustering to ensure the helpful strength of the samples in each cluster. As a result, given a similarity threshold th, the training dataset D is converted into a cluster set $C_{inf} = \{c_1, c_2, ..., c_k\}$. For each $c_i \in C_{inf}$, we measure its helpful strength by

$$strength(c_i) = \frac{2}{|c_i| \times (|c_i| - 1)} \sum_{d_p, d_q \in c_i, p \neq q} mul\_inf_e(d_p, d_q) \tag{6.9}$$

6.4.3 Training Contribution Construction

In this section, we focus on estimating the potential of a training sample $d_{tra} = (c_{tra}, com_{tra})$ to improve the generated comment for a target code c_t with an unknown comment. Using a comment generator g, we evaluate how effectively d_{tra} can be utilized to retrain g to enhance its prediction for c_t.

Semantic Similarity The underlying principle is that training samples whose internal representations, as perceived by the model g, closely resemble that of c_t are more likely to belong to a similar distribution. These semantically similar samples are thus more

informative and better equipped to improve the prediction quality for c_t. The choice of the internal representation allows us to have two benefits:

- **Model-specific Interpretation:** Internal representation lets us capture how the trained comment generator (e.g., CodeBERT or Code T5) interprets the training samples. Note that code that appears similar from a human perspective may not necessarily be identical from the model's perspective. Nevertheless, the model can interpret them differently.
- **Semantic Generalization:** The state-of-the-art approaches with the BERT-based architecture (e.g., Code T5, CodeBERT, GraphCodeBERT) allow us to generalize synonyms (e.g., "admin" and "administrator", "service" and "svc", etc.) more easily in similar contexts.

Representation Assumption We hypothesize that deep learning-based language models generate internal representations for each code sample. Let a code sample be represented as $c = \langle t_1, t_2, ..., t_n \rangle$ and the comment generator as g. When processing token t_i, the model g produces an internal representation denoted as $r_i = h(t_i)$, where $h(.)$ is the internal representation function of g. We think that the assumption holds for most neural network-based models. For instance, both unidirectional and bidirectional recurrent neural networks, as well as transformer-based architectures, satisfy this condition. Figure 6.2 provides an overview of the encoder architecture used in BERT-based transformers. For simplicity, details such as multi-head attention and the computation of query, key, and value vectors are omitted. After tokens are transformed into vector embeddings using the *word2vec* layer, they pass through stacked encoder layers to obtain optimal internal representations, which are then used by the decoder to generate comments. The space R, known as the *representation space*, encompasses all possible internal representations.

Contribution Estimation Formally, given one test sample $c = \langle t_1, t_2, ..., t_n \rangle$ and its semantic internal representation $\hat{c} = \langle h(t_1), h(t_2), ..., h(t_n) \rangle = \langle r_1, r_2, ..., r_n \rangle$, it can be regarded as a dynamic trajectory in the representation space. We show an example in Figure 6.3. To clarify our illustration, we demonstrate the representation space in a 3-dimensional space, where each point represents the internal representation when the model generator parses a token t_i. The black sequence represents the internal representation of the target code, the green one represents the helpful training sample, and the red one

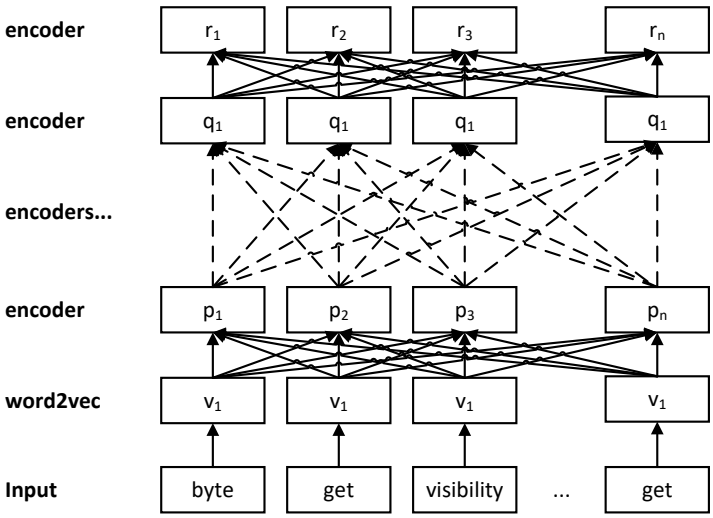

Figure 6.2: A simple illustration of BERT architectures.

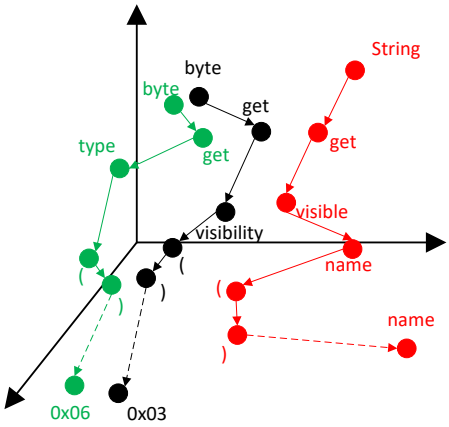

Figure 6.3: We use trace similarity to estimate the contribution of training samples to the test sample.

represents the harmful training sample. Further, the same token can have a different representation (or position in the representation space) because they may have different contexts. This differs from the *word2vec* solution, where a token can always have its fixed representation. For example, the token "get" in the code sample "byte get visibility ..." has a different representation from the token "get" in "String get visible ...".

Representation Comparison. Since a sequence of high-dimensional vectors will represent each code, we compare the target code $c = \langle r_1, r_2, ..., r_n \rangle$ with the training samples. Similar to the classical dynamic algorithm to calculate the longest common subsequence, given two sequences $c = \langle r_1, r_2, ..., r_n \rangle$ and $c' = \langle r'_1, r'_2, ..., r'_m \rangle$, we will still construct

a $n \times m$ matrix. The difference is that there is no exact match between the two token representations. We quantify the contribution of training samples by computing the cosine similarity between the internal representation vectors of two code samples. Generally, we still measure the *minimum editing distance* between two sequences of high-dimensional vectors. Intuitively, Equation 6.10 calculates the accumulated projection of c' on c in the representation space.

The following illustration assumes that readers are familiar with the dynamic programming algorithm for calculating the longest common subsequence. More details can be checked at [132]. Table 6.3 shows how we calculate the pairwise similarity between two representation sequences. Note that the tokens in different sequences have different representations; thus, the similarity between the same token is no longer 1. For example, the similarity between two "byte"-words is 0.87 instead of 1. Then, we introduce a match threshold h_m to only match the representations with pairwise similarity over h_m. We still follow the dynamic programming routine to find the best match between two sequences by constructing the matching matrix as shown in Table 6.4. Specially, given two token representation $v = c[i]$ and $v' = c'[j]$, we have:

1. $M[i,j]$ = $max(M[i-1,j], M[i,j-1])$, if $cos(v,v') > h_m$
2. $M[i,j]$ = $max(M[i,j] + cos(v,v'), M[i-1,j], M[i,j-1])$, otherwise.

As a consequence, given the representation sequences of the target code c and that of a training sample c', we can have a sequence of optimal match $P = \langle (r_{m_1}, r'_{m_1}), (r_{m_2}, r'_{m_2}), ..., (r_{m_k}, r'_{m_k}) \rangle$ where $r_{m_i} \in c$ and $r'_{m_i} \in c'$. Thus, we estimate the training contribution score of c' to c by:

$$tcs(c, c') = \sum_{i=1}^{|P|} cos(r_{m_i}, r'_{m_i}) \tag{6.10}$$

6.4.4 On-the-fly Model Adaptation

Given a target code c and two thresholds th_1 and th_2, Adacom operates as follows: it first identifies a subset of training samples $M = \{m_1, m_2, \ldots\}$ where each sample's training contribution score exceeds the user-defined threshold th_1. If no such samples are found, the model is not retrained. Otherwise, Adacom retrieves additional samples

Table 6.3: The similarity matrix of two sequences of code representations.

	byte	get	visibility	(	)	...	0x03
byte	**0.87**	0.12	0.11	0.12	0.05	...	...
get	0.13	**0.89**	0.21	0.16	0.13	...	...
type	0.07	0.33	**0.56**	0.21	0.19	...	...
(	0.05	0.12	0.18	**0.93**	**0.88**	...	...
)	0.03	0.08	0.15	**0.87**	**0.91**	...	...
...	...	...	...	...	...	...	...
0x06	...	...	...	...	...	...	**0.77**

Table 6.4: The algorithm based on soft-match for the token-level representations.

	byte	get	visibility	(	)	...	0x03
byte	**0.87**	0.87	0.87	0.87	0.87	...	...
get	0.87	**1.76**	1.76	1.76	1.76	...	...
type	0.87	1.76	**2.32**	2.32	2.32	...	...
(	0.87	1.76	2.32	**3.25**	3.25	...	...
)	0.87	1.76	2.32	4.12	**4.16**	...	...
...	...	...	...	...	...	...	...
0x06	...	...	...	...	...	...	**9.76**

$N = \{n_1, n_2, \ldots\}$ from the influence graph that are associated with the nodes in M (see Section 6.4.2). The estimated influence score between the samples in M and N must surpass th_2.

The retraining set is then augmented as $S' = S \cup C_{inf}(c_1) \cup \ldots \cup C_{inf}(c_l)$, where $C_{inf}(c_i)$ represents the influential cluster for c_i. Threshold th_1 filters semantically relevant training samples, while th_2 identifies those that can help the model mitigate the effects of mutually harmful samples and retain mutually beneficial ones. If no mutually harmful samples exist between M and N, as determined by the influence graph, Adacom excludes these samples from further retraining.

Finally, all retrieved samples are used to retrain g, producing an updated model g' that generates a new comment for c as $g'(c)$.

To minimize run-time overhead, Adacom freezes most of the neural network parameters and updates only the last few layers. The number of retrainable layers is treated as a user-configurable hyperparameter. Additionally, dropout and early stopping mechanisms are employed to address overfitting and reduce retraining costs. The retraining process halts either when a user-defined maximum number of epochs is reached or when the retraining loss drops below the specified lower bound th_{lb}.

6.5 Evaluation

We evaluate Adacom by addressing the following research questions:

- **RQ1**: Can Adacom enhance the performance of various comment generation models?
- **RQ2**: Can Adacom improve comment generation across multiple datasets?
- **RQ3**: Does Adacom outperform neuron-based approaches augmented with retrieval techniques?
- **RQ4**: How well does Adacom generalize to improve performance on out-of-distribution samples?
- **RQ5**: What is the runtime overhead of Adacom when enhancing performance?
- **RQ6**: What role does each component of Adacom play in achieving performance improvements?

6.5.1 Experiment Setup

Our experiments were performed on two Ubuntu 20.04 servers, each featuring dual AMD Ryzen™ 9 5950X 16-Core CPUs, 128 GB of RAM, and two Nvidia RTX™ A4000 GPUs. The software environment was set up with Python 3.7 and the Anaconda distribution. PyTorch served as the primary deep-learning framework, and pre-trained models were obtained from the Transformers library (version 4.13.0). All models were fine-tuned using the provided training datasets to achieve optimal performance.

Measurement Following existing literature [7, 109, 3, 206, 89, 6, 25], we use smoothing BLEU4 [160][120], METEOR [12], and ROUGE-L [119] to evaluate the performance of code comment generation.

