An Empirical Study of LLM for Code Analysis: Understanding Syntax and Semantics

Anonymous ACL submission

Abstract

Large language models (LLMs) demonstrate significant potential to revolutionize software engineering (SE). However, the high reliability and risk control requirements in software engineering raise concerns about the need for interpretability of LLMs. To address this concern, we conducted a study to evaluate the capabilities of LLMs and their limitations for code analysis in SE. Code analysis is essential in software development. It identifies bugs, security, and compliance problems and evaluates code quality and performance. We break down the abilities needed for LLMs to address SE tasks related to code analysis into three categories: 1) syntax understanding, 2) static behaviour understanding, and 3) dynamic behaviour understanding. We used four foundational models and assessed the performance of LLMs on multiple-language tasks. We found that, while LLMs are good at understanding code syntax, they struggle with comprehending code semantics, particularly dynamic semantics. Furthermore, our study highlights that LLMs are susceptible to hallucinations when interpreting code semantic structures. It is necessary to explore methods to verify the correctness of LLM's output to ensure its dependability in SE. More importantly, our study provides an initial answer to why the codes generated by LLM are usually syntax-correct but are possibly vulnerable.

1 Introduction

The ability of the large language model (LLM) to comprehend context, align instructions, and produce coherent content has attracted widespread attention from the software engineering (SE) community. Researchers started exploring how to use LLM in SE tasks related to code analysis (Xia and Zhang, 2023; Tian et al., 2023). However, although LLM is widely used and discussed in software engineering, a deep and systematic analysis of LLM's code analysis capabilities is vital and worthy of in-depth study. Code analysis is significant in modern software development to ensure the creation of secure, high-quality, and performant software. The basic ability to understand code syntax and semantics for code analysis is important for LLM in SE. Previous works have confirmed that some simple modifications without changing code semantics can mislead LLM to produce unexpected outputs (Yang et al., 2022; Liu et al., 2023c). As a result, there are two questions, *1) can LLM comprehend program semantics? 2) To what extent does LLM understand the code?* Firstly, it is unclear whether these LLMs can comprehend the syntax and semantics of the code for code analysis. Secondly, if these LLMs have a specific capability to comprehend syntax and semantics, the extent to which they can understand the semantics is also unknown.

To address the two issues, in this paper, we explore the ability of LLMs for code analysis in terms of understanding program syntax, static behaviours, and dynamic behaviours. Our work includes 4 state-of-the-art (SOTA) large language models, GPT4 (OpenAI, 2023), GPT3.5, StarCoder (Li et al., 2023) and CodeLlama-13b-instruct (Roziere et al., 2023). We design a set of code-related tasks (9 different tasks) on 2,560 code samples. Specifically, we design two tasks for code syntax understanding: Abstract Syntax Tree (AST) generation and expression matching to determine whether LLM can comprehend program syntax. Besides, we design five tasks, including Control Flow Graph (CFG) generation, Call Graph (CG) generation, data dependency analysis, taint analysis, and pointer analysis to explore whether LLM can statically approximate program behaviour similar to the traditional static analysis tools (Feist et al., 2019a; Cuoq et al., 2012). We further design and study two challenging tasks: code behaviour change detection and code behaviour variability reasoning to analyze the capability of LLM in dy-

namically analyzing program behaviours. Overall, the main contributions of our paper are summarized as follows: 1) We conduct a comprehensive study from different aspects to explore the capability of LLM for code analysis. We are the first to explore LLM's capability to understand code syntax, static, and dynamic behaviors. We study four state-of-the-art models: GPT4, GPT3.5, StarCoder and CodeLlama. We have made our code and data public on our website[1]. 2) We analyze LLM to understand code syntax, code static semantic structures, and code dynamic behaviours through diverse tasks. Our study suggests that LLM can comprehend code syntax rules and has certain abilities to understand code static behaviours but fails to comprehend dynamic behaviours. GPT4 is the best one to understand code syntax and semantics.

2 Motivation

We use ChatGPT to show our motivation example, a buggy function "bucketsort" in Figure 1 from QuixBugs (Lin et al., 2017). The Bucketsort algorithm requires splitting the array (i.e., "arr" in this function) into several buckets (i.e., "counts") and then sorting each bucket individually. The correct version is to replace the variable "arr" in the second loop with the variable "counts". ChatGPT can automatically fix this bug (Sobania et al., 2023). It seems that ChatGPT correctly comprehends this function semantics. However, a simple mutation can cause ChatGPT to produce incorrect results as shown in Figure 1 if we replace "arr" with "ccounts". If ChatGPT really understands the logic of the code, it should not be confused by the name change. Understanding the capabilities and limitations of LLM for code syntax and semantics is important, which can ensure that we can use LLM correctly and reasonably for code-related tasks. To address this challenge, in this paper, we provide a systematic and comprehensive study to investigate the capabilities of LLM for code analysis, i.e., what it can do and what its limitations are.

