The University of Tokyo
Graduate School of Information Science and Technology
Department of Creative Informatics

博士論文

A Study on Solving Software Engineering Problems by Selectively Applying LLMs through Two Practical Cases

(LLMの選択的適用によるソフトウェア工学上の課題の解決：
2つの実践的事例を通じて)

Doctoral Dissertation of:
Feng DAI
(戴 峰)

Academic Advisor:
Shigeru Chiba
(千葉 滋)

Abstract

This dissertation presents our study on effectively using large language models (LLMs) in software engineering tasks. Currently, the use of LLMs in software engineering remains ad-hoc and is constrained by factors such as input length limitations and limited capabilities of reasoning. We propose to selectively apply LLMs to software engineering tasks by decomposing the original task into smaller subtasks, enabling LLMs to process one or a few of them more effectively. Specifically, we leverage LLMs as tools for processing raw data or analyzing intermediate data. In this dissertation, we examine two representative tasks through empirical studies to assess the effectiveness of our proposed methodology. The tasks are: (1) detection of structurally-similar cross-language code pairs among many candidates, and (2) retrieval of the causes of software regressions in Python and JavaScript projects.

In study (1), we present a fast and reliable tool for detecting structurally-similar cross-language code pairs by adopting a two-stage approach. Our approach utilizes LLMs as a tool for processing raw data and to generate more informative intermediate representations. Our approach first uses models trained on two-level generic abstract syntax trees (ASTs) to filter candidates, and further uses tree-based editing-distance algorithm for accurate comparison. Specifically, we adopt LLMs when designing two kinds of generic ASTs to avoid human bias. To evaluate our tool, we create a new cross-language dataset, including Java-Python, Java-C, Python-C structurally-similar method body pairs. Our results show that LLMs generate satisfying designs of two kinds of generic ASTs, and we manage to obtain a much faster speed and similar detection accuracy in detecting structurally-similar cross-language code pairs compared with the state-of-the-art technique.

In study (2), we provide a tool for locating the root causes of software regressions, addressing the limitations of existing tools, such as low detection accuracy. Our tool improves the accuracy of locating causes of software regressions by proposing three new techniques. In this tool, LLMs are utilized as pre-trained models for analyzing processed intermediate data. Our tech-

niques are based on tracing source code changes during program execution. To evaluate our techniques, we extend the current benchmark of software regressions to include two programming languages, Python and JavaScript, and compare our techniques against existing techniques on the extended benchmark. Specifically, our LLM-powered technique achieves the state-of-the-art accuracy in retrieving causes of software regressions.

Both studies presented in this dissertation provide evidence that our proposal, which selectively applies LLMs to software engineering problems, is effective in addressing tasks that are otherwise challenging for LLMs to handle directly. Our approach may serve as a foundation or valuable reference for future research in this area.

Acknowledgements

During this journey, I owe a great debt of gratitude to many people.

First, I would like to express my sincere gratitude to Prof. Chiba, who has supported me throughout my entire time in this country. Our discussions have always been inspiring and helpful. He has been not only my supervisor during my postgraduate studies, but also a mentor who will continuously influence me in the future.

Second, I would like to express my gratitude to all my defense committee members: Prof. Esaki, Prof. Nakayama, Prof. Ma, Prof. Shioya and Prof. Ugawa. All of their suggestions and criticisms are insightful and thoughtful. They help me make this thesis polished and complete. I would also like to thank Dr. Yamazaki, Senxi Li, Yilin Zhang and Yuefeng Hu for their help during this research. In addition, I would like to thank everyone who has made advice throughout the writing of this thesis.

Third, I would like to express my gratitude to all my friends and lab members, especially Prof. Akiyama, Prof. Nakamaru, Prof. Tung D. Ta, Xuyang Yao, Wennong Cai and Fumika Maejima. Their help has been indispensable to me during my studies. I may not have mentioned everyone by name, but I hold deep appreciation for everyone who has supported me throughout this journey.

Last but not least, I want to thank my parents. They have never questioned my decisions and have always supported me with no hesitation.

谢谢你们, 爸爸妈妈。

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Software engineering in the era of LLMs

Enabling software developers to efficiently create and maintain high-quality software has been a long-standing and challenging research problem. Software engineering as a discipline is dedicated to addressing these challenges. It relates to the entire life cycle of software, including design, development, testing, and maintenance, each involving a wide range of tasks, techniques, considerations and trade-offs. As a discipline, software engineering has a very long history and numerous methodologies and practices have been proposed over time, enriching the discipline.

Traditionally, software engineering practices have often included heuristic techniques and experiential knowledge, especially in areas lacking formalized procedures. For example, when designing test cases, although test frameworks can automate the execution of tests, designing high-coverage and efficient test cases still heavily relies on the experience of test engineers. For another example from the area of code clone detection, traditional approaches [4] rely on comparing source code or abstract syntax trees using heuristic-based algorithms. These heuristic-based algorithms are often tailored to specific scenarios, such as particular clone types, and are difficult to extend to other contexts.

Briefly, traditional approaches in software engineering, which depend on heuristic techniques and experiential knowledge, although valuable in practice, often rely heavily on the intuition and expertise of individual developers or teams. Such reliance can lead to inconsistencies in implementation quality, difficulties in knowledge transfer, and limited scalability across different projects. Moreover, heuristic methods may not always generalize well

to novel or complex scenarios, especially when dealing with fast-changing environments, such as the context of rapidly evolving software engineering practices. These limitations have motivated researchers to explore more automated and data-driven techniques in software engineering. New techniques include the use of machine learning and, more recently, large language models (LLMs).

In the past decade, as machine learning techniques have gained popularity in many research domains, software engineering has also begun to adopt these techniques. Learning-based approaches typically rely on the collection of large-scale datasets, from which machine learning algorithms or neural network models can extract underlying patterns to address specific tasks or problems. For example, researchers collect datasets consisting of faulty code snippets and manually annotate them with bug types to create a constructed dataset for studying automated program repair techniques [71, 12]. Following data collection, the data must be preprocessed, and a suitable model should be selected to learn from the data. Naturally, such learning-based workflows consist of several stages: acquiring and cleaning relevant datasets, performing feature engineering, selecting and fine-tuning appropriate models, and accurately formulating the task to ensure effective learning. In the area of software engineering, extensive studies have been conducted focusing on each stage of this workflow and have contributed a lot to the automation and efficiency of software engineering.

Among many learning-based techniques, large language models, emerging as a more recent advancement, have attracted significant attention in the past few years in the research of software engineering. Large language models, abbreviated as LLMs, represent a major technological breakthrough enabled by the combination of increased computational power, advances in machine learning techniques, and access to massive-scale datasets. This technology has already evolved into a foundational element in many fields, such as natural language processing and multi-modal image generation. In the context of software engineering, LLMs have also demonstrated remarkable capabilities in code generation, documentation, and even automated software maintenance.

According to a recent survey [23], there has been a significant increase in the number of studies focusing on the utilization of large language models for software engineering (LLMs for SE). The number of papers on this topic (applying LLMs to software engineering) published in major software engineering venues has increased rapidly in recent years: 13 in 2021, 56 in 2022, 273 in 2023, and over 300 in 2024. Following the surge in 2023, research activity has remained at a consistently high level.

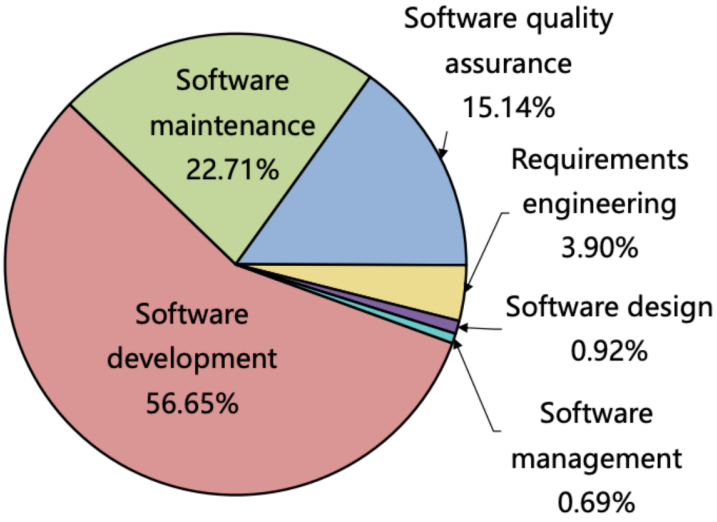


Figure 1.1: The distribution of papers in SE using LLMs

Research on LLMs for software engineering spans all stages of the software engineering lifecycle. In Figure 1.1, it can be seen that researchers have explored the use of LLMs to improve performance across the entire software engineering lifecycle [23]. Among these studies, 56.65% of the studies focus on software development, primarily involving code generation and code completion. Research on software maintenance accounts for 22.71% of the total, primarily focusing on tasks such as code review, clone detection, and several other related activities. Research on software quality assurance accounts for 15.14% of the total, mainly focusing on tasks such as test generation, bug localization, and test automation. In addition to the three major stages of software engineering, there are also studies targeting minor stages such as requirements engineering, software design, and software management. In summary, LLMs have been widely applied across various aspects of software engineering activities. Compared with traditional machine learning approaches, one major advantage of LLMs is that they have already been pre-trained on massive corpora and therefore can be adapted to a variety of downstream tasks with little supervision.

However, this does not necessarily imply that LLMs can be directly and effectively applied to all software engineering problems. In many cases, their performance may still fall short due to the complexity of the task, the structure of the input data, or the need for multi-step reasoning and domain-

specific knowledge. For example, in the software development stage, code generation remains an important and challenging problem. While significant progress has been made in generating code for major programming languages such as Java and Python, code generation for less common or domain-specific languages, such as the Sparrow language [1], remains insufficiently addressed by current LLMs. Although Sparrow is designed as a general-purpose programming language, it currently lacks a large user base and sufficient high-quality code available for training. As a result, when attempting to generate code in Sparrow, LLMs often produce incorrect syntax by confusing it with constructs from more widely-used languages. Such phenomenon is regarded as LLM hallucination. This highlights a key limitation of LLMs in handling low-resource or underrepresented languages.

Another example arises in the software maintenance stage, where code clone analysis plays a critical role. While code clone detection, which is typically formulated as a binary classification problem, has been extensively studied and shows promising results, code search, which involves ranking a set of candidate code fragments based on their relevance to a given query, remains a more challenging and less well-solved problem. The inherent complexity in ranking problems pose difficulties for current LLM-based solutions. A similar situation is observed in the software quality assurance stage. While bug localization within a single file is relatively well-handled by current LLMs, identifying the root causes of software regressions, which represent a specific and often more complex type of software bugs, remains a difficult challenge. Software regressions typically involve complex interactions across multiple files and revisions, which are hard for LLMs to reason about. Overall, LLMs are not a silver bullet when it comes to software engineering.

In this sense, exploring how to effectively and efficiently apply large language models to software engineering problems remains a largely unexplored area. Our research aims to explore more effective ways of applying large language models to software engineering problems, especially for tasks where large language models cannot be directly applied due to various reasons, such as input limits of large language models.

1.2 Contributions

The primary contribution of this dissertation lies in proposing an approach to effectively apply LLMs to software engineering problems that are otherwise difficult to address using LLMs directly. Our main idea is to decompose

[1] https://github.com/Sparrow-lang/sparrow

the original task into smaller sub-tasks and selectively apply LLMs to sub-tasks. Specifically, we decompose the task following the traditional pipeline and consider stages of raw data processing and the application of appropriate algorithms to intermediate representations. We use LLMs as tools to process raw data or analyze intermediate data to avoid their limitations and make the problems easier to solve for them. By our approach, we can improve the performance of individual sub-tasks using LLMs, thereby enhancing the overall performance of the original task. In this dissertation, we select two practical and representative tasks and conduct empirical studies to demonstrate the effectiveness of our idea.

Specifically, our first task is to retrieve structurally-similar cross-language code pair from a number of candidates. The number of candidates can be as large as several hundreds. Finding such code pair is important in software maintenance. Traditionally, comparing a pair of code snippets requires identifying appropriate intermediate representations and applying appropriate algorithms. By using our proposal, we decompose the solution into raw data processing and intermediate data analysis in this task, following the traditional pipeline. Importantly, we utilize LLMs as a tool for helping process raw data and generate informative intermediate data representations. In detail, we develop a two-stage approach for detecting cross-language code pairs with similar control structures. Our approach largely improves the speed compared with the state-of-the-art technique [39], while keeps very close accuracy. In the first stage, our approach exploits neural-network models to pre-determine a few most-likely candidates. In the second stage, it calculates tree-based editing-distances deterministically to select the most similar code fragment in control structures. To promote the accuracy of the first stage, we develop a new code representation technique called two-level generic AST representation for neural-network models. We leverage LLMs to design generic ASTs between different programming languages to avoid human bias. We discover that constructing two different kinds of generic ASTs and giving them to a neural-network model can significantly increase the possibility that the true pair is within the pre-determined candidates, and thus improve final accuracy. We also find that LLMs can assist in designing unbiased generic ASTs without relying on ad-hoc design choices. The contributions of our work to this task are two-fold:

- We improve the speed of detecting pairs of structurally-similar method bodies written in different programming languages by developing a two-stage approach. A unique feature of our approach is the use of two different kinds of generic ASTs as inputs to neural network mod-

els. To avoid ad-hoc design choices in constructing these ASTs, we leverage LLMs to generate them in a more unbiased manner. Our approach for detecting structurally-similar cross-language code pairs in Java, Python and C pairwise achieves comparable accuracy with the state-of-the-art technique, but a much faster speed.

- We create a code-pair dataset for evaluating a tool for detecting cross-language code pairs with similar control structures. The dataset includes similar Java-Python, Java-C and Python-C code pairs. To the best of our knowledge, we are the first to create a dataset specially targeting at structurally-similar code pairs.

Our second task is to locate the root cause of software regressions. Software regressions are a type of software bugs in which previously functioning software exhibits some faulty behaviour after an update. Finding the cause of software regressions is important in software quality assurance. Traditionally, locating the root cause of software regressions requires analyzing the complex relationships between source files, both statically and dynamically, and heavily relies on human instinct. By using our proposed approach, we delegate the analysis of complex relationships to LLMs by providing them with intermediate data that is relatively easy to obtain. In other words, we utilize LLMs to analyze pre-processed intermediate data in order to obtain more accurate and reliable results. We extend the existing work [24] of this task to address some previously unsolved limitations such as low accuracy and limited coverage of programming languages. To do so, we dynamically trace program executions between the base program and the faulty program, and utilize only the order of function executions instead of input and output information. We analyze the changes of source code in function execution traces instead of input and output changes. We provide three new techniques to improve the retrieval accuracy of the root causes of software regressions, including two heuristic algorithms and one LLM-powered technique. Specifically, our LLM-powered technique achieves the state-of-the-art accuracy compared with other existing algorithms. We also extend current benchmark and support another programming language, Python. To do so, we extract 12 real-world Python regressions from the BugsInPy [70] benchmark, and implement a Python-based tracer both for our algorithms and for algorithms introduced by the previous work [24]. Along with 11 JavaScript regressions introduced in the previous work [24], we conduct a detailed analysis about locating the cause of software regressions for both Python and JavaScript projects. Our contributions to this task are also two-fold:

- We improve the accuracy of existing approaches by providing three new techniques. We validate our techniques on both Python and JavaScript benchmark. Among these techniques, the one that is empowered by a large language model achieves the best accuracy.

- We create a new dataset containing software regressions written in Python. This is aimed for an extensive evaluation for tools that try to detect the cause of software regressions. All regressions in the dataset are based on open-source software projects and are suitable for real-world analysis.

Through two practical example tasks, we empirically demonstrate the effectiveness of our proposed approach, which selectively applies LLMs to software engineering problems. Our approach successfully broadens the range of tasks that can be addressed by leveraging LLMs, even when applied only partially. Our research provides a foundation for future work on effectively applying LLMs to software engineering problems, such as AI agents, by demonstrating a principled approach to task decomposition and selective LLM application.

1.3 Organization

In this section, we present the overall organization of this dissertation. The structure is summarized in Table 1.1. Specifically, the dissertation incorporates content from several previously published works. The correspondence between these publications and the respective chapters is also outlined in the table.

In chapter 2, we present the background knowledge leveraged in this dissertation. This includes fundamental concepts essential for understanding the work, as well as key techniques used in the detailed implementation. In chapter 3, we present the motivation of this dissertation. We aim to solve software engineering problems that are difficult to address using LLMs directly. In chapter 4, we give our proposal for addressing software engineering problems that are difficult to solve when using LLMs directly. Specifically, our proposal is to selectively apply LLMs to certain software engineering sub-tasks in order to improve the overall performance of the original task.

In chapter 5 and chapter 6, we analyze the first and second exemplified task: structurally-similar cross-language code search and retrieval of causes of software regressions. In each chapter, we begin by presenting the motivation for addressing the problem and discussing the corresponding sig-

Table 1.1: Organization and corresponding publications

Chapter	Topic	Publications
Chapter 1	Introduction	–
Chapter 2	Preliminary knowledge	–
Chapter 3	Motivation of the dissertation	–
Chapter 4	Our proposal: selectively apply LLMs	–
Chapter 5	Structurally-similar cross-language code search	SAC'24, SAC'25
Chapter 6	Retrieval of causes of software regressions	QRS'25
Chapter 7	Conclusions	–

nificance. We then introduce our proposal, explaining the system design and technical implementation. Finally, we present the experimental results, including comparisons against baseline methods. In chapter 7, we present our conclusions and summarize the key contributions of this dissertation. Additionally, we discuss future directions and highlight potential areas for further investigation.

Chapter 2

Preliminaries

In this chapter, we will introduce some background knowledge of our study and position it within the broader context of research on software engineering. The background knowledge discussed is not merely introductory but provides essential context and conceptual foundations that will be leveraged in addressing the specific tasks mentioned before and presented later in this dissertation.

2.1 Machine learning techniques

Machine learning is a field concerned with the development and study of statistical algorithms that can learn from data and generalize to unseen data, and thus perform tasks without explicit instructions. Traditional machine learning algorithms include decision trees, supporting vector machines, and Bayesian methods. Beyond traditional machine learning, deep learning has emerged as a major direction in machine learning in recent years. Deep learning, as a subdiscipline in machine learning, focuses on neural networks and surpasses many previous machine learning approaches in performance. A neural network consists of connected units or nodes called artificial neurons, which loosely model the neurons in the brain. These are connected by edges, which model the synapses in the brain. Each artificial neuron receives signals from connected neurons, then processes them and sends a signal to other connected neurons. The signal is a real number, and the output of each neuron is computed by some non-linear function of the sum of its inputs, called the activation function. The strength of the signal at each connection is determined by a weight, which adjusts during the learning process.

There are different types of neural networks, such as convolutional neural

networks (CNNs) and recurrent neural networks (RNNs). One typical and famous kind of neural network is long short-term memory (LSTM), which is a type of recurrent neural network (RNN). The structure of an LSTM [31] is shown in Figure 2.1. An LSTM model performs well in processing sequential data and capturing long-term dependencies by maintaining a hidden state that captures information from previous time steps. The architecture of LSTM involves a special module called the memory cell, including multiple artificial neurons, and is controlled by three gates: the input gate, the forget gate and the output gate. These gates decide what information to add to, remove from and output from the memory cell. In specific, the input gate controls what information is added to the memory cell; the forget gate determines what information is removed from the memory cell; the output gate controls what information is output from the memory cell. This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated by using the current input, the previous hidden state and the current state of the memory cell.

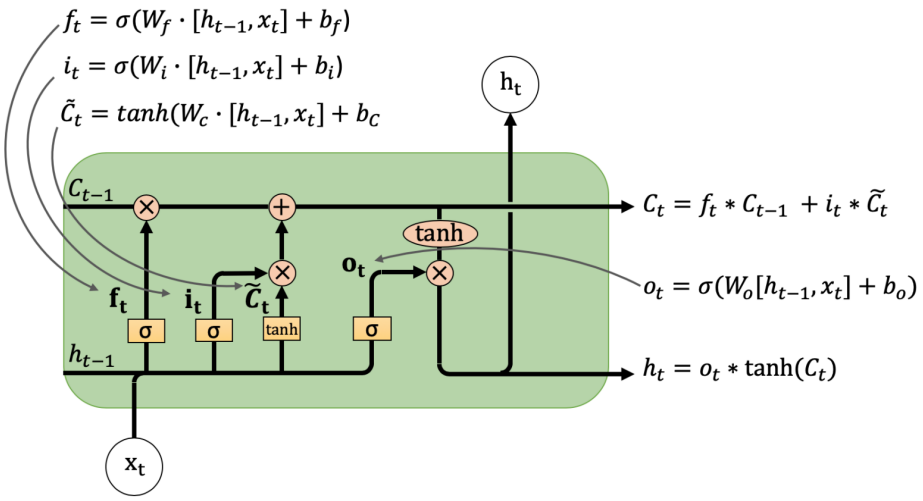


Figure 2.1: The architecture of an LSTM with equations

Another typical and famous kind of neural network is Transformer [64], which is based on the multi-head attention mechanism. The structure of a Transformer is shown in Figure 2.2. Transformer is a neural network archi-

tecture that can process sequential data such as texts, audios, videos, and images (as a sequence of image patches) without using any recurrent or convolution layers. The fundamental layer of a Transformer is called Attention. It also contains other basic layers such as fully-connected layers, normalization layers, embedding layers, and positional encoding layers. Attention is a mechanism that enables a neural network to focus more on relevant parts of the input data and pay less attention to the rest of the input data by computing the similarity between a query vector and a set of key vectors. These similarity scores are then normalized (typically using a softmax function) to produce attention weights, which are used to compute a weighted sum of value vectors. A higher similarity score leads to a higher attention weight after normalization. The original Transformer architecture has both encoder and decoder blocks, designed for sequence-to-sequence tasks such as neural machine translation. However, subsequent research has introduced encoder-only and decoder-only variants, which have been widely adopted in practice to address different classes of problems.