Baselines We selected seven widely-used language models, including RoBERTa [127], CodeBERT (small and base versions) [57], GraphCodeBERT [74], and CodeT5 (small, base, and large versions) [204]. These models were chosen to ensure diversity in their pre-training corpora (e.g., RoBERTa for natural language and CodeBERT for code), architectural designs, and scalability across different sizes. Roberta, Codebert, and

Table 6.5: The Statistics of the Experiment Datasets

Dataset	Train	Valid	Test
FunCom	1,954,807	104,273	90,908
CosBench	296,425	42,348	84,694
CodeKG	161,857	20,282	40,512
CSN-Python	251,820	13,914	14,918
CSN-PHP	241,241	12,982	14,014
CSN-Go	167,288	7,325	8,122
CSN-Java	164,923	5,183	10,955
CSN-JavaScript	58,025	3,885	3,291
CSN-Ruby	24,927	1,400	1,261

GraphCodebert follow the same architecture but have different pre-trained tasks and parameters. Roberta is pre-trained only on natural language, while all other models are originally pre-trained on the CodeSearchNet dataset. Besides, we use the variants - Codebert-small and CodeT5-small to test how the Adacom improves the performance of small models. CodeT5-large is our experiment's largest model, and it is a comparison example of efficiency and resources.

6.5.2 Datasets

We select four public datasets with six different kinds of program languages, including FunCom [109], CodeSearchNet [88] (CSN), CodeKG [39] and CosBench [213]. Table 6.5 shows the details.

CodeKG Dataset The CodeKG dataset selects the top 100 Java projects based on their GitHub popularity (measured by the number of stars). These projects contain:

- 140,000 Java source files,
- 184,000 classes,
- 16,000 interfaces,
- 1.3 million methods, and
- 5.9 million fields (including formal parameters and local variables).

Automatically generated comments (e.g., those with `@parameter`, `@return`, and `@author` annotations) and comments with copyright claims are excluded. After pre-processing, the dataset contains 202,000 Java methods with both code and associated

comments. The dataset randomly selects 141 thousand (∼70%) code-comment pairs as the training set for those methods with available comments. For the rest of the code-comment pairs, it chooses the code-comment pairs with no graphical connection to any of the nodes in the training set as the testing set. As a result, it has 40 thousand (∼20%) code-comment pairs as the testing set.

CodeSearchNet Dataset The CodeSearchNet dataset, introduced by [88], comprises code-comment pairs collected from publicly available, non-fork GitHub repositories. These repositories must be used by at least one other project and have a license permitting the redistribution of project components. The dataset spans six programming languages: Go, Java, JavaScript, Python, PHP, and Ruby. Only functions (or methods) with associated documentation are included. Each code-comment pair is processed as follows:

The code is tokenized using TreeSitter [24], GitHub's universal parser. The comment is truncated to the first whole paragraph. Code-comment pairs are filtered based on three rules: (1) Documentation must be at least three tokens long. (2) The function name must not contain the substring "test." (3) Identical codes are removed. Dataset sizes for each language are detailed in Table 6.5.

Funcom Dataset The Funcom dataset, prepared by LeClair *et al.* [109] from the Sourcerer repository provided by Lopes *et al.* [130], which contains over 51 million Java methods from 50,000+ projects. The authors applied several filtering steps: (1) Methods preceded by JavaDoc comments (indicated by /**) were retained, with the first sentence extracted as the comment. (2) Comments containing non-English words were excluded. After these steps, 4 million methods remained. Further filtering removed auto-generated code by excluding files with phrases like "generated by." However, unique auto-generated codes were retained. Ultimately, the dataset consists of about 2.1 million method-comment pairs. The dataset is split by project, meaning every project's methods are divided into training, validation, and testing subsets by 90%-5%-5%. Specifically, the size of the training set is 1,954,807; the testing set size is 90,908, and 104,273 for the validation set.

CosBench Dataset The CosBench dataset, created by [213], is designed for evaluating code search methods. It includes three components: Codebase, QAset, and Metrics.

Since QAset and Metrics are irrelevant to this study, we focus on the Codebase. The Codebase comprises 1,000 popular GitHub Java projects, containing 475,783 Java files and 4,199,769 methods, totaling 1.4 Gigabytes. For data cleaning and preprocessing, the authors followed the approach in [72]: (1) Java method names and bodies were split by camel case. (2) Duplicate tokens and stopwords (e.g., "the," "in") were removed. (3) JavaDoc comments were extracted from the Abstract Syntax Tree (AST) of the methods. We split the dataset by 70%-20%-10%, and the training set has 296,425 instances, the test set has 84,694 cases, and the validation set has 42,348 samples.

6.5.3 Experiment Design

To address RQ1 (cross-model evaluation), we fine-tuned multiple language models using the CodeKG dataset. The CodeKG dataset was chosen for its diversity across selected projects and its detailed project-code relationships, which support deeper analysis [39]. Considering the randomness introduced by the dropout mechanism during on-the-fly retraining, each experiment was repeated five times to assess the consistency of Adacom.

For RQ2 (cross-dataset evaluation), we assessed the ability of Adacom to enhance CodeT5-small across four datasets. The CodeT5-small model was fine-tuned individually on each dataset, after which Adacom was applied to evaluate its impact across all test datasets.

To investigate RQ3 (retrieval-methods comparison), we compared Adacom to Retro [76], a retrieval-augmented approach, and a technique that retrieves helpful examples using [CLS] tokens with cosine similarity. Unlike Retro, which retrieves and concatenates similar samples during training, Adacom focuses on retrieving helpful samples for each test case during the testing phase without altering the training stage.

For RQ4 (generalization), we followed the methodology in [199] and created a dataset with separate source and target splits. The source and target datasets featured distinct *types* of training and testing samples. We evaluated Adacom by applying it to the target testing dataset, identifying helpful samples in the target training dataset, and adapting the model using only those samples during on-the-fly training. Crucially, no fine-tuning was performed on the target domain; instead, the estimated influence and representations of the target training dataset were cached during the offline stage. This setup allowed us to evaluate Adacom 's ability to adapt to new domains by selectively retraining on helpful

samples rather than the entire dataset. We defined the dataset splits across three levels of granularity:

- Cross-Language Generalization: Adacom was applied to T5 [170] and Roberta models pre-trained exclusively on natural language corpora. These models (source training datasets) were used to generate code comments on the CodeKG dataset (target testing dataset). Adacom 's performance was evaluated by on-the-fly retraining using a small subset of helpful samples from the CodeKG training dataset.
- Cross-Programming Language Generalization: Adacom was used with a Roberta model trained on CSN-Java to generate comments for Python, PHP, Go, JavaScript, and Ruby code in the CSN dataset. The Roberta model was trained on CSN-Java's training data (source dataset), and Adacom 's performance was evaluated by identifying helpful subsets from the training datasets of other programming languages.
- Cross-Project Generalization: Using project-level information from CodeKG, we split the code corpus into D_{pj_1} and D_{pj_2}. The Roberta model trained on D_{pj_1} was evaluated on D_{pj_2}, with Adacom enhancing performance by leveraging helpful samples from D_{pj_2}.

For RQ5 (runtime overhead evaluation), we measured runtime performance during the RQ2 experiments. Two configurations were used: an RTX 3080 GPU on a Windows platform to simulate a developer's working environment and an A4000 GPU on Ubuntu for lab testing. We recorded the average improvement in smoothing BLEU-4 scores across all datasets and the total time required for testing. Cost performance was then calculated by determining the improvement in smoothing BLEU-4 per second.

To answer RQ6 (ablation study), we compared Adacom 's performance across three scenarios: (1) Using the comment from the most helpful sample as the prediction. (2) Employing a standalone model without adaptation to generate predictions. (3) Comparing CodeT5+[203] with CodeT5-small and CodeT5-base models equipped with Adacom.

6.5.4 Experiment Results

The following section shows the experiment results for each research question.

Table 6.6: Boosting performance of Adacom: Cross-model Evaluation

Model	Scale	Parameter	BLEU4			METEOR			ROUGE-L		
			before	after	bst (%)	before	after	bst (%)	before	after	bst (%)
codeT5	small	60M	34.89±0.26	49.05±0.38	40.6	43.74±0.00	56.73±0.08	29.7	50.98±0.00	61.23±0.09	20.1
CodeBERT		84M	40.83±0.31	48.84±0.72	19.6	48.84±0.32	57.07±0.50	16.9	54.82±0.28	60.40±0.47	10.2
RoBERTa	base	173M	44.73±0.08	48.71±0.45	8.9	52.67±0.17	57.23±0.60	8.7	57.78±0.15	60.28±0.59	4.3
CodeBERT		173M	44.30±0.34	48.35±0.43	9.1	52.35±0.38	56.74±0.26	8.4	57.71±0.39	59.81±0.26	3.6
Graph		173M	45.51±0.48	49.40±0.53	8.5	53.68±0.58	57.82±0.46	7.7	58.89±0.60	61.13±0.48	3.8
codeT5		223M	45.53±0.00	49.79±0.03	9.4	54.19±0.00	57.63±0.15	6.3	58.48±0.00	61.85±0.09	5.8
codeT5	large	738M	45.99±0.65	49.87±0.66	8.4	54.09±0.69	58.27±0.47	7.7	59.29±0.57	61.73±0.46	4.1

RQ1: Cross-model Evaluation Table 6.6 demonstrates that Adacom significantly enhances the performance of state-of-the-art comment generators with notable consistency. On average, Adacom achieves a 14.9% improvement in the smoothing BLEU-4 score, a 12.2% increase in the METEOR score, and a 7.4% boost in the ROUGE-L score. Interestingly, the impact of Adacom is more pronounced on smaller models compared to larger ones. This aligns with expectations, as smaller models tend to struggle more with conflicting subsets of code samples, making them more receptive to Adacom 's optimization. Moreover, Adacom exhibits consistent performance across experiments. The observed deviations fall within a narrow range: between 0 and 0.72 for BLEU-4, 0.08 and 0.5 for METEOR, and 0.09 and 0.59 for ROUGE-L. Adacom has improvement on all different language models. Before the Adacom, although CodeT5-large has the best performance, all the base-size models, including Roberta, Codebert-base, GraphCodebert, and CodeT5-base, have a similar performance, which is close to the best. The small models CodeBert-small and CodeT5-small clearly show much worse scores. However, after the Adacom, all the models' performance are increasing, especially the small-size models, which approach the larger-size models. Due to the much shorter training time(all epochs) and test time(per sample), the smaller models can gain much attention if they are equipped with Adacom.

RQ2: Cross-dataset Evaluation Table 6.7 highlights that Adacom performs effectively across various code datasets when applied to CodeT5-small, achieving an average improvement of 8.3% in BLEU-4, 5.0% in METEOR, and 3.2% in ROUGE-L. However, the results reveal some variability in performance across datasets. Adacom demonstrates significantly better enhancements on datasets such as CodeKG, Cosbench, and CSN-js. Upon analysis, we found that datasets like `CSN-python` contain fewer helpful training

Table 6.7: Boosting Performance of Adacom: Cross-Dataset Evaluation

Dataset	BLEU4			METEOR			ROUGE-L		
	before	after	bst (%)	before	after	bst (%)	before	after	bst (%)
CodeKG	34.89	49.05	40.6	43.74	56.73	29.7	50.98	61.23	20.1
Cosbench	29.22	31.31	7.15	35.86	37.07	3.37	37.23	37.48	0.67
FunCom	33.32	33.76	1.32	41.71	42.04	0.79	49.23	49.53	0.61
CSN-java	19.17	20.06	4.64	32.09	32.84	2.34	38.28	38.88	1.57
CSN-js	15.15	17.06	12.61	22.84	24.40	6.83	30.38	31.18	2.63
CSN-python	19.71	19.92	1.07	30.57	30.68	0.36	37.23	37.48	0.67
CSN-go	18.88	19.15	1.43	33.95	34.10	0.44	41.21	41.40	0.46
CSN-php	24.70	25.76	4.29	36.34	36.96	1.71	44.53	45.40	1.95
CSN-ruby	14.78	15.03	1.69	25.11	25.05	-0.24	31.77	31.94	0.54

Table 6.8: Performance Analysis: Retrieval Model vs. Semantic Embedding with Cosine Similarity on CodeT5-Base Model

Dataset	Retro 223M	CLS-Cosine 223M	Adacom 223M
CSN-java	20.05	20.14	20.85
CSN-js	16.15	17.35	18.81
CSN-python	19.67	19.58	20.46
CSN-go	19.46	19.12	19.61
CSN-php	24.91	26.45	26.90
CSN-ruby	14.91	14.45	15.39
overall	19.19	19.52	20.34

samples, which limits the potential for boosting model performance. Additionally, as our hyperparameter tuning primarily focuses on optimizing BLEU scores, the BLEU-4 improvements are more pronounced compared to those of METEOR and ROUGE-L. One practical mitigation strategy is to apply a stricter threshold for selecting helpful samples. A higher threshold can reduce the inclusion of irrelevant training samples, thereby improving the effectiveness of on-the-fly retraining.