3 Study Design

3.1 Overview

We begin by examining the elemental abilities that LLMs must possess to tackle SE tasks related to code analysis effectively. Code analysis needs

[1] https://sites.google.com/view/chatgpt4se

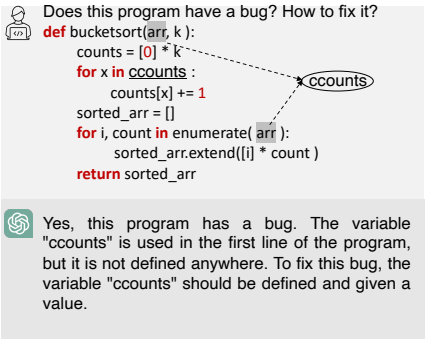


Figure 1: An semantic equivalent version of the buggy function by replacing "arr" with "ccounts" (May 2023).

code syntax, static behaviour, and running-time behaviour. In the previous works about code learning, instead of naively using code sequence, abstract syntax tree (AST)(Feng et al., 2020; Zhang et al., 2019; Wang et al., 2022a; Niu et al., 2022; Jiang et al., 2021; Wang et al., 2021), control and data flow (Guo et al., 2020; Ma et al., 2022a; Ahmad et al., 2021; Wang et al., 2022b; Zhou et al., 2019), dynamic execution trace (Ye et al., 2022; Jin et al., 2022; Pei et al., 2020) are used to build good code models. They have proven helpful for code models to learn code features for SE tasks. Since three types of information are critical to solving SE tasks, we try to answer the following three research questions (RQs): RQ1). Can LLM understand code syntax well? RQ2). Can LLM understand code static behaviors? RQ3). Can LLM understand code dynamic behaviors?

3.2 Code Syntax Understanding (RQ1)

AST Generation AST is the core structure in code analysis (Baxter et al., 1998b; Zhang et al., 2019). We prompt LLM to parse code into an AST, and then compare these ASTs with those generated by AST parsers to determine their meaningfulness. The ability to comprehend ASTs is fundamental for code models, as tokens in code serve distinct syntax roles. Understanding code syntax is crucial for addressing certain SE tasks, such as generating syntactically correct code.

Expression Matching This task aims to find a similar expression to the target mathematics expression. It is related to code-clone detection for Type-2 and Type-3 that requires understanding the syntax of the code (Baxter et al., 1998a; Koschke et al., 2006). The matched expression should have almost the same operators as the target expression. Figure 2 presents an example of this task, in which we try to find a similar expression with

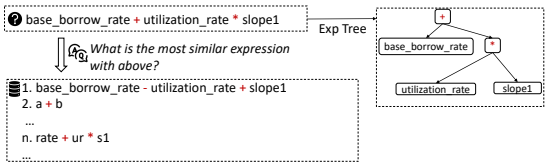


Figure 2: An example of the task of Expression Matching.

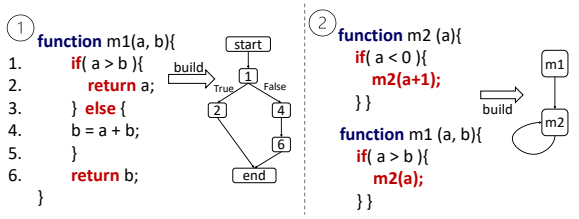


Figure 3: Examples of (1) Code Control Flow Graph (Python) and (2) Code Call Graph (Python).

"base_borrow_rate+utilization_rate*slope1". Without understanding the syntax role of tokens, finding similar expressions is not feasible. For instance, "base_borrow_rate-utilization_rate+slope1" may be incorrectly identified as a more similar expression to it than to "rate+ur*s1", if the operators "+" and "*" are not recognized.

3.3 Code Static Behavior (RQ2)

Control Flow Graph (CFG) Analysis Control flow graph analysis (CFG) is typically the first step in program analysis and understanding. We prompt LLM to construct the CFG from the input code. Figure 3 provides an example ① of this process. Understanding the CFG is critical for code models to identify relationships among statements. CFG is a core code structure in static analysis and is widely employed in software engineering to address various tasks (Cheng et al., 2019; Ferrante et al., 1987; Allen, 1970).

Call Graph (CG) Analysis The Call Graph is a data structure that depicts the invocation relationship among functions in a program. It is extensively employed in software engineering to understand program behaviors (Murphy et al., 1998). Figure 3 presents an example ② of a call graph with two methods. We prompt LLM to construct the call graph for the given code. Understanding the call graph is significant as it provides insights into the function relationships in the code.

Data Dependency Figure 4 provides an example ① in which "d" is data-dependent on "a". We prompt LLM to determine whether two given variables are data dependent in the code. Data dependency analysis is a powerful technique for code understanding (Guo et al., 2020) and optimizing

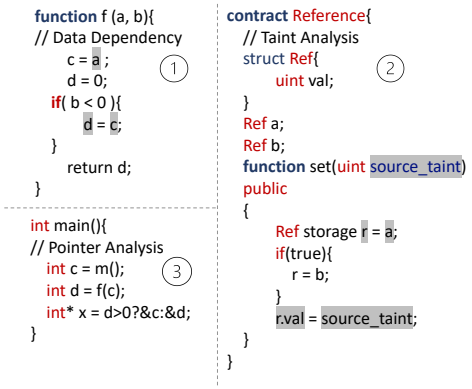


Figure 4: (1) Data Dependency Example (Python), (2) Taint Analysis Example (Solidity) and (3) Pointer Analysis (C).

code (Ferrante et al., 1984), as it can reveal data relationships among different variables in a program. Data dependency illustrates how data are propagated in the program, and it is extremely useful for code models to solve SE tasks such as vulnerability detection (Guo et al., 2020).

