# Program Slicing in the Era of Large Language Models

KIMYA KHAKZAD SHAHANDASHTI, York University, Canada MOHAMMAD MAHDI MOHAJER, York University, Canada ALVINE BOAYE BELLE, York University, Canada SONG WANG, York University, Canada HADI HEMMATI, York University, Canada

Program slicing is a critical technique in software engineering, enabling developers to isolate relevant portions of code for tasks such as bug detection, code comprehension, and debugging. In this study, we investigate the application of large language models (LLMs) to both static and dynamic program slicing, with a focus on Java programs. We evaluate the performance of four state-of-the-art LLMs-GPT-40, GPT-3.5 Turbo, Llama-2, and Gemma-7B— by leveraging advanced prompting techniques, including few-shot learning and chain-of-thought reasoning. Using a dataset of 100 Java programs derived from LeetCode problems, our experiments reveal that GPT-40 performs the best in both static and dynamic slicing across other LLMs, achieving an accuracy of 60.84% and 59.69%, respectively. Our results also show that the LLMs we experimented with are yet to achieve reasonable performance for either static slicing or dynamic slicing. Through a rigorous manual analysis, we developed a taxonomy of root causes and failure locations to explore the unsuccessful cases in more depth. We identified Complex Control Flow as the most frequent root cause of failures, with the majority of issues occurring in Variable Declarations and Assignments locations. To improve the performance of LLMs, we further examined two independent strategies for prompting guided by our taxonomy, including prompt crafting, which involved refining the prompts to better guide the LLM through the slicing process and *iterative prompting*, where the model receives feedback on the root cause and location of the failure and re-generates its responses. Our evaluation shows these two prompting enhancement approaches can improve accuracy by 4% and 3.9%, respectively.

CCS Concepts: • Software and its engineering → Software maintenance tools; Automatic programming; • Computing methodologies → Natural language generation.

Additional Key Words and Phrases: Large Language Models, Program Slicing, Empirical Study

#### **ACM Reference Format:**

### 1 Introduction

Program slicing [41, 50] is a crucial technique in software engineering, enabling early bug detection, enhancing code comprehension, and improving debugging effectiveness [46]. By isolating relevant code portions based on a *slicing criterion*, such as a variable or line number [50], developers can better

Authors' Contact Information: Kimya Khakzad Shahandashti, kimya@yorku.ca, York University, Toronto, Ontario, Canada; Mohammad Mahdi Mohajer, mmm98@yorku.ca, York University, Toronto, Ontario, Canada; Alvine Boaye Belle, alvine. belle@lassonde.yorku.ca, York University, Toronto, Ontario, Canada; Song Wang, wangsong@yorku.ca, York University, Toronto, Ontario, Canada; Hadi Hemmati, hemmati@yorku.ca, York University, Toronto, Ontario, Canada.

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

© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM XXXX-XXXX/2024/9-ART

https://doi.org/XXXXXXXXXXXXXXX

understand complex systems and pinpoint issues. This process is instrumental in simplifying the maintenance and evolution of complex systems [25]. In general, two primary approaches to program slicing exist, i.e., static slicing [2], which analyzes code without execution, and dynamic slicing [28], which considers program execution to capture runtime behavior [3]. Traditional program slicing tools [17, 34, 39, 56] have been extensively studied and continually refined, making them reliable for a wide range of slicing tasks. However, the emergence of large language models (LLMs) in natural language processing (NLP) has introduced new possibilities in software engineering [5, 11, 14, 23, 44], particularly for tasks like bug detection [7] and static analysis [33]. Despite growing interest in LLMs, there has yet to be a comprehensive evaluation of their effectiveness in performing program slicing. This study aims to bridge that gap by assessing the performance of LLMs in program slicing tasks. In this work, we explore the capabilities of LLMs in both static and dynamic program slicing, with a focus on Java programs. The novelty of this study lies in the development of a taxonomy of unsuccessful slicing cases, providing a structured framework for understanding where and why LLMs fail in generating accurate slices. This taxonomy represents a critical step toward improving LLM-based slicing techniques and serves as a foundation for future research in the field. We also propose strategies, guided by this taxonomy, to improve the accuracy of LLMs in program slicing tasks. Our study is driven by the following research questions (RQs):

RQ1: How do LLMs perform in static program slicing?

RQ2: How do LLMs perform in dynamic program slicing?

RQ3: What are the characteristics of unsuccessful static program slices?

RQ4: How do prompt enhancement strategies improve the accuracy of LLM-based static program slicing?

To address these research questions, we utilize a dataset derived from solutions to LeetCode problems [22], which offers various programs across multiple programming languages. For this study, we focus on a randomly selected subset of 100 Java programs, preprocessed for compatibility with traditional static and dynamic slicing tools. We employ advanced prompting techniques, such as one-shot learning [9] and chain-of-thought reasoning [49], using four state-of-the-art models: GPT-40 [37], GPT-3.5 Turbo [36], Llama-2-7B-Chat [6], and Gemma-7B [18]. These models are evaluated against traditional slicing tools to assess their accuracy in generating program slices. Our experiments reveal that GPT-40 achieved the highest performance in both static and dynamic slicing tasks, with an accuracy of 60.84% in static slicing and 59.69% in dynamic slicing. However, our results also indicate that the subject LLMs have yet to reach reasonable performance levels for either task, underscoring the inherent challenges LLMs face in generating precise program slices.

Furthermore, we conduct an in-depth manual analysis of the unsuccessful slices to explore why and where LLMs fail to perform slicing, categorizing the root causes and identifying failure locations within the code. This analysis culminated in the creation of a novel taxonomy of failures in LLM-based program slicing, which constitutes a key contribution of this study. We concentrated our taxonomy-building efforts on static slicing, primarily for its predictability and the more well-defined dependencies it involves. The taxonomy categorizes root causes into three key subcategories: Lack of Logic Understanding, which includes Conditional Statements, Loops, and Method Invocations; Code Complexity, encompassing Complex Control Flow and Ambiguity in Code; and Model-specific Constraints. Among these, Complex Control Flow emerged as the most frequent root cause of failures. Additionally, we identified six categories regarding the locations of failures: Conditional Statements, Loop Constructs, Method Invocations and Returns, Variable Declarations and Assignments, Class Declarations, and Imports. Variable Declarations and Assignments were identified as the most common failure location, where incorrect or incomplete slices were generated. These detailed categories provide valuable insights into where and why LLMs fall short in program slicing,

serving as a foundation for future improvements and targeted strategies. Leveraging these insights, we examined two independent prompt enhancement strategies to improve accuracy for static slicing. The first strategy, *prompt crafting*, involves refining the structure and content of the prompts based on our taxonomy of failures. This approach helps guide the LLMs more effectively through the slicing process and results in an accuracy improvement of 4% across GPT-40. The second strategy, *iterative prompting*, provides the model with feedback on the root cause and location of its errors and allows it to re-generate its response. This technique yielded a 3.9% improvement in accuracy.

Our contributions are four-fold:

- Evaluation of LLM-based program slicing: We evaluate the performance of state-of-theart LLMs in static and dynamic slicing tasks.
- A taxonomy of unsuccessful program slicing: We provide a novel taxonomy of the root causes and common fault locations behind unsuccessful static program slices.
- Prompt strategies guided by our taxonomy to enhance LLM-based static program slicing: We further examined enhanced prompt crafting and iterative prompting strategies, guided by the taxonomy, to improve the accuracy of LLMs in static program slicing.
- **Artifacts.** We release the dataset and source code of our experiments to help other researchers replicate and extend our study<sup>1</sup>.