RQ3: Retrieval-methods Comparison Table 6.8 illustrates that Adacom generally outperforms retrieval-augmented approaches. Traditional retrieval-based methods struggle to adapt the model to a specific test sample using only similar code-comment pairs. During training, these methods remain susceptible to biases introduced by contradictory samples, and simply appending similar examples to the input does not resolve these compromises. In contrast, Adacom excels by adapting the model during testing. It selectively re-learns from helpful samples while mitigating the influence of harmful ones, resulting in improved performance tailored to individual test cases.

Table 6.9: Cross-Domain Generalizability of Adacom: Language, Programming Language, and Project Evaluation

Type	Option	BLEU4			METEOR			ROUGE-L		
		before	after	bst (%)	before	after	bst (%)	before	after	bst (%)
Cross language	T5-base	10.44	41.97	302.01	24.91	57.27	129.91	18.99	50.72	167.09
	Roberta-base	6.42	37.51	484.27	10.24	42.85	318.46	11.45	38.77	238.60
Cross PL	CSN-JS	7.00	15.50	121.43	14.37	23.48	63.40	7.90	18.16	129.87
	CSN-Python	7.42	12.91	73.99	16.34	22.61	38.37	8.54	16.71	95.67
	CSN-Go	4.20	11.32	169.52	9.95	21.99	121.01	4.91	17.14	249.08
	CSN-PHP	7.56	16.71	121.03	16.39	28.61	74.56	8.66	22.38	158.43
	CSN-Ruby	7.71	9.95	29.05	16.18	20.18	24.72	8.63	13.69	58.63
Cross project	CodeKG	11.52	35.19	205.47	20.82	45.57	118.88	28.03	48.64	73.53

RQ4: Generalization Evaluation Table 6.9 highlights the strong generalizability of Adacom, demonstrating significant performance improvements across cross-language, cross-programming-language (PL), and cross-project scenarios.

For cross-language adaptation, Adacom leverages on-the-fly retraining to adjust the model using a small, helpful subset of the target samples. When adapting from natural language to programming language, models like T5 and Roberta—pre-trained exclusively on natural language—show marked performance gains, approaching the level of fine-tuned models. This indicates that language models can bypass the resource-intensive fine-tuning process by instead identifying a few relevant samples in the target dataset and applying Adacom for adaptation to specific test cases.

For cross-PL generalization, we evaluated Adacom on the challenging CodeSearchNet dataset. The results confirm its capability to enhance performance across different programming languages, reinforcing its adaptability in multilingual coding environments.

For cross-project scenarios, Adacom was tested on the diverse Java projects in the CodeKG dataset. It maintained competitive performance, particularly on out-of-distribution data, which includes code from previously unseen projects. Interestingly, performance within the same project often exceeded baseline results, likely because fewer projects reduced the compromises made by the large language model during training.

Overall, Adacom has proven effective in addressing distribution shifts across a variety of challenging scenarios, making it a versatile tool for improving model robustness.

RQ5: Runtime Overhead Evaluation To evaluate the time consumption of Adacom, we conducted an experiment using GraphCodeBERT on the CodeKG dataset, configuring retraining parameters to include 20 epochs with the top 5 most similar training samples.

Table 6.10: Efficiency Comparison: BLEU-4 Score Enhancement for Run-Time Overhead with Baselines

	Model Name	**CodeT5**	**CodeT5+**	**Adacom-small**	**Adacom-base**
	Parameter Size	223M	770M	60M	223M
Windows	Time (second)	1.21	4.06	2.34	4.81
	Time per sample (second)	0	2.85	1.13	3.60
	Average boost BLEU score	0	0.63	0.45	1.28
	Boost BLEU score per second	-	22.11%	39.82%	35.56%
Ubuntu	Time (second)	0.38	1.91	1.46	3.16
	Time per sample (second)	0	1.53	1.08	2.78
	Average boost BLEU score	0	0.63	0.45	1.28
	Boost BLEU score per second	-	41.18%	41.67%	46.04%

After extensive hyperparameter ablation, we selected the CodeT5-small and CodeT5-base models for further testing, setting the retraining parameters to 15 epochs, 3 helpful samples, a learning rate of 5×10^{-5}, and freezing the encoder along with half of the decoder.

Table 6.10 details the runtime overhead relative to the performance boost in smoothing BLEU-4 scores, using the CodeT5-base model as the baseline. The results show that Adacom delivers significant performance improvements with relatively modest computational effort. On the Windows platform, both Adacom-small and Adacom-base outperform CodeT5+. On the Ubuntu platform, Adacom-base demonstrates superior performance compared to CodeT5+. These findings suggest that Adacom is well-suited for practical applications, particularly as a cost-efficient solution for enhancing the performance of smaller models.

RQ6: Ablation study Table 6.11 highlights that relying solely on helpful comments or standalone models, as shown in the first column, results in reduced performance for Adacom. In contrast, the last two columns demonstrate how Adacom effectively integrates helpful samples and adapts the model, achieving improved performance. The third column presents the results for the standalone fine-tuned CodeT5-small model. Here, the prediction is made by directly using the comment from the most helpful sample with the highest contribution score. Notably, Adacom, when applied to CodeT5-small, outperforms even the standalone CodeT5+ large model, as shown in the fourth column.

6.6 Case Study

This section further demonstrates our comparison between Adacom and ChatGPT. ChatGPT is a popular deep language model that provides a practical and intuitive explana-

Table 6.11: Ablation Study on Adacom

	Retrieved Comment 223M	**Model Only** 223M	**Large Model** 770M	**Adacom-small** 60M	**Adacom-base** 223M
CSN-java	13.18	19.17	20.83	20.06	20.85
CSN-js	13.96	15.15	17.93	17.06	18.81
CSN-python	11.70	19.71	20.47	19.92	20.46
CSN-go	11.17	18.88	19.64	19.15	19.61
CSN-php	17.47	24.70	26.39	25.76	26.90
CSN-ruby	8.91	14.78	15.63	15.03	15.39
overall	12.73	18.73	20.15	19.50	20.34

tion for any code. Generally, ChatGPT has shown its powerful capability of generating great interpretation. Thus, we prepare 100 code examples to compare ChatGPT with Adacom and report the following findings.

Table 6.12: We compare our AdaCom with ChatGPT for code comment generation.

Sample	Test Sample1	Test Sample2	Test Sample3
Code Text	`public int getDataLength() { return dataLength; }`	`public static ClusterNode getClusterNode(String id) { if (id == null) {return null;} ClusterNode clusterNode = null; for (EntryType nodeType : EntryType.values()) { clusterNode = clusterNodeMap.get(new StringResourceWrapper(id, nodeType)); if (clusterNode != null) { break; }} return clusterNode;}`	`public int getRootOffset() { return parent.getRootOffset() + getParentOffset(); }`
Ground Truth	Returns the data length	Get @link ClusterNode of the resource name	Get the offset of this Data item from the start of some hierarchy of structures
Adacom	returns the data length	get @ link cluster node of the resource	get the offset of this data item from the start of some hierarchy of structures
ChatGPT	This code snippet is a getter method that returns the value of the "dataLength" field of the class.	This code snippet retrieves a ClusterNode object associated with a given id from a map of ClusterNode objects by iterating over the values of EntryType enum and using an object to access the appropriate key in the clusterNodeMap. If the id passed as a parameter is null, it will return null.	This code returns the offset of a node in the root of the tree by adding the offset of its parent to the offset of its parent in the root.

6.6.1 Abstract versus Detailed Explanation

In this case study, we find 71% of the ChatGPT-generated comments are “monologue”. Generally, when given a piece of code, ChatGPT intends to explain it line by line, even

when we ask it to answer with only one sentence. As shown in the first and second sample in Table 6.12 and all the samples in Table 6.13, ChatGPT explains the code in a very detailed way but in one sentence. ChatGPT usually translates the code line by line and follows a structure like "defines a method called X," "takes X and X as parameters," or "returns the X object." We deem this helpful for novice programmers while not necessary for a senior software engineer who may need to know the brief function of the code. Many researchers are working on a deep language model for *general* comment generation. Nevertheless, software engineering tasks (e.g., software debugging, repair, and testing) might require task-specific comment generation. Thus, we foresee that customized small- to medium-sized comment generators in the integrated development environment (IDE) could have an advantage over a general-purposed large model.

6.6.2 Explicit and Implicit Mistakes

However, after we manually compare 100 examples between ChatGPT and Adacom, we find 71% of the prediction of ChatGPT suffers the verbose problem while software engineers need the comment to be more abstract and concise. Moreover, we find that ChatGPT still makes mistakes. However, its latent mistakes are more inclined to hide with its eloquent appearance. In this study, we label 26% samples as plausible and misleading to the users. Table 6.14 and the third piece of code in Table 6.12 showcases four examples. The explanation is misleading compared to the ground-truth comment, especially for programmers with little expertise in the domain. In contrast, the mistakes made by the classical comment generator are easier to detect. We foresee that a differential testing technique for comment generators can be applied to compare with multiple generators, including classical comment generators and large language models such as ChatGPT, to help programmers in practice.

These weaknesses are also reported on the official website [153]. We show three examples in the 6.12. The first and second samples show that ChatGPT gives a correct but lengthy answer, and the longer the comment, the more complex the code. ChatGPT almost follows each code line and generates comments one by one but lacks global thinking of the whole code snippet. The third sample shows that ChatGPT sometimes gives a well-organized natural language description but incorrect comments. The comment of ChatGPT misuses the "root" information while the code gets the offset from the start of

some hierarchy of structures but maybe not the root. Overall, Adacom can give more concise and abstract comments with the helpful sample set to help users understand the prediction of the large language models.

Table 6.13: The predictions of ChatGPT are more verbose compared to AdaCom's and the ground truth.

Sample	Test Sample1	Test Sample2	Test Sample3
Code Text	`public boolean isEmpty() { return size == 0; }`	`public okhttp3.Call createUsersWithArrayInputAsync (List<User> body, final ApiCallback<Void> _callback) throws ApiException { okhttp3.Call localVarCall = createUsers(body, _callback); localVarApiClient. executeAsync(localVarCall, _callback); return localVarCall; }`	`public static String getDefaultHost( @Nullable String strInterface) throws UnknownException { return getDefaultHost (strInterface, null, false); }`
Ground Truth	Returns true if the array is empty.	Creates list of users with given input array (asynchronously).	Returns the default (first) host name associated by the default nameserver with the address bound to the specified network interface.
AdaCom	Returns true if the array is empty.	Creates list of users with the given input array asynchronously	Returns the default (first) host name associated by the provided nameserver with the address bound to the specified network interface
ChatGPT	This code snippet defines a method that returns a boolean indicating whether the collection is empty or not by checking if the size of the collection is equal to zero.	This code defines a public method called "createUsersWithArrayInputAsync" that takes a list of "User" objects and an "ApiCallback" object as parameters, validates the input before making an asynchronous call, and returns an okhttp3 Call object.	This code defines a method called "getDefaultHost" which takes a single parameter "strInterface" and returns the default host for that interface, potentially throwing an exception if the host is unknown.

6.7 Threats to Validity

Scalability Adacom requires additional time for model adaptation, particularly when dealing with billion-parameter models. To address this, freezing techniques are essential, specifically by freezing the encoder's parameters and selectively training only a subset of the decoder's parameters.