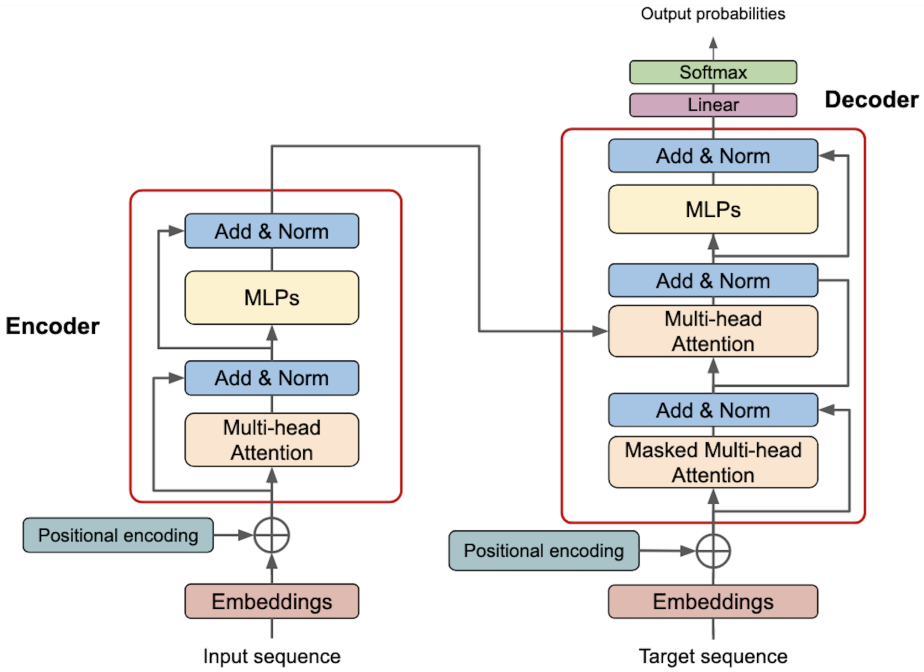


Figure 2.2: The architecture of a Transformer

2.2 Large language models

Large language models (LLMs) are a type of machine learning models designed for natural language processing tasks such as language generation, translation and question answering. Fundamentally, an LLM is a pretrained model that acquires knowledge by learning from massive text corpora. A key innovation of LLMs lies in their scale. They consist of an enormous number of parameters, often ranging from hundreds of millions to hundreds of billions. These models are typically trained using self-supervised learning on large-scale datasets. Their performance tends to improve significantly as model size increases. The phenomenon of rapidly improving performance with increasing model size is referred to as the scaling law [28]. We present some representative models along with their number of parameters in Table 2.1.

As the generative capabilities of LLMs continue to improve, a growing concern is that the content they produce may sometimes violate human moral or ethical standards. To address the misalignment between the outputs generated by LLMs and human preferences, researchers have investigated this problem and developed several mitigation techniques, one of the most notable being reinforcement learning from human feedback (RLHF) [45]. In brief, RLHF is a method that combines supervised learning and reinforcement learning to fine-tune LLMs so that their outputs better align with human preferences. The core process of RLHF includes three steps. The first step is to collect human feedback. Human annotators are required to rank different outputs generated by the language model based on quality or preference. The second step is to train a reward model. The rankings from the first step are used to train the reward model that predicts how well an output aligns with human preferences. The reward model takes the language model's output and assigns a score representing how favorable it is according to human feedback. The third step is to fine-tune the language model with reinforcement learning. A reinforcement learning algorithm, typically the proximal policy optimization algorithm(PPO), is used to fine-tune the language model based on the reward from the reward model and the goal is to maximize the reward as predicted by the reward model. RLHF guides a language model to produce more human-aligned outputs.

Although a large language model is a pretrained model capable of performing many tasks directly, researchers can further fine-tune these models to better adapt them to specific downstream tasks. For example, CodeBERT [13] is built upon BERT [9] and further trained on a large corpus of source code, enabling it to handle programming-related tasks more effectively.

Table 2.1: Number of parameters of some representative LLMs

Name	Release year	Organization	No. Params	Architecture
GPT-1	2018	OpenAI	117M	Decoder-only
BERT (base)	2018	Google	110M	Encoder-only
GPT-2	2019	OpenAI	1.5B	Decoder-only
T5	2020	Google	11B	Encoder-Decoder
GPT-3	2020	OpenAI	175B	Decoder-only
Gopher	2021	DeepMind	280B	Decoder-only
PaLM	2022	Google	540B	Decoder-only
OPT	2022	Meta	175B	Decoder-only
LLaMA 1	2023	Meta	65B	Decoder-only
GPT-4	2023	OpenAI	> 500B	Multi-modal
LLaMA 2	2023	Meta	70B	Decoder-only
Mistral	2023	Mistral AI	7B	Decoder-only
Gemini 1.5 Pro	2024	Google DeepMind	Unknown	Multi-modal
Claude 3 Opus	2024	Anthropic	Unknown	Multi-modal

Table 2.2: Input limits of major LLMs

Name	Input limits	Output limits
ChatGPT-4o	128,000	16,384
GPT-3.5 Turbo	16,385	4,096
GPT-4	8,192	8,192
GPT-4 Turbo	128,000	4,096
OpenAI o1	200,000	100,000
OpenAI o3	200,000	100,000
Claude Opus 4	200,000	32,000
Claude Sonnet 4	200,000	64,000
Claude Sonnet 3.7	200,000	64,000
Gemini 2.5 Pro Preview	1,048,576	65,536
Gemini 2.0 Flash	1,048,576	8,192
deepseek-chat	64,000	64,000
deepseek-reasoner	64,000	64,000

Almost all large language models are subject to limitations on the length of both input and output context, including some commonly-used LLMs such as ChatGPT by OpenAI, Claude by Anthropic and Gemini by Google. They are constrained by maximum input and output lengths, which can impact their ability to handle long or complex tasks. In Table 2.2, we present some of the commonly-used large language models and their input and output limits. As of the time of writing this dissertation, current models support a maximum input size of 1,048,576 tokens, which corresponds to approximately 750,000 English words. It is important to note that even though some models claim to handle nearly one million English words in context, their actual performance can still be affected, as they may sometimes generate hallucinations.

To improve the effectiveness of LLMs, a range of techniques, collectively referred to as prompt engineering, have been developed. Some well-known prompt engineering techniques include in-context learning [5] and chain-of-thought prompting [69]. In-context learning refers to the ability of LLMs to learn from examples provided directly in the input prompt, without any parameter updates. By supplying a few input-output pairs, or known as demonstrations, LLMs can generalize and produce appropriate outputs for new inputs. Chain-of-thought prompting, on the other hand, guides LLMs to generate intermediate reasoning steps before arriving at a final answer.

This approach is particularly effective for tasks that require logical reasoning or multi-step problem solving. However, it typically requires human intervention in refining the prompts, and thus is not yet fully automated.

2.3 Abstract syntax trees

An abstract syntax tree (AST) is a data structure used to represent the structure of a program or code snippet. It is a tree representation of the abstract syntactic structure of source code written in a formal language. Each node of the tree denotes a syntax construct occurring in the text. An abstract syntax tree is different from concrete syntax trees, or parse tree, because it does not represent every detail, but rather just the structural details. For instance, grouping parentheses are implicit in the tree structure, so they do not have to be represented as separate nodes. Similarly, a syntactic construct like an if-condition-then statement may be denoted by means of a single node with three branches. Abstract syntax trees are often used in program analysis for various purposes, such as clone detection.

Chapter 3

Motivation

In recent years, large language model (LLM) technologies, exemplified by ChatGPT, have demonstrated strong capabilities in handling a wide range of text-based tasks, such as text generation, question answering, language translation, text summarization, and dialogue. In addition to natural language text, LLMs are also capable of handling numerous code-related tasks in programming languages, such as code generation, code completion, and code understanding. Solving software engineering problems is one of the key applications of LLMs. A survey [23] has shown that there has been a significant increase in the number of studies focused on the utilization of large language models in software engineering (LLM for SE).

However, LLMs are not a silver bullet when it comes to software engineering. Based on our literature review, there are still several problems that LLMs struggle to handle effectively at this stage. Although improving the capabilities of LLMs is a popular research topic, it is not the focus of this dissertation. In this dissertation, we primarily focus on software engineering problems that LLMs are not yet able to handle effectively and explore how LLMs can be applied to address these challenges. In this chapter, we begin with a retrospective overview of approaches to solve software engineering problems. We then examine the limitations of current LLMs and identify common characteristics of tasks that LLMs struggle to address. Finally, we present our proposal for better applying LLMs to software engineering problems.

3.1 A retrospective overview of solving software engineerting problems

Traditionally, before the usage of LLMs, software engineering problems were usually solved through a systematic procedure. We present the traditional solution of software engineering problems in Figure 3.1a. First, raw data, such as source code, commit logs, or documentation, is collected from the software system or development environment. This raw data is often unstructured or noisy, so it must be processed and transformed into more informative and structured representations to facilitate effective analysis. For example, abstract syntax trees (ASTs) are a widely used data structure in software engineering that captures the syntactic structure of source code, enabling further tasks such as code analysis, transformation, and verification.

Once appropriate data structures are obtained, the next step involves selecting suitable algorithms to solve the given tasks. These algorithms typically fall into two broad categories: heuristic-based algorithms, which rely on manually crafted rules and domain knowledge, and learning-based algorithms, which leverage machine learning techniques to automatically infer patterns from data. The choice of algorithm depends on the specific requirements of the task.

By combining well-designed data representations with appropriate algorithms, software engineers can derive meaningful insights or generate desired outputs, thus effectively addressing various software engineering problems. This traditional pipeline forms the foundation of many classic software engineering tools and frameworks. In the following subsections, we present related work that follows the traditional pipeline, focusing on the two example tasks introduced earlier.

3.1.1 Related work for code clones

Similar code pair detection is a popular topic with a long history. Before, people were more interested in single-language code clone detection, and many techniques were developed for different types of code clones. For example, some works [27, 58] develop token-based matching algorithms to find clones, while other work [26] develops an AST-based matching algorithm to detect clones. These approaches rely on pre-defined rules to generate matching patterns, and use static-analysis algorithms to compare patterns. After machine learning shows its dominating power in other fields, researchers want to apply its power to code clone detection. One work [34] develops a solely token-based clone detection approach using deep learning. Another

(a) The traditional procedures of solving SE problems with no LLMs

(b) The straight-forward use of LLMs in SE problems

Figure 3.1: A comparison between different ways of solving SE problems

work [68] builds a tree-LSTM model based on hashed features of information about code structure and grammar extracted from ASTs. Specifically, this work [74] splits ASTs into trees consisting of statement nodes as roots and corresponding AST nodes of statements, and builds a Bi-GRU model upon the encoded vectors of these statement trees. These approaches build neural network models based on code-related data structures, and rely on true code clone labels. In the above works, researchers process raw data, specifically source code in a single programming language, into more informative structures, such as key tokens or ASTs. They then apply algorithms, such as pattern matching or neural models, to analyze the processed data and obtain final results. This is consistent with the traditional pipeline discussed previously.

Compared with single-language code clone detection, cross-language code clone detection has a shorter history, but faster growth in attention since development in multiple programming languages becomes trendy. Perez et al. [48] train an LSTM-based neural network using ASTs and uses tree-based skip-gram algorithm to initialize node vectors. It also provides a large cross-language code clone dataset based on competitive programming contest submissions. Nafi et al. [41] train a model based on API documentation to detect clones. But these approaches still require true labels of code clones. Tao et al. [62] adopt a contrastive learning objective to fine-tune the pre-trained model CodeBert to detect clones. Specially, one work [59] questions the generalizability of CodeBert, and points out that CodeBert performs less than expected on unseen data in code clone detection task. There are also some works avoiding using machine learning models. Cheng et al. [7] analyze revision logs and version control histories to find cross-language clones. Vislavski et al. [65] analyze an enriched Concrete Syntax Tree, but requires two code fragments to have the same code length. Kraft et al. [32] detect cross-language code clones in the .NET language family by translating a program in a .NET language into the .NET Code Document Object Model (CodeDOM).

The above works also align with the traditional pipeline. Although the raw data has evolved into cross-language source code and the intermediate representations have become more diverse, such as incorporating API documentation and revision logs, the fundamental workflow remains unchanged. It is worth noting that CodeBert is used to convert raw source code into numerical vector representations [62]. Although CodeBert can be considered as an early form of a language model, its capabilities are not sufficient to support general-purpose dialogue tasks. Therefore, we regard this work as part of the traditional pipeline.

All above works [44, 32, 65, 41, 7, 48, 62] for traditional cross-language code clone detection are not satisfying for various reasons. Some of these works focus on functionality and neglect structural similarity, therefore are not specialized for structurally-similar cross-language code pairs. Some of them are only limited in .NET family of programming languages. Some of them are not applicable for large structurally-similar cross-language code pair detection. Specifically, these works [44, 32] detect cross-language code clones in the .NET language family by translating a program in a .NET language into the .NET Code Document Object Model (CodeDOM). They can represent the logical structure of source code, but are limited in .NET language family. Vislavski et al. [65] analyze an enriched Concrete Syntax Tree [51], which is an intermediate state used by Budimac et al. [6], to detect cross-language code clones. This method gives some consideration to control structures, but still has some limitations. For example, this method requires two code fragments to have the same code length. Cheng et al. [7] analyze commit logs recorded in version control systems to detect clones, which does not consider control structure similarity. Some works [62, 48, 41] train neural network models for traditional cross-language code clone detection by using data collected from competitive programming contest websites. The problem of these methods is that they require large labeled datasets for training. The dataset they use consists of different solutions by different programmers to the same problem. These code fragments have the same functionality, but not necessarily the similar control structures. It is difficult to collect a large dataset with structurally-similar cross-language code pairs for supervised model training. Moreover, the features these models choose for training the models are more focused on code semantics instead of structures.

One of the most recent works is done by Mathew and Stolee [39], which utilizes static analysis of code text and ASTs and dynamic analysis of input/output results to find clones. This work provides an approach to detect structurally-similar cross-language code pairs, which is consistent with our first example problem. However, the drawback of this approach is that it is very slow. We include this work as a baseline in our comparative analysis. Its technical details and relevance to our study are further elaborated in chapter 5. As far as we know, there is not yet a fast and accurate tool targeting at detecting structurally-similar cross-language code pairs.

3.1.2 Related work for software regressions

Regression testing techniques are getting more important because of the emergence of new approaches for developing software, such as the agile move-

ment [54]. Regression testing consists of rerunning the previously executed tests when the software under test evolves to verify that no new failures, or regressions, has been introduced [16]. Unit testing frameworks [14, 63, 3] play an important role in regression testing. Almost all major programming languages have one or more corresponding unit testing frameworks.

A number of approaches have been investigated to make regression testing more effective and efficient [16]. For example, test case prioritization [30] determines an execution ordering that ideally gives precedence to the most effective test cases; test case selection [29] takes a sample of test cases for execution generally based on the recently introduced changes; and test suite reduction [53] aims at removing reduntant test cases according to some criterion, for instance code or requirements coverage. Moreover, Elbaum et al. [11] provide techniques for improving regression testing in continuous integration development environments. While these works are not directly aligned with our example problem, they broadly adhere to the same traditional procedures, which process raw data and select suitable algorithms to derive results, without incorporating LLMs.

Although detecting regressions is well studied [60], localizing the causes of regressions requires further exploration. A number of techniques focus on preventing regressions before release [15], but fewer tools effectively address the causes of regressions post-release [46]. This work [47] provides a tool for regressions in C/C++ software. This work traces predicates and reports suspicious differences between the base version and the upgraded version by using GDB debugger. The drawback of this work is that it is unable to implement universal tracing with complete data serialization because of the limitations of a low-level and procedural programming language. As a result, only a small subset of programs can be applied using this technique to localize the causes of regressions. Maksimovic et al. [38] provides a tool that adopts machine learning techniques to debug regressions. It utilizes historical data from a version control system and the results from functional debugging. They train models to rank revisions based on their probability to be the cause of a certain failure. This work is also consistent with the traditional pipeline. It leverages version control information as intermediate data and employs a ranking algorithm to select appropriate results. However, they use an undisclosed dataset to test their tool. Moreover, developers still have to manually check function calls based on testing outputs and revision histories. Such debugging is especially tedious when the project is large and different components are interconnected with each other.

One of the most recent works [24] follows the traditional pipeline and proposes a trace-based approach for locating the root causes of software

regressions in JavaScript projects. They have proved that locating causes of regressions by tracing runtime information is possible, but the accuracy is not satisfying. Also, their approach is limited in JavaScript projects. We provide more details about this work in chapter 6 where it is treated as a competing approach in our evaluation.

3.1.3 Limitations of traditional procedures

All related works mentioned before follow the traditional procedures for solving software engineering problems, as illustrated in Figure 3.1a. This typically involves transforming raw data into structured representations, such as abstract syntax trees, followed by the application of heuristic or learning-based algorithms to solve specific tasks.

While effective in certain well-defined scenarios, these methods fail to effectively address the two example tasks discussed in this work, namely structurally-similar cross-language code search and retrieval of causes of software regressions. The limitations include two aspects. The first one is related to performance. Traditional methods often exhibit unsatisfactory results in terms of either accuracy or computational overhead, and in some cases, both. The second one is related to limited generalizability. It is often difficult to adapt them to new settings, such as supporting additional programming languages. As a result, their applicability remains constrained. In the following chapters, we provide overviews of related state-of-the-art approaches and explain how our proposed methods address the example tasks.

3.2 The use of LLMs in software engineering

Large language models have been foundational for natural language text generation and understanding. Due to the success of LLMs in natural language processing, many researchers have sought to apply them to source code, or software engineering problems. The number of studies exploring the application of LLMs in software engineering has been increasing year by year, and many of these studies have produced promising results [23]. The utilization of LLMs in software engineering is largely driven by a novel perspective that reinterprets many software engineering challenges as problems of data, code, and/or natural language analysis [67]. Typical tasks include code summarization and code generation, which yield a natural language description from a code fragment and a code fragment from a natural language description, respectively.

In most cases, the popular utilization of LLMs involves presenting tasks in a plain-text format, with corresponding data that is raw or minimally processed attached directly to the prompt. This approach tends to treat LLMs as an all-powerful solution, often overlooking their inherent limitations. Since such a method is direct and straightforward, as shown in Figure 3.1b, and it may fail to take advantage of structured information and may result in inadequate performance on complex tasks.

These tasks often share common characteristics that align with the known limitations of LLMs. As discussed in chapter 1, examples of such tasks include code generation for less common programming languages, code search, and localization of the root causes of software regressions. A key challenge across these tasks is the involvement of a large number of files, with complex interactions occurring between the files, either statically (e.g., text similarity comparison) or dynamically (e.g., through runtime behaviors). Additionally, some tasks may require domain-specific or rarely encountered knowledge, which LLMs may not have been sufficiently exposed to during training. This lack of such knowledge further limits their effectiveness in handling scenarios where minor knowledge is needed.

These characteristics correspond to several known weaknesses of LLMs. First, context limits remain a fundamental constraint. Although state-of-the-art LLMs can now handle inputs of up to one or two million tokens, this is still insufficient for processing entire codebases in large-scale software projects, where relevant information may be distributed across numerous files and modules. Even in cases where an LLM is technically capable of processing the entire codebase, the results are often unsatisfactory, which leads to the second weakness. Limited reasoning capability is another challenge. Even when large inputs are supported, LLMs often struggle to maintain a coherent understanding of complex inter-file relationships. This frequently leads to reasoning failures and hallucinated outputs, particularly when the dependencies between files are implicit. Third, generalization beyond training data is not guaranteed. When LLMs are exposed to knowledge that was absent in their training datasets, such as domain-specific code or obscure programming languages, their performance becomes unreliable.