# 2 Background and Related Work

# 2.1 Program Slicing

Program slicing, introduced by Weiser [50], is a fundamental technique in software engineering that isolates parts of a program influencing a particular computation, referred to as the *slicing criterion* [8]. This method is widely employed in debugging, testing, and maintenance tasks, allowing developers to hone in on the relevant portions of code while excluding unrelated parts [4]. There are two primary approaches to program slicing: static and dynamic. Static slicing analyzes the program without executing it, providing insights into the overall code structure and dependencies. This approach is particularly useful for optimization and enhancing code comprehension [46]. In contrast, dynamic slicing takes into account the program's execution with specific input values, making it highly effective for debugging and addressing runtime issues [3, 28]. Although dynamic slicing offers more precision in capturing execution-specific behavior, it is resource-intensive due to the need for actual execution data and test cases [29]. Both slicing techniques have been extensively studied and refined over the years. Static slicing tends to be more efficient but may produce larger-than-needed slices since it includes all potential execution paths, whereas dynamic slicing is more precise but requires considerable resources for execution [28, 46].

When generating a program slice, two key types can be constructed: backward slicing and forward slicing [21]. A backward slice captures all the statements that can potentially affect the slicing criterion, making it valuable for understanding the origins of values or bugs. In contrast, a forward slice identifies the statements that are affected by the slicing criterion, which is useful for tracking the downstream impacts of specific computations. This study focuses exclusively on backward slicing, as it is particularly effective for debugging by tracing the dependencies leading up to a specified behavior.

### 2.2 LLMs in SE Tasks

Recently, significant research has explored the potential of LLMs in Software Engineering (SE) tasks. Fan et al. [14] provide a comprehensive survey of LLMs for SE tasks. Several studies have focused

<sup>&</sup>lt;sup>1</sup>https://anonymous.4open.science/r/ProgramSlicingLLMs-4E60

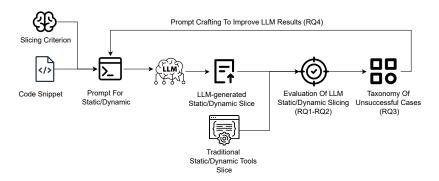


Fig. 1. Overview of this study

on utilizing LLMs for software testing and fuzzing [12, 24, 52, 54]. For instance, Deng et al. [12] introduce TitanFuzz, a novel approach that harnesses LLMs to generate and mutate input programs for fuzzing deep learning libraries, achieving significantly higher code coverage and identifying numerous previously unknown bugs compared to traditional fuzzing techniques. Additionally, some research has concentrated on using LLMs in Requirements Engineering tasks [26, 31, 42, 55]. Zhang et al. [55] evaluate ChatGPT's performance on requirements analysis tasks to gain insights into the impact of LLMs on both research and the practice of natural language processing within requirements engineering. In the realm of program analysis, several studies have examined the application of LLMs [10, 15, 30, 33]. For example, Mohajer et al. [33] assess ChatGPT's effectiveness in static analysis tasks, such as static bug detection and false positive warning removal. Yadavally et al. [53] introduce NS-Slicer which is a notable study that leverages pre-trained language models, specifically targeting the task of static program slicing. NS-Slicer uses GraphCodeBERT [20] and CodeBERT [16] to predict static slices for both complete and partial code. This study serves as an important reference point for our research. We plan to use NS-Slicer as a baseline for evaluating the effectiveness of our LLM-based approach.

# 3 Study Design

This section outlines the methodology and setup used to evaluate LLMs for program slicing tasks. Our study involves generating program slices using LLMs, comparing them against ground-truth slices produced by traditional slicing tools, and assessing their accuracy using various metrics. In addition to the automated evaluation, we have conducted a manual analysis of unsuccessful slices, which helped us identify common root causes and failure locations. As a result of our analysis, we developed a novel taxonomy of unsuccessful slicing cases regarding the root causes and locations. Based on the taxonomy, we further examined two prompting strategies to improve the performance of LLMs, including enhanced prompt crafting and iterative prompting techniques. These strategies were aimed at addressing the specific failure patterns identified during the manual analysis and resulted in improvements in the models' accuracy in program slicing tasks.

## 3.1 Overview

Figure 1 illustrates the overall process of our LLM-based program slicing study. The process begins with two key inputs: the *Slicing Criterion* (which specifies the variable or line to slice) and a code snippet. Depending on whether static or dynamic slicing is required, a corresponding prompt is generated and fed into an LLM. The LLM then produces a program slice based on the prompt. In parallel, to accelerate the process of labeling ground truth, we leverage traditional program slicing

tools (i.e., we use JavaSlicer [13] for static slicing and Slicer4J [39] for dynamic slicing) to generate slices that serve as the candidate ground truth for each program. We then manually verified the slices generated by these traditional tools, cross-referencing them with program logic to create the ground truth slicing for each program. Next, we evaluate the LLM-generated slices for both static and dynamic slicing by comparing them with the manually verified ground-truth slices (**RQ1** and **RQ2**). For cases where the LLM-generated slices are unsuccessful, we perform a detailed analysis to identify and categorize the underlying root causes and failure locations (**RQ3**). Finally, we apply iterative prompt refinement, adjusting and improving prompts based on the identified failures to enhance the LLM's slicing performance (**RQ4**). This iterative process allows us to address the challenges and limitations identified in previous steps, ultimately improving the accuracy and robustness of LLM-generated program slices.

# 3.2 Dataset Description

The dataset used in this study consists of a diverse collection of solutions and explanations for various LeetCode problems, covering multiple programming languages such as Python, Java, C, and JavaScript. For our experiments, we focused exclusively on Java, randomly selecting 100 samples. This selection was made to balance the computational cost of running LLMs and the scope of the analysis while ensuring a representative sample size. The dataset was preprocessed to be compatible with both static and dynamic analysis tools. For static slicing, we added necessary libraries, adjusted function names, and ensured the code was properly structured for slicing tools. For dynamic slicing, we manually inserted a main function into each program to enable execution-based analysis. Moreover, our *slicing criterion* focused on manually selecting the key variables and lines of code that were central to the program's logic.

# 3.3 Labeling Ground Truth

To evaluate the performance of LLMs, it was essential to establish ground-truth program slices for comparison. We adopted a two-step process to generate and validate these ground-truth slices. First, we used traditional program slicing tools to automatically generate candidate ground-truth slices. These tools provide accurate slices based on well-established slicing algorithms, serving as a benchmark for assessing the LLM-generated slices. In the second step, we manually verified and, when necessary, adapted these slices to ensure correctness and alignment with the program's intended behavior. We employed two primary tools: JavaSlicer and Slicer4J. JavaSlicer [13] is a static program slicing tool that analyzes Java bytecode using the system dependence graph (SDG) to extract program slices. Slicer4J [39], on the other hand, is a dynamic program slicing tool designed specifically for Java programs. It captures runtime information to produce program slices reflecting the program's execution.