Taint Analysis This task is to find if a variable can be tainted by an external source. Figure 4 illustrates an example ② in which the variable "a" can be overwritten by "source_taint" via the storage variable "r". This task necessitates the reasoning ability of LLM based on data dependency analysis. Taint analysis (Kim et al., 2014) is strongly related to data dependency but also needs information from the call graph and the control flow graph to track how one data point is propagated in the program. It requires a deep understanding and reasoning of the semantics of the code in terms of execution order and relationship.

Pointer Analysis Pointer analysis is to find the data type of the pointers in code. Figure 4 illustrates an example ③ of pointer analysis. The pointer "x" can potentially point to either "c" or "d". Pointer analysis is widely used to detect vulnerabilities such as memory leakage. Pointer analysis (Smaragdakis et al., 2015; Hind and Pioli, 2000) requires an understanding of the dependency of data, the control flow, and the call graph. Point analysis also requires the inference to figure out what the current variable refers to. We prompt LLM to infer the referents of pointers. This task requires LLM to comprehend the code syntax and semantics in-depth.

3.4 Code Dynamic Behavior (RQ3)

Code Behavior Change Detection Any code change is highly possible to change the dynamic

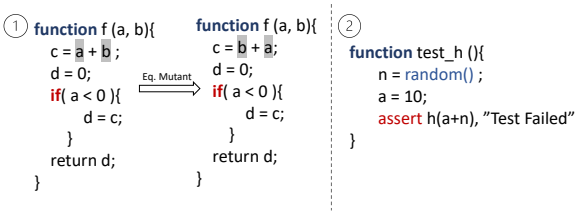


Figure 5: (1) Equivalent Mutant Example (Python) and (2) Flaky Test Reasoning Example (Python).

behaviors of the code. If LLM can really sense the behavior difference due to the minor code change, it can be as evidence that LLM can understand the change of code dynamic behavior well. This is called detecting equivalent mutants which is a critical problem in Mutation Testing (Papadakis et al., 2019). We check if LLM can find if the code minor change can change the code behavior. Figure 5 presents an example of an equivalent mutant ① by switching two variables.

Code Behavior Variability Reasoning Sometimes, the same code runs multiple times could behave differently. If LLM can understand the code dynamic behavior, it should know why the multiple-running outputs are inconsistent. In software testing, we call it Flaky Test. Flaky test is one challenging problem related to code dynamic behavior. Flaky test means the output inconsistency of one test when running multiple times. Thirty flaky reasons are summarized (Akli et al., 2022). Flaky tests are usually caused by some undetermined functions, the environment state and the execution schedule. Figure 5 presents one flaky example ② due to randomness. We prompt LLM to tell the reason why one test is flaky.

4 Evaluation Setup

We created a new dataset that was generated by the program analysis tools. The tasks and datasets utilized in our study are summarized in Table 1. We study two closed OpenAI models, GPT4 (OpenAI, 2023) and GPT3.5, and two open source models, StarCoder (Li et al., 2023) and CodeLlama-13b-Instruct (Roziere et al., 2023). For StarCoder, we use its conversational version, StarChat (Tunstall et al., 2023). For the evaluation, we employ two experts for code analysis. They first check the model output and then discuss how good the output is. We have different evaluation criteria for tasks that require manual inspection. We carefully design our prompt and please see more details in Appendix A. For dynamic behavior in code, we conducted an additional trial and provided examples, primarily

Table 1: Tasks and Datasets used in this study.

Task	Level	Programs	Dataset Size	LoC
AST	syntax	75	75	1,059
Expression Matching		4	32	4,238
CFG	static	75	75	1,059
CG		24	24	1,609
Data Dependence		13	992	62,606
Taint Analysis		13	830	62,052
Pointer Analysis		40	342	2,726
Code Behavior Variability	dynamic	13	65	1,615
Code Behavior Change		35	200	15728
Total		217	2,560	151,633

focusing on zero-shot learning. While it is acknowledged that some techniques such as RAG and SFG can enhance the model performance, we opted not to utilize them for two main reasons. Firstly, they introduce the potential for bias. It remains unclear whether improved outcomes are mainly attributable to which part. Secondly, zero-shot learning lies in the capacity of the models to comprehend and process code across various programming languages without the need for explicit, task-specific learning. This approach emphasizes the model's innate ability to generalize and adapt.

4.1 Evaluation Metrics

AST Generation We analyzed programs containing diverse syntax structures to effectively assess LLMs ability to understand code syntax. We classified LLM output as reasonable or not by analyzing the entire structure with a tolerance for the minor issues (missing trivial leaf nodes); 1) lack leaf nodes but keep the overall structure, we labeled it as 'Yes'. It means that LLM correctly generates the syntax type for the code token but does not give the token itself. 2) If the output provided a wrong structure with incorrect edges or lacked non-leaf nodes, we labeled it as 'No'. In the end, we count how many programs are reasonably handled, and also record and categorize the issues we find.