Construct Validity Adacom relies on the assumption that it can identify helpful examples in the training dataset using two defined metrics to improve model performance. However, certain test cases may lack any relevant or helpful examples, and these are excluded from consideration. Additionally, there is a possibility that the proposed method might inadvertently select harmful samples. This limitation emphasizes the need for

Table 6.14: The ChatGPT gives incorrect predictions compared to AdaCom's and the ground truth.

Sample	Test Sample1	Test Sample2	Test Sample3
Code Text	`public void stop() { if (this.started) { this.started = false; this.cumulate += System.nanoTime() - this.last; } }`	`public static _Fields findByThriftIdOrThrow (int fieldId) { _Fields fields = findByThriftId( fieldId); if (fields == null ) throw new IllegalArgument Exception("Field " + fieldId + " doesn' t exist!"); return fields; }`	`protected DatanodeInfo takeNodeOutofService(int nnIndex, String datanodeUuid, long maintenanceExpirationInMS, List<DatanodeInfo> decommissionedNodes, Map<DatanodeInfo, Long> inMaintenanceNodes, AdminStates waitForState) throws IOException { return takeNodeOutofService(nnIndex, (datanodeUuid != null ? Lists.newArrayList(datanodeUuid) : null), maintenanceExpirationInMS, decommissionedNodes, inMaintenanceNodes, waitForState).get(0); }`
Ground Truth	Stop the timer. Note: No effect if timer is already stopped.	Find the _Fields constant that matches fieldId, throwing an exception if it is not found.	Decommission or perform Maintenance for DataNodes and wait for them to reach the expected state.
AdaCom	Stop the timer	Find the fields that matches field id, throwing an exception if it is not found	Decommission or perform maintenance for data nodes and wait to reach the expected state
ChatGPT	This code is stopping a currently running process and updating the cumulative time spent running by adding the difference between the current time and the last time the process was started.	This code is a method for finding a field in a thrift object by its ID and throwing an exception if it is not found.	This code takes a node out of service by calling another method, "takeNodeOutofService," with the given arguments and returns the first element of the list returned by that method.

diverse and comprehensive training datasets that encompass a wide range of projects and scenarios.

Internal Validity We also examined scenarios where Adacom exhibits limited effectiveness. One issue arises from overfitting during retraining. As shown in Table 6.15, even when Adacom correctly identifies a useful reference c_{help}, it struggles to determine the optimal extent to which the comment generator should learn from that reference. For instance, in the example, Adacom retrains the deep comment generator for 15 epochs, but the optimal stopping point occurs at epoch 11. Stopping earlier would have yielded higher accuracy and reduced runtime.

To address this, Adacom could present candidates to users, enabling them to select and use helpful samples to better interpret the model's behavior. Future work will focus on refining the relationship between the performance metrics and the stopping criteria for

Table 6.15: Illustration of Potential Overfitting in Adacom

Sample	Target Test Sample	Harmful Training Sample	Helpful Training Sample
Code	`ReceiptViewModel purchase(...) { Db.User user = Db.getInstance().findUserByUserName(userName); if (user == null) { ... } Db.Account account = findAccount(user); return purchase(user, account, itemName); }`	`String receiveRequest(Object... parameters) throws DbUnavailableException { var id = generateId(); var req = new PaymentRequest(id, (float)parameters[0]); return updateDb(req); }`	`ReceiptViewModel purchase(...) { Db.Product item = Db.get().find(itemName); if (item == null) { ... } Receipt receipt = ...; if (transaction == null) { ... } return receipt; }`
Ground Truth	domain purchase with userName and itemName, with validation for userName	public method which will receive request from @link com. iluwatar. commander. Commander	domain purchase with user, account and itemName, with validation for whether product is out of stock and whether user has insufficient funds in the account
Origin	(BLEU 16.78) register purchase with user name		
Adacom Epoch8	(BLEU 66.43) domain purchase with user name and item name , with validation for whether user is enabled or not		
Adacom Epoch11	(BLEU 84.84) domain purchase with user name and item name , with validation for the account		
Adacom Epoch15	(BLEU 47.79) domain purchase with user , account and item name , with validation for whether user is enabled or not		

on-the-fly retraining, aiming to improve the process further.

Chapter 7

Conclusion and Future Work

This chapter concludes the thesis by summarizing its contributions and outlining directions for future research. Section 7.1 summarizes the main contributions of this thesis, and Section 7.2 discusses some ongoing and future directions.

7.1 Summary of the Thesis

This thesis includes program refinement, documentation, evolution, and real-time adaptation throughout the key phases of the software development life cycle. We built several tools and conducted comprehensive experiments to show their effectiveness.

First, we introduced LLM4PR, a system designed for the *automated* generation of *verified* code, integrating LLMs and formal methods like automated theorem provers. Our method transforms formal specifications into executable code through a process guided by refinement laws and enhanced interaction with LLMs. We have extended the formal refinement calculus to better accommodate the informal nature of LLMs by constructing active prompts that effectively guide the model. LLM4PR leverages the capabilities of GPT-4 to predict the applicable refinement laws intelligently and to assist in generating code suitable for formal verification. Following the code generation phase, LLM4PR employs ATPs to rigorously verify the refinement conditions and ensure that the generated code adheres to the specified preconditions and postconditions. Our experimental results validate that LLM4PR not only enhances the robustness of the generated code but also significantly improves correctness over existing state-of-the-art LLM-based approaches. This demonstrates LLM4PR's effectiveness in producing high-quality, verified software, marking a substantial advancement in the field of automated code generation.

Secondly, we propose CProSum, including a graph-structure context extractor, a structure-centric context evaluator, and a context-aware comment generator to enhance existing code summarization techniques regarding their code context. We transform the whole code project into a code knowledge graph and then use a context-evaluating and prompting framework adaptable to encoder-decoder summarization architectures. We further build a large graph dataset to facilitate the training and application of retrieved augmented generator-based solutions. Our extensive experiment shows that the graph contextual information is helpful for both selecting the related code examples and boosting the performance of comment generation. Our approach outperforms existing baselines by effectively utilizing structural and contextual information. Moreover, our experiment shows that all existing techniques can be further improved with a graph-based design.

Thirdly, we present CoEdPilot, a comprehensive end-to-end framework designed to interactively generate code edits by coordinating a series of neural transformers, each responsible for analyzing previous edits, subsequent edits, and generating new edits. Our extensive experimental results demonstrate that CoEdPilot excels in accurately predicting edit locations and generating viable edit options. Additionally, this framework enhances the capabilities of several leading-edge edit generators, significantly boosting their performance. A user study confirms that when integrated as a VS Code plugin, CoEdPilot effectively aids programmers in real-world coding tasks, streamlining the editing process and improving efficiency.

Finally, we presented Adacom, a novel solution of real-time performance enhancement for neural network models. We observed that models tend to compromise the performance of individual samples to improve the overall generalization ability, as the conflicting effect is universal across different training sets. To address this issue, Adacom is designed to further improve the model's performance on specific samples by utilizing the potential of the training set. We build the influence graph and estimate the contribution of training samples to generate a helpful set for each test sample. Once the model is deployed, we retrain it on the fly with the selected helpful training samples for each test sample and generate the code comment based on that new model. Extensive experiments have shown that Adacom can effectively improve the performance of the code comment generation on diverse datasets, programming languages, and different deep language models. Further, Adacom also shows its decent generalizability in cross-language, cross-programming-language, and cross-project settings.

7.2 On-going and Future Works

We are actively developing LLM with formal methods for trustworthy LLM-assistant tools. This section discusses some ongoing and future works surrounding the LLM and formal methods. The road map of the formal methods agents with LLM is published on the arxiv [231].

7.2.1 Tool Development

To bridge the gap between theoretical advancements and practical applications, developing and deploying our tools in real-world scenarios is a crucial part of our future work. We have collaborated with companies to deploy our tools as industrial products. We are committed to enhancing tools that leverage the latest innovations in LLMs and formal methods and are built more robustly and reliably for use in diverse environments. A key focus of our upcoming initiatives will be to address and mitigate the problem of over-fitting. This involves refining our methods to enhance the understanding and scientific control of the adaptation and fine-tuning process, ensuring that our models generalize well and remain effective under varied conditions. By doing so, we aim to provide tools that perform well on laboratory benchmark tests and deliver consistent and reliable performance in real-world applications.

7.2.2 Program Generation Techniques

Developing new program generation algorithms and refining related techniques remains our top priority. We have identified several promising directions to guide our future efforts:

Expansion of Refinement Laws and Program Structures Expanding the scope of refinement laws and program structures is a strategic priority to address more complex and diverse software engineering challenges. By deepening our engagement with formal methods and broadening the applicability of our tools, we strive to handle intricate questions and specifications that arise in industrial software projects. Enhancing these tools will facilitate the development of highly sophisticated, automated solutions that push the boundaries of current technology. We envision these formal method-based tools as becoming crucial in the software industry's evolution, delivering solutions that are not

only technically robust but also reliable and indispensable for developers across various domains. This advancement will solidify our contribution to setting new standards for accuracy and dependability in software development.

Development of a Larger Knowledge Graph The construction of an expanded knowledge graph represents a crucial step forward in our approach to software development. This enhanced knowledge graph will enable a deeper and more detailed analysis of software projects by mapping out a more extensive graph of code dependencies and interactions. Such an expansion will significantly improve our understanding of the nuanced relationships within codebases, facilitating more informed decision-making in the program generation process. The larger knowledge graph will boost the accuracy and effectiveness of our algorithms, enabling them to handle increasingly complex scenarios with greater precision. Through this advanced tool, we aim to revolutionize how developers interact with and manage large-scale software projects, ultimately leading to more robust and efficient software systems.

Innovation in Future Code Edits We are exploring new methodologies for predicting and implementing code edits. Incorporating real-time analytics into development tools can provide immediate insights into how code changes affect software performance and stability. By analyzing code behavior in real-time, developers can receive proactive suggestions for necessary edits before issues become problematic, effectively reducing debugging time and enhancing code quality. Drawing insights from version control systems can help predict future code edits by analyzing historical data on code changes. By understanding patterns in how software has evolved, predictive models can suggest future changes that align with long-term development trends or correct recurring issues.

Advancement in Adaptation Methods Our goal is to significantly advance the adaptation methods used within our tool sets so that they more effectively align with the dynamic and ever-evolving nature of software development projects. By meticulously refining these methods, we aim to equip our tools with the latest real-time adaptation techniques to seamlessly adjust to changes in project requirements and shifts in the technological landscape. This enhancement will ensure that the programs generated by our tools meet current needs and adapt proactively to future demands, maintaining their relevance and

effectiveness over time. This strategic improvement will enable developers to manage and anticipate changes more efficiently, thus fostering more resilient and adaptable software solutions.

7.2.3 LLM Agents

LLM Agents are advanced AI systems that utilize large language models to interpret and generate human language with a high degree of contextual understanding and sophistication. These agents maintain the continuity of conversations, recall past interactions, and dynamically adjust their responses in varying tones and styles to suit different situations. LLM agents' advanced capabilities make them highly effective for complex tasks such as problem-solving and engaging in nuanced dialogues. Consequently, they are increasingly utilized in diverse fields, including data analysis, education, and healthcare. Moreover, LLM Agents possess a degree of autonomy, enabling them to self-navigate within the scope of their programming. This autonomous capability allows them to effectively assist human users by streamlining productivity, alleviating mundane tasks, and tackling complex challenges. However, despite their sophisticated linguistic abilities, LLM Agents are vulnerable to misinformation, inherent biases, privacy breaches, and the potential propagation of harmful content. We are building the PAT [190] based LLM agent to remedy these weaknesses. PAT is a toolkit for flexible and efficient system analysis under fairness. It proposes a unified algorithm to effectively model check systems with a variety of fairness in different settings. We add another safety layer using PAT to recognize the potential for misuse and the hallucination of the LLM agents for code generation, mitigate the risk of spreading misinformation, and ensure responses are contextually appropriate and correct.