# 3.4 Experimental LLMs

In this study, we selected four state-of-the-art large language models (LLMs) for evaluation: Llama-2-7B-Chat, Gemma-7B, GPT-3.5 Turbo, and GPT-4o. **Llama-2-7B-Chat**, developed by Meta AI [47], is fine-tuned for chat-based applications and coding assistance, containing 7 billion parameters. It is designed to provide high-quality conversational and coding capabilities, making it particularly useful for tasks requiring the generation of programming instructions. **Gemma-7B**, developed by Google as part of the Gemma family [45], is an open-source, decoder-only text-to-text model optimized for English language tasks, with a context length of up to 8192 tokens. Its versatility allows it to excel in tasks such as summarization, question answering, and reasoning. **GPT-3.5 Turbo**, a variant of the GPT-3.5 series from OpenAI [35], is designed for high efficiency and strong performance in text generation tasks. It is well-suited for real-world applications that require

Model	Llama-2-7B-Chat	Gemma-7B	GPT-3.5 Turbo	GPT-40
Size	7B	7B	-	-
Context Window	4K	8K	4K	8K
Release Date	Jul 2023	Feb 2024	March 2023	March 2024

Table 1. Overview of experimental LLMs

high-quality text output at a lower computational cost. Lastly, **GPT-4o**, the flagship model in the GPT-4 series by OpenAI, is engineered to handle complex tasks with superior accuracy. This model is known for its powerful capabilities in both natural language and programming code generation, making it an ideal choice for our programming-related tasks. Table 1 summarizes the characteristics of the mentioned LLMs. Each model was used with its default settings for our experiments, with GPT-4o configured at a temperature of 0.7 and Llama-2-7B-Chat at a temperature of 0.8, ensuring consistency across all evaluations.

### 3.5 Evaluation Metrics

To capture the models' performance at the slice level, we used two metrics to evaluate our experiments, both adopted from the work of Yadavally et al. [53]. The first metric, **Exact-Match Accuracy (Accuracy-EM)**, evaluates the model's performance by counting the number of times the predicted slices exactly match the ground-truth slices. The second metric, **Dependence Accuracy (Accuracy-D)**, measures how accurately the inter-statement dependencies are predicted. Accuracy-D is calculated as the ratio of correctly predicted dependencies to actual dependencies for all slicing criteria in a given program.

# 4 LLM-Based Program Slicing

# 4.1 Prompting Techniques

In this study, we employed several commonly used prompting techniques to evaluate the performance of the LLMs. First, **zero-shot prompting** involves providing the model with a prompt without any examples or additional context, relying solely on the model's pre-trained knowledge to generate a response based on the input alone [27]. Next, we used **one-shot prompting**, where the model is given a prompt along with a single example to guide its response. This technique helps the model understand the task better by showcasing the desired output format, allowing it to leverage its ability to learn from minimal data to improve task performance [19]. Lastly, we employed **Chain-of-Thought (CoT)** prompting [49], which breaks down complex tasks into intermediate reasoning steps. This method guides the model through a sequence of logical steps to arrive at the final answer, enhancing its ability to handle tasks that require multi-step reasoning [48, 57].

### 4.2 Experimental Setup

We designed three experiments to investigate the effectiveness of different prompting techniques in generating accurate and relevant program slices.

**Experiment 1: zero-shot.** This experiment evaluates the ability of the model to generate program slices using zero-shot prompting, where the model is not provided with any specific examples or additional context. The model relies solely on its pre-trained knowledge to perform the task.

**Experiment 2: one-shot.** This experiment incorporates a one-shot learning approach, where the model is given a single example to guide its response. In this setup, the model is provided with one example of a program slice with its respective output, but no additional context is given. We

selected the specific example that was introduced in the traditional program slicing tools. This example is a pair consisting of a *<code\_snippet*, *slicing\_criterion*, *output>*. This approach ensures that the example used in the one-shot learning setup is both relevant and representative of the typical slicing tasks the LLM is expected to perform.

**Experiment 3: one-shot with CoT.** This experiment combines one-shot learning with the CoT prompting technique. In this setup, the model is provided with a single example that includes a program slice along with a logical sequence of steps or reasoning chains that guide its slicing output. We used the same methodology as in Experiment 2 for selecting the example, relying on the example introduced in the traditional program slicing tools. This example is a pair consisting of a *<code\_snippet*, *slicing\_criterion*, *reasoning*, *output>*. By breaking down the problem into these intermediate steps through the reasoning chain, the Chain-of-Thought technique helps the model to better understand and execute the task [49].

# 4.3 Prompting Templates

This section presents the prompt templates used for RQ1 and RQ2, which guide the LLM in performing slicing tasks while maintaining consistency in responses across different experiments. Since we employ various prompting strategies depending on the experiment, the structure of the prompt changes accordingly, particularly regarding the inclusion of examples. For zero-shot, no example is provided within the prompt. In one-shot, we include an example comprising a code snippet, the associated slicing criterion, and the expected output to guide the LLM's response. In one-shot with CoT, the prompt is further expanded to include an example with a code snippet, the slicing criterion, a reasoning section detailing the steps needed to derive the output, and the expected output itself.

- 4.3.1 RQ1: Static Slicing Template. Table 2 presents the prompt template for static slicing. It provides a structured approach, guiding the LLM to identify all lines that impact the Slicing Criterion and then trace backward to earlier lines. The template ensures that the model fully utilizes its capabilities to generate accurate and comprehensive slices by clearly defining the Slicing Criterion format. To maintain consistency and facilitate easy comparison with ground truth data, the output is formatted in JSON, simplifying the parsing process for subsequent analysis.
- 4.3.2 RQ2: Dynamic Slicing Template. Table 3 presents the prompt template for dynamic slicing. It directs the LLM to focus on the Slicing Criterion, typically corresponding to a critical line such as the return statement in the main function, and to trace backward through the code to identify all lines that directly or indirectly influence it. This template ensures the model accurately captures the runtime dependencies essential for dynamic slicing. We format the output in JSON to enable seamless parsing and direct comparison with ground-truth data during subsequent analysis.

# 4.4 RQ1: Effectiveness of LLMs on Static Program Slicing

Table 4 presents the performance metrics for static program slicing across three experiments (i.e., zero-shot, one-shot, and one-shot with Chain-of-Thought) for selected LLMs, including GPT-40, GPT-3.5 Turbo, Llama-2, and Gemma-7B. The metrics evaluated are Accuracy-D and Accuracy-EM. Moreover, to mitigate the effects of the inherently non-deterministic nature of LLMs, we ran each experiment three times and used the average for each model to present its performance. To benchmark our LLM-based approach, we employed NS-Slicer [53] as the baseline for static slicing of Java programs. NS-Slicer leverages both CodeBERT and GraphCodeBERT, and their pre-trained models are publicly available. We carefully preprocessed our dataset to ensure compatibility with NS-Slicer for execution. However, due to the complexity of certain programs, we were unable to run 15% of the dataset. Despite this, NS-Slicer effectively predicts static program slices, making

### Table 2. Prompt template for RQ1

### **Prompt Template**

You are an AI assistant specialized in performing backward static slicing for Java programs. You are provided with a Java code snippet with the line numbers and a Slicing Criterion.

### **Important Notes:**

- Your task is to identify all lines in the program that may affect the value of the Slicing Criterion variable, following a backward slicing approach.
- You need to first locate the line of the last instruction that affects the Slicing Criterion Variable in the code (That line is always in the output), then trace backward to find all relevant lines.

# Slicing Criterion Format: <Provide Slicing Criterion Format>

### **Output Format:**

- An array of line numbers in plain JSON without any markdown in the following format: {{"output": ["line\_number1", "line\_number2"]}}
- <Provide Example with Code Snippet, Slicing Criterion, and Output for One-shot Experiment. Include Reasoning for One-shot with CoT Prompting. Omit Example for Zero-shot.>

Task:

Now, based on the provided Java program and Slicing Criterion, generate the output in the specified format.

Slicing Criterion: <Provide Slicing Criterion>

Program: <Provide Code Snippet>

Output:

Table 3. Prompt template for RQ2

### **Prompt Template**

You are an AI assistant specialized in performing backward dynamic slicing for Java programs. You are provided with a Java code snippet, which includes line numbers, and a Slicing Criterion line number.