Expression Matching We implemented 32 reward equations strictly from the whitepapers of Web3 projects and then prompted LLM to find the corresponding implementation in the target projects. For each query, we fed the target code to LLM as comprehensively as possible and then checked if the corresponding implementation expression was in the top-5,10 and 20 outputs. For the open source model, we also computed the cosine similarity between our implemented expressions from the whitepapers and the expressions in the project.

Control Flow Graph (CFG) Generation We used the dataset that covers diverse control flows. We labeled the generated CFGs as reasonable or

not by comparing them with their corresponding programs. A CFG was considered reasonable if its overall structure was correct, with tolerating missing 1) start or end nodes, 2) a lack of edges to the end node, or 3) sequence statements being stacked in one node. CFGs that incorrectly represented control structure, 1) wrong branch and loop structures, and 2) fabricated non-existent nodes and edges, were labeled as not reasonable. Finally, we counted the number of reasonable CFGs and recorded and categorized all issues we identified.

Call Graph (CG) Generation We selected 24 public source code programs with at least three function calls. A generated call graph (CG) was considered reasonable if all of its call relationships were correct, even with some missing calling or redundant nodes. However, if the output contains non-existent call edges, we think it is not reasonable. The number of correct CGs generated was recorded.

Data Dependency and Taint Analysis We used Slither (Feist et al., 2019b) to extract 992 pairs of data-dependency variables and 830 externally tainted variables from 13 DeFi projects from Etherscan. We downsampled data to ensure the datasets were balanced. 1 indicated that it had a data-dependency fact or could be externally tainted, otherwise 0. We used F1 as a performance measurement.

Pointer Analysis We used Frama-C (Cuoq et al., 2012) to extract the possible set of variables for each pointer, which served as the ground truth. For each pointer, we collect a set of variables that it may refer to by prompting LLMs. We used the Jaccard index to measure the similarity between the ground truth set and the predicted set. We compute the Jaccard index for two scenarios: 1). Jaccard index for each pointer. It can measure how LLM behaves on this task; 2). Jaccard index for each program. It can measure whether LLM has a data shift problem, that is, behaves differently for different programs.

Code Behavior Change Detection We utilized MutantBench (van Hijfte and Oprescu, 2021) for this task. We use a script to extract the input-label data pairs. We randomly selected 100 equivalent mutants and 100 nonequivalent mutants. We use the F1 score in this task. Two different prompts were used for this task: one type of prompt did not include any example (zero-shot), and the other type included demonstration examples (few-shot).

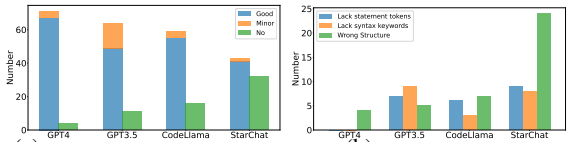
Code Behavior Variability Reasoning This dataset (Akli et al., 2022), which was manually collected, consists of 13 classes. To create our sample set, we randomly selected 5 samples for each class, resulting in a total of 65 samples. We prompted LLM to assign a label to each input and used accuracy as the performance metric. Similarly to the previous task, we employed two different prompts for this task. We also analyzed the prediction details for each class.

5 Experimental Results

All figures and the generated results can be found on our website(Anonymous, 2024).

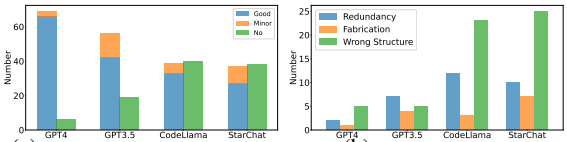
5.1 Code Syntax Understanding (RQ1)

AST Generation Figure 6a displays the number of reasonable and unreasonable ASTs generated by four LLMs. Blue represents the number of correct ASTs, orange represents the number of ASTs with minor issues and green represents unreasonable. We can see for GPT4 and GPT3.5, the majority of the generated ASTs were reasonable and few are minor. However, the open-source models CodeLlama-13-instruct and StarCoder are worse than OpenAI's models and have more unreasonable AST outputs. But CodeLlama is slightly better than StarCoder. We further investigate the issues of the generated ASTs and Figure 6b displays the number of issues that we identified. A single AST may exhibit multiple issues, and even reasonable ASTs might present some minor issues, as explained below. We categorized the found issues into three groups: missing statement tokens (blue bar), missing syntax tokens (orange bar), and wrong structure (green bar) as shown in Figure 6b. The missing-statement-tokens category indicates that some tokens in a statement were missing, such as the token "Printer" in "Printer.out.print(a);". The missing-syntax-tokens category indicates that some syntax keyword tokens were missing, such as "public" and "private" access modifiers. The wrong-structure category indicates that AST structures were incorrect, such as an incorrect if-else structure. The category Wrong Structure is serious and it means that AST contains faulty syntax structures. Reasonable ASTs with minor issues refer to these ASTs that typically had missing tokens of the statement or missing syntax trivial tokens, such as return type and access modifier. In our evaluation, GPT4 suffers from the least issues. GPT3.5 and CodeLlama have similar performance; GPT3.5 has more missing cases while CodeLlama has more wrong



(a) Reasonable VS. Unreasonable (b) Issue Distribution

Figure 6: AST Generation Result



(a) Reasonable VS. Unreasonable (b) Issue Distribution

Figure 7: CFG Generation Results

Table 2: Expression Matching Results of LLMs

Rank	GPT4	GPT3.5	StarCoder	CodeLlama
Top-5	27	28	4	4
Top-10	27	28	5	5
Top-20	27	28	5	5
Hit Rate	27/32	28/32	5/32	5/32

structures. StarCoder is the worst and suffers from lots of wrong structures.