In the following subsections, we present related work about the use of LLMs with particular relevance to the two example tasks mentioned before. However, it is important to note that these works do not directly address the example tasks themselves, but rather explore adjacent or similar areas that is relevant to our example tasks.

3.2.1 Related work for code clones

Some attempts [10, 76] have been made to use LLMs for code clone detection, which is a binary classification problem. In these works, LLMs are treated merely as simple conversational engines, and the prompt design is relatively straightforward. The raw input data and problem statements are directly provided to LLMs, which are then expected to produce a binary decision regarding whether the given code fragments represent a clone. As a result, these attempts also have some limitations. For example, they are typically designed to determine whether a given pair of code snippets constitutes a specific type of clone. They do not consider scenarios where similar code fragments need to be retrieved from a large pool of candidates. Moreover, these methods are designed for single-language scenarios and do not address cross-language scenarios. According to the results, the ability of LLMs to detect code clones differs among various programming languages. LLMs' performance in code clone detection is unclear and needs more study for accurate assessment. Although some techniques such as chain-of-thought prompting are employed to improve the accuracy of LLM responses, these approaches do not attempt to decompose the problem or perform deeper analysis for a more fine-grained utilization of LLMs. As a result, they remain relatively straightforward. Due to the input limitations of LLMs, it is currently difficult to solve cross-language code search by simply adopting LLMs.

3.2.2 Related work for software regressions

Some works [36, 57] target at regression testing with the help of LLMs, although the problem settings in these works differ from our work. One work [36] provides a feedback-directed, zero-shot LLM-based regression test generation technique, and tries to generate easy-to-understand bug-revealing or bug-reproduction test cases for commits even when only a commit message is given. They try to generate test cases to prevent future potential software regressions instead of locating the causes of software regressions when they occur. Further, this technique has to take highly structured and human-readable inputs, which limits the generalizability of this technique.

Another work [57] presents a code-aware prompting strategy for LLMs in test generation based on recent work that demonstrates LLMs can solve more complex logical problems when prompted to reason about the problem in a multi-step fashion. In specific, they generate test cases by deconstructing the testsuite generation process into a multi-stage sequence, each of which is

driven by a specific prompt aligned with the execution paths of the method under test, and exposing relevant type and dependency focal context to the model. They intend to improve test coverage in a regression setting. In this setting, they assume that the code currently under test is implemented correctly. Their objective is to generate a suite of tests that will effectively detect future bugs that may be introduced into the codebase during later development, causing a regression. They also do not focus on the problem of locating the root cause after a regression occurs.

Although these works are not directly related to our example task, our investigation into their use of LLMs reveals a rather straightforward application. For instance, in the second related work [57], although the task is decomposed and multi-step prompting is employed, the use of LLMs in each step remains relatively direct, as the problem statements are provided to the model without additional intermediate processing or abstraction. This limits the range of problems that LLMs can effectively address. Moreover, repeated use of LLMs increases the likelihood of hallucinations.

3.2.3 Limitations of direct usage of LLMs

From the related works discussed above, we observe that while some software engineering problems have been effectively addressed using LLM-based solutions, our two example tasks remain largely unexplored in this context and currently lack well-established LLM-based approaches.

In our first task, structurally-similar cross-language code search, the goal is to compare a large number of code pairs based on their structural similarity. This process often involves analyzing hundreds of source code files simultaneously. For LLMs, which are static text analyzers, such a task is challenging because the context limit restricts the number of files that can be processed at once, and their limited reasoning capability makes it difficult to accurately capture and compare the structural relationships across multiple files. As a result, LLMs struggle to solve large-scale structural comparison effectively.

In our second task, retrieval of causes of software regressions, the objective is to compare two versions of a codebase in chronological order. This process typically involves analyzing a large number of source code files across both versions. Traditionally, identifying the root cause of a regression requires dynamic analysis and execution tracing, which remain challenging for LLMs. Moreover, the context limits and reasoning capabilities of LLMs are insufficient for analyzing an entire codebase to locate the potential causes of software regressions. As a result, although LLMs can provide general sugges-

tions about code differences or generate test cases to prevent potential future regressions, they are currently unable to accurately pinpoint the exact cause of a regression.

At the same time, the direct use of LLMs also presents several challenges. For example, their performance is not always reliable, and repeated or continuous use can increase the likelihood of hallucinations. Addressing the limitations of current approaches requires a more sophisticated method for leveraging LLMs in software engineering tasks, as opposed to relying on direct and naive usage.

3.3 Summary

In this chapter, we begin with a retrospective overview of how software engineering problems have traditionally been addressed, along with some representative literature that follows the traditional pipeline. We then explore recent research that applies LLMs directly to software engineering problems and discuss the limitations of these approaches, including the weaknesses of current LLMs in handling complex software engineering tasks. These observations motivate our goal: to expand the scope of software engineering problems that can be effectively addressed with LLMs. In next chapter, we will give our proposal about how to effectively address more software engineering problems with LLMs.

Chapter 4

Proposal

In this work, we aim to expand the scope of software engineering problems that can benefit from the use of LLMs. The position of our attempts are illustrated in Figure 4.1. To address the limitations of both traditional approaches and the straight-forward, end-to-end use of LLMs of solving software engineering problems, we propose a selective strategy. In this chapter, we first introduce our proposal in detail. While writing this dissertation, we have identified several related works that share similar ideas with ours but are developed independently and in parallel with us. We introduce these works in the later section.

4.1 Selectively Applying LLMs to SE Problems

To address software engineering problems more effectively, we propose an approach with selective application of LLMs. In this approach, LLMs are selectively applied to specific sub-part of the SE problem where their strengths can be effectively leveraged. An overview of our proposed methodology is illustrated in Figure 4.2.

In short, rather than using LLMs as end-to-end solvers, we integrate LLMs into traditional software engineering pipelines by empowering only certain parts of the original task and the original task is improved correspondingly. By doing so, we leverage the strengths of LLMs, such as code understanding and pattern recognition, without being limited by their weaknesses. More specifically, we use LLMs simply as tools to either assist in processing raw data (e.g., transforming unstructured code into intermediate representations) or serve as pre-trained models like neural networks, rather than making them responsible for solving the entire task.

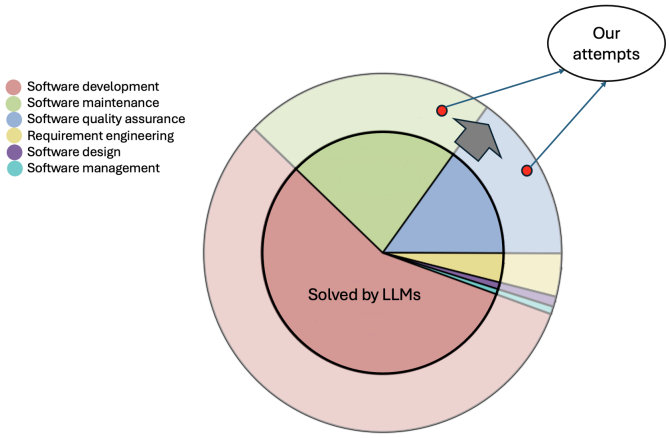


Figure 4.1: The position of our research

Traditionally, solving software engineering problems requires to reformulate tasks and analyze the underlying data structure of the original task. Subsequently, the raw data is transformed into an intermediate data structure that is more informative or easier for processing. Such reformulations can occur at different levels of granularity, depending on the specific task and its objectives.

Due to the impressive performance of LLMs, many people tend to regard them merely as simple conversational engines. They present their questions in a relatively straight-forward manner and expect LLMs to be an end-to-end solver. This approach can sometimes yield satisfactory results for tasks with relatively simple formulations and simple data structures. However, some software engineering problems have complex descriptions and intricate data structures, and they heavily rely on logical reasoning. Without careful task analysis, these problems suffer from the weaknesses of LLMs discussed before. The direct use of LLMs fails to generate effective responses and produce satisfactory results.

Our proposal combines the strengths of traditional procedures with the capabilities of LLMs. In the reformulated subtasks, we leverage LLMs as powerful conversational engines to enhance specific components of the pipeline. By improving the performance of these individual subtasks, we aim to improve the overall performance of the original task.

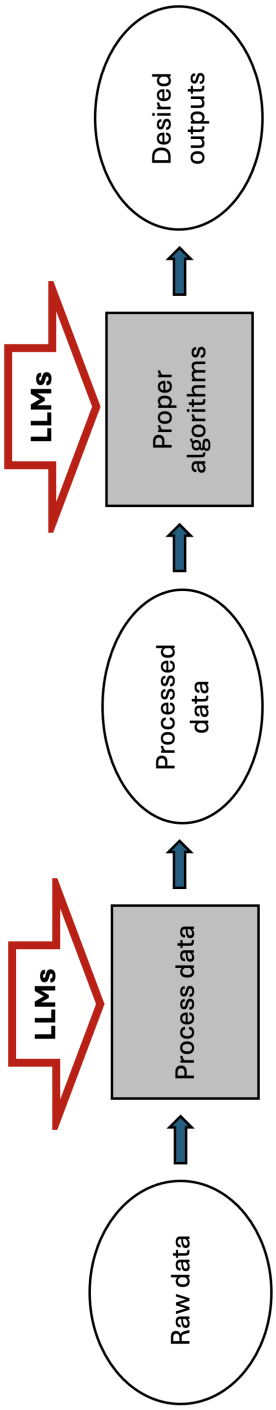


Figure 4.2: Our proposal: selectively applying LLMs to SE problems

4.2 Related work

Some recent works make similar attempts in parallel, although these works do not address our two example tasks directly. For example, two works [50, 35] try to solve the project-level bug localization problem, which localizes a bug within the extire codebase.

Project-level bug localization problem also requires to process a large number of source files simultaneously and analyze the logic between files. The traditional pipeline struggles to address this problem effectively. This is because different bugs exhibit different characteristics, making it difficult to define a suitable intermediate representation and algorithm for accurate bug localization. At the same time, the direct use of LLMs is insufficient for handling a large number of issues and reasoning over complex file relationships.

The approaches proposed in these two related works are consistent with the proposal presented in this dissertation. They both try to filter the large number of files by heuristics and reduce the context length. For example, one method first narrows down relevant class files and then further localizes methods within those classes. Semantic embeddings are employed to match error reports with potentially related source code files. After the input content is reduced, LLMs are applied to a smaller set of files to identify the bug. In other words, they selectively apply LLMs to the problem, using them as an algorithm to analyze a limited number of files. Prior to utilizing LLMs, intermediate files are obtained through heuristic methods. The key to solving the problem is that how to break down the task into manageable subtasks.

In Table 4.1, we provide an overview of existing literature that is relevant to our tasks but does not directly address them. Our goal is to fill in this gap by demonstrating the feasibility and effectiveness of our proposed approach. This table also helps position our research within the broader landscape of software engineering studies involving LLMs.

4.3 Summary

To tackle the challenge, we present our proposal in which LLMs are selectively applied to specific sub-parts of a software engineering task, rather than being used to address the entire task end-to-end. The core of this proposal is to decompose the original complex problem into smaller, more manageable sub-tasks, where LLMs can be utilized more effectively. By improving the

Table 4.1: Related paper in relevant areas

	Code clone detection	Code search	Bug localization	Regression cause localization	other SE problems
Without LLMs	Daniel et al., 2019 [48]	Mathew et al., 2021 [39]	Lam et al., 2017 [33]	Hu et al., 2024 [24]	Szafraniec et al., 2022 [61]
Direct use of LLMs	Zhang et al., 2024 [76]	No relevant literature	Wu et al., 2023 [72]; Hossain et al., 2024 [22]; Yang et al., 2024 [73]	No relevant literature	Jiang et al., 2024 [25]
Indirect use of LLMs	Dou et al., 2023 [10]	Can we?	Qin et al., 2025 [50]; Li et al., 2025 [35]	Can we?	Zhang et al., 2025 [75]

performance of one or more sub-tasks through the use of LLMs, the overall effectiveness and efficiency of solving the original task can be enhanced.

In the following chapters, we present two representative tasks as case studies to demonstrate our approach. We begin by introducing the motivation behind each task and reviewing related work. Then, we analyze each task in detail by examining their underlying data structures. Finally, we show how LLMs can be selectively incorporated into the solution, either in handling raw data and helping generate intermediate representations, or supporting specific sub-tasks, to improve the overall performance of the task.

Chapter 5

Structurally-similar Cross-language Code Search

In this chapter, we will present our approach for structurally-similar cross-language code search. We will start with the motivation for this task by introducing its significance. After that, we introduce the state-of-the-art approach for this task and analyze the limitations of this approach. To address the limitations of the current state-of-the-art approach, we introduce a novel approach that incorporates LLMs as a key component of the solution. Specifically, we will develop a new code intermediate representation called two-level generic ASTs, and train a model based on two-level generic ASTs to improve search speed without losing too much accuracy. During the generation of two-level generic ASTs, we leverage an LLM to perform the mapping between two language-specific ASTs, which results in the final design of the generic AST.

5.1 Significance of structurally-similar cross-language code search

Similar code pairs are a non-negligible issue in software maintenance [18]. Researchers have studied detection of similar code pairs with the same programming language for decades. This task is also referred as code clone detection. Based on the degree of similarity, [56] classifies code pairs with the same language into four types. Among them, type-1 pairs means identical code fragments, ignoring differences in white-space, code formatting and comments. Types-2 pairs means structurally or syntactically identical code fragments, ignoring differences in identifier names and literal values, as well

as differences in white-space, code formatting and comments. Type-3 pairs are these with statement-level insertions, deletions and substitutions, and type-4 pairs are these with the same functionality while ignoring implementation details. Many approaches [27, 58, 26, 55, 34] have been developed to solve the detection for each type of similar code pairs.

With the advancement of software development technologies, detecting similar code pairs across different programming languages is attracting practitioners' interests. This trend is driven by the growing prevalence of polyglot programming in large software projects. A representative scenario is web-service development. A web-service project usually includes a server side and multiple client sides. Programmers often use multiple languages for different platforms, such as Java for servers, JavaScript for browsers, Kotlin for Android and Swift for iOS. In such settings, cross-language code pairs with similar control structures are of particularly interests. These structurally similar pairs are often indicators of reusable functionality. We will elaborate on the practical significance of these code pairs in detail.

When software development is polyglot, cross-language code pairs with similar control structures are important causes of source code refactoring and software maintenance. A typical scenario can be found in micro-service development where multiple languages are used for different modules. In micro-service development, an important principle is single-responsibility principle, which states that a microservice should do one thing only, such as user management, order processing, or email notification, and own all the logic and data related to that responsibility. If a service breaks the single-responsibility principle, code pairs with similar control structures may appear. For instance, a service which is responsible for managing the user posts in a bulletin board system (BBS) may implement some authorization logic, and another service responsible for managing user comments may also implement a similar authorization logic. This will result in code duplication, and such duplication of responsibilities violates the single-responsibility principle. Although the two services implement different functionalities, the authorization logic should ideally be extracted and consolidated into an independent service.

Another typical scenario is modern web-based service development. In web-based service development, different clients often share the same business logic. In such cases, the shared logic should be consolidated to reduce code duplication. For example, client-side functions across platforms that share similar logic may correspond to a common server-side implementation. Usually, such functions share similar structures and they should be reconsidered and grouped into a function on the server side. We provide a concrete

example in a web-based message system as shown in Figure 5.1. In this message system, users can freely send messages through different platforms, such as mobile devices or web browsers. However, the grouping-messgae feature was not part of the initial application design in the system, and has been added later on. In the initial design, the backend API only provides an HTTP endpoint with a GET method to return a list of ungrouped user messages, and the frontend clients need to group the messages by themselves. Since the logic is the same across different clients, this leads to unnecessary duplicated code fragments. For maintenance efficiency and system stability, such pairs of code fragments should be grouped into a common API in the backend server. In Figure 5.2 and Figure 5.3, we show the two code snippets for grouping user messages. The two code snippets are written in different programming languages, and we can see a clear correspondence between their control structures: both the code fragments contain an assignment statement, a for statement and a return statement. Within the for loop body, the logic of processing the dict/HashMap is also similar. This pair of code snippets is supposed to be detected and merged into the server side.

While fine-grained requirements engineering and increased collaboration among development teams can help mitigate code duplication to some extent, these approaches alone are insufficient to completely eliminate the issue. In large-scale software systems, duplication often arises due to evolving requirements, inconsistent team practices, or historical design decisions. Consequently, detecting and reducing code duplication during software maintenance remains a tedious and non-trivial challenge for developers.

One reason why detecting structurally-similar cross-language code pairs is challenging is that, even if two code fragments share the same control structure, they are not identical at the token level. Although there exist some works for traditional cross-language code clone detection, these works have some limitations. For example, some works [65, 41, 7, 48, 62] are not suitable for structurally-similar cross-language code pair detection because they ignore the importance of control structures and only focus on the same functionality. Code pairs with the same functionality do not necessarily share similar control structures, which makes the techniques in these works potentially biased. Some works [44, 32] realize the importance of similar structures. However, they utilize .NET Code Document Object Model (CodeDOM) and thus are only useful in .NET language family. As a result, these techniques are difficult to extend to other programming languages such as Java or Python.

Our goal is similar to traditional cross-language clone detection, as both try to find a *similar* code pair. However, we are more focused on similar

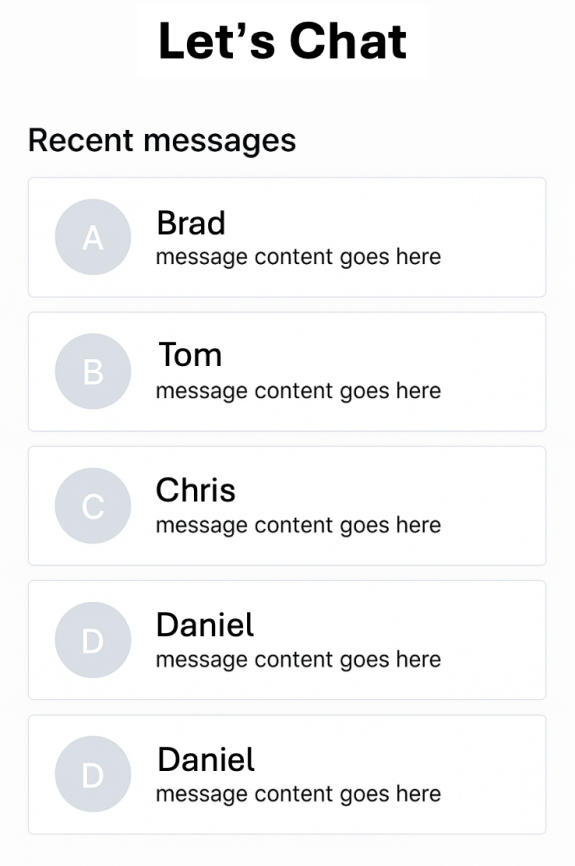


Figure 5.1: The frontend of a message system

control structures while traditional cross-language clone detection mainly considers the same functionality and ignores detailed implementations. In real-world development, it is important to detect similar structures as they are good candidates for code refactoring. Finding a pair of code fragments with only the same functionality is also an important research topic, but it is out of the scope of this paper. If two functions have the same functionality but with different algorithms, such difference might be intended by the programmers, and might not be refactored or merged. Therefore, we are more interested in similar control structures.

```
def group_posts:
  res = {}
  for post in posts:
    bucket = res.setdefault(post.owner, [])
    bucket.append(post)
  return res

```

Figure 5.2: A structurally-similar code pair in web-services: Python code

```
public Map<String, List<Post>> groupPosts(List<Post>
    posts){
  Map<String, List<Post>> grouped = new HashMap<>();
  for (Post post: posts){
    if (!grouped.containsKey(post.getOwner())){
      grouped.put(post.getOwner(), new ArrayList<
   Post>());
    }
    grouped.get(post.getOwner()).add(post);
  }
  return grouped;
}

```

Figure 5.3: A structurally-similar code pair in web-services: Java code

5.2 State-of-the-art method

To the best of our knowledge, the state-of-the-art technique for detecting a structually-similar cross-language code pair is proposed by Mathew and Stolee [39]. It notices the importance of similar control structures between code pairs, and successfully develops a technique to find such similarity. This work develops a common representation across different programming languages called *generic* abstract syntax trees (ASTs), and adopts tree-based editing-distance algorithm to calculate similarity scores between different generic ASTs. It takes control structures into consideration by calculating the number of edits to make two generic ASTs identical. Although a transformer from an original AST to a generic AST must be developed for every source language in advance, developing such a transformer is not expensive since the transformation is simple node-by-node mapping. The accuracy of this technique is ideal, but a drawback of the technique is its detection speed.