Through these directions, we seek to significantly enhance our capabilities in program generation, setting new benchmarks for innovation and reliability in software development tools. These efforts will address current challenges and pave the way for future advancements in the field.

Bibliography

[1] P. Aggarwal, A. Madaan, Y. Yang, and Mausam, "Let's sample step by step: Adaptive-consistency for efficient reasoning and coding with LLMs", in *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, H. Bouamor, J. Pino, and K. Bali, Eds., Singapore: Association for Computational Linguistics, Dec. 2023, pp. 12 375–12 396. [Online]. Available: https://aclanthology.org/2023.emnlp-main.761.

[2] W. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified pre-training for program understanding and generation", in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, Online: Association for Computational Linguistics, Jun. 2021, pp. 2655–2668. [Online]. Available: https://aclanthology.org/2021.naacl-main.211.

[3] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization", *ACL*, 2020.

[4] D. Alfageh, H. Alhakami, A. Baz, E. Alanazi, and T. Alsubait, "Clone detection techniques for javascript and language independence", *International Journal of Advanced Computer Science and Applications*, vol. 11, no. 4, 2020.

[5] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program", in *European Conference on Object-Oriented Programming*, Springer, 2013, pp. 378–400.

[6] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code", in *International conference on machine learning*, PMLR, 2016, pp. 2091–2100.

[7] U. Alon, S. Brody, O. Levy, and E. Yahav, "Code2seq: Generating sequences from structured representations of code", *ICLR*, 2019.

[8] J. Alpuim and W. Swierstra, "Embedding the refinement calculus in Coq", *Science of Computer Programming*, vol. 164, pp. 37–48, 2018, Special issue of selected papers from FLOPS 2016, ISSN: 0167-6423.

[9] C. Angiuli, E. Cavallo, K.-B. Hou (Favonia), R. Harper, and J. Sterling, "The RedPRL proof assistant (invited paper)", *Electronic Proceedings in Theoretical Computer Science*, vol. 274, pp. 1–10, Jul. 2018, ISSN: 2075-2180. [Online]. Available: http://dx.doi.org/10.4204/EPTCS.274.1.

[10] R.-J. J. Back, A. Akademi, J. V. Wright, F. B. Schneider, and D. Gries, "Refinement Calculus: A Systematic Introduction", 1st. Berlin, Heidelberg: Springer-Verlag, 1998, ISBN: 0387984178.

[11] R.-J. R. Back and J. von Wright, "Refinement concepts formalised in higher order logic", *Formal Aspects of Computing*, vol. 2, pp. 247–272, 1990.

[12] S. Banerjee and A. Lavie, "Meteor: An automatic metric for mt evaluation with improved correlation with human judgments", in *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, 2005, pp. 65–72.

[13] A. Bansal, S. Haque, and C. McMillan, "Project-level encoding for neural source code summarization of subroutines", in *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, IEEE, 2021, pp. 253–264.

[14] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, *et al.*, "The Coq proof assistant reference manual", *INRIA, version*, vol. 6, no. 11, 1999.

[15] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4", in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, Springer, 2011, pp. 171–177.

[16] L. Ben Allal, N. Muennighoff, L. Kumar Umapathi, B. Lipkin, and L. von Werra, "A framework for the evaluation of code generation models", https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.

[17] M. Besta, N. Blach, A. Kubicek, R. Gerstenberger, L. Gianinazzi, J. Gajda, T. Lehmann, M. Podstawski, H. Niewiadomski, P. Nyczyk, and T. Hoefler, "Graph of Thoughts: Solving Elaborate Problems with Large Language Models", *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 16, pp. 17 682–17 690, Mar. 2024. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/29720.

[18] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models", *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591300.

[19] L. Beurer-Kellner, M. Fischer, and M. Vechev, "Prompting is programming: A query language for large language models", *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, Jun. 2023. [Online]. Available: https://doi.org/10.1145/3591300.

[20] S. Böhme and T. Nipkow, "Sledgehammer: Judgement day", in *Automated Reasoning: 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings 5*, Springer, 2010, pp. 107–121.

[21] T. Bordis, T. Runge, A. Kittelmann, and I. Schaefer, "Correctness-by-construction: An overview of the corc ecosystem", *Ada Lett.*, vol. 42, no. 2, pp. 75–78, Apr. 2023, ISSN: 1094-3641. [Online]. Available: https://doi.org/10.1145/3591335.3591343.

[22] M. Bowers, T. X. Olausson, L. Wong, G. Grand, J. B. Tenenbaum, K. Ellis, and A. Solar-Lezama, "Top-down synthesis for library learning", *Proc. ACM Program. Lang.*, vol. 7, no. POPL, Jan. 2023. [Online]. Available: https://doi.org/10.1145/3571234.

[23] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners", *Advances in neural information processing systems*, vol. 33, pp. 1877–1901, 2020.

[24] M. Brunsfeld, P. Thomson, A. Hlynskyi, J. Vera, P. Turnbull, T. Clem, D. Creager, A. Helwer, R. Rix, H. van Antwerpen, M. Davis, Ika, T.-A. Nguyen, S. Brunk, N. Hasabnis, bfredl, M. Dong, V. Panteleev, ikrima, S. Kalt, K. Lampe, A. Pinkus, M. Schmitz, M. Krupcale, narpfel, S. Gallegos, V. Martí, Edgar, and G. Fraser, "Tree-sitter/tree-sitter: V0.20.7", version v0.20.7, Sep. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.7045041.

[25] N. D. Bui, Y. Yu, and L. Jiang, "Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations", in *Proceedings of the 44th International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2021, pp. 511–521.

[26] M. Butler and T. Långbacka, "Program derivation using the refinement calculator", in *Theorem Proving in Higher Order Logics*, G. Goos, J. Hartmanis, J. van Leeuwen, J. von Wright, J. Grundy, and J. Harrison, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 93–108, ISBN: 978-3-540-70641-0.

[27] Y. Cai, Y. Lin, C. Liu, J. Wu, Y. Zhang, Y. Liu, Y. Gong, and J. S. Dong, "On-the-fly adapting code summarization on trainable cost-effective language models", in *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., vol. 36, Curran Associates, Inc., 2023, pp. 56 660–56 672. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2023/file/b16e6de5fbbdcb2df237aa66b302bc17-Paper-Conference.pdf.

[28] D. Carrington, I. Hayes, R. Nickson, G. Watson, and J. Welsh, "A program refinement tool", *Formal Aspects of Computing*, vol. 10, pp. 97–124, 1998.

[29] S. Chakraborty and B. Ray, "On multi-modal learning of editing source code", in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Los Alamitos, CA, USA: IEEE Computer Society, 2021, pp. 443–455. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE51524.2021.9678559.

[30] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray, "Codit: Code editing with tree-based neural models", *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1385–1399, 2022.

[31] Y. Charalambous, N. Tihanyi, R. Jain, Y. Sun, M. A. Ferrag, and L. C. Cordeiro, "A new era in software security: Towards self-healing software via large language models and formal verification", 2023. arXiv: 2305.14752 [cs.SE].

[32] B. Chen, F. Zhang, A. Nguyen, D. Zan, Z. Lin, J.-G. Lou, and W. Chen, "Codet: Code generation with generated tests", *arXiv preprint arXiv:2207.10397*, 2022.

[33] H. Chen, Y. Wang, K. Zheng, W. Li, C.-T. Chang, A. P. Harrison, J. Xiao, G. D. Hager, L. Lu, C.-H. Liao, *et al.*, "Anatomy-aware siamese network: Exploiting semantic asymmetry for accurate pelvic fracture detection in x-ray images", in *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXIII 16*, Springer, 2020, pp. 239–255.

[34] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code", 2021. arXiv: 2107.03374 [cs.LG].

[35] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation", 2014. arXiv: 1406.1078 [cs.CL].

[36] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches", *arXiv preprint arXiv:1409.1259*, 2014.

[37] K. Claessen, R. Hähnle, and J. Mårtensson, "Verification of hardware systems with first-order logic", in *Proceedings of the CADE-18 Workshop-Problem and Problem Sets for ATP*, Citeseer, 2002.

[38] C. B. Clement, D. Drain, J. Timcheck, A. Svyatkovskiy, and N. Sundaresan, "Pymt5: Multi-mode translation of natural language and python code with transformers", *arXiv preprint arXiv:2010.03150*, 2020.

[39] CodeKG, "Codekg", 2022. [Online]. Available: https://sites.google.com/view/code-kg/home.

[40] "Coedpilot website", https://sites.google.com/view/coedpilot/home, 2024.

[41] "Crossentropyloss — PyTorch 2.1 documentation", https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html, 2023.

[42] Q. Cutts, R. Connor, G. Michaelson, and P. Donaldson, "Code or (not code) separating formal and natural language in cs education", in *Proceedings of the 9th Workshop in Primary and Secondary Computing Education*, 2014, pp. 20–28.

[43] Ł. Czajka and C. Kaliszyk, "Hammer for Coq: Automation for dependent type theory", *Journal of automated reasoning*, vol. 61, pp. 423–453, 2018.

[44] H. Daumé III and E. Brill, "Web search intent induction via automatic query reformulation", in *Proceedings of HLT-NAACL 2004: Short Papers*, 2004, pp. 49–52.

[45] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver", in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, pp. 337–340.

[46] B. Delaware, C. Pit-Claudel, J. Gross, and A. Chlipala, "Fiat: Deductive synthesis of abstract data types in a proof assistant", in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '15, Mumbai, India: ACM, 2015, pp. 689–700, ISBN: 978-1-4503-3300-9. [Online]. Available: http://doi.acm.org/10.1145/2676726.2677006.

[47] D. Detlefs, G. Nelson, and J. B. Saxe, "Simplify: A theorem prover for program checking", *J. ACM*, vol. 52, no. 3, pp. 365–473, May 2005, ISSN: 0004-5411. [Online]. Available: https://doi.org/10.1145/1066100.1066102.

[48] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding",, J. Burstein, C. Doran, and T. Solorio, Eds., pp. 4171–4186, Jun. 2019. [Online]. Available: https://aclanthology.org/N19-1423.

[49] E. W. Dijkstra, E. W. Dijkstra, E. W. Dijkstra, and E. W. Dijkstra, "A discipline of programming", prentice-hall Englewood Cliffs, 1976, vol. 613924118.

[50] H. Ding, V. Kumar, Y. Tian, Z. Wang, R. Kwiatkowski, X. Li, M. K. Ramanathan, B. Ray, P. Bhatia, S. Sengupta, D. Roth, and B. Xiang, "A static evaluation of code completion by large language models", 2023. arXiv: 2306.03203 [cs.CL].

[51] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "Decaf: A deep convolutional activation feature for generic visual recognition", in *Proceedings of the 31st International Conference on Machine Learning*, E. P. Xing and T. Jebara, Eds., ser. Proceedings of Machine Learning Research, vol. 32, Bejing, China: PMLR, 2014, pp. 647–655. [Online]. Available: https://proceedings.mlr.press/v32/donahue14.html.

[52] I. Dragomir, V. Preoteasa, and S. Tripakis, "The refinement calculus of reactive systems toolset", *International Journal on Software Tools for Technology Transfer*, vol. 22, pp. 689–708, 2020.

[53] S. Duncan, A. Walker, C. DeHaan, S. Alvord, T. Cerny, and P. Tisnovsky, "Pyclone: A python code clone test bank generator", in *Information Science and Applications: Proceedings of ICISA 2020*, Springer, 2021, pp. 235–243.

[54] K. Ellis, C. Wong, M. Nye, M. Sablé-Meyer, L. Morales, L. Hewitt, L. Cary, A. Solar-Lezama, and J. B. Tenenbaum, "Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning",, ser. PLDI 2021, Virtual, Canada: Association for Computing Machinery, 2021, pp. 835–850, ISBN: 9781450383912. [Online]. Available: https://doi.org/10.1145/3453483.3454080.