Expression Matching Table 2 presents the results obtained using four LLMs. The first column shows the number of ranked equations that were considered in the results. GPT4 and GPT3.5 have a very close hit-rate performance, 27/32 and 28/32 in the top-5, top-10 and top-20. In contrast, StarCoder and CodeLlama have the same hit-rate performance, 5/32. Notice that, the number of StarCoder and CodeLlama in Table 2 are based on the consine similarity. The prompt results of both are quite bad. For StarCoder, none of the outputs is correct. 26/32 outputs are fabricated and 6/32 cases are wrongly matched. For CodeLlama, its prompt outputs only match 1 case and fabricate 28/32 cases.

In our investigation of the answers of GPT4 and GPT3.5, we discovered that both consider two expressions to be similar if they use similar operators, have similar orders, and have a similar number of variables. An intriguing observation is that, while GPT4 and GPT3.5 can identify the line number where the matched expression is located or the starting line number of the function containing the matched expression, none of the line numbers were accurate for these 32 expression matching samples in either case.

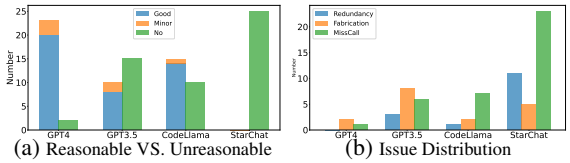
In general, LLM can understand the syntax structure of the code and the syntax roles of the code tokens. This ability allows it to act as an Abstract Syntax Tree (AST) parser.

5.2 Code Static Understanding (RQ2)

CFG Generation Figure 7a shows the number of reasonable and unreasonable CFGs generated by ChatGPT according to the predefined criteria. It can be seen that GPT4 achieves the best performance and majority of GPT4 outputs are correct, and few suffer from minor problems or wrong results. GPT3.5 is worse than GPT4 but is better than CodeLlama and StarCoder. CodeLlama and StarCoder have very close performances. Figure 7b shows the issues we identified, which we categorized into three groups: redundancy, fabrication, and wrong structure. Reasonable CFGs only can have the redundancy issue. The redundancy category means that there are meaningless nodes and removing them does not change anything such as null nodes, while the fabrication category means there are non-existent nodes (statements). The wrong-structure category refers to CFGs with incorrect structures (incorrectly represented loop statements and if-else statements). Redundancy issues are minor because they do not affect the control flow. Fabrication and wrong-structure issues are serious because they alter the control flow. We observe that GPT4 is still the best one and then the next is GPT3.5. Most results of CodeLlama and StarCoder suffer from the wrong structure. StarCoder suffers from more hallucinations than others. Upon examining the identified issues in the AST and CFG generation tasks, an interesting observation is that some serious problems typically arise with loop or if-else statements. LLM appears to have a weaker understanding of the syntax and static behavior of loop and if-else statements.

Call Graph Generation Figure 8a shows that GPT3.5, CodeLlama, and StarCoder were unable to generate reasonable CGs for most of the samples while GPT4 still performed well. StarCoder did not generate any reasonable output. Figure 8b illustrates the issues we found and categorized into 3 groups: redundancy, fabrication, and missing call. Redundancy means there are multiple edges between the functions that have the calling relationship. Fabrication means there are non-exist function calling. A missing call means the call edge is missed. It can be seen that GPT3.5 suffered from hallucination and GPT4 is obviously better than others. Missing calling is one common issue for the four LLMs. It indicates that GPT4 has a strong ability to understand code semantics.

Data Dependency and Taint Analysis Table 3 demonstrates the F1 score of the four LLMs on the data dependency task in the second row. GPT4



(a) Reasonable VS. Unreasonable (b) Issue Distribution

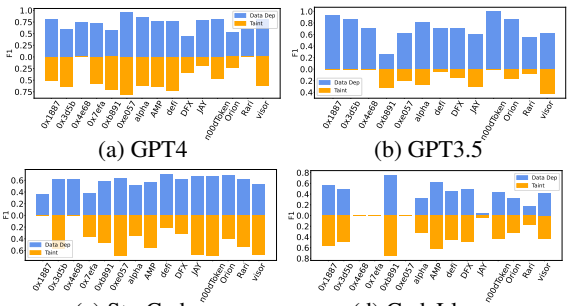
Figure 8: CG Generation Result

Table 3: Prediction Performance (F1) on Data Dependence and Taint Analysis on the entire datasets.