Since computing a tree-based editing distance is highly time-consuming, the technique is not applicable for a large code base.

In detail, this technique first transforms a given AST in a source language into *a generic AST*, which is common tree-representation among multiple languages to absorb syntactical differences between them. Then, *tree-based editing distances* are computed to discover similar trees. The code fragments with similar trees are a potential pair with similar control structures. By mapping features from different programming languages into generic nodes, this technique creates language-agnostic common intermediate representations, and can encompass syntactic features from different languages while abstracting away syntactical differences. For example, when detecting code pairs between Java and Python, the *if* statements in Java and Python are mapped to the same kind of tree node representing an *if* statement. A list comprehension in Python is mapped to a *loop* node in the generic AST. If a feature is only supported in some of the languages, an exclusive node is still created in the generic AST, such as a *switch* statement in Java. The capability of this technique relies on how the transformation is done and how the generic ASTs are designed. This work provides a simple design and a straight-forward node-to-node mapping and shows that their design works well. Although designing such a generic AST for Java and Prolog would be difficult, it is feasible for an Algol-like language family, in which a program consists of nested blocks and procedures.

This technique achieves good accuracy in finding cross-language code pairs with similar control structures. However, there are some drawbacks. The first one is the speed of detection. Constructing a generic AST is not expensive but computing editing distances between generic ASTs is heavily compute-intensive. If the number of generic ASTs is n, $n \times n$ distances must be computed. Due to this problem of being time-intensive, it is unrealistic to use this technique for large-scale detection. The second drawback is that their design of generic ASTs is ad-hoc, which makes it a bit difficult to migrate to a third language.

5.3 Our proposal

In this section, we present our proposed approach to overcome the limitations of the existing state-of-the-art solution. Before the system design details, we first explain how our approach aligns with the general proposal of this dissertation, namely, selectively applying LLMs to specific sub-components of the solution pipeline to improve the overall performance of the task.

As illustrated in Figure 5.4, we leverage LLMs as a tool to assist in generating an intermediate representation from raw data, which can be further processed by later algorithms. With the help of this intermediate representation, we develop a two-stage approach that enhances both the efficiency and accuracy of structurally-similar cross-language code search.

5.3.1 System overview

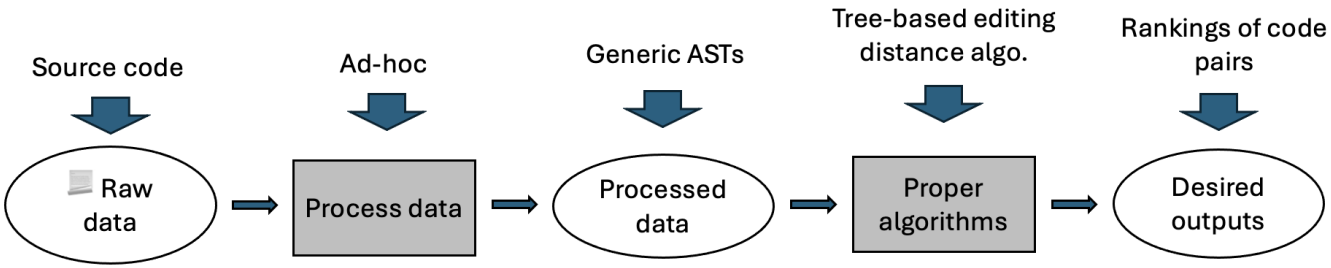
We develop a two-stage approach to detect structurally-similar cross-language code pairs. The system overview is shown in Figure 5.5. In the first stage, we use neural network models to pre-determine a few most-likely candidates, instead of calculating editing-distances of all candidates. In the second stage, we only calculate editing-distances with the pre-determined candidates. Our uniqueness is leveraging LLMs to develop a new code representation technique called *two-level generic ASTs* as inputs for models to improve the accuracy of the first stage. The technique uses two kinds of generic ASTs with different abstraction levels, one is coarse-grained and another is fine-grained, to model source code. After getting vector representations of source code, we calculate *aggregated cosine similarity scores* and compare them to pre-determine candidates. In this work, we use Java, Python and C as examples for structurally-similar cross-language code pair detection.

5.3.2 Two-level generic ASTs

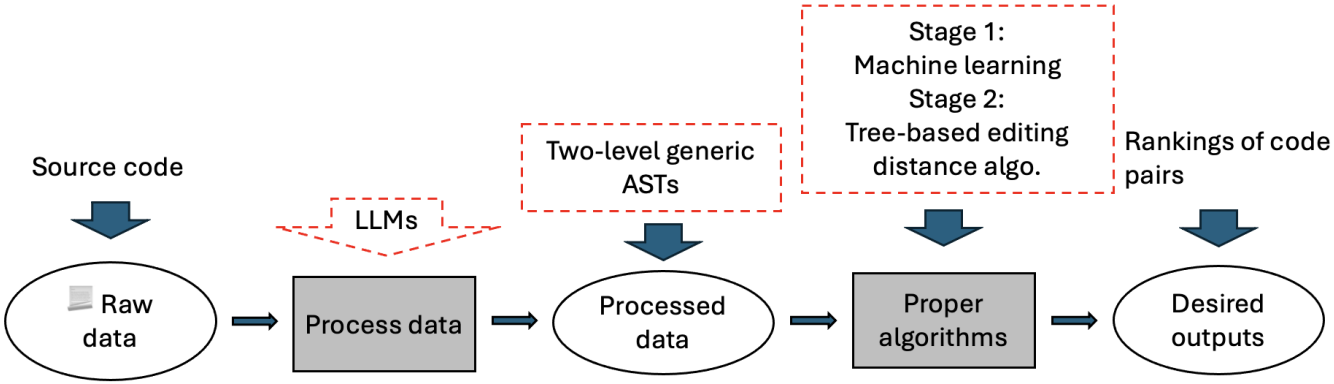
In the first stage, we use two-level generic ASTs as inputs for models to focus on different aspects of syntactic features. A generic AST is a mapping from language-specific AST nodes to generic AST nodes with deletion, addition and modification. The coarse-grained generic ASTs focus on high-level structural features, and the fine-grained generic ASTs focus on low-level semantic features. The mapping rules for coarse-grained generic ASTs and fine-grained generic ASTs are different. In this work, we use *JavaParser*[1], Python's *ast*[2] library, and *pycparser*[3] to generate original Java-, Python- and C-specific ASTs. We will introduce how to generate generic ASTs in detail.

A fine-grained generic AST is a rough union of syntactic features from two languages, and it preserves both common syntactic features and language-specific syntactic features. A language-specific syntactic feature means a

[1]https://javaparser.org/
[2]https://docs.python.org/3.7/library/ast.html
[3]https://pypi.org/project/pycparser/



(a) The workflow of the state-of-the-art approach



(b) The workflow of our proposal

Figure 5.4: How we selectively apply LLMs in the pipeline of the first task

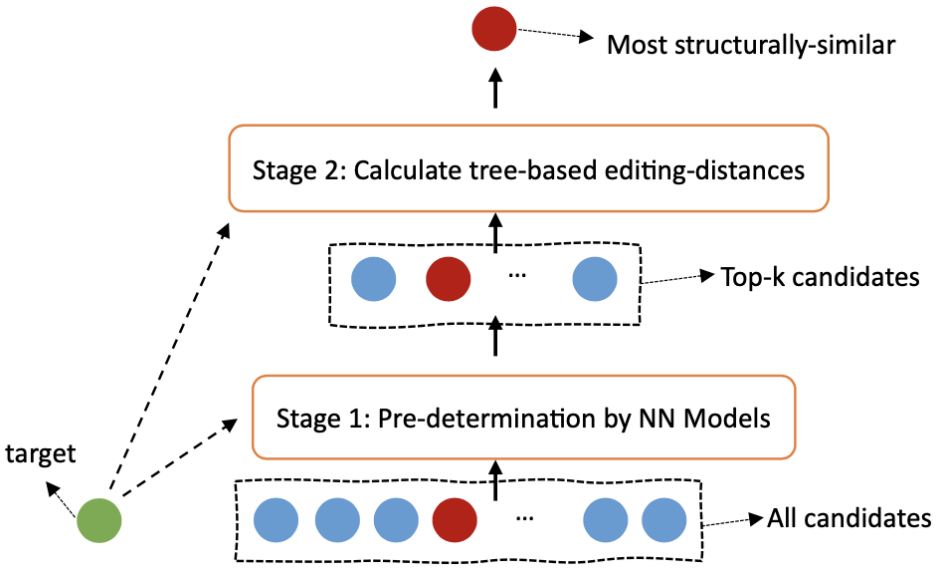


Figure 5.5: System overview

syntactic feature that exists in one programming language, but not necessarily exists in other languages. For example, type declaration is enforced in Java, but Python does not have static types. Therefore, type declaration is a language-specific feature for Java. A common syntactic feature means a syntactic feature that exists in both languages. For example, a *for* statement exists in Java, Python and C. A common syntactic feature for two programming languages could also be a language-specific feature for a third language. For example, both Java and Python are object-oriented languages and support class definitions. However, such feature does not exist in C.

The way of generating a fine-grained generic AST is to map common syntactic features into common generic AST nodes, and keep all other language-specific syntactic features as what they are. To do so, we look into the documentation of original ASTs and extract all syntactic features. Then, we adopt a large language model (LLM) [4] to generate a mapping between syntactic features from two languages. We simply adopt the mapping by the LLM. If one syntactic feature does not have a corresponding mapping in another language, we regard it as a language-specific feature. The prompt we use is shown in Figure 6.10:

A coarse-grained generic AST is a rough intersection of syntactic fea-

[4]We use gpt-4-turbo API from OpenAI in this work.

```
1 There are some AST node types in C:ID, Constant,
    BinaryOpBitXor, UnaryOpInvert, Goto, ...; Also,
    there are some AST node types in Python: Name,
    Call, Num, FunctionDef, body, Str, BinOpSub, ...
2 Can you generate a mapping of the common syntactic
    features between two languages? If some node
    types are specific in one particular language,
    just ignore them.
3
```

Figure 5.6: An example of prompt in generating mappings between two languages

tures from two languages. It ignores language-specific syntactic features, and keeps essential features that refer to common key control structures. The way of generating a coarse-grained generic AST is as follows. We first use a large amount of source code and generate original ASTs for all the source code. Then, we count the number of occurrences of each syntactic feature, and sort them based on their number of occurrences. Finally, based on the mapping in fine-grained ASTs and a hyper-parameter *node_num*, we only keep the most-frequently occurring common nodes, and ignore all other nodes. If one node is not included in coarse-grained ASTs, its children are directly concatenated to its parent, and this node is deleted. One certain rule about coarse-grained ASTs is that all kinds of binary operations and unary operations are ignored and they are mapped into common *BinaryOp* and *UnaryOp* nodes. This is to increase the variety of key control structures in the most-frequently occurring nodes. After a coarse-grained generic AST is generated, unnecessary information is ignored, and only key control structures are kept.

5.3.3 Model training and usage

To train the models in the first stage, a code fragment is first transformed into two generic ASTs designed in subsection 5.3.2, and these ASTs are given as inputs to neural models after being vectorized by the path-based encoding [2]. We use an LSTM-based encoder-decoder model [8] to generate a vector representation for the given AST, since LSTM processes sequences of data with long-range dependencies. This model is trained to predict a method name for the given AST representing its method body. In other words, it is trained so that its decoder's output will be a method name. Thus, the

model training involves no true structurally-similar code pair labels.

After the training, when we detect code pairs, the decoder is not used; only the encoder is used. The vector representation from the encoder's output is used as the representation of the given AST to calculate similarity between coarse-grained or fine-grained ASTs, and finally to calculate similarity between code fragments. We below give details about how the models are trained and used. The model overview is shown in Figure 5.7.

We use path-based encoding [2] to transform ASTs into model inputs. A path means a shortest sequence of AST nodes from a leaf node to another leaf node with all intermediate nodes through parent-child connections. Since every node can only have one parent, every pair of leaf nodes must be able to trace back to one common ancestor, making a path between two leaf nodes unique. If an AST has n leaf nodes, it has $n \times (n-1) \div 2$ paths. For practical training, We randomly sample m paths as inputs.

After paths are extracted from an AST, we use look-up tables [40] to vectorize the paths, which transfer tokens in paths into numerical vectors for models. In our system, there are three different look-up tables. Two of them are for generic AST nodes. As the mapping rules for coarse-grained generic ASTs and fine-grained generic ASTs are different, they have independent look-up tables, and each of them has all nodes in corresponding generic ASTs. Besides them, there is a method name look-up table. A method name look-up table is consisted of method names either by camel-case or by snake-case. We split them into subtokens and use subtokens for the look-up table. Specifically, $<UNK>$ is used for unknown subtokens that are not in the look-up table.

After paths and method names are vectorized, we put them into the encoder-decoder model for training. Both the encoder and the decoder are based on LSTM [20], and the architectures are referred from [2]. Given a generic AST path, the encoder gives nodes embedded by the look-up table to a bi-directional LSTM (bi-LSTM) sequentially, and use the final state from the bi-LSTM as the vector representation of this path. We incorporate attention mechanism to aggregate vector representations of paths from the same generic AST as the vector representation of this generic AST. This representation will also be used for further calculation of cosine similarity scores. The decoder gets the vector representation from the encoder for a generic AST, and it predicts a method name. During training, the predicted method name is compared with the true method name, and if it is not correct, back-propagation is used to modify the weights of models. When the model converges, the vector representation of a given generic AST can successfully be used to predict a method name.

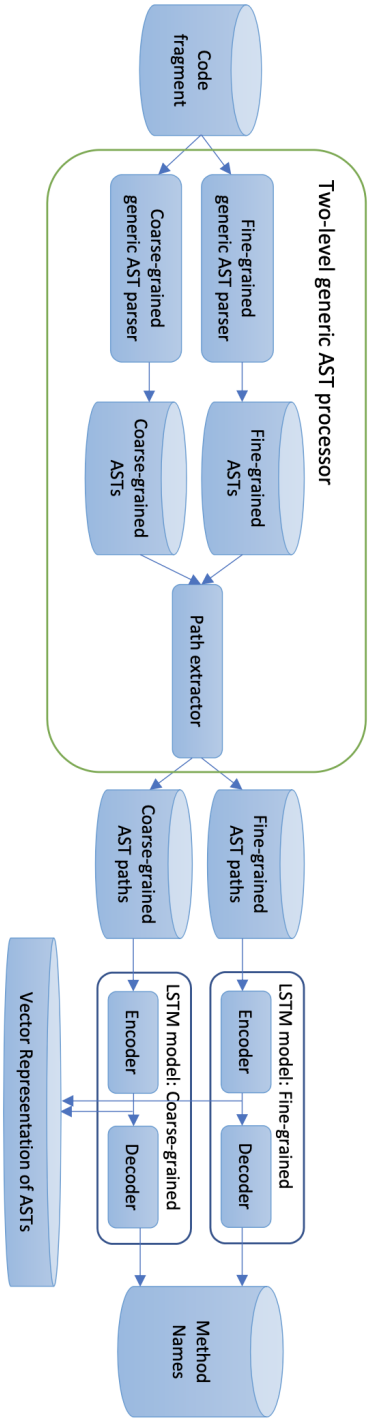


Figure 5.7: Model structure overview

When using the trained models to detect structurally-similar code pairs, we use the vector representations from the encoders to calculate cosine similarity scores. We get two vector representations of a code fragment, one from the model for coarse-grained generic ASTs and one from the model for fine-grained generic ASTs. To compare two code fragments, four generic ASTs are generated, and then four vector representations are generated from the ASTs by the models. By using *cosine similarity*, their similarity scores are calculated for the pair of coarse-grained ASTs and the pair of fine-grained ASTs. The final score is the sum of the two scores. In detail, given a target code fragment t and a candidate code fragment c, we calculate $score_{t-c}^{fine}$ and $score_{t-c}^{coarse}$, and add the fine-grained similarity score and coarse-grained similarity score together to get an aggregated similarity score $score_{t-c}^{aggr}$. The higher the score, the higher possibility the candidate is similar to the target. Since cosine similarity is always between 0 and 1, the aggregated similarity score $score_{t-c}^{aggr}$ is always no larger than 2.

5.3.4 Design rationale

The rationale behind our system design is that, to achieve our goal, we first need to consider the underlying data structure of the task itself. Since our focus is on structurally similar code pairs, the Abstract Syntax Tree (AST) is the most appropriate representation for analyzing code structure. Furthermore, because we are dealing with cross-language code pairs, it is necessary to develop a method that enables effective comparison between ASTs from different programming languages. However, according to the findings reported in [39], generic AST comparison based solely on heuristic algorithms is computationally expensive and suffers from substantial time consumption, making it impractical for large-scale or real-time applications. Therefore, we adopt a machine learning-based approach to reduce the time consumption.

Moreover, using a single generic AST often leads to significant information loss, which can reduce search accuracy in a machine learning context. To address this issue, we decompose the overall problem into two sub-problems. First, we design two types of generic ASTs and apply parallel learning on both representations to improve search accuracy. Second, we combine learning-based and heuristic-based approaches in a two-stage framework, further enhancing the precision of the code search.

Finally, when designing generic ASTs, we aim to minimize subjective bias, as we are dealing with multiple programming languages. Since LLMs possess prior knowledge of various languages and their syntax, we leverage

LLMs to assist in the design of generic ASTs. This helps reduce the influence of individual subjectivity and enhances the consistency and objectivity of the overall algorithm. In fact, our experiments show that the generic ASTs designed with the help of LLMs outperform those manually designed by the authors in terms of performance.

5.4 Experiments

We want to evaluate both accuracy and speed of our tool against other approaches in detecting structurally-similar cross-language code pairs. The first competitor is tree-based editing-distance algorithm from [39], and it is the state-of-the-art approach. It calculates the edits needed to transform an ordered labeled tree to another. Specifically, this work uses Zhang-Sasha algorithm to calculate tree-based editing distance between two generic ASTs. Since [39] provides their source code to generate Java and Python generic ASTs, we reuse their design in this algorithm. However, [39] does not provide code to generate generic ASTs for C language. We use our coarse-grained ASTs for C in this algorithm.

The second competitor is token-based Jaccard similarity algorithm from [39]. It calculates the frequency of common tokens between two code fragments by Jaccard similarity coefficient. To avoid rare tokens and limit vocabulary size, it splits identifiers to address nomenclature, and removes unnecessary tokens from source code, such as English stopwords and tokens shorter than a certain length.

The third and fourth competitors are pre-trained large neural network models. One is CodeBert [13]. The other one is Unixcoder [17], which is fine-tuned on code search dataset based on CodeBert. They both use a linear token sequence of source code as input. The sequence is generated by tokenizing source code using a lexical analyzer, and splitting camel-case or snake-case identifiers into subtokens. The models will output a vector representation of the sequence. We calculate cosine similarity of two vectors as the similarity of two code fragments.

We also do ablation study for our approach. The fifth to seventh competitors utilize neural network models only and do not combine editing-distance algorithm. We train the models on only one kind of generic ASTs (fine-grained and coarse-grained) and two-level generic ASTs respectively, and evaluate the performance of these models. The rest two competitors combine neural network models with editing-distance algorithm, but the models are trained on single kind of generic ASTs.

To train models on generic ASTs, we use Java-small [1], py150k [52] and CodeNet [49] datasets. They are large code collections based on open-source projects. The training requires no similarity labels. We only need the source files and process them into corresponding generic AST formats.

5.4.1 Evaluation metrics

Our approach outputs a similar code fragment among a set of candidates, which is similar to an information retrieval system. Therefore, we use average rank (AR), quantile rank, SuccessRate@k and mean reciprocal rank (MRR) [66] to measure the accuracy of approaches. Consider a given target and multiple candidates, we try to find the most structurally-similar code fragment to the target among all the candidates. In our experiment setting, there is one and only one ground truth among the candidates in a single detection. By sorting all candidates according to their similarity scores with the target, the ground truth is ranked among the candidates. If the ground truth is ranked higher, the approach is more powerful in structurally-similar code pair detection.

Consider a detection d in a set of detections D. $R(d)$ is the rank of ground truth. δ_k is the indicator function which returns 1 if the input is less than or equal to k and 0 otherwise. SuccessRate@k or SR@k is the percentage of detections for which the ground truth exists among the top-k candidates. Mathematically, AR, MRR and SR@k are defined as $AR = \frac{\sum_{d \in D} R(d)}{|D|}, MRR = \frac{\sum_{d \in D} \frac{1}{R(d)}}{|D|}, SR@k = \frac{\sum_{d \in D} \delta_k(R(d))}{|D|}$.

Quantile rank means the rank at a certain point after sorting the ranks of all detections ascendingly. For example, 25% quantile rank means the worst rank of the most well-performing 25% detections. Quantile rank represents the accuracy of the system on a certain proportion of well-performing cases.