[55] J. L. Elman, "Finding structure in time", *Cognitive science*, vol. 14, no. 2, pp. 179–211, 1990.

[56] Y. Elrakaiby, A. Borgida, A. Ferrari, and J. Mylopoulos, "CaRE: A refinement calculus for requirements engineering based on argumentation theory", *Software and Systems Modeling*, vol. 21, no. 6, pp. 2113–2132, 2022.

[57] 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", *EMNLP*, 2020.

[58] P. Fernandes, M. Allamanis, and M. Brockschmidt, "Structured neural summarization", *ICLR*, 2019.

[59] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization", *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 9, no. 3, pp. 319–349, 1987.

[60] E. First, M. Rabe, T. Ringer, and Y. Brun, "Baldur: Whole-proof generation and repair with large language models", in *The 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2023, pp. 1229–1241.

[61] R. W. Floyd, "Assigning meanings to programs", in *Program Verification: Fundamental Issues in Computer Science*, Springer, 1993, pp. 65–81.

[62] S. Foster, J. J. Huerta y Munive, and G. Struth, "Differential Hoare logics and refinement calculi for hybrid systems with Isabelle/HOL", in *Relational and Algebraic Methods in Computer Science: 18th International Conference, RAMiCS 2020, Palaiseau, France, October 26–29, 2020, Proceedings 18*, Springer, 2020, pp. 169–186.

[63] M. Fowler, "Refactoring: improving the design of existing code", Addison-Wesley Professional, 2018.

[64] M. Freitag and Y. Al-Onaizan, "Beam search strategies for neural machine translation", in *Proceedings of the First Workshop on Neural Machine Translation*, Vancouver: Association for Computational Linguistics, Aug. 2017, pp. 56–60. [Online]. Available: `https://aclanthology.org/W17-3207`.

[65] D. Fried, A. Aghajanyan, J. Lin, S. Wang, E. Wallace, F. Shi, R. Zhong, S. Yih, L. Zettlemoyer, and M. Lewis, "Incoder: A generative model for code infilling and synthesis", in *The Eleventh International Conference on Learning Representations*, 2023.

[66] Y. Ganin and V. Lempitsky, "Unsupervised domain adaptation by backpropagation", in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15, Lille, France: JMLR.org, 2015, pp. 1180–1189.

[67] H. Ganzinger, F. Pfenning, and C. Schürmann, "System description: Twelf—a meta-logical framework for deductive systems", in *Automated Deduction—CADE-16: 16th International Conference on Automated Deduction Trento, Italy, July 7–10, 1999 Proceedings 16*, Springer, 1999, pp. 202–206.

[68] 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*, ser. ICSE '24, <conf-loc>, <city>Lisbon</city>, <country>Portugal</country>, </conf-loc>: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3608134.

[69] 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*, 2024, pp. 1–13.

[70] GitHub, "GitHub Copilot", 2023. [Online]. Available: https://github.com/features/copilot.

[71] R. Gozalo-Brizuela and E. C. Garrido-Merchan, "Chatgpt is not all you need. a state of the art review of large generative ai models", *arXiv preprint arXiv:2301.04655*, 2023.

[72] X. Gu, H. Zhang, and S. Kim, "Deep code search", in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 933–944.

[73] D. Guo, S. Lu, N. Duan, Y. Wang, M. Zhou, and J. Yin, "Unixcoder: Unified cross-modal pre-training for code representation", *arXiv preprint arXiv:2203.03850*, 2022.

[74] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, *et al.*, "Graphcodebert: Pre-training code representations with data flow", *The International Conference on Learning Representations*, 2020.

[75] P. Gupta, A. Khare, Y. Bajpai, S. Chakraborty, S. Gulwani, A. Kanade, A. Radhakrishna, G. Soares, and A. Tiwari, "Grace: Language models meet code edits", in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2023, New York, NY, USA: Association for Computing Machinery, 2023, pp. 1483–1495. [Online]. Available: https://doi.org/10.1145/3611643.3616253.

[76] V. Gupta, A. Shrivastava, A. Sagar, A. Aghajanyan, and D. Savenkov, "RetroNLU: Retrieval augmented task-oriented semantic parsing", in *Proceedings of the 4th Workshop on NLP for Conversational AI*, Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 184–196. [Online]. Available: https://aclanthology.org/2022.nlp4convai-1.15.

[77] R. Hähnle, "Quo vadis formal verification?" In *Deductive Software Verification – The KeY Book: From Theory to Practice*, W. Ahrendt, B. Beckert, R. Bubel, R. Hähnle, P. H. Schmitt, and M. Ulbrich, Eds. Cham: Springer International Publishing, 2016, pp. 1–19, ISBN: 978-3-319-49812-6. [Online]. Available: https://doi.org/10.1007/978-3-319-49812-6_1.

[78] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code", in *2010 17th Working Conference on Reverse Engineering*, IEEE, 2010, pp. 35–44.

[79] J. M. Han, J. Rute, Y. Wu, E. W. Ayers, and S. Polu, "Proof artifact co-training for theorem proving with language models", *arXiv preprint arXiv:2102.06203*, 2021.

[80] X. Han, Z. Zhang, N. Ding, Y. Gu, X. Liu, Y. Huo, J. Qiu, Y. Yao, A. Zhang, L. Zhang, *et al.*, "Pre-trained models: Past, present and future", *AI Open*, vol. 2, pp. 225–250, 2021.

[81] S. Haque, A. LeClair, L. Wu, and C. McMillan, "Improved automatic summarization of subroutines via attention to file context", in *Proceedings of the 17th International Conference on Mining Software Repositories*, 2020, pp. 300–310.

[82] C. A. R. Hoare, "An axiomatic basis for computer programming", *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[83] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[84] S. Hochreiter and J. Schmidhuber, "Long short-term memory", *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[85] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation", in *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, IEEE, 2018, pp. 200–20 010.

[86] P. Huang, H. Wu, Y. Yang, I. Daukantas, M. Wu, Y. Zhang, and C. Barrett, "Towards efficient verification of quantized neural networks", *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 38, no. 19, pp. 21 152–21 160, Mar. 2024. [Online]. Available: `https://ojs.aaai.org/index.php/AAAI/article/view/30108`.

[87] Y. Huang, M. Wei, S. Wang, J. Wang, and Q. Wang, "Yet another combination of ir-and neural-based comment generation", *Information and Software Technology*, vol. 152, p. 107 001, 2022.

[88] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search", *arXiv preprint arXiv:1909.09436*, 2019.

[89] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model", in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2016, pp. 2073–2083.

[90] S. Jain and B. C. Wallace, "Attention is not explanation", *arXiv preprint arXiv:1902.10186*, 2019.

[91] S. Jha, S. K. Jha, P. Lincoln, N. D. Bastian, A. Velasquez, and S. Neema, "Dehallucinating large language models using formal methods guided iterative prompting", in *2023 IEEE International Conference on Assured Autonomy (ICAA)*, 2023, pp. 149–152.

[92] A. Q. Jiang, W. Li, S. Tworkowski, K. Czechowski, T. Odrzygóźdź, P. Miłoś, Y. Wu, and M. Jamnik, "Thor: Wielding hammers to integrate language models and automated theorem provers", *Advances in Neural Information Processing Systems*, vol. 35, pp. 8360–8373, 2022.

[93] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair", in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1161–1173.

[94] Z. Jiang, J. Araki, H. Ding, and G. Neubig, "How can we know when language models know? on the calibration of language models for question answering", *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 962–977, 2021.

[95] W. Jin, Y. Cai, R. Kazman, Q. Zheng, D. Cui, and T. Liu, "Enre: A tool framework for extensible entity relation extraction", in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 67–70.

[96] W. Jin, D. Zhong, Y. Cai, R. Kazman, and T. Liu, "Evaluating the impact of possible dependencies on architecture-level maintainability", *IEEE Transactions on Software Engineering*, pp. 1–1, 2022.

[97] S. Kabir, D. N. Udo-Imeh, B. Kou, and T. Zhang, "Is stack overflow obsolete? an empirical study of the characteristics of chatgpt answers to stack overflow questions", in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. CHI '24, Honolulu, HI, USA: Association for Computing Machinery, 2024, ISBN: 9798400703300. [Online]. Available: https://doi.org/10.1145/3613904.3642596.

[98] Ł. Kaiser, O. Nachum, A. Roy, and S. Bengio, "Learning to remember rare events", *arXiv preprint arXiv:1703.03129*, 2017.

[99] M. Kaufmann and J. S. Moore, "An ACL2 tutorial", in *International Conference on Theorem Proving in Higher Order Logics*, Springer, 2008, pp. 17–21.

[100] I. Keivanloo, F. Zhang, and Y. Zou, "Threshold-free code clone detection for a large-scale heterogeneous java repository", in *2015 IEEE 22nd International*

Conference on Software Analysis, Evolution, and Reengineering (SANER), IEEE, 2015, pp. 201–210.

[101] R. Kitchin and M. Dodge, "Code/space: Software and everyday life", Mit Press, 2014.

[102] M. Kodetzki, T. Bordis, T. Runge, and I. Schaefer, "Partial proofs to optimize deductive verification of feature-oriented software product lines", in *Proceedings of the 18th International Working Conference on Variability Modelling of Software-Intensive Systems*, ser. VaMoS '24, Bern, Switzerland: Association for Computing Machinery, 2024, pp. 17–26, ISBN: 9798400708770. [Online]. Available: https://doi.org/10.1145/3634713.3634714.

[103] M. A. Köhl, "An executable structural operational formal semantics for Python", 2021. arXiv: 2109.03139 [cs.PL].

[104] L. Kovács and A. Voronkov, "First-order theorem proving and vampire", in *International Conference on Computer Aided Verification*, Springer, 2013, pp. 1–35.

[105] G. Lample and A. Conneau, "Cross-lingual language model pretraining", 2019. arXiv: 1901.07291 [cs.CL].

[106] G. Lample, T. Lacroix, M.-A. Lachaux, A. Rodriguez, A. Hayat, T. Lavril, G. Ebner, and X. Martinet, "Hypertree proof search for neural theorem proving", *Advances in Neural Information Processing Systems*, vol. 35, pp. 26 337–26 349, 2022.

[107] T. D. LaToza and B. A. Myers, "Developers ask reachability questions", in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, 2010, pp. 185–194.

[108] A. LeClair, S. Haque, L. Wu, and C. McMillan, "Improved code summarization via a graph neural network", in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 184–195.

[109] A. LeClair, S. Jiang, and C. McMillan, "A neural model for generating natural language summaries of program subroutines", in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, pp. 795–806.

[110] A. LeClair and C. McMillan, "Recommendations for datasets for source code summarization", in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, J. Burstein, C. Doran, and T. Solorio, Eds., Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 3931–3937. [Online]. Available: https://aclanthology.org/N19-1394.

[111] T. Lecomte, "Atelier b", *Formal Methods Applied to Complex Systems: Implementation of the B Method*, pp. 35–46, 2014.

[112] J. Li, G. Li, Z. Li, Z. Jin, X. Hu, K. Zhang, and Z. Fu, "Codeeditor: Learning to edit source code with pre-trained models", *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 6, 2023, ISSN: 1049-331X. [Online]. Available: https://doi.org/10.1145/3597207.

[113] J. Li, Y. Li, G. Li, X. Hu, X. Xia, and Z. Jin, "Editsum: A retrieve-and-edit framework for source code summarization", in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2021, pp. 155–166.

[114] 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.

[115] Z. Li, Y. Wu, B. Peng, X. Chen, Z. Sun, Y. Liu, and D. Yu, "Secnn: A semantic cnn parser for code comment generation", *Journal of Systems and Software*, vol. 181, p. 111 036, 2021, ISSN: 0164-1212. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001333.