Task	GPT4	GPT3.5	CodeLLama	StarCoder
Data Dependency	**0.69**	0.68	0.44	0.6
Taint Analysis	0.44	0.15	0.39	**0.47**

achieves the best F1 score. GPT3.5 is inferior to GPT4. StarCoder is better than CodeLlama for this task, and worse than OpenAI's models. The comparison of the taint analysis is presented in the last row of Table 3. It can be seen that the performance of LLMs on this task is inferior to the data dependency analysis in terms of F1 (please note that we downsampled the taint dataset to make it balanced). All of them are worse than the random guess classifier that should have a 0.5 F1 score. Taint analysis is based on data dependency and requires the reasoning ability about the data flow. The results show that the studied models lack in-depth reasoning capabilities about the data flow. To assess whether LLM is suffering from the data-shift problem, we conducted an investigation as shown by Figure 9. We computed F1 for each project. Our findings indicate that LLM is significantly affected by the data-shift problem. As illustrated in the upper part (marked in blue) of Figure 9 on the data dependency, F1 scores differ for different projects, with a wide variance ranging from approximately 0. to 1.0. The lower part (marked in orange) in Figure 9 shows F1 scores on taint analysis, which display a variation ranging from 0 to about 0.8.

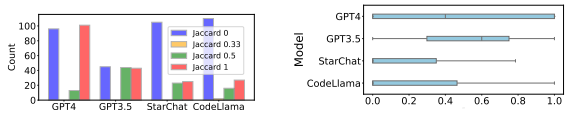
Pointer Analysis Initially, we determined the number of pointers for which LLMs fully predicted pointer analysis. Out of a total of 342 pointer samples from the 40 programs, only 105 were correctly



(a) GPT4 (b) GPT3.5

(c) StarCoder (d) CodeLlama

Figure 9: F1 of each project on Data Dependency (blue) and Taint Analysis (orange).



(a) GPT4, Jaccard Index of Each Pointer (b) GPT4, Jaccard Index of Each Pointer

Figure 10: Jaccard Index of Pointer Analysis.

predicted by GPT4, 43 were correctly predicted by GPT3.5, 27 and 25 were correctly predicted by CodeLlama and StarCoder respectively. We also noticed that some pointers were missed by LLMs. GPT4 predicts 211/342 pointers, GPT3.5 predicts 133/342 pointers, CodeLlama predicts 156/342 pointers and StarCoder predicts 153/342 pointers. Since one pointer can point to multiple variables, we computed the Jaccard Index between the predicted set and the ground truth set for each pointer, as shown in Figure 10a. We discovered that GPT4 is half good and half bad. It has the largest number for Jaccard Index 1 for each pointer but almost the same number of pointers have Jaccard Index with 0 values. GPT3.5 is inferior to GPT4. CodeLLama and StarCoder are not good in terms of the Jacrad Index for each pointer. We also computed the average Jaccard index for each program to assess whether LLM is affected by the data shift issue. We created a box plot of the mean Jaccard index of pointers from each project, which is illustrated in Figure 10b. We can see that the Jaccard Index variance is quite varied and suggests that LLMs are indeed affected by the data-shift problem.

GPT4 and GPT3.5 have the primary ability to perform code static analysis. CodeLlama and StarCoder are not as good as OpenAI models. However, during the analysis process, we find that all of them experience the issue of hallucination. Furthermore, the performance of LLMs can vary for a given task due to the data shift.

5.3 Code Dynamic Understanding (RQ3)

Code Behavior Change Detection Table 4 illustrates the performance F1 score of four LLMs in the detection of equivalent mutants, based on two types of prompts: prompt learning with or without example code (few-shot v.s. zero-shot) because examples are not always helpful to improve model performance (Wang et al., 2023). GPT4 and StarCoder perform well. CodeLlama failed to distinguish the equivalent mutant and non-equivalent mutant that has one small difference. Even if we tell CodeLlama that the two are equal, it still answers no. Although CodeLlama has the ability to

Table 4: Performance (F1) about Equivalent Mutant Detection.

Type	GPT4	GPT3.5	StarCoder	CodeLlama
few-shot	0.56	0.55	0.57	0.
zero-shot	0.67	0.54	0.62	0.

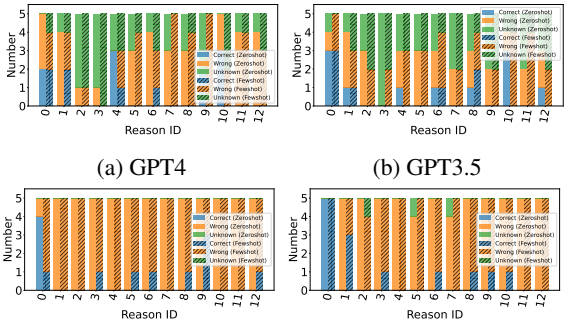
understand code syntax and the limited ability for static analysis as shown in the previous sections, it lacks the reasoning ability for code dynamic behaviors based on the code syntax and static structure understanding. StarCoder demonstrates a better dynamic-understanding ability than CodeLlama due to its pre-training data and tasks. To validate our hypothesis, we investigated the pre-training data and pre-training tasks of StarCoder (Li et al., 2023) and CodeLlama (Roziere et al., 2023). StarCoder uses the code commit data from GitHub while CodeLlama learns publicly available code only without any additional meta-level or temporal information such as git-commit info. Code commit info involves the code behavior change description and its impact.