We use the average inferring time of all detections to measure the speed of approaches. The inferring time of one detection is the time to calculate the similarity scores between the target and all candidates. To note, for machine learning models, we do not consider training time. Once a model finishes training, it can be used multiple times, and the inferring time is independent of training process.

For hardware configurations, we use an 11th Gen Intel(R) Core(TM) i5-11400 CPU with a total of 6 cores and 12 threads, a 32GB memory, and an Nvidia RTX A6000 GPU. For software configurations, we use PyTorch[5] to

[5]PyTorch version: 1.10.0

implement all machine learning models, and Numpy[6] for calculation.

5.4.2 Evaluation dataset

This work focuses on detecting structurally-similar cross-language code pairs. To this extend, we find that there is not an appropriate dataset for evaluation. One of the most famous dataset available for cross-language *code clone detection* is AtCoder dataset organized by Perez et al. [48] and used by Mathew and Stolee [39]. It comprises solutions of competitive programming problems from a competitive programming contest platform AtCoder, and the solutions are contributed by different programmers independently. Although solutions from the same problem implement the same functionality, they might have different algorithms, structures and number of code lines. Therefore, it is not suitable for evaluation of structurally-similar cross-language code pair detection. We did rough estimation and over 70 percent cases in current dataset are not suitable.

To conduct proper evaluation, one of the authors organized a new dataset based on the original AtCoder dataset with the help of a graduate student with industrial experiences. Specifically, we look into all programming contest problems in current dataset, and select solutions written by the same author. We find that the same author tends to use the same algorithm but different programming languages for the same problem. We manually check the solutions for each problem to ensure that they have similar control structures. Our criteria include number of code lines, number of key structures such as loops and conditions, and how they are structured. As a contribution, we manage to organize a dataset with 310 pairs of code fragments written in Java and Python, 100 pairs of code fragments written in Java and C and 100 pairs of code fragments written in Python and C. All these pairs have the same functionality with similar control structures. We show the number of lines of code (LoC) in Table 5.1.

5.4.3 Results

The results are presented in Table 5.2, Table 5.3 and Table 5.4. In Java-Python experiments, our approach can successfully pick up the most structurally-similar code fragment to the target in 63% of the cases. It achieves the best accuracy in terms of AR, SR@1, SR@3 and MRR, and outperforms all other approaches including editing-distance algorithm. Meanwhile, our approach

[6]Numpy version: 1.17.2

Table 5.1: Lines of code of datasets

Metrics	Java	Python	C
Average	17	12	18
25% quantile	10	6	10
50% quantile	14	10	15
75% quantile	21	15	23

is 20 times faster for a single detection compared with editing-distance algorithm. In Java-C experiments and Python-C experiments, our approach also leads a large advance over all other approaches except editing-distance algorithm, but is very close to the tree-based editing-distance algorithm. However, compared with tree-based editing-distance algorithm, our tool is 10 times faster for a single detection, and still keeps a very close accuracy.

The results show that our tool improves the accuracy a lot compared with other approaches in detecting structurally-similar cross-language code pairs. It achieves close accuracy compared with tree-based editing-distance algorithm, but with a much faster speed. Although our model is trained on a single Nvidia RTX A6000 GPU card for around 15 hours, the model can be used to inferring cases at a large scale. Tree-based editing-distance algorithm can not be used at the same scale because the time consumption is unacceptable. Therefore, our tool is of practical use in detecting structurally-similar cross-language code pairs.

Moreover, from the ablation study, we find that using two kinds of generic ASTs improves the accuracy compared to using only one kind of generic ASTs for all language combinations. It indicates that using two kinds of generic ASTs improves the representation quality for neural network models, and helps better find code pairs with similar structures. We will provide a discussion about this in later section. We also conduct an experiment to analyze the influence of the hyper-parameter k and see how the changes of it can affect the results. The results are conducted on Java-Python experiments and are presented in Table 5.5. With the increase of k, the accuracy improves, but the time also increases. The marginal change of accuracy decreases with respect to k, and the time spent is linear to k. We notice that when k increases to a certain extend, the changes of SR and MRR is very small or even zero. Overall, there is a trade-off between accuracy and time when using our tool. We should choose an appropriate hyper-parameter k to balance the speed and accuracy.

Table 5.2: Comparison between our approach and other approaches for Java-Python pairs

Methods	AR	25% Quan.	50% Quan.	SR@1	SR@3	MRR	Time
Token-based Jaccard similarity algorithm	67	1	15	0.34	0.41	0.391	0.36s
Tree-based editing distance algorithm	**11**	**1**	**1**	**0.51**	**0.64**	**0.605**	131s
CodeBert	95	27	73	0.03	0.06	0.068	0.018s
Unixcoder	63	5	31	0.16	0.21	0.223	0.022s
Single-level (Coarse-grained generic ASTs)	32	3	14	0.14	0.25	0.238	0.021s
Single-level (Fine-grained generic ASTs)	16	2	8	0.21	0.34	0.326	0.026s
Two-level generic ASTs	12	2	6	0.24	0.41	0.366	0.047s
Coarse-grained + Editing distance (Top-15)	30	1	3	0.48	0.5	0.511	6.36s
Fine-grained + Editing distance (Top-15)	14	1	2	0.49	0.61	0.596	6.37s
Our tool (Top-15)	**11**	**1**	**1**	**0.63**	**0.67**	**0.665**	**6.39s**

Table 5.3: Comparison between our approach and other approaches for Java-C pairs

Methods	AR	25% Quan.	50% Quan.	SR@1	SR@3	MRR	Time
Token-based Jaccard similarity algorithm	28	1	10	0.32	0.42	0.393	0.14
Tree-based editing distance algorithm	**3**	**1**	**1**	**0.75**	**0.86**	**0.82**	282s
CodeBert	17	3	8	0.19	0.35	0.369	0.023s
Unixcoder	18	2	7	0.24	0.44	0.439	0.028s
Only coarse-grained generic ASTs	8	1	3	0.35	0.52	0.48	0.027s
Only fine-grained generic ASTs	10	1	5	0.31	0.43	0.41	0.029s
Two-level generic ASTs	7	1	3	0.44	0.55	0.53	0.056s
Coarse-grained + Editing distance (Top-10)	7	1	1	0.6	0.75	0.71	28.2s
Fine-grained + Editing distance (Top-10)	10	1	2	0.49	0.59	0.56	28.23s
Our tool (Top-10)	**3**	**1**	**1**	**0.73**	**0.89**	**0.81**	**28.3s**

Table 5.4: Comparison between our approach and other approaches for Python-C pairs

Methods	AR	25% Quan.	50% Quan.	SR@1	SR@3	MRR	Time
Token-based Jaccard similarity algorithm	24	1	16	0.25	0.33	0.32	0.14
Tree-based editing distance algorithm	**3**	**1**	**1**	**0.75**	**0.88**	**0.81**	354s
CodeBert	12	5	6	0.21	0.24	0.329	0.022s
Unixcoder	13	4	6	0.16	0.21	0.223	0.027s
Single-level (Coarse-grained generic ASTs)	14	2	7	0.19	0.39	0.33	0.025s
Single-level (Fine-grained generic ASTs)	12	2	7	0.22	0.36	0.34	0.028s
Two-level generic ASTs	10	2	5	0.24	0.43	0.39	0.052s
Coarse-grained + Editing distance (Top-10)	12	1	1	0.57	0.61	0.6	35.42s
Fine-grained + Editing distance (Top-10)	10	1	1	0.6	0.63	0.63	35.43s
Our tool (Top-10)	**3**	**1**	**1**	**0.76**	**0.88**	**0.82**	**35.49s**

Table 5.5: Comparison between different top-k

Methods	AR	25% Q.	50% Q.	SR@1	SR@3	MRR	Time
Top-5	15	1	6	0.47	0.49	0.51	2.16s
Top-10	14	1	1	0.57	0.61	0.61	4.27s
Top-12	13	1	1	0.6	0.64	0.63	5.12s
Top-15	11	1	1	0.63	0.67	0.67	6.39s
Top-20	11	1	1	0.7	0.78	0.75	8.5s
Top-30	10	1	1	0.75	0.81	0.80	12.72s
Top-40	9	1	1	0.75	0.82	0.80	63.43s
Top-50	9	1	1	0.75	0.83	0.80	21.18s
Top-60	9	1	1	0.76	0.84	0.81	31.74s

5.5 Discussion

We empirically show that combining a neural-network model with heuristic algorithms works well for detecting structurally-similar cross-language code pairs. Here, we provide a discussion about our generic ASTs and our two-stage approach. The results show that two kinds of generic ASTs work better than single kind of generic ASTs. In fact, this phenomenon can be seen as data augmentation. On the one hand, with only fine-grained generic ASTs, we do focus more on low-level semantic features in certain languages, but there are also some syntactic features that occur very rarely in the whole corpus. Such features may add extra noise when trying to find code pairs with similar structures. On the other hand, with only coarse-grained generic ASTs, we focus on a high-level structure of source code, but some language-specific code details are lost. For example, Python supports list comprehension that offers a shorter syntax to allow programmers to manipulate a list within one line, but such syntactic feature is ignored in coarse-grained generic ASTs. These details also help train a better model in finding code pairs with similar structures. Finding a best design of generic ASTs is subjective and difficult. We show that using two kinds of generic ASTs in an objective way also works.

We also show that a two-stage approach improves the accuracy compared with only using one stage. It is interesting that our two-stage approach works better than the pure editing-distance algorithm. This is because the editing distances of some candidates are smaller than the ground truth. After filtering in the first stage, these candidates are blocked outside top-k candidates. Therefore, the editing-distance algorithm in the second stage can success-

Table 5.6: Comparison between two-level generic ASTs with other approaches on AtCoder Dataset

Methods	AR	25% Q.	50% Q.	75% Q.	Time
Jaccard sim. algorithm	86	6	45	158	0.35s
Tree-editing-distance	97	23	66	153	121s
Two-level generic ASTs	89	15	54	149	0.05s
Coarse-grained ASTs	98	27	72	155	0.016s
Fine-grained ASTs	104	26	72	174	0.33s
CodeBert	125	47	109	204	0.019s
Unixcoder	108	25	86	185	0.022s

fully pick up the ground truth. In fact, if we enlarge the hyper-parameter k, the accuracy of our two-stage approach will not increase ultimately, but approach the accuracy of the pure editing-distance algorithm.

One threat of validity is our dataset. It contains code pairs with both similar control structures and the same functionality. Although we are targeting at structurally-similar code pairs, it is extremely hard or impossible to create a dataset with only similar structures without considering functionality. Therefore, we use this dataset for experiments. However, we conduct another experiment to prove that the same functionality has limited impact in the detection. The results are shown in Table 5.6. This experiment is conducted on Java-Python code pairs without similar control structures, but only the same functionality. This dataset is also based on the original AtCoder dataset and is manually created. We can see that all approaches work badly on these code pairs, indicating that structural dissimilarity could interfere the ability of these approaches a lot. It also indicates that the same functionality has very limited influence on these approaches in finding structurally-similar code pairs.

Further, the dataset is relatively small. Although we select the similar code pairs based on their authors, we still need human experts to read code solutions, and make final decisions. These experts have experiences of working in industries and writing code in multiple programming languages and are trustworthy. But it is still very time-consuming to create a dataset that contains structurally-similar cross-language code pairs in our work. To better validate the effectiveness of our approach against other approaches, a larger dataset is needed for massive evaluation.

Chapter 6

Retrieval of Causes of Software Regressions

In this chapter, we will present our approach for retrieval of causes of software regressions. We will start with the motivation for this task and introduce the significance. We then introduce the state-of-the-art approach, along with the limitations of this approach. To address the limitations, we present our approach with LLMs incorporated. Specifically, instead of tracing inputs and outputs as mentioned in the existing work [24], we trace only the execution order of functions and analyze the corresponding source code changes of these functions. To further investigate these changes, we propose two heuristic algorithms and one LLM-powered algorithm. Moreover, we extend our implementation and experiments to another language, Python, to further evaluate the generalizability of our algorithms.

6.1 Significance of retrieving root causes of software regressions

Software regressions have been a persistent issue in software development. Software regressions typically mean a certain type of software bugs where previously functional software features exhibit new faults and stop functioning after updates. While many works have focused on detecting software regressions or preventing regressions before release, locating the root cause of software regressions is important but challenging [54, 16], and still needs exploration.

In Figure 6.1, we show an example of a real-world regression from a JavaScript project called Express [21], which is a web framework for Node.js

providing a set of features for web and mobile applications. This example involves the version with commit id 997a558 (referred as the base version) and its subsequent commit with id c6e6203 (referred as the faulty version). In this example, the green code belongs to the base version and the red code belongs to the faulty version.

In this example, the faulty version implicitly introduces a regression by mistakenly encoding the uniform resource identifier (URI) from the arguments provided in the function. As shown in Figure 6.1, line 3 from the faulty version assigns the argument to a variable *address*. Then the *if* statements check whether there are two arguments and if the first argument is a number. If yes, the first argument is assigned to *status* as status code and the second argument is assigned to *address* as the url address. If there is only one argument, the code would not enter these conditional branches. However, later at line 20, the faulty version does not use the parsed *address* but the original argument *url* to encode the uniform resource identifier, which makes the URI incorrect. To side with the modifications in Figure 6.1, the developers also change the corresponding test case in Figure 6.2 by deleting the status code in line 7. Along with many other changes, the developers commit this change without carefully reviewing the code. This bug is identified as a regression later and gets fixed by developers at a later commit (6f91416).

In most cases, addressing the root cause of software regressions requires manually checking source code, and can be burdensome. Sometimes, even if a programmer knows which unit test fails, it is still tedious. In the given example, *res.redirect* is just a middle-ware function in *response.js* to handle web requests, while *response.js* also relates to many other functions. All these functions could be a potential candidate of the root cause of the software regression when developers are given limited information from the failures of unit tests by testing frameworks such as Mocha [14]. Moreover, the regression is usually committed with many other changes. In Table 6.1, we present an analysis of code changes between a base version and a updated but faulty version, which is collected from real-world regressions. It can be seen that in average, 7 files are changed for Python projects and 5 files are changed for JavaScript projects when regressions happen, and more than 300 lines of code are changed for both languages. It is difficult for developers to locate the root cause of a software regression among these changes manually. The table shows the number of changed files and lines of code between two consecutive commits. It is even harder if the regression is discovered after a long period of time along with numerous commits covering the regression. Given the complexity of pinpointing software regressions, an automated tool

would become greatly convenient and time-saving for developers to resolve the task efficiently.

6.2 State-of-the-art method

To the best of our knowledge, one recent work [24] targeting at locating the causes of software regressions is done by providing an automatic tool based on runtime tracing. This work traces dynamic information during program execution, including changes of arguments and return values. Specifically,

```
1  res.redirect = function(url) {
2  - var head = 'HEAD' == this.req.method;
3  + var address = url;
4    ...
5  - if (2 == arguments.length) {
6  -   if ('number' == typeof url) {
7  -     status = url;
8  -     url = arguments[1];
9  + if (arguments.length === 2) {
10 +   if (typeof arguments[0] === 'number') {
11 +     status = arguments[0];
12 +     address = arguments[1];
13     } else {
14       ...
15     }
16   }
17
18   this.format({
19     text: function(){
20       body = statusCodes[status] + '. Redirecting to
       ' + encodeURI(url);
21     },
22     ...
23   });
24
25 - ...
26 + ...
27 }
```

Figure 6.1: Base (green) and updated (red) faulty version of function *res.redirect* in response.js.

Table 6.1: Changes between commits for software regressions

Project Name	Avg. files	Avg. lines of code involved
Regressions in youtube-dl (Python)	3	31
Regressions in Tornado (Python)	20	1420
Regressions in black (Python)	2	70
All regressions in Python projects	7	384
Regressions in Express (JavaScript)	4	188
Regressions in ESLint (JavaScript)	6	655
Regressions in hessian.js (JavaScript)	7	134
All regressions of JavaScript projects	5	305

the main idea of this approach is to trace the inputs and outputs of each function invocation during the execution of both a base version and an updated but faulty version of the program when a regression occurs. By analyzing the collected dynamic information, they aim to identify discrepancies between the two versions. To achieve this, tracing statements are inserted into both the base and faulty versions of the source code to record function behaviors during runtime. Based on the recorded dynamic behaviors, a data structure

```
1  it('should include the redirect type', function(done
       ){
2    ...
3  - app.use(function(req, res){
4  -   res.redirect(301, 'http://google.com');
5  - });
6  + app.use(function(req, res){
7  +   res.redirect('http://google.com');
8  + });
9    ...
10
11 })
```

Figure 6.2: Base (green) and updated (red) faulty version of test case related to function *res.redirect*.

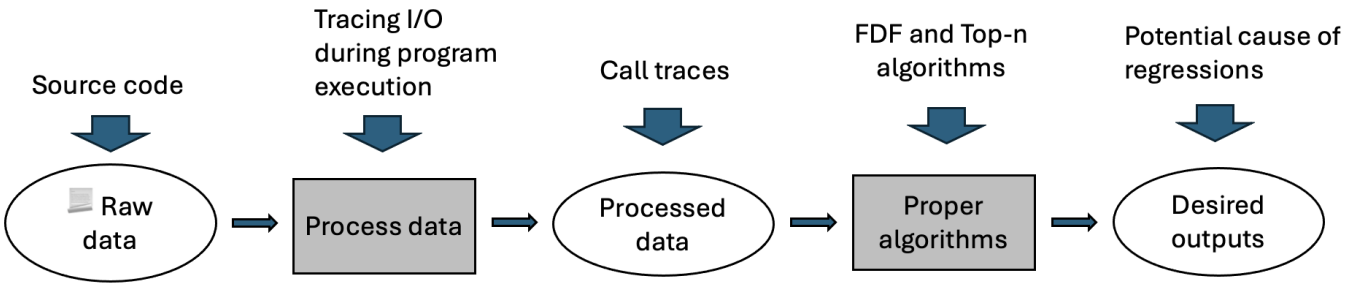
called a function call graph is constructed for each version, and the differences between the call graphs are then compared. Finally, using the identified differences, two heuristic strategies are applied to locate the root function that may have caused the software regression. Their heuristic strategies are straight-forward. The first one is called the first-deepest-function algorithm and the second one is called the top-n algorithm. The first algorithm identifies the starting point of deviated behavior of the faulty program, and the second algorithm identifies the most-occurred deviated function during execution.

They implemented their approach in JavaScript and were able to successfully address the causes of several regressions. This work demonstrates that detecting causes of software regressions by tracing software execution is possible. However, there are still some limitations about their approach. First, the accuracy of their algorithms is less satisfying. Each of their heuristics discovers only around half of real causes of regressions even by giving many candidates. This still requires developers to manually check these candidates to locate the true cause, which is trivial and time-consuming. Second, their approach only covers projects written by JavaScript and heavily depends on the serialization feature of JavaScript. It might not be easy to implement such a tracer for other languages, especially on top of the off-the-shelf engines. It is unknown how their algorithms perform when applied to projects written in other programming languages.

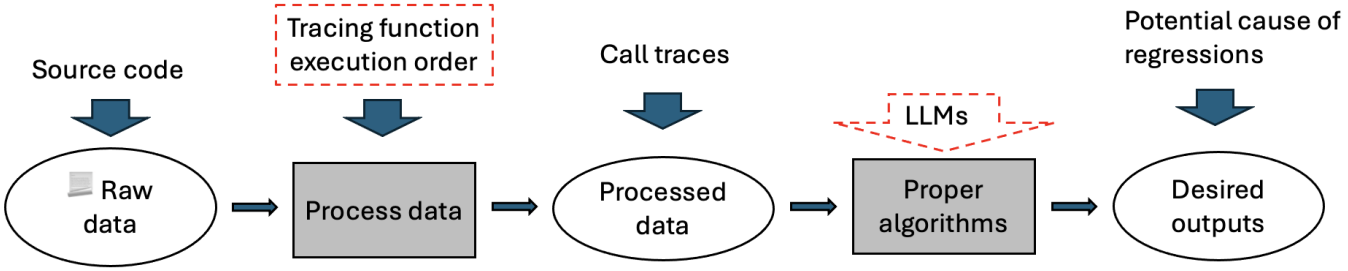
6.3 Our proposal

In this section, we present our proposed approach to overcome the limitations of the existing state-of-the-art solution. At first, we explain how our approach aligns with the general proposal of this dissertation by selectively applying LLMs to specific sub-components in the solution of this task.

The overview is illustrated in Figure 6.3. We leverage LLMs as pre-trained models to analyze the processed intermediate data, rather than directly handling large volumes of raw source code. By operating on a more informative and structured representation of the input, LLMs are relieved from the burden of extensive context and are able to perform more effectively in reasoning and comparison. With the integration of LLMs, we are able to improve the accuracy in identifying the root causes of software regressions.