[116] Z. Li, S. Lu, D. Guo, N. Duan, S. Jannu, G. Jenks, D. Majumder, J. Green, A. Svyatkovskiy, S. Fu, and N. Sundaresan, "Automating code review activities by large-scale pre-training", in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2022, New York, NY, USA: Association for Computing Machinery, 2022, pp. 1035–1047, ISBN: 9781450394130. [Online]. Available: https://doi.org/10.1145/3540250.3549081.

[117] B. Y. Lin, W. Zhou, M. Shen, P. Zhou, C. Bhagavatula, Y. Choi, and X. Ren, "Commongen: A constrained text generation challenge for generative commonsense reasoning", *arXiv preprint arXiv:1911.03705*, 2019.

[118] B. Lin, S. Wang, Z. Liu, Y. Liu, X. Xia, and X. Mao, "Cct5: A code-change-oriented pre-trained model", *arXiv preprint arXiv:2305.10785*, 2023.

[119] C.-Y. Lin, "ROUGE: A package for automatic evaluation of summaries", in *Text Summarization Branches Out*, Barcelona, Spain: Association for Computational Linguistics, Jul. 2004, pp. 74–81. [Online]. Available: https://aclanthology.org/W04-1013.

[120] C.-Y. Lin and F. J. Och, "ORANGE: A method for evaluating automatic evaluation metrics for machine translation", in *COLING 2004: Proceedings of the 20th International Conference on Computational Linguistics*, Geneva, Switzerland: COLING, 2004, pp. 501–507. [Online]. Available: https://aclanthology.org/C04-1072.

[121] Y. Lin, X. Peng, Z. Xing, D. Zheng, and W. Zhao, "Clone-based and interactive recommendation for modifying pasted code", in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 520–531.

[122] W. Ling, P. Blunsom, E. Grefenstette, K. M. Hermann, T. Kočiský, F. Wang, and A. Senior, "Latent predictor networks for code generation", in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 599–609. [Online]. Available: https://aclanthology.org/P16-1057.

[123] C. Liu, Y. Cai, Y. Lin, Y. Huang, Y. Pei, B. Jiang, P. Yang, J. S. Dong, and H. Mei, "Coedpilot: Recommending code edits with learned prior edit relevance, project-wise awareness, and interactive nature",, 2024.

[124] 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", in *Thirty-seventh Conference on Neural Information Processing Systems*, 2023. [Online]. Available: https://openreview.net/forum?id=1qvx610Cu7.

[125] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing", 2021. arXiv: 2107.13586 [cs.CL].

[126] S. Liu, Y. Chen, X. Xie, J. Siow, and Y. Liu, "Retrieval-augmented generation for code summarization via hybrid gnn", *arXiv preprint arXiv:2006.05405*, 2020.

[127] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach", 2019. arXiv: 1907.11692 [cs.CL].

[128] Y. Liu, Z.-Y. Dou, and P. Liu, "Refsum: Refactoring neural summarization", *arXiv preprint arXiv:2104.07210*, 2021.

[129] Z. Liu, X. Xia, M. Yan, and S. Li, "Automating just-in-time comment updating", in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 585–597.

[130] C. Lopes, S. Bajracharya, J. Ossher, and P. Baldi, "UCI source code data sets", 2010. [Online]. Available: http://www.ics.uci.edu/$%5Csim$lopes/datasets/.

[131] J. Mahmud, F. Faisal, R. I. Arnob, A. Anastasopoulos, and K. Moran, "Code to comment translation: A comparative study on model effectiveness & errors", in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, Online: Association for Computational Linguistics, Aug. 2021, pp. 1–16. [Online]. Available: https://aclanthology.org/2021.nlp4prog-1.1.

[132] D. Maier, "The complexity of some problems on subsequences and supersequences", *Journal of the ACM (JACM)*, vol. 25, no. 2, pp. 322–336, 1978.

[133] A. Martino, M. Iannelli, and C. Truong, "Knowledge injection to counter large language model (llm) hallucination", in *The Semantic Web: ESWC 2023 Satellite Events*, C. Pesquita, H. Skaf-Molli, V. Efthymiou, S. Kirrane, A. Ngonga, D. Collarana, R. Cerqueira, M. Alam, C. Trojahn, and S. Hertling, Eds., Cham: Springer Nature Switzerland, 2023, pp. 182–185, ISBN: 978-3-031-43458-7.

[134] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context", in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014, pp. 279–290.

[135] N. Megill and D. A. Wheeler, "Metamath: a computer language for mathematical proofs", Lulu. com, 2019.

[136] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space", 2013. arXiv: 1301.3781 [cs.CL]. [Online]. Available: https://arxiv.org/abs/1301.3781.

[137] M. J. Min, Y. Ding, L. Buratti, S. Pujar, G. Kaiser, S. Jana, and B. Ray, "Beyond accuracy: Evaluating self-consistency of code large language models with identitychain", in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=caW7LdAALh.

[138] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for java classes", in *2013 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 23–32.

[139] C. Morgan, K. Robinson, and P. Gardiner, "On the Refinement Calculus", Techllical Monograph, 1988, ISBN: 0-902928-52-X. [Online]. Available: https://web.comlab.ox.ac.uk/files/3391/PRG70.pdf.

[140] C. Morgan, "Programming from Specifications", USA: Prentice-Hall, Inc., 1990, ISBN: 0137262256.

[141] L. de Moura, S. Kong, J. Avigad, F. Van Doorn, and J. von Raumer, "The lean theorem prover (system description)", in *Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25*, Springer, 2015, pp. 378–388.

[142] D. Movshovitz-Attias and W. Cohen, "Natural language models for predicting programming comments", in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, 2013, pp. 35–40.

[143] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming", *arXiv preprint arXiv:2210.14306*, 2022.

[144] F. Mu, X. Chen, L. Shi, S. Wang, and Q. Wang, "Automatic comment generation via multi-pass deliberation", in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[145] N. Nashid, M. Sintaha, and A. Mesbah, "Retrieval-based prompt selection for code-related few-shot learning", in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23, Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 2450–2462, ISBN: 9781665457019. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00205.

[146] M. S. Nawaz, M. Malik, Y. Li, M. Sun, and M. I. U. Lali, "A survey on theorem provers in formal methods", 2019. arXiv: 1912.03028 [cs.SE].

[147] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution", in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2013, pp. 180–190.

[148] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions", in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5.

[149] U. Norell, "Dependently typed programming in Agda", in *Proceedings of the 4th international workshop on Types in language design and implementation*, 2009, pp. 1–2.

[150] M. Norrish, "C formalised in HOL", University of Cambridge, Computer Laboratory, Tech. Rep., 1998.

[151] OpenAI, "Codex", https://openai.com/blog/openai-codex, 2020.

[152] OpenAI, "Gpt3.5", https://platform.openai.com/docs/models/gpt-3-5, 2020.

[153] OpenAI, "OpenAI GPT-3 Model", 2020. [Online]. Available: https://openai.com/gpt-3/.

[154] OpenAI, "Chatgpt", https://openai.com/chatgpt, Accessed on March 29, 2023, 2021.

[155] OpenAI and co., "GPT-4 technical report", 2023. arXiv: 2303.08774 [cs.CL].

[156] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, *et al.*, "Training language models to follow instructions with human feedback", *arXiv preprint arXiv:2203.02155*, 2022.

[157] S. Owre, J. Rushby, N. Shankar, and F. von Henke, "Formal verification for fault-tolerant architectures: Prolegomena to the design of PVS", *IEEE Transactions on Software Engineering*, vol. 21, no. 2, pp. 107–125, 1995.

[158] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[159] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation", in *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, P. Isabelle, E. Charniak, and D. Lin, Eds., Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, Jul. 2002, pp. 311–318. [Online]. Available: https://aclanthology.org/P02-1040.

[160] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "Bleu: A method for automatic evaluation of machine translation", in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.

[161] M. R. Parvez, W. U. Ahmad, S. Chakraborty, B. Ray, and K. Chang, "Retrieval augmented code generation and summarization", *CoRR*, vol. abs/2108.11601, 2021. arXiv: 2108.11601. [Online]. Available: https://arxiv.org/abs/2108.11601.

[162] L. C. Paulson, "Isabelle: A generic theorem prover", Springer, 1994.

[163] J. Pennington, R. Socher, and C. Manning, "GloVe: Global vectors for word representation", in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, A. Moschitti, B. Pang, and W. Daelemans, Eds., Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1532–1543. [Online]. Available: https://aclanthology.org/D14-1162.

[164] L. Phan, H. Tran, D. Le, H. Nguyen, J. Anibal, A. Peltekian, and Y. Ye, "Co-text: Multi-task learning with code-text transformer", 2021. [Online]. Available: https://arxiv.org/abs/2105.08645.

[165] N. Polikarpova, I. Kuraj, and A. Solar-Lezama, "Program synthesis from polymorphic refinement types", *SIGPLAN Not.*, vol. 51, no. 6, pp. 522–538, Jun. 2016, ISSN: 0362-1340. [Online]. Available: http://doi.acm.org/10.1145/2980983.2908093.

[166] S. Polu, J. M. Han, K. Zheng, M. Baksys, I. Babuschkin, and I. Sutskever, "Formal mathematics statement curriculum learning", *arXiv preprint arXiv:2202.01344*, 2022.

[167] S. Polu and I. Sutskever, "Generative language modeling for automated theorem proving", *arXiv preprint arXiv:2009.03393*, 2020.

[168] E. M. Ponti, G. Glavaš, O. Majewska, Q. Liu, I. Vulić, and A. Korhonen, "Xcopa: A multilingual dataset for causal commonsense reasoning", *arXiv preprint arXiv:2005.00333*, 2020.

[169] M. Püschel, J. M. F. Moura, J. R. Johnson, D. A. Padua, M. M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: code generation for DSP transforms", *Proceedings of the IEEE*, vol. 93, no. 2, pp. 232–275, 2005. [Online]. Available: https://doi.org/10.1109/JPROC.2004.840306.

[170] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, P. J. Liu, *et al.*, "Exploring the limits of transfer learning with a unified text-to-text transformer." *J. Mach. Learn. Res.*, vol. 21, no. 140, pp. 1–67, 2020.

[171] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer", *Journal of Machine Learning Research*, vol. 21, no. 140, pp. 1–67, 2020. [Online]. Available: http://jmlr.org/papers/v21/20-074.html.

[172] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer",, vol. 21, no. 1, 2020, ISSN: 1532-4435.

[173] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks", in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, Nov. 2019. [Online]. Available: https://arxiv.org/abs/1908.10084.

[174] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, *et al.*, "Mathematical discoveries from program search with large language models", *Nature*, pp. 1–3, 2023.

[175] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. C. Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve, "Code llama: Open foundation models for code", 2023. arXiv: 2308.12950 [cs.CL].

[176] P. Rudnicki, "An overview of the Mizar project", in *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 1992, pp. 311–330.

[177] T. Runge, I. Schaefer, L. Cleophas, T. Thüm, D. Kourie, and B. W. Watson, "Tool support for correctness-by-construction", in *Fundamental Approaches to Software Engineering*, R. Hähnle and W. van der Aalst, Eds., Cham: Springer International Publishing, 2019, pp. 25–42.

[178] C. Sadowski, E. Söderberg, L. Church, M. Sipko, and A. Bacchelli, "Modern code review: A case study at google", in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, 2018, pp. 181–190.

[179] J. Salazar, D. Liang, T. Q. Nguyen, and K. Kirchhoff, "Masked language model scoring", *arXiv preprint arXiv:1910.14659*, 2019.

[180] B. D. Sall, F. Peschanski, and E. Chailloux, "A mechanized theory of program refinement", in *Formal Methods and Software Engineering: 21st International Conference on Formal Engineering Methods, ICFEM 2019, Shenzhen, China, November 5–9, 2019, Proceedings*, Shenzhen, China: Springer-Verlag, 2019, pp. 305–321, ISBN: 978-3-030-32408-7. [Online]. Available: https://doi.org/10.1007/978-3-030-32409-4_19.