Code Behavior Variability Reasoning For this task, we also employed two types of prompts, few-shot and zero-shot. The task comprises 13 classes, with each class containing five samples. To visualize the predicted label number for each class, we used bar figures, which are presented in Figure 11 (bar with slash for few-shot learning and bar without slash for zero-shot learning). Y-axis is the number and X-axis is the reason id. In these figures, the green bar represents the number of predictions that LLM is uncertain about, the orange bar represents the number of incorrect predictions, and the blue bar represents the number of correct predictions. We find that GPT4 and GPT3.5 are more conservative and prefer to answer unknown for most cases, while StarCoder and CodeLlama are quite confident about their outputs. We also find that the demo examples in the prompt can help StarCoder and CodeLlama improve their performance. However, overall, the four models do not perform well for this task.

LLM has limited ability to approximate code dynamic behavior and also suffers from the data shift problem.

6 Related Work

Probing Analysis: Probing analysis (Rogers et al., 2021) is a technique employed to examine and interpret the mechanisms of large language models (LLMs). Recently, some works have started to



(a) GPT4 (b) GPT3.5

(c) CodeLlama (d) StarCoder

Figure 11: Predictions of LLMs for Flaky Test

analyze code models, like CodeBERT, CodeT5, and UnixCoder. Wan et al. (2022) and Hernández López et al. (2022) analyze how code models learn syntax. Troshin and Chirkova (2022) and Ma et al. (2022b) analyze how code models represent semantics. But all of them study non-foundational models based on tuning classifiers. Our work studies the generation ability of LLM on code analysis.

LLM for SE: Tian et al. (2023); Xia and Zhang (2023); Sobania et al. (2023) investigated LLM's capabilities in code generation, program repair, and code summarization. Hou et al. (2023) systematically reviews the various applications of language models in software engineering. Fan et al. (2023) and Nguyen-Duc et al. (2023) discuss how LLM can change software engineering. None of them study the interpretability and reliability of LLM on code analysis, and our work is to estimate how LLM understands code that provides the interpretability of LLM for SE tasks. We are the first to study LLM on code analysis, especially for understanding code syntax and semantics, providing some confidence when employing LLMs to solve code tasks.

7 Conclusion

In this paper, we conduct a comprehensive empirical study to investigate the capabilities of LLM for code analysis. We used the new datasets created by the code tools, including four programming languages: Python, Java, C, and Solidity. Overall, our study indicates that LLM is capable of comprehending code syntax rules and has certain abilities to understand static behaviors of the code, but it cannot understand dynamic behaviors well. GPT4 achieves the best performance among all four models we included. We believe that our findings offer insights into LLM performance on SE tasks related to code analysis and guide follow-up researchers to effectively utilize LLM in the future to solve SE tasks.

8 Limitations

There are several limitations in this study. First, this study does not employ very large datasets for the analysis of each SE task. The reason is that we consider multiple tasks in this study and the dataset preparation is already very expensive. But these tasks are diverse and the analyzed data are created by ourselves via tools, which can help us conduct a comprehensive evaluation of LLMs in various scenarios related to software engineering on code analysis. Second, this study adopts manually designed prompts, however, there are several techniques for automatically designing prompts (Shin et al., 2020). It is possible that superior prompts can have better results in this study. We also do not explore how the examples in the prompt affect LLM. In our experiments, we observe that the choice of examples matters and how to find a good example for the prompt is the future independent work. Third, as a transformer architecture, LLM imposes a maximum token limitation, restricting the input context in our study. Lastly, we use nine basic and challenging tasks that are related to code analysis in Software Engineering. Some other code analysis tasks are also important but are not included in this study, such as dead code elimination. However, these tasks usually will use dependency and data flow analysis. It may limit our insights to LLMs on code analysis.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*.

Amal Akli, Guillaume Haben, Sarra Habchi, Mike Papadakis, and Yves Le Traon. 2022. Predicting flaky tests categories using few-shot learning. *arXiv preprint arXiv:2208.14799*.

Frances E. Allen. 1970. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, page 1–19, New York, NY, USA. Association for Computing Machinery.

Anonymous. 2024. Capabilities of chatgpt for code analysis: An empirical study.

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

Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998b. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377. IEEE.

Xiao Cheng, Haoyu Wang, Jiayi Hua, Miao Zhang, Guoai Xu, Li Yi, and Yulei Sui. 2019. Static detection of control-flow-related vulnerabilities using graph embedding. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 41–50. IEEE.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 2012. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247, Berlin, Heidelberg. Springer Berlin Heidelberg.

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. *arXiv preprint arXiv:2310.03533*.

Josselin Feist, Gustavo Greico, and Alex Groce. 2019a. Slither: A static analysis framework for smart contracts. In *Proceedings of the 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*, WETSEB '19, page 8–15. IEEE Press.

Josselin Feist, Gustavo Grieco, and Alex Groce. 2019b. Slither: a static analysis framework for smart contracts. In *2019 IEEE/ACM 2nd International Workshop on Emerging Trends in Software Engineering for Blockchain (WETSEB)*, pages 8–15. IEEE.