(a) The workflow of the state-of-the-art approach



(b) The workflow of our proposal

Figure 6.3: How we selectively apply LLMs in the pipeline of the second task

6.3.1 System overview

We extend the existing approach [24] to address the their limitations by proposing three algorithms. Unlike the existing method, our approach traces changes in the source code itself rather than relying on changes in inputs and outputs. The system overview is shown in Figure 6.4. We first generate execution traces of both the base program and the faulty program, and then compare the execution traces. Instead of collecting inputs and outputs of functions, we only requires the execution order of functions in each program. We then analyze the corresponding source code changes of these functions. Based on the hypothesis that the cause of a software regression could be located in the differences between the base program and the faulty program, we present three algorithms to improve the accuracy compared to the existing algorithms [24]. Specifically, our algorithms include two heuristic algorithms and one LLM-powered technique. Since we do not collect functions' inputs and outputs but only trace their execution order, we implement a customized program tracer that differs from the one used in the existing work [24]. Our implementation is simpler and more compatible with various off-the-shelf engines.

It is important to note that our approach relies on a version control system, such as Git, to manage source code versions. In our scenario, a software regression is defined as a case where a previous version of the software passes a given unit test successfully, but a later updated version fails the same test due to some faulty behavior. We refer to the former version as the base version and the latter one as the faulty version. Our approach aims to resolve the root cause of such regressions.

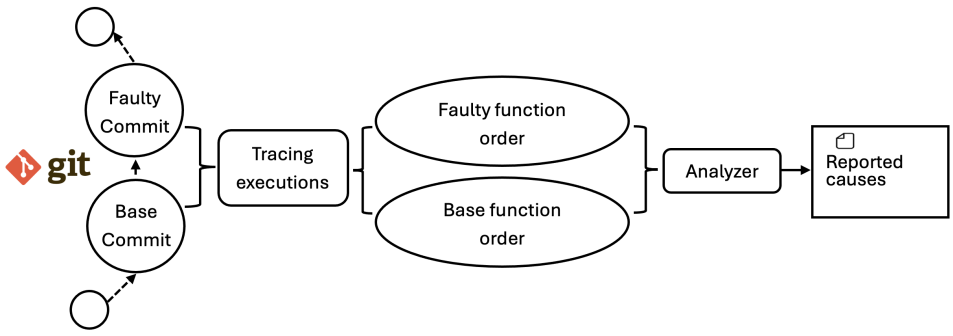


Figure 6.4: System overview

6.3.2 Trace execution order of functions

First, we need to trace the execution order of functions as dynamic information for analysis. In our approach, we trace the function execution order of the base version and the faulty version seperately. We will obtain a tree structure that represents the invoking relationships between functions in the program. In this tree, a parent node is a function to invoke others and a child node is a function being invoked by its parent within the function body. Specifically, the root node indicates the entry point of a program, which is artificially constructed. An example of the execution trace is shown in Figure 6.5. In this example, the program executes function A, B and D sequentially. Before the program executes function B, function A will invoke function B and C first. To note, the same function can be invoked multiple times by the same or by different functions.

The tracing is done by modifying the functions and inserting tracing statements into the original source code. These statements will record function order during program execution. Moreover, when modifying the functions, the original source code of each function is recorded for further comparison. In our experiments, we only modify the source code of each project itself, without considering the libraries it depends on. An example of the modification is shown in Figure 6.6. The original function is wrapped and bound to its context so that its behavior remains unaffected. We do some operations before and after its execution in order to construct a tree-like execution structure. Specifically, Python does not have a *this* context like JavaScript; instead, it uses *self* to refer to the instance within class methods. We consider such things when modifying the source code.

Our tracer supports various types of functions in a program, including module-level functions, methods, lambda functions in Python, function expressions and arrow functions in JavaScript, and so on. A unique identifier is given to each function to represent this function. This identifier includes the path of the source file, the line number in the source file, the name of the function if any, and hash value of the function definition. The hash value is calculated after pretty-printing to avoid the influence of formatting. An example is shown below:

```
tornado/options.py@line99/Func@__init__,
75f93a90c.
```

It shows that the function is defined in line 99 of the file *tornado/options.py*. It is a module-level function and its function name is $\_\_init\_\_$. The hash value of this function is *75f93a90c*.

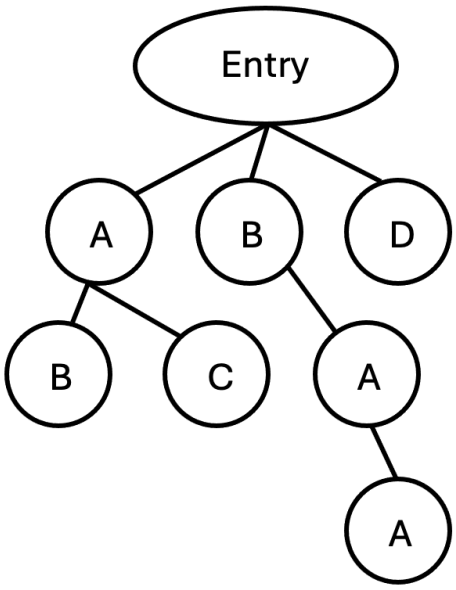


Figure 6.5: An example of the execution trace.

6.3.3 Compare execution traces

We propose three algorithms to compare two execution traces and retrieve the cause of software regressions. Two of them are based on heuristic algorithms and the other one is powered by a large language model (LLM).

Our heuristic algorithms are based on the assumption that a regression is usually caused by the changes made in a certain function. We want to locate such a change that causes the regression among various changes in various files. In order to compare two different but homologous execution traces, we set up two rules to avoid the influence of irrelevant functions. First, only a pair of nodes with the same position in traces can be compared. The same position means they have the same depth and breadth. Second, only a pair of nodes with the same function type and name can be compared. These two rules guarantee that a pair of nodes are always following the same execution logic, despite the fact that they may have different behaviour. We compare each pair of nodes in depth-first order. We give an example in Figure 6.7. In this example, the blue node C on the left means a function with a name C, and the blue node D on the right means a function with a name D. In this case, they are at the same position, but have different function names. Therefore, they are not compared although they are at the same position.

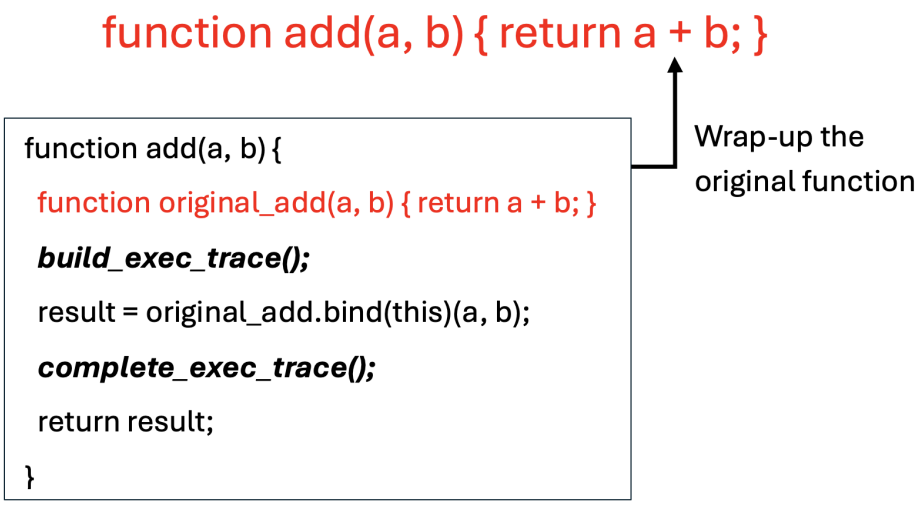
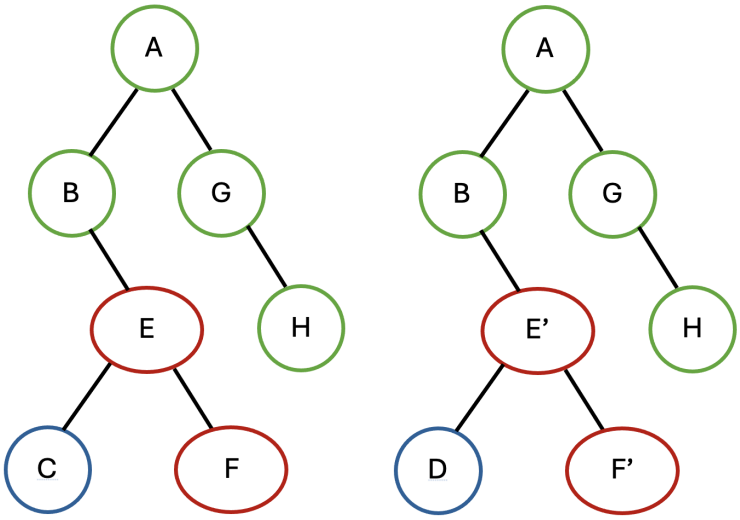


Figure 6.6: An example of tracing statement insertion.

The red nodes E and E' mean that they have the same function name E, but the source code is changed. The red nodes F and F' also have the same name F but changed source code. In this case, both function pairs, E/E' and F/F', are compared as they are at the same position and have the same name.

The other technique utilizes the power of LLMs to understand source code. Recently, LLMs have demonstrated abilities in processing source code and finishing code-related tasks, such as code completion [77]. Given two execution traces and the source code, we want to know whether LLMs can understand the logic of the programs and pinpoint the function that introduces faulty behaviour. To this extend, we design the third technique as a comparison. We will introduce our techniques in detail.

Deepest change with the same position The first heuristic technique tries to find the deepest pair of functions in execution traces that have a change in the source code. The deepest pair means that they have the deepest depth compared with other different pairs, and they do not have descendants that are different in their subtrees. If two pair has the same depth, the first pair is reported. As shown in Figure 6.7, function F and F' display discrepancy in source code and are the deepest in execution traces. This pair is reported by this technique. The reasoning behind this technique is that the deepest pair contains no further deviated behaviour in their nested



(a) The base version. (b) The faulty version.

Figure 6.7: A comparison between two callgraphs

calls to other functions, and they are responsible for the faulty behaviour between the base version and the faulty version.

An example is shown in Figure 6.8. In this example, the *create()* function from *rule-tester.js* calls the *create()* function from *no-multi-spaces.js*, and the *create()* function from *no-multi-spaces.js* calls the *Program()* function which is defined inside the *create()* function from *no-multi-spaces.js*. As the source code of the *Program()* changes, the source code of the *create()* function from *no-multi-spaces.js* changes correspondingly. By the deepest-change algorithm, the *Program()* function is considered as the cause of this regression. This is a real example from the project ESLint with bug id 307. According to the results in section 6.4, the deepest-change algorithm reports a correct cause of the regression.

First change with the same position The second heuristic technique tries to find the first pair of functions in execution traces that has a change. The first pair means that they appear first in the order of depth-first search following the previous rules. As shown in Figure 6.7, function E and E' are the first pair with the same name and the same position, but are different in source code. This pair is reported. The reasoning behind this technique is that the first chronologically different pair identifies the beginning of deviated

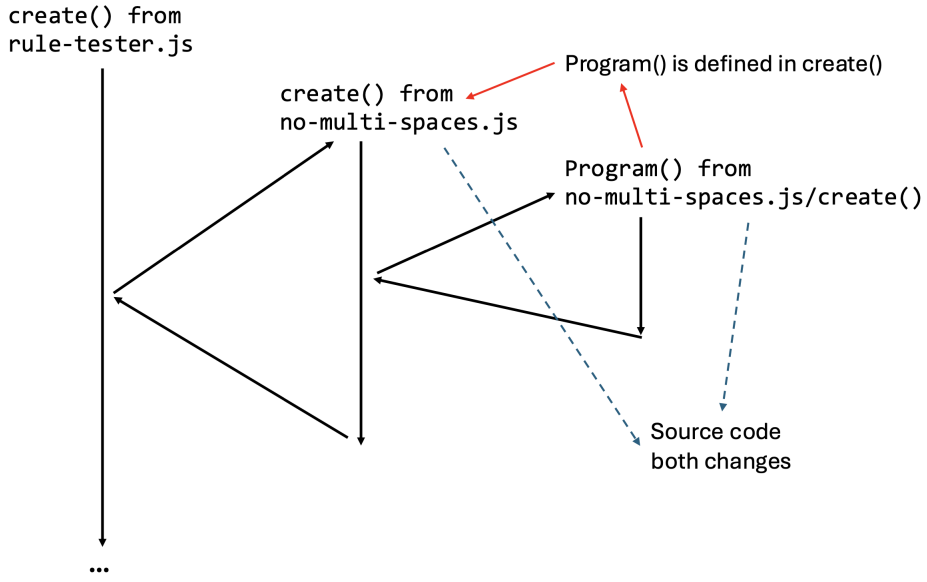


Figure 6.8: Deepest change with the same position.

behaviour between the base version and the faulty version and is the root cause of the bug.

In the example shown in Figure 6.8, the first-change algorithm should report the *create()* function from *no-multi-spaces.js* as the cause of the regression. However, this is not a correct report.

An LLM-powered technique The third technique utilizes an LLM to locate the function that causes a regression. It relies on the power of LLMs to understand source code. We want to know whether an LLM can correctly pinpoint the cause of a regression if it is provided with the source code, the order of functions executed and probably some other information. However, due to context limits of LLMs, it is not possible to input source code of the whole project into an LLM. Therefore, we use a sliding window through the linearized execution trace to reduce the context.

The basic idea is to linearize the execution trace by depth-first-search order, and extract a sub-trace from the linearized trace to give to the LLM. As shown in Figure 6.9, the execution traces in Figure 6.7 are linearized by depth-first-search order. To extract a sub-trace, we compare the functions in two linearized traces one-by-one to find the first pair of functions that are

different. This time, the pair of functions does not have to have the same name or the same position in the original tree structures. The two linearized execution traces start to deviate by such the first pair of different functions. After getting such a pair, we expand the pair into a sub-execution trace by including some pairs before and after this pair. This is for an LLM to better understand the context of function executions. In this example, the rectangle with dotted lines is the sub-execution traces we obtain. The sub-execution traces are given to an LLM to locate the cause of the regression. The window size is adjustable depending on the input size of the LLM. Also, it is possible that the root cause of a regression is not contained in the sub-execution trace. In such cases, the LLM is not able to locate the root cause of a regression. A large window size reduces the chance of occurrences of such cases. In our experiments, we include five functions before the starting point and twenty functions after the starting point.

Moreover, to help an LLM better analyze the source code, we provide the error messages of the unit test from the faulty version as an enhancement. In real development, developers usually need error messages to help them better locate where the bug happens. We wonder whether such messages can improve the accuracy of an LLM as they usually help developers. To find out, we do both experiments of using an LLM with and without the error messages. We provide the prompt of using an LLM with error messages in Figure 6.10a. As a comparison, we also provide the prompt of using an LLM without error messages in Figure 6.10b for experimental purpose.

6.3.4 Design rationale

To better solve the problem of locating causes of software regressions, we need to consider the inherent data structure of software regressions. A software regression fundamentally arises from changes introduced in the source code. However, identifying the specific change responsible for the regression among a large number of modifications goes beyond the capabilities of existing static analysis techniques. Even with the aid of text-based large language models, it is currently infeasible to statically process and reason over an entire codebase at once. Therefore, dynamic analysis with runtime information becomes necessary. Existing dynamic analysis based on function inputs and outputs is essentially a form of data-flow analysis. However, this approach can generate a large amount of potentially irrelevant information, especially when inputs contain random values or large binary files. Therefore, using function execution order as a higher-level abstraction in place of function inputs and outputs presents a viable alternative for analysis. In this work,

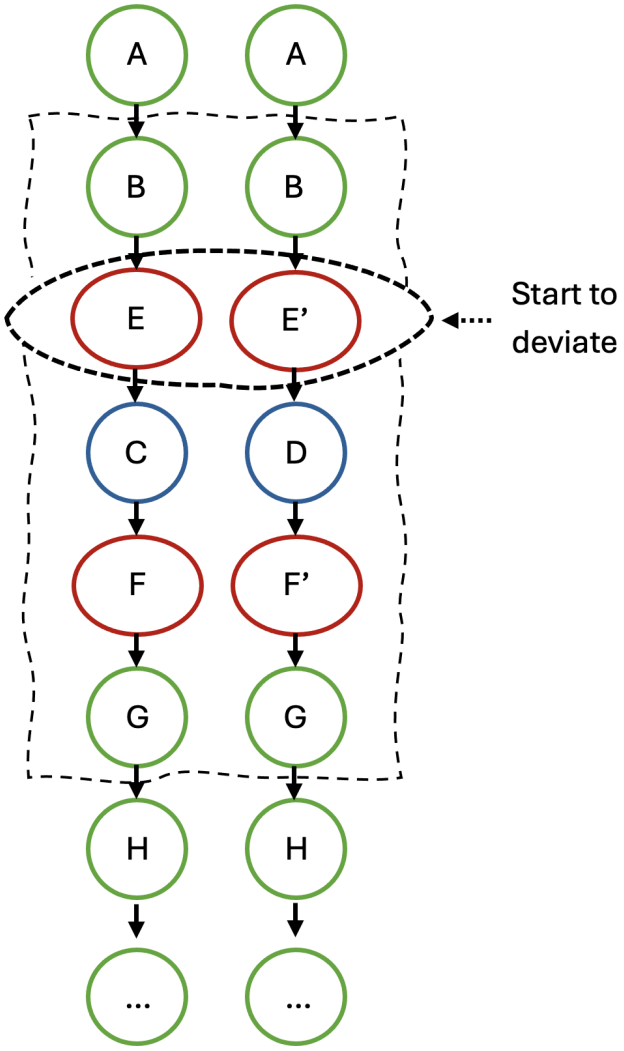


Figure 6.9: A sliding window through the linearized execution trace.

we adopt a tracer to trace execution order of functions instead of inputs and outputs to improve efficiency of locating causes of software regressions.

LLMs, on the other hand, although are not capable of analyzing an entire codebase in a static manner, by providing the execution order of functions, we can significantly reduce the search space and make it more feasible for LLMs to reason about the relevant code fragments. Therefore, after tracing execution order of functions, we provide such runtime context to LLMs

and let LLMs offer general suggestions or summarize potential issues from the differences. Our experimental results show that an LLM can achieve higher accuracy than heuristic algorithms when provided with well-curated information.

6.4 Experiments

In this work, we evaluate the accuracy and overhead of locating the cause of software regressions of our techniques against the algorithms mentioned in [24]. The authors mention two algorithms, first-deepest-function (FDF) algorithm and Top-n algorithm. First deepest function algorithm identifies the starting point of deviated behaviour of the faulty version. However, they consider inputs and outputs of functions instead of source code. Top-n algorithm counts the occurrences of each function that has different behaviour, and selects the most-occurred n functions. This algorithm requires users to spend more time investigating the candidates and determine the true cause. In this work, we choose 2 as the number of n as we think that such kind of algorithms should not provide many candidates for users to choose to avoid false-positive issue. Since [24] does not provide a Python version tracer, we implement a Python tracer to trace inputs/outputs based on the description and source code provided in the work.

In this section, we will introduce our experiment settings, datasets and experiment results.

6.4.1 Experiment settings

The experiments are conducted on a machine with an *11th Gen Intel(R) Core(TM) i7-11700 @ 2.50GHz* CPU and 32GB RAM, and the operating system is Ubuntu 20.04.4 LTS. The versions of Node.js and Python we use vary according to the requirements of each project of each respective regression. Our third technique utilizes a large language model for analysis. In this work, we use *gpt-4-turbo* API from OpenAI as our backend model as it is one of the most easily-available and state-of-the-art models on the market.

6.4.2 Evaluation dataset

Currently, there is no dataset with software regressions covering multiple programming languages. The authors in [24] construct a dataset with 12 regressions written in JavaScript based on an existing dataset BugsJS [19], which contains over 450 manually-validated JavaScript bugs from 10 projects. The

authors manually screen and extract 12 regression bugs. In this work, we reuse the dataset but remove one regression from the original dataset (The bug in Express with Bug-id 1). The reason of removal is that we find that this regressions is not caused by the function provided in the dataset. In fact, the causing function provided in the dataset is never invoked in both base version and faulty version of this regression. In addition, we re-check all other JavaScript bugs and make sure no further regressions can be found in the dataset.

Moreover, we extend the existing dataset to cover a new programming language, Python. We extract regressions based on an existing Python bug dataset BugsInPy [70], which contains over 400 real-world Python bugs from 17 Python projects. The approach we adopt to extract regressions is as follows. In detail, each bug in BugsInPy corresponds to a real Git commit, a unit test and a fix. Starting from the commit of the bug, we traverse the Git tree in reverse chronological order and execute the fixed unit test on all preceding commits. All preceding commits should report faulty. If we ever get a test success after a test failure in reverse chronological order, it could be considered a regression as the same functionality works properly before it stops working. The commit with the successful test result is the base version, while the commit with the failed test result is the faulty version. The function being updated in a later fix is identified as the cause of the regression. We manually verify the extracted regressions again to ensure the validity of the regressions, along with their corresponding unit tests and fixes.