[181] T. Schick and H. Schütze, "Few-shot text generation with natural language instructions", in *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 2021, pp. 390–402.

[182] T. Schick, S. Udupa, and H. Schütze, "Self-diagnosis and self-debiasing: A proposal for reducing corpus-based bias in nlp", *Transactions of the Association for Computational Linguistics*, vol. 9, pp. 1408–1424, 2021.

[183] S. Schulz, "E - a brainiac theorem prover", *AI Commun.*, vol. 15, no. 2,3, pp. 111–126, 2002, ISSN: 0921-7126.

[184] R. Sharma, F. Chen, and F. Fard, "Lamner: Code comment generation using character language model and named entity recognition", 2022. arXiv: 2204.09654 [cs.CL].

[185] H. Shiina, A. Takahashi, R. Ito, and N. Kobayashi, "Comment generation system for program procedure learning", in *2018 7th International Congress on Advanced Applied Informatics (IIAI-AAI)*, IEEE, 2018, pp. 38–42.

[186] J. Shin and J. Nam, "A survey of automatic code generation from natural language", *Journal of Information Processing Systems*, vol. 17, no. 3, pp. 537–555, 2021.

[187] A. Solar-Lezama, C. G. Jones, and R. Bodík, "Sketching concurrent data structures", in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 2008, pp. 136–148. [Online]. Available: http://doi.acm.org/10.1145/1375581.1375599.

[188] S. Srivastava, S. Gulwani, and J. S. Foster, "From program verification to program synthesis", in *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, 2010, pp. 313–326. [Online]. Available: http://doi.acm.org/10.1145/1706299.1706337.

[189] J. Su, J. Cao, W. Liu, and Y. Ou, "Whitening sentence representations for better semantics and faster retrieval", *arXiv preprint arXiv:2103.15316*, 2021.

[190] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "Pat: Towards flexible verification under fairness", in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 709–714, ISBN: 978-3-642-02658-4.

[191] Y. Sun, X. Wang, Z. Liu, J. Miller, A. Efros, and M. Hardt, "Test-time training with self-supervision for generalization under distribution shifts", in *Proceedings of the 37th International Conference on Machine Learning*, H. D. III and A. Singh, Eds., ser. Proceedings of Machine Learning Research, vol. 119, PMLR, 2020, pp. 9229–9248. [Online]. Available: https://proceedings.mlr.press/v119/sun20b.html.

[192] Z. Sun, Q. Zhu, Y. Xiong, Y. Sun, L. Mou, and L. Zhang, "Treegen: A tree-based transformer architecture for code generation", in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, 2020, pp. 8984–8991.

[193] W. Swierstra and J. Alpuim, "From proposition to program: Embedding the refinement calculus in Coq", in *Functional and Logic Programming: 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings 13*, Springer, 2016, pp. 29–44.

[194] S. Team, "Spring framework", 2022. [Online]. Available: https://spring.io/projects/spring-framework.

[195] E. Ufuktepe, T. Tuglular, and K. Palaniappan, "Tracking code bug fix ripple effects based on change patterns using markov chain models", *IEEE Transactions on Reliability*, vol. 71, no. 2, pp. 1141–1156, 2022.

[196] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models", in *Chi conference on human factors in computing systems extended abstracts*, 2022, pp. 1–7.

[197] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework", in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[198] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need", *Advances in neural information processing systems*, vol. 30, 2017.

[199] D. Wang, E. Shelhamer, S. Liu, B. Olshausen, and T. Darrell, "Tent: Fully test-time adaptation by entropy minimization", in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=uXl3bZLkr3c.

[200] H. Wang, Y. Yuan, Z. Liu, J. Shen, Y. Yin, J. Xiong, E. Xie, H. Shi, Y. Li, L. Li, *et al.*, "Dt-solver: Automated theorem proving with dynamic-tree sampling guided by proof-level value function", in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2023, pp. 12 632–12 646.

[201] Q. Wang, C. Kaliszyk, and J. Urban, "First experiments with neural translation of informal to formal mathematics", in *Intelligent Computer Mathematics: 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings 11*, Springer, 2018, pp. 255–270.

[202] Y. Wang, E. Shi, L. Du, X. Yang, Y. Hu, S. Han, H. Zhang, and D. Zhang, "Cocosum: Contextual code summarization with multi-relational graph neural network", 2021. [Online]. Available: https://arxiv.org/abs/2107.01933.

[203] Y. Wang, H. Le, A. D. Gotmare, N. D. Bui, J. Li, and S. C. Hoi, "Codet5+: Open code large language models for code understanding and generation", *arXiv preprint arXiv:2305.07922*, 2023.

[204] 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.

[205] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization", *NeurIPS*, 2019.

[206] B. Wei, Y. Li, G. Li, X. Xia, and Z. Jin, "Retrieve and refine: Exemplar-based neural comment generation", in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2020, pp. 349–360.

[207] 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*, vol. 35, pp. 24 824–24 837, 2022.

[208] E. Wong, T. Liu, and L. Tan, "Clocom: Mining existing source code for automatic comment generation", in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, IEEE, 2015, pp. 380–389.

[209] E. Wong, J. Yang, and L. Tan, "Autocomment: Mining question and answer sites for automatic comment generation", in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2013, pp. 562–567.

[210] Y. Wu, A. Q. Jiang, W. Li, M. Rabe, C. Staats, M. Jamnik, and C. Szegedy, "Autoformalization with large language models", *Advances in Neural Information Processing Systems*, vol. 35, pp. 32 353–32 368, 2022.

[211] H. Xin, H. Wang, C. Zheng, L. Li, Z. Liu, Q. Cao, Y. Huang, J. Xiong, H. Shi, E. Xie, *et al.*, "LEGO-prover: Neural theorem proving with growing libraries", *arXiv preprint arXiv:2310.00656*, 2023.

[212] Z. Xu, S. Jain, and M. Kankanhalli, "Hallucination is inevitable: An innate limitation of large language models", 2024. arXiv: 2401.11817 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2401.11817.

[213] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries", in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, IEEE, 2020, pp. 344–354.

[214] C. Yang, Y. Liu, and C. Yin, "Recent advances in intelligent source code generation: A survey on natural language based studies", *Entropy*, vol. 23, no. 9, p. 1174, 2021.

[215] G. Yang, X. Chen, J. Cao, S. Xu, Z. Cui, C. Yu, and K. Liu, "Comformer: Code comment generation via transformer and fusion method-based hybrid code representation", 2021. [Online]. Available: https://arxiv.org/abs/2107.03644.

[216] G. Yang, K. Liu, X. Chen, Y. Zhou, C. Yu, and H. Lin, "Ccgir: Information retrieval-based code comment generation method for smart contracts", *Knowledge-Based Systems*, vol. 237, p. 107 858, 2022.

[217] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar, "LeanDojo: Theorem proving with retrieval-augmented language models", *arXiv preprint arXiv:2306.15626*, 2023.

[218] S. Yao, D. Yu, J. Zhao, I. Shafran, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models", English (US), *Advances in Neural Information Processing Systems*, vol. 36, 2023, Publisher Copyright: © 2023 Neural information processing systems foundation. All rights reserved.; 37th Conference on Neural Information Processing Systems, NeurIPS 2023 ; Conference date: 10-12-2023 Through 16-12-2023, ISSN: 1049-5258.

[219] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning", in *Proceedings of The Web Conference 2020*, 2020, pp. 2309–2319.

[220] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14, Montreal, Canada: MIT Press, 2014, pp. 3320–3328.

[221] C. Yu, G. Yang, X. Chen, K. Liu, and Y. Zhou, "Bashexplainer: Retrieval-augmented bash code comment generation based on fine-tuned codebert", *arXiv preprint arXiv:2206.13325*, 2022.

[222] H. Yuchao, W. Moshi, W. Song, W. Junjie, and W. Qing, "Yet another combination of ir- and neural-based comment generation", 2021. [Online]. Available: `https://arxiv.org/abs/2107.12938`.

[223] B. Zhang, B. Haddow, and A. Birch, "Prompting large language model for machine translation: A case study", *arXiv preprint arXiv:2301.07069*, 2023.

[224] C. Zhang, Q. Zhou, M. Qiao, K. Tang, L. Xu, and F. Liu, "Re_trans: Combined retrieval and transformer model for source code summarization", *Entropy*, vol. 24, no. 10, p. 1372, 2022.

[225] J. Zhang, X. Wang, H. Zhang, H. Sun, and X. Liu, "Retrieval-based neural source code summarization", in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, IEEE, 2020, pp. 1385–1397.

[226] J. Zhang, S. Panthaplackel, P. Nie, J. J. Li, and M. Gligoric, "CoditT5: Pretraining for source code and natural language editing", in *International Conference on Automated Software Engineering*, 2022.

[227] J. Zhang, S. Panthaplackel, P. Nie, R. J. Mooney, J. J. Li, and M. Gligoric, "Learning to generate code comments from class hierarchies", 2021. [Online]. Available: https://arxiv.org/abs/2103.13426.

[228] R. Zhang, A. Saran, B. Liu, Y. Zhu, S. Guo, S. Niekum, D. Ballard, and M. Hayhoe, "Human gaze assisted artificial intelligence: A review", in *IJCAI: Proceedings of the Conference*, NIH Public Access, vol. 2020, 2020, p. 4951.

[229] S. Zhang, L. Dong, X. Li, S. Zhang, X. Sun, S. Wang, J. Li, R. Hu, T. Zhang, F. Wu, and G. Wang, "Instruction tuning for large language models: A survey", 2024. arXiv: 2308.10792 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2308.10792.

[230] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: A new data clustering algorithm and its applications", *Data mining and knowledge discovery*, vol. 1, pp. 141–182, 1997.

[231] Y. Zhang, Y. Cai, X. Zuo, X. Luan, K. Wang, Z. Hou, Y. Zhang, Z. Wei, M. Sun, J. Sun, J. Sun, and J. S. Dong, "The fusion of large language models and formal methods for trustworthy ai agents: A roadmap", 2024. arXiv: 2412.06512 [cs.AI]. [Online]. Available: https://arxiv.org/abs/2412.06512.

[232] Y. Zhang, J. Yang, Y. Yuan, and A. C.-C. Yao, "Cumulative reasoning with large language models", *arXiv preprint arXiv:2308.04371*, 2023.

[233] Y. Zhang, Y. Bajpai, P. Gupta, A. Ketkar, M. Allamanis, T. Barik, S. Gulwani, A. Radhakrishna, M. Raza, G. Soares, and A. Tiwari, "Overwatch: Learning patterns in code edit sequences", *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, 2022. [Online]. Available: https://doi.org/10.1145/3563302.

[234] R. Zhao, X. Li, S. Joty, C. Qin, and L. Bing, "Verify-and-edit: A knowledge-enhanced chain-of-thought framework", in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds., Toronto, Canada: Association for Computational Linguistics, Jul. 2023, pp. 5823–5840. [Online]. Available: https://aclanthology.org/2023.acl-long.320.

[235] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models", 2023. arXiv: 2303.18223 [cs.CL].

[236] X. Zhao, W. Li, and L. Kong, "Decomposing the Enigma: Subgoal-based demonstration learning for formal theorem proving", *arXiv preprint arXiv:2305.16366*, 2023.

[237] Y. Zhou, X. Zhang, J. Shen, T. Han, T. Chen, and H. Gall, "Adversarial robustness of deep code comment generation", 2021. [Online]. Available: https://arxiv.org/abs/2108.00213.

[238] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair", in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021, Athens, Greece: Association for Computing Machinery, 2021, pp. 341–353, ISBN: 9781450385626. [Online]. Available: https://doi.org/10.1145/3468264.3468544.