#### **Important Notes:**

- The Slicing Criterion line number corresponds to the return statement in the  ${\tt main}$  function.
- Your task is to start with the Slicing Criterion line number itself in the output and then trace backward through the code to identify all relevant lines that directly or indirectly influence the value at the Slicing Criterion.

### **Output Format:**

- An array of line numbers in plain JSON without any markdown in the following format:  ${\text{["output": ["line_number1", "line_number2"]}}}$
- <Provide Example with Code Snippet, Slicing Criterion, and Output for One-shot Experiment. Include Reasoning for One-shot with CoT Prompting. Omit Example for Zero-shot.>

Task:

Now, based on the provided Java program and Slicing Criterion, generate the output in the specified format.

Slicing Criterion: <Provide Slicing Criterion>

Program: <Provide Code Snippet>

Output:

it a valuable benchmark for comparison with our LLM-based approach. Table 5 presents the performance of NS-Slicer on our dataset. It is worth noting that the performance of NS-Slicer in our study differs from the results reported in its original paper. This discrepancy can be attributed to several factors. First, our dataset is different, as we selected Java programs from LeetCode, which vary in complexity compared to the dataset used in the original NS-Slicer study, which is IBM's Project CodeNet dataset [40]. Another possible factor is the preprocessing steps we performed to ensure compatibility with the traditional tools, which may have introduced variations in the code format that could influence the performance.

As we can see from Table 4, GPT-40 demonstrated the best performance across all models, particularly in the one-shot with CoT prompting strategy, achieving an Accuracy-D of **60.84**% and an Accuracy-EM of **7.33**%. This is notable because GPT-40 performs better than NS-Slicer concerning

Model	Zero-shot		One-shot		One-shot with CoT	
	Acc-D	Acc-EM	Acc-D	Acc-EM	Acc-D	Acc-EM
GPT-40	30.91%	0%	54.09%	4.0%	60.84%	7.33%
GPT-3.5 Turbo	29.60%	0%	37.33%	0%	46.43%	1%
Llama-2	4.75%	0%	10.54%	0%	17.57%	0%
Gemma-7B	38.65%	0%	35.22%	0%	43.43%	0.66%

Table 4. Effectiveness of LLMs on static program slicing across different prompt strategies

Table 5. Effectiveness of NS-Slicer on our dataset

Model	Acc-D	Acc-EM
NS-Slicer (CodeBERT)	60.52%	0%
NS-Slicer (GraphCodeBERT)	55.33%	0%

exact match accuracy, as neither NS-Slicer (CodeBERT) nor NS-Slicer (GraphCodeBERT) generated any exact matches. Regarding Accuracy-D, GPT-40 also slightly outperforms NS-Slicer (CodeBERT) and NS-Slicer (GraphCodeBERT). NS-Slicer (GraphCodeBERT) shows lower performance, achieving an Accuracy-D of 55.33%. GPT-3.5 Turbo also showed improvement across the experiments, with its highest performance in the one-shot with CoT prompting strategy, achieving an Accuracy-D of 46.43%. Llama-2 consistently performed weaker than the other models, with its best result in the one-shot with CoT strategy reaching only 17.57% for Accuracy-D. Gemma-7B exhibited moderate performance, performing best in the one-shot with CoT prompting strategy, achieving an Accuracy-D of 43.43%.

**Key Findings:** GPT-40 emerged as the best-performing model for static slicing, especially under the one-shot with chain-of-thought prompting strategy, which resulted in the Accuracy-D of 60.84% and an Accuracy-EM of 7.33%. NS-Slicer (CodeBERT) achieved comparable slice Accuracy-D of 60.52% but could not generate any exact matches.

# 4.5 RQ2: Effectiveness of LLMs on Dynamic Program Slicing

Table 6 presents the performance metrics for dynamic program slicing across different prompt strategies (i.e., zero-shot, one-shot, and one-shot with Chain-of-Thought) for various LLMs, including GPT-40, GPT-3.5 Turbo, Llama-2, and Gemma-7B, in Java. The evaluated metrics are Accuracy-D and Accuracy-EM. Similar to RQ1, we ran each experiment three times and selected the average for each model to report its performance. The findings from our analysis of dynamic slicing results indicate that GPT-40 consistently outperforms the other models in dynamic slicing, achieving the highest Accuracy-D score of **59.69%** under the zero-shot prompting strategy. However, GPT-40, like the other models, struggles to achieve exact matches, as indicated by its Accuracy-EM of 0.0% across all experiments. GPT-3.5 Turbo also performs relatively well, particularly in the zero-shot strategy, where it achieves an Accuracy-D of 44.84%, though its performance drops slightly in one-shot with chain-of-thought strategies. Llama-2 performs the best under the one-shot with chain-of-thought strategy with an Accuracy-D of 36.72%, although it shows lower performance in other strategies. Gemma-7B demonstrates mixed performance, achieving its best Accuracy-D of 41.90% in the one-shot strategy, but its performance drops significantly in the one-shot with

Model	Zero-shot		One-shot		One-shot with CoT	
	Acc-D	Acc-EM	Acc-D	Acc-EM	Acc-D	Acc-EM
GPT-40	59.69%	0%	41.77%	0%	58.62%	0%
GPT-3.5 Turbo	44.84%	0%	42.44%	0%	36.16%	0%
Llama-2	25.26%	0%	26.85%	0%	36.72%	0%
Gemma-7B	33.41%	0%	41.90%	0%	25.49%	0%

Table 6. Effectiveness of different prompting techniques for dynamic program slicing

chain-of-thought strategy, reaching 25.49%. The performance inconsistency across experiments can be attributed to the nature of dynamic slicing tasks. Dynamic slicing relies heavily on execution contexts and runtime behavior, which may not be adequately captured in all prompting strategies. The additional context provided in the prompts appears to introduce confusion. This suggests that, for dynamic slicing, more context does not always lead to better performance, likely due to the complexity of runtime behavior and the challenges faced by models when handling additional information in such cases.

**Key Findings:** GPT-40 remains the most effective model for dynamic slicing, particularly under the zero-shot strategy, achieving an Accuracy-D of **59.69**%. However, all models across all experiments struggle with exact matches, as indicated by the consistent 0% Accuracy-EM.

# 4.6 Comparison of Static and Dynamic Slicing

In this section, we compare the effectiveness of static and dynamic slicing across the evaluated LLMs. To assess the statistical significance of the performance difference between static and dynamic slicing, we applied the Mann-Whitney U test [32], a non-parametric statistical test that does not assume normal distribution. This test is particularly suited for our data, given the small sample size and the fact that accuracy values may not follow a normal distribution. We used the Accuracy-D from Tables 4 and 6, which represent the performance of LLMs on static and dynamic slicing, respectively. The results of the Mann-Whitney U test yielded a U statistic of **63.0** and a p-value of **0.62**. As the p-value exceeds the commonly accepted threshold of 0.05, we concluded that there is no statistically significant difference between the effectiveness of static slicing and dynamic slicing. However, it is important to note that while no significant difference was found in Accuracy-D, all models demonstrated an advantage in static slicing compared to dynamic slicing when looking at Accuracy-EM.