We manage to extract 12 regressions from three projects from BugsInPy. We collect the commits of the base and faulty version, along with their fixes. In detail, 7 regressions are from a project called youtube-dl, 3 regressions are from a project called tornado and 2 regressions are from a project called black. We also collect the error messages from the faulty version as supporting materials, including the regressions from JavaScript projects. An example of an error message is shown in Figure 6.11. In total, we have 23 real-world software regressions from two different programming languages.

The number of regressions is relatively small. This is because that in a typical software development process, a software regression should be addressed before it is committed. In other words, software regressions should not appear in the version control system such as Git. Therefore, it is highly challenging to identify software regressions in publicly available Git repositories. However, this does not mean that our research topic is less important. In fact, our tool is designed to help developers fix such regressions before they are committed to the repository, which improves the quality of the soft-

ware development process. Based on this reason, we construct a dataset covering two programming languages to test the validity of our approach. Some details of these regressions are summarized in Table 6.2. The table shows the number of changed files and lines of code between two consecutive commits. Considering that regressions are often discovered several commits after they are introduced, the actual numbers are likely even higher.

6.4.3 Results

We present the experiment results in Table 6.3 and Table 6.4. From the results, we successfully locate more causes of regressions against the algorithms that trace I/O from [24]. For each regression, we demonstrate whether each technique can successfully locate the cause of the regression. In Python experiments, both our first-change algorithm and the LLM-powered technique with error messages achieve the best accuracy against other techniques. Our deepest-change algorithm can resolve 7 regressions, and first-change algorithm can resolve 9 regressions. Moreover, our LLM-powered technique resolves 7 regressions if not provided with error messages, and 9 regressions if provided with the error messages. As a comparison, the FDF algorithm from [24] can locate 6 root causes of regressions, while Top-2 algorithm can locate 8 causes.

In JavaScript experiments, our LLM-powered technique can successfully locate 9 causes of regressions when provided with error messages, which outperforms all other techniques. Our deepest-change algorithm and first-change algorithm resolve 6 and 8 regressions respectively, and also outperform existing techniques from [24]. As a comparison, the FDF algorithm resolves 6 regressions, while Top-2 algorithm only resolves 3 regressions. However, our LLM-powered technique performs differently with and without error messages. When not provided with error messages, LLM-powered technique only resolves 3 regressions.

We also present the overhead of our techniques in Table 6.5. The transformation time indicates the time for modifying the source code to insert tracing statements. The runtime of modified program indicates the time for running the unit tests. The analysis time indicates the time for analyzing the differences between the base version and the faulty version. Specifically, all reported times correspond to the combined time of the base and faulty versions. The size of trace files is also the combined size of the base and faulty version. For easier understanding, we show the aggregation time of all steps in Figure 6.12. It can be seen that in most cases, techniques that trace source code consume less time than tracing inputs/outputs. This is

Table 6.2: Number of files and lines changed in each regression

(a) Number of files and lines changed in Python regressions

Bug id	No. files changed	No. lines changed
youtube-dl-19	3	15
youtube-dl-20	3	79
youtube-dl-25	4	34
youtube-dl-26	4	34
youtube-dl-29	3	15
youtube-dl-37	3	18
youtube-dl-41	5	23
tornado-2	29	2618
tornado-9	14	155
tornado-12	19	1487
black-3	2	28
black-19	3	113

(b) Number of files and lines changed in JavaScript regressions

Bug id	No. files changed	No. lines changed
Express-8	4	35
Express-9	6	418
Express-13	5	186
Express-16	4	232
Express-18	4	63
Express-27	5	196
ESLint-10	12	1612
ESLint-134	4	93
ESLint-307	4	260
hessian.js-2	8	112
hessian.js-8	7	156

Table 6.3: The results for regressions in Python

Bug id	Tracing I/O (FDF)	Tracing I/O (Top 2)	Deepest code change	First code change	LLM only	LLM with error message
youtube-dl-19	no	yes	yes	yes	no	yes
youtube-dl-20	yes	yes	yes	yes	yes	yes
youtube-dl-25	no	yes	no	yes	yes	yes
youtube-dl-26	no	yes	no	yes	yes	yes
youtube-dl-29	yes	yes	yes	yes	yes	yes
youtube-dl-37	yes	yes	yes	yes	yes	yes
youtube-dl-41	yes	yes	yes	yes	yes	yes
tornado-2	no	no	no	no	no	no
tornado-9	yes	yes	yes	yes	yes	yes
tornado-12	no	no	no	no	no	no
black-3	yes	no	yes	yes	no	yes
black-19	no	no	no	no	no	no
total	6	8	7	9	7	9

Table 6.4: The results for regressions in JavaScript

Bug id	Bugfox [24] (FDF)	Bugfox [24] (Top 2)	Deepest code change	First code change	LLM only	LLM with error message
Express-8	yes	yes	yes	yes	no	yes
Express-9	no	no	no	yes	no	yes
Express-13	no	no	no	no	no	no
Express-16	no	no	no	yes	no	yes
Express-18	no	yes	no	yes	no	yes
Express-27	no	no	no	no	no	no
ESLint-10	yes	yes	yes	yes	yes	yes
ESLint-134	yes	no	yes	yes	no	yes
ESLint-307	yes	no	yes	no	no	yes
hessian.js-2	yes	no	yes	yes	yes	yes
hessian.js-8	yes	no	yes	yes	yes	yes
total	6	3	6	8	3	9

because less time is spent on program modification and program execution. However, in some cases, analyzing source code change might take slightly longer time than analyzing changes between inputs and outputs. Moreover, tracing source code consumes less memory than tracing I/O. This is because we do not need to record the source code every time a function gets executed. Instead, we just need to record the identifier of the function. The source code of the executed functions only needs to be saved once.

There are five regressions where all techniques fail to resolve, including two cases in tornado by Python with id 2 and 12, one case in black by Python with id 19 and two cases in Express by JavaScript with id 13 and 27. We look into these cases and find that the failures of these cases are related to large-scale refactoring and many scattered changes in single commit.

For example, both bug 2 and bug 12 from tornado introduce over a thousand changes in lines of code. These changes make the execution traces vary a lot between the base version and the faulty version. It is difficult for both heuristic algorithms and an LLM to locate the correct cause of the regression from such a lot of changes. Although the large amount of changes does not necessarily mean the impossibility of locating the causes of regressions, it surely increases the difficulty for analyzing the tracing results and locating the true cause.

Specifically, as mentioned by [24], Bugfox cannot deal with cases in which random data such as IP addresses or port numbers are generated. Such randomly-generated data will interfere the judgement of algorithms when tracing I/O. Our techniques are not influenced by these randomly-generated data. An example is the bug 9 from the project Express, which contains a lot of randomly-generated values. Our techniques successfully locate the cause of this regressions while algorithms tracing I/O fail.

6.5 Discussion

We empirically show that tracing source code change instead of tracing inputs and outputs of functions provides better accuracy in locating the cause of software regressions. Also, we empirically show that an LLM-powered technique with error messages achieves the best accuracy. In this section, we provide a discussion about our approach, as well as the threats to validity.

6.5.1 Tracing source code instead of I/O

There are several reasons about why we choose to compare source code changes instead of comparing I/O of functions. Generally, we think locating

Table 6.5: The overhead for regressions in JavaScript

Bug id	Transformation time		Runtime of modified program		Analysis time		Size of trace files	
	Tracing I/O	Tracing source code	Tracing I/O	Tracing source code	Bugfox	Our tool (LLM-powered)	Tracing I/O	Tracing source code
youtube-dl-19	49.7s	43.1s	1.6s	1.4s	1.25s	2.6s	50.9 MB	6.8 MB
youtube-dl-20	35.8s	32.3s	0.2s	0.2s	0.24s	2.3s	3 MB	0.34 MB
youtube-dl-25	43.4s	38.5s	0.7s	0.5s	0.31s	3.0s	5.4 MB	0.86 MB
youtube-dl-26	44.3s	39.1s	0.7s	0.4s	0.18s	3.1s	5.4 MB	0.84 MB
youtube-dl-29	36.1s	32.6s	1.0s	0.5s	0.28s	1.5s	4.6 MB	0.36 MB
youtube-dl-37	26.5s	24.2s	1.3s	0.6s	0.27s	1.6s	2.2 MB	0.04 MB
youtube-dl-41	26.7s	25.6s	0.6s	0.2s	0.13s	2.7s	1.9 MB	0.08 MB
tornado-2	25.8s	24.1s	26.1s	5.7s	6.07s	3.2s	253.8 MB	2.5 MB
tornado-9	36.6s	29.1s	0.8s	0.3s	0.93s	0.8s	33.4 MB	0.56 MB
tornado-12	22.1s	21.1s	2.1s	0.7s	1.09s	3.5s	51.8 MB	2 MB
black-3	8.2s	6.7s	3.4s	0.7s	0.91s	1.6s	53.4 MB	0.49 MB
black-19	4.1s	3.6s	19.2s	1.1s	10.73s	3.7s	670.1 MB	0.36 MB
Express-8	4.2 s	3.9s	16.2s	8.3s	1.46s	3.4s	125 MB	0.27 MB
Express-9	4.1 s	4.1s	3.1s	1.2s	0.59s	2.4s	41 MB	0.34 MB
Express-13	3.5 s	3.4s	17.1s	11.1s	1.85s	3.2s	93 MB	0.31 MB
Express-16	2.7 s	2.2s	20.5s	10.2s	2.86s	1.7s	132 MB	0.29 MB
Express-18	3.9 s	3.7s	12.6s	7.2s	1.44s	2.5s	80 MB	0.36 MB
Express-27	3.6 s	3.5s	21.4s	15.8s	3.26s	2.4s	99 MB	0.58 MB
ESLint-10	17.9 s	15.9s	1.1s	0.8s	0.43s	1.4s	24 MB	0.94 MB
ESLint-134	15.5 s	15.2s	2.3s	1.6s	0.56s	2.4s	7 MB	3.6 MB
ESLint-307	18.2 s	17.5s	1.1s	0.9s	0.55s	2.2s	8 MB	1.5 MB
hessian.js-2	1.2 s	1.1s	1.2s	0.9s	0.47s	1.9s	39 MB	0.08 MB
hessian.js-8	1.1 s	1.1s	1.5s	0.6s	0.54s	1.8s	41 MB	0.16 MB

the cause of a software regression is about locating the key change in source code that leads to the malfunctioning of a feature. First, some functions do not have an explicit input or output. As an example, the function provided in Section **??** does not have an explicit return value. In such cases, the inputs or outputs never change and will cause false positives when analyzing the changes of I/O. Such cases often happen when a function modifies some global values or tries to pass wrong arguments to another function. When a function invokes another function but passes wrong arguments to that function, the I/O of the invoked function changes. The invoked function is reported as the cause of the regression by tracing I/O. However, this is wrong. The function that passes wrong arguments, instead of the function being invoked, should be blamed. In this case, tracing inputs/outputs would report a wrong cause of the regression, which is not what we expect. The algorithms mentioned in [24] do not consider such cases where inputs and outputs never change.

Second, tracing inputs and outputs may generate large tracing logs. Our statistics in Table 6.5 demonstrates the reduction in memory usage achieved by tracing source code. Some benefits are introduced by smaller tracing logs. One benefit is the time consumption. It can be seen that analyzing source code spends less time than analyzing I/O to locate the causes of regressions in most cases. Another benefit is that analyzing source code improves the accuracy. For example, the regression from Express with bug id 16 generates a large tracing log when tracing I/O, but a relatively smaller one when tracing source code. However, both algorithms analyzing I/O fail while our algorithm analyzing source code succeeds in locating the cause.

Third, as mentioned in [24], randomly-generated values sometimes prevent tracing I/O from successfully locating the true causes. Tracing source code is not affected by such randomly-generated values.

Moreover, tracing inputs and outputs is dependent on the serialization feature of each programming language. However, it may not be friendly to some off-the-shelf engines. For example, a generator function in Python is difficult to be serialized without changing its inner states, thus difficult to be traced. Without tracing such features, the accuracy may be compromised. Although we can develop some techniques to bypass these limits, it is out of the scope of this work. In this work, we intend to develop a lightweight and easy-to-use tool for detecting the causes of software regressions. Modifying compilers or interpreters would be a harder and more systematic job, which could be the future work of this work.

Although we prefer source code changes instead of input/output changes, it is still necessary to trace program execution dynamically because we need

the execution order of functions to locate the causes of regressions. In this task, the execution order helps us exclude some functions with source code changes but are irrelevant to the current test case, and helps us better locate a true cause of the regression. In other words, a dynamic execution order helps narrow the decision space.

6.5.2 The LLM-powered technique

A large language model combined with error messages shows better accuracy in locating the causes of regressions. From Table 6.3 and Table 6.4, it can be seen that most regressions can be resolved by our deepest-change-with-the-same-position and first-change-with-the-same-position algorithms. However, when these two algorithms do not align with each other, it is difficult to make a decision about the true cause. Large language models have demonstrated abilities in understanding source code [42], and dominated many code-related tasks [43, 37]. In this regard, we try to examine whether utilizing an LLM to help determine the true cause of software regressions is feasible.

Simply adopting an LLM is arbitrary and does not provide good accuracy. We provide error messages as supporting materials to an LLM, and find out that it achieves better accuracy than heuristic algorithms. The experiment shows that naively using an LLM is not enough for locating the true causes of regressions. Error messages help an LLM better analyze the source code and make the decisions.

The window size of the sub-execution trace is another concern. From our experiements, we find that a smaller window size would decrease the accuracy when the nested function call is long. However, a larger window size does not necessarily increase the accuracy. When the window size is as large as 10 before the starting point and 50 after the starting point, the accuracy decreases again due to LLM hallucination. Our choice of window size is based on our experiment results. A systematic analysis of the window size and prompting engineering might be the future work.

6.5.3 Threats to validity

There are some threats to validity of our work. First of all, the number of regressions in our dataset is small. Although we cover two programming languages and collect these regressions from real-world projects, it is still limited to evaluate tools specifically for locating the causes of regressions. This is due to the difficulty of identifying software regressions in publicly available datasets. Further benchmarking with more real-world regressions as well as

more programming languages is needed to conduct a more comprehensive evaluation.

Second, there are still some regressions that our techniques cannot resolve. One reason is the large-scale refactoring. When there are massive scattered changes and the execution traces vary a lot, it is difficult for both heurictic algorithms and LLMs to locate the true causes. Also, semantically-equivalent changes may cause false positives by our algorithms. Although we empirically show that our techniques perform better than previous techniques, the boundary conditions of our techniques associated with these refactorings require further research.

```
You will be given a list of function
call pairs and an error message
{error_messages}. In each pair, one
is a function call from the correct
version, and the other one is a
function call from the faulty version.
The list is in the order of function
calls. Based on the list and the error
message, please tell me which function
might be the cause of the regression
bug.
The list is
{str(list_of_function_pairs)}.
Please be as precise as possible, and
just give me the function name.

```

(a) Prompts with error messages

```
You will be given a list of function
call pairs. In each pair, one is a
function call from the correct version,
and the other one is a function call
from the faulty version. The list is
in the order of function calls. Based
on the list, please tell me which
function might be the cause of the
regression bug.
The list is
{str(list_of_function_pairs)}.
Please be as precise as possible, and
just give me the function name.

```

(b) Prompt without error messages as a comparison

Figure 6.10: An example of prompts in finding the cause of software regressions

```
1) app should support empty string path:
  TypeError: Router.use() requires callback function but
   got a [object String]
    at /.../lib/router/index.js:389:17
    at /.../lib/router/index.js:405:113
    at Array.forEach (<anonymous>)
    at Function.INNERFUNC (/.../lib/router/index.js
   :384:15)
    at Function.use (/.../lib/router/index.js:418:45)
    at /.../lib/application.js:174:25
    at /.../lib/application.js:218:113
    at Array.forEach (<anonymous>)
    at Function.INNERFUNC (/.../lib/application.js:171:9)
    at Function.use (/.../lib/application.js:231:45)
    at context.Original (/.../test/app.use.js:10:13)
    at context.<anonymous> (/.../test/app.use.js:32:113)
    at Runnable.run (/.../node_modules/mocha/lib/runnable
   .js:217:15)
    at Runner.runTest (/.../node_modules/mocha/lib/runner
   .js:373:10)
    at /.../node_modules/mocha/lib/runner.js:451:12
    at next (/.../node_modules/mocha/lib/runner.js
   :298:14)
    at /.../node_modules/mocha/lib/runner.js:308:7
    at next (/.../node_modules/mocha/lib/runner.js
   :246:23)
    at Immediate.<anonymous> (/.../node_modules/mocha/lib
   /runner.js:275:5)
    at process.processImmediate (node:internal/timers
   :476:21)

```

Figure 6.11: An example of an error message

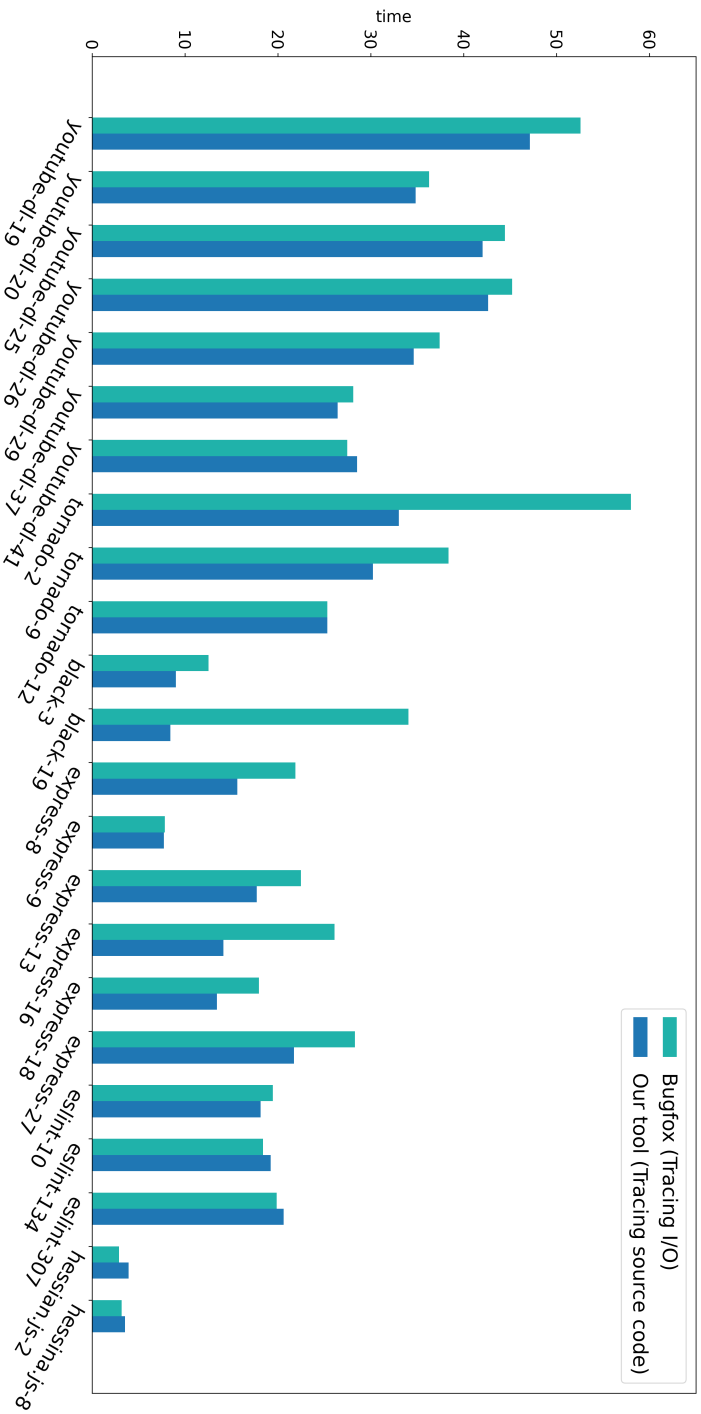


Figure 6.12: The aggregation time for analyzing a regression.

Chapter 7

Conclusions

In this dissertation, we present our proposal for effectively leveraging LLMs in software engineering problems. As discussed in chapter 3, our approach is to decompose the original task and selectively apply LLMs to particular subtasks.

Our approach integrates the traditional pipeline for solving software engineering problems with the use of LLMs, guided by the data structure of the original task. Specifically, we decompose the task into stages of raw data processing and the application of appropriate algorithms to intermediate representations. LLMs are employed either as tools to process raw data or as reasoning engines to analyze intermediate data, thereby mitigating their limitations and making the problems more manageable for them. By improving LLM performance on individual subtasks, we aim to enhance the overall effectiveness on the original task. To validate our proposal, we select two practical representative tasks as case studies.