# 5 RQ3: Taxonomy of Unsuccessful Program Slicing

In RQ1 and RQ2, our experimental results show that most of the subject LLMs are yet to achieve a reasonable performance for either static slicing or dynamic slicing. In this section, we set out to demystify the reasons behind these unsuccessful slicing cases. We concentrated our taxonomy-building efforts on static slicing, primarily for its predictability and the more well-defined dependencies it involves. Moreover, the complexities of dynamic slicing—such as the need for runtime execution—introduce additional challenges that make manual analysis and taxonomy building more difficult. The runtime behavior adds layers of variability that complicate the identification of consistent failure patterns. Focusing first on static slicing, we establish a robust taxonomy that addresses core issues, which can later be adapted and extended to more complex, dynamic slicing scenarios. Our study identified GPT-40 as the best-performing model in static slicing, producing

slices with higher Accuracy-D compared to other models. To streamline the manual effort of labeling unsuccessful cases for taxonomy creation, we focused on GPT-40 under the prompting strategy of one-shot with CoT, which consistently generated higher-quality slices. Additionally, we concentrated on the best iteration of the model's output, allowing us to manage the manual effort more effectively while still targeting the most promising results. By concentrating on the best-performing model and its best iterations, we ensured that the taxonomy reflects the most nuanced and sophisticated challenges encountered in program slicing, rather than simply cataloging basic errors from less capable models.

# 5.1 Methodology

To systematically identify unsuccessful slicing, we focused on the cases where the Accuracy-D of the slicing produced by the LLM was not 100%. This approach allowed us to ensure a comprehensive analysis of the LLM's performance, capturing every instance where the model did not perfectly replicate the ground-truth slices. By including all non-perfect slices, we could thoroughly investigate the specific areas where the LLMs struggled, ensuring that no potential issue was overlooked. A total of 92 unsuccessful cases required manual analysis. Two of the authors, each with over 5 years of experience in software development, independently reviewed the unsuccessful slicing cases to ensure objective classification. The authors carefully analyzed each case, focusing on identifying the root causes and error locations within the LLM-generated slices. On average, each sample took approximately 5 minutes to assess, which involved understanding the code, comparing the LLM-generated slice with the ground truth, and labeling both the root cause and error location. When a new root cause or fault location category emerged that was not accounted for in the taxonomy, the authors temporarily paused their labeling efforts and held meetings with each other. These discussions allowed for the validation of new categories, updates to the taxonomy, and re-labeling of affected cases, ensuring consistency and accuracy throughout the process. After completing their reviews, the two experts met to discuss and resolve any discrepancies in their classifications. Through collaborative discussion, they reached a consensus on each case. This rigorous process resulted in a refined taxonomy that categorizes root causes into three main subcategories: Lack of Logic Understanding (including Conditional Statements, Loops, and Method Invocations), Code Complexity (including Complex Control Flow and Ambiguity in Code), and Model-specific Constraints. In addition, the authors identified six distinct failure location categories: Conditional Statements, Loop Constructs, Method Invocations and Returns, Variable Declarations and Assignments, and Class Declarations and Imports.

### 5.2 Taxonomy of Fault Locations

Faults within a codebase can occur at various critical locations, each presenting unique challenges for accurate program slicing by LLMs. Identifying and categorizing these fault locations is essential for understanding the slicing failures and developing strategies to improve the accuracy of LLM-generated slices. Figure 2 illustrates the six categories of fault locations in unsuccessful slicing cases. Below, we describe the primary categories of fault locations that commonly affect slicing accuracy. We also provide an example to illustrate each category.

**A1. Conditional Statements.** Conditional statements, such as if, else, and switch, are common locations for slicing faults. These faults typically arise when the LLM incorrectly evaluates the branching logic, leading to the misinterpretation of which branches are relevant to the slicing criterion. As a result, the generated slice may either over-include irrelevant branches or omit necessary ones, causing incomplete or inaccurate slices.

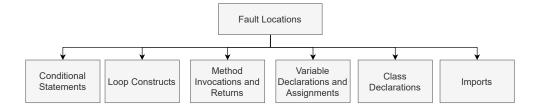


Fig. 2. Categories of locations where slicing faults occur

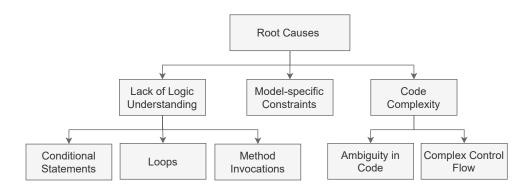


Fig. 3. Categories of the root causes

- **A2. Loop Constructs.** Loops, including for, while, and do-while loops, are common locations for slicing errors. These faults lead to slices that either exclude essential loop iterations or unnecessarily include irrelevant ones, failing to correctly propagate the slicing criterion across the loop structure. **A3. Method Invocations and Returns.** Method Invocations and return statements are another critical location for faults.
- **A4. Variable Declarations and Assignments.** Variable declarations and assignments are frequent locations of slicing failures. Ambiguities in how variables are declared or initialized can lead to data dependency errors, where the LLM fail to accurately track variable states throughout the program. **A5. Class Declarations.** Faults occurring in class declarations can have an impact on slicing accuracy.
- **A6. Imports.** Imports represent a location where slicing errors occur. When external modules or libraries are imported, the LLM may not fully understand how these imports impact the program, leading to incomplete slices that fail to include the necessary functionality brought in by the imports.

# 5.3 Taxonomy of Root Causes

Figure 3 illustrates the categories of root causes for unsuccessful cases, which consists of three subcategories, i.e., Lack of Logic Understanding (Section 5.3.1), Code Complexity (Section 5.3.2), and Model-specific Constraints (Section 5.3.3).

5.3.1 Lack of Logic Understanding. The analysis revealed that unsuccessful slicing cases often arise from the LLM's inability to fully understand the logical structure of the program. These logic-related failures typically occur in code segments that involve decision-making and control flow. The LLM

struggles to accurately reason about the dependencies in these regions, which hinders the proper propagation of slicing criteria and leads to incomplete or incorrect slices. The key logical challenges where LLMs frequently struggle include conditional statements, loops, and method invocations.

**B1.** Conditional Statements. Errors related to conditional statements, such as if and switch, were commonly observed. These issues typically arise when the LLM fails to capture the entire control flow relevant to the slicing criterion. For instance, the LLM may incorrectly include only the if branch, while omitting necessary else or else if branches, resulting in an incomplete slice. In this example, when slicing based on the free variable, both branches influence its value. However, the LLM might incorrectly slice only the if branch, leading to an incomplete representation of the program logic.

```
if (target[i] > free) {
    req += target[i] - free;
    free = target[i];
} else if (target[i] < free) {
    free = target[i];
}

// Expected: The slice should include both branches when slicing on the 'free' variable.
// LLM Error: The slice may only include the 'if' branch and ignore the 'else' branch.</pre>
```

**B2. Loops.** Slicing errors in loops (such as while and for) frequently arise from the LLM's difficulty in understanding complex loop conditions, invariants, or exit criteria. This often results in slices that are either incomplete or incorrectly exclude significant parts of the loop's execution, such as intermediate calculations or modifications to data structures. In this example, both the sum and queue statements are crucial for the program's logic. However, when slicing based on the queue variable, the LLM might exclude the sum operation, leading to an incomplete slice and missing the interdependencies within the loop.

```
for (int num : nums) {
    sum = sum.add(BigDecimal.valueOf(num).divide(BigDecimal.valueOf(2)));
    queue.add(BigDecimal.valueOf(-num));
}

// Expected: The slice should include both statements in the loop when slicing on the 'queue' variable.
// LLM Error: The slice may only include the 'queue' statement, ignoring the 'sum' operation.
```

**B3. Method Invocations.** Method invocations, especially those involving built-in data structures, often present significant challenges for LLMs. The LLM may fail to propagate the slicing criterion through these method calls, which can affect the internal state of data structures. These methods often manipulate key variables, and failing to track their effects can result in incomplete slices. In this case, the add(), remove(), and size() method invocations directly influence the internal state of the numbers list. When slicing based on the size variable, the LLM needs to capture all these method invocations. Failure to do so would result in an incomplete representation of the program, as key changes to the numbers list would be missed, affecting the final output.

```
List<Integer> numbers = new ArrayList<>();
numbers.add(5);
numbers.remove(0);
int size = numbers.size();
return size;
// Expected: The slice should include the 'add', 'remove', and 'size' method invocations as they modify or access the internal state of 'numbers'.
// LLM Error: The slice may fail to capture one or more method invocations, missing their impact on the 'numbers' list.
```

5.3.2 Code Complexity. The analysis also identified several root causes related to the complexity of the code that contributed to unsuccessful slicing. These root causes were consistently linked to the LLM's limitations in understanding and processing complex programming constructs. Two main categories of root causes were identified:

**C1. Ambiguity in Code.** Ambiguous code constructs were a significant source of errors in program slicing. These ambiguities often arise when the code can be interpreted in multiple ways, causing the LLM to make incorrect assumptions about how the slicing criterion should propagate. A common scenario involves variables that are initialized across multiple lines, or expressions that involve complex or unclear logic, leading to incomplete or incorrect slices. In this example, the value of 'result[i]' is determined by a multi-line ternary expression, which introduces ambiguity. The slicing criterion focuses on the variable 'result[i]', but due to the way the expression spans several lines, the LLM might fail to capture the entire initialization process.

```
result[i] =
s.length() > (intLength + 1) / 2
? -1
: someFunction(s, intLength);
// Expected: The slice should correctly propagate the value of 'result[i]'.
// LLM Error: The LLM might fail to capture the full expression due to the multi-line structure.
```

**C2. Complex Control Flow.** The LLM exhibited difficulties when handling complex control flows, particularly in the case of nested loops and conditionals, where dependencies and data flow become intricate and challenging to track. Although loops and conditionals are already part of the sub-category of Lack of Logic Understanding, we introduce this category because the complexity increases significantly when these constructs are nested. This added complexity often led to incomplete or incorrect slices, as the LLM failed to accurately propagate the slicing criterion across all relevant paths within the code. The nested structure of the loops, as shown above, can confuse the LLM, particularly when tracking how the variables 'z', 'i', and 'y' depend on each other.

```
for (int z = m - 1; z >= 0; --z) {
   for (int i = 0; i <= z; ++i) {
      int y = z - i;
   }
}
// Expected: The slice should capture the relationship between 'z', 'i', and 'y' across all iterations.
// LLM Error: The LLM fail to track dependencies between the variables, especially in nested loops.</pre>
```

5.3.3 Model-Specific Constraints. Several unsuccessful slices stemmed from inherent limitations in the design of the LLMs used. These model-specific constraints often resulted in slicing errors and reduced the overall effectiveness of the LLM in accurately processing complex code. The LLM's internal context window can cause errors in cases where the code being sliced exceeds the model's token limit, leading to incomplete slices as critical code dependencies are truncated. Additionally, when text is intermixed with code, such as in comments, documentation, or string literals, the LLM can mistakenly include irrelevant text as part of the slice, resulting in incorrect slices. Parsing failures also occur due to the LLM's reliance on JSON-based outputs for processing results. In some cases, the JSON parser can misinterpret or fail to process the LLM's output correctly, especially when the model generates malformed or incomplete JSON structures. These model-specific constraints represent inherent challenges that limit the LLM's ability to slice complex programs fully and accurately.

### 5.4 Discussion

Figure 4 illustrates the distribution of root causes and location across these slices, highlighting the potential impact each category has on the accuracy of LLM-generated slices. Out of the 92 unsuccessful slicing cases that required manual analysis, several key patterns emerged. The most frequent **root cause** of errors is *Complex Control Flow*, identified in 39 cases. This suggests that LLMs struggle with handling intricate control flow structures such as nested loops and conditionals, which are prevalent in more complex codebases. In terms of **location**, the most common location of errors is *Variable Declarations and Assignments*, identified in 78 cases. This points to significant challenges LLMs face in correctly slicing code around variable declaration and assignments. These findings highlight key areas where LLMs struggle, particularly when dealing with complex control structures and dependencies. Understanding these challenges is crucial for refining LLM-based slicing techniques and improving their overall accuracy. We also created a Sankey diagram to illustrate the relationship between root causes and failure locations in LLM-based program slicing, as shown in Figure 5. The diagram provides a depiction of how root causes are linked to fault locations. For example, the diagram highlights that the majority of errors related to *Complex Control Flow* happen in *Loop Constructs* and *Variable Declarations and Assignments* locations.

**Key Findings:** Through manual analysis, we proposed a taxonomy for root causes and locations of failures in LLM-based static program slicing. Our findings reveal that *Complex Control Flow* is the most common root cause, while *Variable Declarations and Assignments* is the most frequent error location.

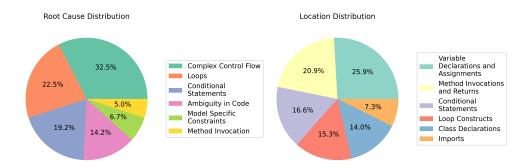


Fig. 4. Distribution of unsuccessful slicing cases

### 6 RQ4: Strategies for Improvement

Rangeet et al. [38] proposed prompt crafting and iterative prompting to improve LLM-based code translation tasks. Motivated by [38], in this section, we set out to examine the effectiveness of these two approaches on LLM-based static program slicing. In this context, we refer to the results from RQ1 in one-shot with CoT prompting as *Vanilla*, which served as the baseline for comparison as it has the best Accuracy-D among other experiments. Our goal is to build upon this baseline and enhance performance through targeted improvements. This taxonomy enables researchers to accurately guide LLMs to avoid the most common root causes and fault locations leading to unsuccessful slicing across four subject LLMs: GPT-4, GPT-3.5 Turbo, Llama-2, and Gemma-7B.

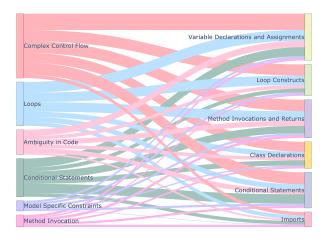


Fig. 5. Mapping between root causes and failure locations

■ Vanilla ■ Enhanced Prompt Crafting ■ Iterative Prompting

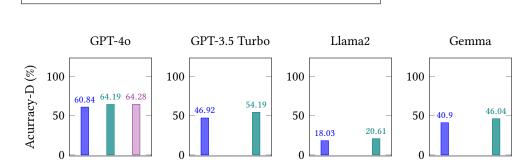


Fig. 6. Comparison of accuracy improvements for different LLMs (GPT-4o, GPT-3.5 Turbo, Llama2, and Gemma) under various strategies: Vanilla, Enhanced Prompt Crafting, and Iterative Prompting

**1. Enhanced Prompt Crafting:** Refining the prompts used to guide LLMs has been shown to enhance their accuracy [51]. As highlighted in Section 5.4, the most common root cause for failures is *Complex Control Flow*, while the most frequent error location occurs in *Variable Declarations and Assignments*. To mitigate these issues, we enriched the prompts by adding an example that includes a code snippet, detailed reasoning steps explaining how the slicing should be correctly performed, and the expected output. This example contains common failure scenarios identified in our manual analysis, such as *Complex Control Flow* and *Variable Declaration and Assignment*, where we identified LLMs as having historically struggled. The results demonstrate that this enhanced prompt crafting approach consistently improves performance across different LLMs. For GPT-40, accuracy increased by approximately 4%, while GPT-3.5 Turbo showed a similar improvement, with accuracy rising by 6.4%. Gemma-7B also experienced a gain in performance, with an increase of 5.1%, and Llama2 saw an improvement of 2.5%. These results indicate that prompt crafting, when guided by insights from our taxonomy of common failure points, can significantly enhance the LLMs' ability to handle complex static program slicing tasks. Figure 6 demonstrates the effectiveness

of this enhanced prompt crafting strategy, showcasing the Accuracy-D gains achieved across the selected LLMs compared to the Vanilla approach.