In the first task, we improve the speed of detecting structurally similar cross-language code pairs between Java, Python, and C (in pairwise combinations), while maintaining accuracy comparable to the state-of-the-art approach. Our solution is a two-stage method: the first stage employs neural network models to identify a small number of high-probability candidate pairs, and the second stage applies a deterministic edit distance algorithm to select the most similar code fragment. To enhance the performance of the neural models, we introduce a two-level generic AST representation as model input. More importantly, in designing the two-level generic ASTs, we leverage LLMs to reduce subjective bias in the design process. We evaluate our approach on Java–Python, Java–C, and Python–C code pairs, and the experimental results demonstrate that our method is effective and general-

izable across these language combinations.

In the second task, we provide techniques to improve the accuracy of locating the root cause of software regressions compared with the existing approach. First, we extend the existing dataset from JavaScript to Python for better comparison, and conduct the experiments on both languages. We achieve the improvement by tracing the order of function execution dynamically, and comparing the source code of functions in the execution trace. In detail, we provide three techniques. Two of them are based on heuristic algorithms, namely the first code change with the same position and the deepest code change with the same position. The other one is powered by a large language model combined with error messages. Specifically, our LLM-powered technique achieves the best accuracy compared with other techniques on both JavaScript dataset and Python dataset.

Both tasks empirically demonstrate that selectively applying LLMs to sub-tasks can improve the overall performance of the original task. Instead of letting LLMs deal with the whole task, we propose to use LLMs as a powerful but backend conversational engine and simply extract relevant knowledge from LLMs. Our proposal provides a foundation for future research in this area, such as the development of AI agents for effectively and efficiently solving software engineering problems.

Future Work

The experimental results of the two tasks provide preliminary evidence supporting the effectiveness of our proposal. To further validate and generalize our approach, we aim to identify and formally describe the common characteristics shared by these example tasks. By doing so, we hope to derive general principles that reveal the typical properties of software engineering problems that current LLMs struggle to solve effectively. In addition, we aim to systematically investigate how task reformulation and decomposition can improve the effectiveness of LLMs in solving software engineering tasks. This may require additional case studies and a deeper investigation into the inherent characteristics of software engineering tasks.

For the first task, addressing structurally-similar cross-language code search more effectively requires larger and more diverse datasets. This includes not only increasing the number of code pairs but also incorporating a wider range of programming languages. Currently, our two-level generic ASTs are designed in a pairwise manner for specific language combinations. An important direction for future work is to explore whether it is possible to

design a unified generic AST representation that can simultaneously support three or more languages.

For the second task, in order to more effectively address the problem of locating the causes of software regressions, larger and more diverse datasets are needed to further validate the effectiveness of our approach. Additionally, our current method still faces challenges in handling large-scale software refactorings. An important direction for future research is to explore how to effectively integrate dynamic analysis with LLM-based static code understanding, in order to improve the robustness and generalizability of regression localization techniques.

Bibliography

[1] Miltiadis Allamanis, Hao Peng, and Charles Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code". In: *International Conference on Machine Learning*. PMLR. 2016, pp. 2091–2100.

[2] Uri Alon et al. "Code2vec: Learning Distributed Representations of Code". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290353. URL: https://doi.org/10.1145/3290353.

[3] Kent Beck et al. *JUnit - a programmer-friendly testing framework for Java and the JVM*. https://junit.org/junit5/. 2024.

[4] Stefan Bellon et al. "Comparison and Evaluation of Clone Detection Tools". In: *IEEE Transactions on Software Engineering* 33.9 (2007), pp. 577–591. DOI: 10.1109/TSE.2007.70725.

[5] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: 2005.14165 [cs.CL]. URL: https://arxiv.org/abs/2005.14165.

[6] Zoran Budimac, Gordana Rakic, and Miloš Savic. "SSQSA Architecture". In: *Proceedings of the Fifth Balkan Conference in Informatics*. BCI '12. Novi Sad, Serbia: Association for Computing Machinery, 2012, pp. 287–290. ISBN: 9781450312400. DOI: 10.1145/2371316.2371380. URL: https://doi.org/10.1145/2371316.2371380.

[7] Xiao Cheng et al. "CLCMiner: Detecting Cross-Language Clones without Intermediates". In: *IEICE Transactions on Information and Systems* E100.D (Feb. 2017), pp. 273–284. DOI: 10.1587/transinf.2016EDP7334.

[8] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734. DOI: 10.3115/v1/D14-1179. URL: https://aclanthology.org/D14-1179.

[9] Jacob Devlin et al. *BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding*. 2019. arXiv: 1810.04805 [cs.CL]. URL: https://arxiv.org/abs/1810.04805.

[10] Shihan Dou et al. *Towards Understanding the Capability of Large Language Models on Code Clone Detection: A Survey*. 2023. arXiv: 2308.01191 [cs.SE]. URL: https://arxiv.org/abs/2308.01191.

[11] Sebastian Elbaum, Gregg Rothermel, and John Penix. "Techniques for improving regression testing in continuous integration development environments". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2014. Hong Kong, China: Association for Computing Machinery, 2014, pp. 235–245. ISBN: 9781450330565. DOI: 10.1145/2635868.2635910. URL: https://doi.org/10.1145/2635868.2635910.

[12] Zhiyu Fan et al. "Automated Repair of Programs from Large Language Models". In: *Proceedings of the 45th International Conference on Software Engineering*. ICSE '23. Melbourne, Victoria, Australia: IEEE Press, 2023, pp. 1469–1481. ISBN: 9781665457019. DOI: 10.1109/ICSE48619.2023.00128. URL: https://doi.org/10.1109/ICSE48619.2023.00128.

[13] Zhangyin Feng et al. "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*. Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547. DOI: 10.18653/v1/2020.findings-emnlp.139. URL: https://aclanthology.org/2020.findings-emnlp.139.

[14] OpenJS Foundation. *Mocha - the fun, simple, flexible test framework*. https://mochajs.org/. 2024.

[15] Todd L. Graves et al. "An empirical study of regression test selection techniques". In: *ACM Trans. Softw. Eng. Methodol.* 10.2 (Apr. 2001), pp. 184–208. ISSN: 1049-331X. DOI: 10.1145/367008.367020. URL: https://doi.org/10.1145/367008.367020.

[16] Renan Greca, Breno Miranda, and Antonia Bertolino. "State of Practical Applicability of Regression Testing Research: A Live Systematic Literature Review". In: *ACM Comput. Surv.* 55.13s (July 2023). ISSN: 0360-0300. DOI: 10.1145/3579851. URL: https://doi.org/10.1145/3579851.

[17] Daya Guo et al. "UniXcoder: Unified Cross-Modal Pre-training for Code Representation". In: Jan. 2022, pp. 7212–7225. DOI: 10.18653/v1/2022.acl-long.499.

[18] Aakanshi Gupta and Bharti Suri. "A Survey on Code Clone, Its Behavior and Applications". In: Jan. 2018, pp. 27–39. ISBN: 978-981-10-4599-8. DOI: 10.1007/978-981-10-4600-1_3.

[19] Peter Gyimesi et al. "BugsJS: a Benchmark of JavaScript Bugs". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. 2019, pp. 90–101. DOI: 10.1109/ICST.2019.00019.

[20] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Computation* 9.8 (1997), pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[21] TJ Holowaychuk, StrongLoop, et al. *Express - fast, unopinionated, minimalist web framework for Node.js*. https://expressjs.com/. 2024.

[22] Soneya Binta Hossain et al. "A Deep Dive into Large Language Models for Automated Bug Localization and Repair". In: *Proc. ACM Softw. Eng.* 1.FSE (July 2024). DOI: 10.1145/3660773. URL: https://doi.org/10.1145/3660773.

[23] Xinyi Hou et al. "Large Language Models for Software Engineering: A Systematic Literature Review". In: *ACM Trans. Softw. Eng. Methodol.* 33.8 (Dec. 2024). ISSN: 1049-331X. DOI: 10.1145/3695988. URL: https://doi.org/10.1145/3695988.

[24] Yuefeng Hu et al. "Bugfox: A Trace-Based Analyzer for Localizing the Cause of Software Regression in JavaScript". In: *Proceedings of the 17th ACM SIGPLAN International Conference on Software Language Engineering*. SLE '24. Pasadena, CA, USA: Association for Computing Machinery, 2024, pp. 224–233. ISBN: 9798400711800. DOI: 10.1145/3687997.3695648. URL: https://doi.org/10.1145/3687997.3695648.

[25] Juyong Jiang et al. *A Survey on Large Language Models for Code Generation*. 2024. arXiv: 2406.00515 [cs.CL]. URL: https://arxiv.org/abs/2406.00515.

[26] Lingxiao Jiang et al. "DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones". In: *29th International Conference on Software Engineering (ICSE'07)*. 2007, pp. 96–105. DOI: 10.1109/ICSE.2007.30.

[27] T. Kamiya, S. Kusumoto, and K. Inoue. "CCFinder: a multilinguistic token-based code clone detection system for large scale source code". In: *IEEE Transactions on Software Engineering* 28.7 (2002), pp. 654–670. DOI: 10.1109/TSE.2002.1019480.

[28] Jared Kaplan et al. *Scaling Laws for Neural Language Models*. 2020. arXiv: 2001.08361 [cs.LG]. URL: https://arxiv.org/abs/2001.08361.

[29] Rafaqut Kazmi et al. "Effective Regression Test Case Selection: A Systematic Literature Review". In: *ACM Comput. Surv.* 50.2 (May 2017). ISSN: 0360-0300. DOI: 10.1145/3057269. URL: https://doi.org/10.1145/3057269.

[30] Muhammad Khatibsyarbini et al. "Test case prioritization approaches in regression testing". In: *Inf. Softw. Technol.* 93.C (Jan. 2018), pp. 74–93. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2017.08.014. URL: https://doi.org/10.1016/j.infsof.2017.08.014.

[31] Rodney Kizito et al. "Long Short-Term Memory Networks for Facility Infrastructure Failure and Remaining Useful Life Prediction". In: *IEEE Access* PP (May 2021), pp. 1–1. DOI: 10.1109/ACCESS.2021.3077192.

[32] N.A. Kraft, B.W. Bonds, and R.K. Smith. "Cross-Language Clone Detection". In: *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE'08)*. San Francisco, CA, USA, July 2008, pp. 54–59.

[33] An Ngoc Lam et al. "Bug Localization with Combination of Deep Learning and Information Retrieval". In: *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 2017, pp. 218–229. DOI: 10.1109/ICPC.2017.24.

[34] Liuqing Li et al. "CCLearner: A Deep Learning-Based Clone Detection Approach". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 249–260. DOI: 10.1109/ICSME.2017.46.

[35] Zhengliang Li et al. "LLM-BL: Large Language Models are Zero-Shot Rankers for Bug Localization". In: *2025 IEEE/ACM 33rd International Conference on Program Comprehension (ICPC)*. Los Alamitos, CA, USA: IEEE Computer Society, Apr. 2025, pp. 548–559. DOI: 10.1109/ICPC66645.2025.00064. URL: https://doi.ieeecomputersociety.org/10.1109/ICPC66645.2025.00064.

[36] Jing Liu et al. *Can LLM Generate Regression Tests for Software Commits?* 2025. arXiv: 2501.11086 [cs.SE]. URL: https://arxiv.org/abs/2501.11086.

[37] Qinyu Luo et al. *RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation*. 2024. arXiv: 2402.16667 [cs.CL].

[38] Djordje Maksimovic, Andreas Veneris, and Zissis Poulos. "Clustering-based revision debug in regression verification". In: *2015 33rd IEEE International Conference on Computer Design (ICCD)*. 2015, pp. 32–37. DOI: 10.1109/ICCD.2015.7357081.

[39] George Mathew and Kathryn T. Stolee. "Cross-Language Code Search Using Static and Dynamic Analyses". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2021. Athens, Greece: Association for Computing Machinery, 2021, pp. 205–217. DOI: 10.1145/3468264.3468538. URL: https://doi.org/10.1145/3468264.3468538.

[40] Tomas Mikolov et al. *Efficient Estimation of Word Representations in Vector Space*. 2013. arXiv: 1301.3781 [cs.CL]. URL: https://arxiv.org/abs/1301.3781.

[41] Kawser Wazed Nafi et al. "CLCDSA: Cross Language Code Clone Detection using Syntactical Features and API Documentation". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering*. 2019, pp. 1026–1037. DOI: 10.1109/ASE.2019.00099.

[42] Daye Nam et al. "Using an LLM to Help With Code Understanding". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3639187. URL: https://doi.org/10.1145/3597503.3639187.

[43] Ansong Ni et al. *L2CEval: Evaluating Language-to-Code Generation Capabilities of Large Language Models*. 2023. arXiv: 2309.17446 [cs.CL].

[44] Farouq Al-omari et al. "Detecting Clones Across Microsoft .NET Programming Languages". In: Oct. 2012, pp. 405–414. ISBN: 978-1-4673-4536-1. DOI: 10.1109/WCRE.2012.50.

[45] Long Ouyang et al. *Training language models to follow instructions with human feedback*. 2022. arXiv: 2203.02155 [cs.CL]. URL: https://arxiv.org/abs/2203.02155.

[46] Chris Parnin and Alessandro Orso. "Are automated debugging techniques actually helping programmers?" In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 199–209. ISBN: 9781450305624. DOI: 10.1145/2001420.2001445. URL: https://doi.org/10.1145/2001420.2001445.

[47] Fabrizio Pastore, Leonardo Mariani, and Alberto Goffi. "RADAR: A tool for debugging regression problems in C/C++ Software". In: *2013 35th International Conference on Software Engineering (ICSE)*. 2013, pp. 1335–1338. DOI: 10.1109/ICSE.2013.6606711.

[48] Daniel Perez and Shigeru Chiba. "Cross-language Clone Detection by Learning over Abstract Syntax Trees". In: *Proceedings of the 16th International Conference on Mining Software Repositories*. MSR '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 518–528. DOI: 10.1109/MSR.2019.00078. URL: https://doi.org/10.1109/MSR.2019.00078.

[49] Ruchir Puri et al. *CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks*. 2021. arXiv: 2105.12655 [cs.SE]. URL: https://arxiv.org/abs/2105.12655.

[50] Yihao Qin et al. *AgentFL: Scaling LLM-based Fault Localization to Project-Level Context*. 2025. arXiv: 2403.16362 [cs.SE]. URL: https://arxiv.org/abs/2403.16362.

[51] Gordana Rakic and Zoran Budimac. "Introducing enriched concrete syntax trees". In: *Proc. of the 14th International Multiconference on Information Society (IS), Collaboration, Software And Services In Information Society (CSS)* A (Jan. 2011), pp. 211–214.

[52] Veselin Raychev, Pavol Bielik, and Martin Vechev. "Probabilistic Model for Code with Decision Trees". In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2016. Amsterdam, Netherlands: Association for Computing Machinery, 2016, pp. 731–747. DOI: 10.1145/2983990.2984041. URL: https://doi.org/10.1145/2983990.2984041.

[53] Saif Ur Rehman Khan et al. "A Systematic Review on Test Suite Reduction: Approaches, Experiment's Quality Evaluation, and Guidelines". In: *IEEE Access* 6 (2018), pp. 11816–11841. DOI: 10.1109/ACCESS.2018.2809600.

[54] Raul H. Rosero, Omar S. Gomez, and Glen Rodriguez. "15 Years of Software Regression Testing Techniques — A Survey". In: *International Journal of Software Engineering and Knowledge Engineering* 26.05 (2016), pp. 675–689. DOI: 10.1142/S0218194016300013. eprint: https://doi.org/10.1142/S0218194016300013. URL: https://doi.org/10.1142/S0218194016300013.

[55] Chanchal K. Roy and James R. Cordy. "NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization". In: *2008 16th IEEE International Conference on Program Comprehension*. 2008, pp. 172–181. DOI: 10.1109/ICPC.2008.41.

[56] Chanchal Kumar Roy and James R Cordy. "A survey on software clone detection research". In: *Queen's School of Computing TR* 541.115 (2007), pp. 64–68.

[57] Gabriel Ryan et al. "Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM". In: *Proc. ACM Softw. Eng.* 1.FSE (July 2024). DOI: 10.1145/3643769. URL: https://doi.org/10.1145/3643769.

[58] Hitesh Sajnani et al. "SourcererCC: Scaling Code Clone Detection to Big-Code". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016, pp. 1157–1168. DOI: 10.1145/2884781.2884877.

[59] Tim Sonnekalb et al. "Generalizability of Code Clone Detection on CodeBERT". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. New York, NY, USA: Association for Computing Machinery, 2023. ISBN: 9781450394758. URL: https://doi.org/10.1145/3551349.3561165.

[60] Hema Srikanth and Myra B. Cohen. "Regression testing in Software as a Service: An industrial case study". In: *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. 2011, pp. 372–381. DOI: 10.1109/ICSM.2011.6080804.

[61] Marc Szafraniec et al. *Code Translation with Compiler Representations*. 2023. arXiv: 2207.03578 [cs.PL]. URL: https://arxiv.org/abs/2207.03578.

[62] Chenning Tao et al. "C4: Contrastive Cross-Language Code Clone Detection". In: *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*. ICPC '22. Virtual Event: Association for Computing Machinery, 2022, pp. 413–424. ISBN: 9781450392983. DOI: 10.1145/3524610.3527911. URL: https://doi.org/10.1145/3524610.3527911.

[63] pytest-dev Team. *pytest: helps you write better programs*. https://pytest.org. 2024.

[64] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[65] Tijana Vislavski et al. "LICCA: A tool for cross-language clone detection". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering*. 2018, pp. 512–516. DOI: 10.1109/SANER.2018.8330250.

[66] Ellen M. Voorhees and Dawn M. Tice. "The TREC-8 Question Answering Track". In: *Proceedings of the Second International Conference on Language Resources and Evaluation (LREC'00)*. Ed. by M. Gavrilidou et al. Athens, Greece: European Language Resources Association (ELRA), May 2000. URL: http://www.lrec-conf.org/proceedings/lrec2000/pdf/26.pdf.

[67] Simin Wang et al. "Machine/Deep Learning for Software Engineering: A Systematic Literature Review". In: *IEEE Trans. Softw. Eng.* 49.3 (Mar. 2023), pp. 1188–1231. ISSN: 0098-5589. DOI: 10.1109/TSE.2022.3173346. URL: https://doi.org/10.1109/TSE.2022.3173346.

[68] Hui-Hui Wei and Ming Li. "Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code". In: *Proceedings of the 26th International Joint Conference on Artificial Intelligence*. IJCAI'17. Melbourne, Australia: AAAI Press, 2017, pp. 3034–3040. ISBN: 9780999241103.

[69] Jason Wei et al. *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. 2023. arXiv: 2201.11903 [cs.CL]. URL: https://arxiv.org/abs/2201.11903.

[70] Ratnadira Widyasari et al. "BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association

for Computing Machinery, 2020, pp. 1556–1560. ISBN: 9781450370431. DOI: 10.1145/3368089.3417943. URL: https://doi.org/10.1145/3368089.3417943.

[71] Yi Wu et al. "How Effective Are Neural Networks for Fixing Security Vulnerabilities". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2023. Seattle, WA, USA: Association for Computing Machinery, 2023, pp. 1282–1294. ISBN: 9798400702211. DOI: 10.1145/3597926.3598135. URL: https://doi.org/10.1145/3597926.3598135.

[72] Yonghao Wu et al. *Large Language Models in Fault Localisation*. 2023. arXiv: 2308.15276 [cs.SE]. URL: https://arxiv.org/abs/2308.15276.

[73] Aidan Z. H. Yang et al. "Large Language Models for Test-Free Fault Localization". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE '24. Lisbon, Portugal: Association for Computing Machinery, 2024. ISBN: 9798400702174. DOI: 10.1145/3597503.3623342. URL: https://doi.org/10.1145/3597503.3623342.

[74] Jian Zhang et al. "A Novel Neural Source Code Representation Based on Abstract Syntax Tree". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019, pp. 783–794. DOI: 10.1109/ICSE.2019.00086.

[75] Yuanliang Zhang et al. *Unseen Horizons: Unveiling the Real Capability of LLM Code Generation Beyond the Familiar*. 2025. arXiv: 2412.08109 [cs.SE]. URL: https://arxiv.org/abs/2412.08109.

[76] Zixian Zhang and Takfarinas Saber. *Assessing the Code Clone Detection Capability of Large Language Models*. 2024. arXiv: 2407.02402 [cs.SE]. URL: https://arxiv.org/abs/2407.02402.

[77] Ziyin Zhang et al. *Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code*. 2024. arXiv: 2311.07989 [cs.CL]. URL: https://arxiv.org/abs/2311.07989.