2. Iterative Prompting: Building on the approach outlined in [43], we employed iterative prompting to improve the performance of LLM-generated program slices. This technique was applied exclusively to GPT-40 after we identified specific failure cases through manual analysis. In this process, we provided feedback directly in the prompt, explaining why the initial slices were incorrect by detailing the root causes and failure locations identified during the manual review. By explicitly guiding the LLM to address these issues, we instructed it to avoid the same pitfalls while re-generating the slices. This iterative feedback loop allowed the model to focus on the areas where it had previously struggled, enabling it to refine its performance in a targeted manner. In this single iteration, we observed an increase in Accuracy-D by 3.9%, demonstrating the effectiveness of targeted feedback in addressing the specific issues the model encountered. While only one iteration was performed, further iterations could improve the model's performance even more. Each iteration would involve identifying new failed cases, labeling them, and providing feedback based on root causes and failure locations, ultimately improving the LLM's ability to generate accurate slices. Although the iterative approach is independent of enhanced prompt crafting, both techniques complement each other by targeting different aspects of the LLM's limitations. It is important to note that, while this approach can improve performance, it may not be practical as it requires a human-in-the-loop to verify the results of each iteration to provide feedback. Figure 6 demonstrates the effectiveness of iterative prompting for GPT-40, highlighting the gains achieved compared to the Vanilla approach.

**Key Findings:** Our investigation into enhancement strategies revealed that both *enhanced prompt crafting* and *iterative prompting*, guided by our proposed taxonomy, improved the Accuracy-D of LLM-based static program slicing. Enhanced prompt crafting increased the Accuracy-D by up to 4% for GPT-40, while iterative prompting improved GPT-40's Accuracy-D by 3.9%.

### 7 Threats to Validity

In this section, we outline potential threats to the validity of our study and explain the steps we took to address them.

**External Validity**. A key threat to external validity is the limited range of programming languages we used in our experiments. Our study focused exclusively on Java, which may restrict the generalizability of our findings to other languages. One reason for this choice is the availability of traditional slicing tools like JavaSlicer and Slicer4J, which are well-tested and maintained. Expanding the study to other languages, such as C, would require identifying or developing reliable slicing tools for those languages. Another external validity concern relates to the limited number of LLMs selected for our experiments. Future work could involve a broader range of LLMs to determine if our findings hold across different architectures and model versions. Another potential threat arises from using the same dataset to develop the taxonomy and evaluate the approach. This overlap could affect the generalizability of our findings. To mitigate this issue, future work could involve using separate datasets: one for developing the taxonomy and another for evaluating the approach.

**Internal Validity**. One internal threat arises from the possibility that the LLMs had seen our dataset during their training. To mitigate this, we preprocessed the dataset to ensure that the LLMs had not encountered it before. This step helped prevent the models from generating responses based on previously seen data, ensuring a more accurate evaluation of their slicing capabilities on

unseen code. Furthermore, our results could be impacted by potential errors in our automation scripts. To minimize this risk, we performed extensive testing of the scripts and conducted spot checks to verify the correctness of the outputs. Additionally, we have made our artifacts publicly accessible to support the review and replication of our findings [1].

Construct Validity. A threat to construct validity stems from the manual analysis of unsuccessful slices. The subjective nature of manual labeling can introduce bias. To address this, we employed two independent reviewers to evaluate the results, followed by a discussion to resolve any discrepancies. This collaborative process ensured a more consistent and objective analysis. Another potential issue lies in the reliability of the traditional program slicing tools we used as the ground truth. Any inaccuracies in these tools could affect our evaluation of the LLM-generated slices. We mitigated this by using well-established tools such as JavaSlicer and Slicer4J, which are trusted for their accuracy and reliability.

### 8 Conclusion

Program slicing is essential in software engineering, aiding in tasks like bug detection. In this study, we set out to investigate how LLMs could be applied to program slicing, specifically focusing on both static and dynamic slicing for Java programs. Our findings reveal that GPT-40 demonstrated the best performance among the models tested, but challenges remain, particularly in achieving exact matches in more complex scenarios. Recognizing the need to understand these failures better, we examined the unsuccessful cases and characterized the root causes and locations of such failures. Our taxonomies offer a structured framework for researchers to identify common patterns in unsuccessful static program slices, providing insights into where LLMs struggle most in handling program-slicing tasks. Furthermore, we leveraged this taxonomy to further examine two prompt enhancement strategies, i.e., enhanced prompt crafting and iterative prompting.

For future work, we plan to expand the scope of this study to include other programming languages to provide a more comprehensive evaluation of LLM performance. Additionally, we plan to explore alternative prompt engineering techniques to improve model accuracy.

### 9 Data Availability

All the scripts and data to reproduce our experiments can be found in our replication package [1].

# References

- [1] 2024. Program Slicing with LLMs: Replication Package. https://anonymous.4open.science/r/ProgramSlicingLLMs-4E60.
- [2] Mithun Acharya and Brian Robinson. 2011. Practical change impact analysis based on static program slicing for industrial software systems. In *Proceedings of the 33rd international conference on software engineering*. 746–755.
- [3] Hiralal Agrawal and Joseph R Horgan. 1990. Dynamic program slicing. ACM SIGPlan Notices 25, 6 (1990), 246-256.
- [4] Khaled Ahmed, Mieszko Lis, and Julia Rubin. 2021. Mandoline: Dynamic slicing of android applications with trace-based alias analysis. In 2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST). IEEE, 105–115.
- [5] Tameem Ahmed and Premkumar T Devanbu. 2023. Few-shot training LLMs for project-specific code-summarization. In Proceedings of the 37th International Conference on Automated Software Engineering (ASE).
- [6] Meta AI. 2024. Llama-2-7B-Chat-HF. https://huggingface.co/meta-llama/Llama-2-7b-chat-hf Accessed: 2024-08-07.
- [7] Kamel Alrashedy. 2023. Language Models are Better Bug Detector Through Code-Pair Classification. arXiv preprint arXiv:2311.07957 (2023).
- [8] David W Binkley and Keith Brian Gallagher. 1996. Program slicing. Advances in computers 43 (1996), 1-50.
- [9] Tom B Brown et al. 2020. Language models are few-shot learners. Advances in neural information processing systems 33 (2020), 1877–1901.
- [10] Patrick J Chapman, Cindy Rubio-González, and Aditya V Thakur. 2024. Interleaving Static Analysis and LLM Prompting. In Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis. 9–17.
- [11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021).

- [12] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis. 423–435.
- [13] JavaSlicer Developers. 2024. JavaSlicer: A Program Slicer for Java. https://github.com/mistupv/JavaSlicer.
- [14] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In 2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE). IEEE, 31–53.
- [15] Chongzhou Fang, Ning Miao, Shaurya Srivastav, Jialin Liu, Ruoyu Zhang, Ruijie Fang, Asmita Asmita, Ryan Tsang, Najmeh Nazari, Han Wang, et al. 2023. Large language models for code analysis: Do llms really do their job? arXiv preprint arXiv:2310.12357 (2023).
- [16] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155 (2020).
- [17] Carlos Galindo, Sergio Perez, and Josep Silva. 2022. A program slicer for java (tool paper). In International Conference on Software Engineering and Formal Methods. Springer, 146–151.
- [18] Google. 2024. Gemma-7B. https://huggingface.co/google/gemma-7b
- [19] Yuxian Gu, Xu Han, Zhiyuan Liu, and Minlie Huang. 2021. Ppt: Pre-trained prompt tuning for few-shot learning. arXiv preprint arXiv:2109.04332 (2021).
- [20] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366 (2020).
- [21] Mark Harman and Robert Hierons. 2001. An overview of program slicing. software focus 2, 3 (2001), 85-92.
- [22] Eric Hartford. 2024. LeetCode Solutions Dataset. https://www.kaggle.com/datasets/erichartford/leetcode-solutions/data Accessed: 2024-08-07.
- [23] Xueqi Hou, Yu Zhao, Yanlin Liu, Zexuan Yang, Kai Wang, Ling Li, Xiaodong Luo, David Lo, John Grundy, and Hao Wang. 2023. Large language models for software engineering: A systematic literature review. arXiv preprint arXiv:2308.10620 (2023).
- [24] Jie Hu, Qian Zhang, and Heng Yin. 2023. Augmenting greybox fuzzing with generative ai. arXiv preprint arXiv:2306.06782 (2023).
- [25] Takashi Ishio, Shinji Kusumoto, and Katsuro Inoue. 2003. Program slicing tool for effective software evolution using aspect-oriented technique. In Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings. IEEE, 3–12.
- [26] Kimya Khakzad Shahandashti, Mithila Sivakumar, Mohammad Mahdi Mohajer, Alvine Boaye Belle, Song Wang, and Timothy Lethbridge. 2024. Assessing the Impact of GPT-4 Turbo in Generating Defeaters for Assurance Cases. In Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering. 52–56.
- [27] T. Kojima, S. S. Gu, M. Reid, Y. Matsuo, and Y. Iwasawa. 2022. Large language models are zero-shot reasoners. *NeuRIPS* 35 (2022), 22199–22213.
- [28] Bogdan Korel and Janusz Laski. 1988. Dynamic program slicing. Information processing letters 29, 3 (1988), 155–163.
- [29] Bogdan Korel and Juergen Rilling. 1997. Dynamic program slicing in understanding of program execution. In *Proceedings Fifth International Workshop on Program Comprehension. IWPC'97*. IEEE, 80–89.
- [30] Haonan Li, Yu Hao, Yizhuo Zhai, and Zhiyun Qian. 2023. The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models. arXiv preprint arXiv:2308.00245 (2023).
- [31] Dipeeka Luitel, Shabnam Hassani, and Mehrdad Sabetzadeh. 2024. Improving requirements completeness: Automated assistance through large language models. *Requirements Engineering* 29, 1 (2024), 73–95.
- [32] Thomas W MacFarland, Jan M Yates, Thomas W MacFarland, and Jan M Yates. 2016. Mann–whitney u test. *Introduction to nonparametric statistics for the biological sciences using R* (2016), 103–132.
- [33] Mohammad Mahdi Mohajer, Reem Aleithan, Nima Shiri Harzevili, Moshi Wei, Alvine Boaye Belle, Hung Viet Pham, and Song Wang. 2024. Effectiveness of ChatGPT for Static Analysis: How Far Are We?. In Proceedings of the 1st ACM International Conference on AI-Powered Software. 151–160.
- [34] Jagadiswar Mohapatra, Soumyajit Giri, Kumar Debasis, and Sujit Kumar Das. 2014. Slicing complex C++ program dynamically. In 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies. IEEE, 1765–1770.
- [35] OpenAI. [n.d.]. OpenAI: Advancing Digital Intelligence. https://www.openai.com/
- [36] OpenAI. 2024. GPT-3.5 Turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo
- [37] OpenAI. 2024. Hello GPT-4o. https://openai.com/index/hello-gpt-4o/ Accessed: 2024-08-07.

[38] Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. 2024. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13.

- [39] Ivan Pi et al. 2024. Slicer4J: A Dynamic Program Slicing Tool for Java. https://github.com/resess/Slicer4J.
- [40] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. arXiv preprint arXiv:2105.12655 (2021).
- [41] N Sasirekha, A Edwin Robert, and Dr M Hemalatha. 2011. Program slicing techniques and its applications. arXiv preprint arXiv:1108.1352 (2011).
- [42] Kimya Khakzad Shahandashti, Alvine Boaye Belle, Mohammad Mahdi Mohajer, Oluwafemi Odu, Timothy C Lethbridge, Hadi Hemmati, and Song Wang. 2024. Using GPT-4 Turbo to Automatically Identify Defeaters in Assurance Cases. In 2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW). IEEE, 46–56.
- [43] Kaya Stechly, Matthew Marquez, and Subbarao Kambhampati. 2023. Gpt-4 doesn't know it's wrong: An analysis of iterative prompting for reasoning problems. arXiv preprint arXiv:2310.12397 (2023).
- [44] Wenhao Sun, Chenchu Fang, Yang You, Yu Miao, Yuzhi Liu, Yiming Li, Guanjie Deng, Sheng Huang, Yifeng Chen, Qiushi Zhang, Hongyu Qian, Yang Liu, and Zhiyuan Chen. 2023. Automatic code summarization via ChatGPT: How far are we? *CoRR* abs/2305.12865 (2023), 1–13.
- [45] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. arXiv preprint arXiv:2403.08295 (2024).
- [46] Frank Tip. 1995. A survey of program slicing techniques. Journal of Programming Languages 3, 3 (1995), 121-189.
- [47] H. Touvron et al. 2023. Llama 2: Open and Efficient Foundation Language Models. Available at: https://arxiv.org/abs/2307.09288.
- [48] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. arXiv preprint arXiv:2203.11171 (2022).
- [49] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *arXiv preprint arXiv:2201.11903* (2022).
- [50] Mark Weiser. 1984. Program slicing. IEEE Transactions on software engineering 4 (1984), 352-357.
- [51] Jules White, Quchen Fu, Sam Hays, Michael Sandborn, Carlos Olea, Henry Gilbert, Ashraf Elnashar, Jesse Spencer-Smith, and Douglas C Schmidt. 2023. A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382 (2023).
- [52] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. 2023. ChatUniTest: a ChatGPT-based automated unit test generation tool. arXiv preprint arXiv:2305.04764 (2023).
- [53] Aashish Yadavally, Yi Li, Shaohua Wang, and Tien N Nguyen. 2024. A Learning-Based Approach to Static Program Slicing. Proceedings of the ACM on Programming Languages 8, OOPSLA1 (2024), 83–109.
- [54] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2023. No more manual tests? evaluating and improving chatgpt for unit test generation. *arXiv preprint arXiv:2305.04207* (2023).
- [55] Jianzhang Zhang, Yiyang Chen, Nan Niu, and Chuang Liu. 2023. Evaluation of chatgpt on requirements information retrieval under zero-shot setting. Available at SSRN 4450322 (2023).
- [56] Ying-Zhou Zhang. 2021. SymPas: symbolic program slicing. Journal of Computer Science and Technology 36 (2021), 397–418.
- [57] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2022. Automatic chain of thought prompting in large language models. arXiv preprint arXiv:2210.03493 (2022).