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*.

Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1984. The program dependence graph and its use in optimization. In *International Symposium on Programming*, pages 125–132. Springer.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349.

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*.

José Antonio Hernández López, Martin Weyssow, Jesús Sánchez Cuadrado, and Houari Sahraoui. 2022. Ast-probe: Recovering abstract syntax trees from hidden representations of pre-trained language models. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–11.

Michael Hind and Anthony Pioli. 2000. Which pointer analysis should i use? *SIGSOFT Softw. Eng. Notes*, 25(5):113–123.

Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2023. Large language models for software engineering: A systematic literature review. *arXiv preprint arXiv:2308.10620*.

Xue Jiang, Zhuoran Zheng, Chen Lyu, Liang Li, and Lei Lyu. 2021. Treebert: A tree-based pre-trained model for programming language. In *Uncertainty in Artificial Intelligence*, pages 54–63. PMLR.

Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 1631–1645, New York, NY, USA. Association for Computing Machinery.

Junhyoung Kim, TaeGuen Kim, and Eul Gyu Im. 2014. Survey of dynamic taint analysis. In *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*, pages 269–272. IEEE.

Rainer Koschke, Raimar Falke, and Pierre Frenzel. 2006. Clone detection using abstract syntax suffix trees. In *2006 13th Working Conference on Reverse Engineering*, pages 253–262. IEEE.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. 2017. Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, SPLASH Companion 2017, page 55–56, New York, NY, USA. Association for Computing Machinery.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023a. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35.

Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023b. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. *ACM Comput. Surv.*, 55(9).

Shangqing Liu, Bozhi Wu, Xiaofei Xie, Guozhu Meng, and Yang Liu. 2023c. Contrabert: Enhancing code pre-trained models via contrastive learning. *arXiv preprint arXiv:2301.09072*.

Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie M Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. 2022a. Graphcode2vec: generic code embedding via lexical and program dependence analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories*, pages 524–536.

Wei Ma, Mengjie Zhao, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiexin Zhang, Wenhan Wang, and Yang Liu. 2022b. Are code pre-trained models powerful to learn code syntax and semantics?

Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. 1998. An empirical study of static call graph extractors. *ACM Trans. Softw. Eng. Methodol.*, 7(2):158–191.

Anh Nguyen-Duc, Beatriz Cabrero-Daniel, Adam Przybylek, Chetan Arora, Dron Khanna, Tomas Herda, Usman Rafiq, Jorge Melegati, Eduardo Guerra, Kai-Kristian Kemell, et al. 2023. Generative artificial intelligence for software engineering–a research agenda. *arXiv preprint arXiv:2310.18648*.

Changan Niu, Chuanyi Li, Vincent Ng, Jidong Ge, Liguo Huang, and Bin Luo. 2022. Spt-code: Sequence-to-sequence pre-training for learning source code representations. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 2006–2018, New York, NY, USA. Association for Computing Machinery.

OpenAI. 2023. Gpt-4 technical report. *ArXiv*, abs/2303.08774.

Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers*, volume 112, pages 275–378. Elsevier.

Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680*.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2021. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Taylor Shin, Yasaman Razeghi, Robert L Logan IV, Eric Wallace, and Sameer Singh. 2020. Autoprompt: Eliciting knowledge from language models with automatically generated prompts. *arXiv preprint arXiv:2010.15980*.

Yannis Smaragdakis, George Balatsouras, et al. 2015. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69.

Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. 2023. An analysis of the automatic bug fixing performance of chatgpt. *arXiv preprint arXiv:2301.08653*.

Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F Bissyandé. 2023. Is chatgpt the ultimate programming assistant–how far is it? *arXiv preprint arXiv:2304.11938*.

Sergey Troshin and Nadezhda Chirkova. 2022. Probing pretrained models of source codes. In *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP*, pages 371–383, Abu Dhabi, United Arab Emirates (Hybrid). Association for Computational Linguistics.

Lewis Tunstall, Nathan Lambert, Nazneen Rajani, Edward Beeching, Teven Le Scao, Leandro von Werra, Sheon Han, Philipp Schmid, and Alexander Rush. 2023. Creating a coding assistant with starcoder. *Hugging Face Blog*. Https://huggingface.co/blog/starchat.

Lars van Hijfte and Ana Oprescu. 2021. Mutantbench: an equivalent mutant problem comparison framework. In *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 7–12.

Yao Wan, Wei Zhao, Hongyu Zhang, Yulei Sui, Guandong Xu, and Hai Jin. 2022. What do they capture? a structural analysis of pre-trained language models for source code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 2377–2388.

Weishi Wang, Yue Wang, Steven Hoi, and Shafiq Joty. 2023. Towards low-resource automatic program repair with meta-learning and pretrained language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 6954–6968, Singapore. Association for Computational Linguistics.

Wenhan Wang, Kechi Zhang, Ge Li, Shangqing Liu, Zhi Jin, and Yang Liu. 2022a. A tree-structured transformer for program representation learning. *arXiv preprint arXiv:2208.08643*.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. 2021. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *arXiv preprint arXiv:2108.04556*.

Xin Wang, Yasheng Wang, Yao Wan, Jiawei Wang, Pingyi Zhou, Li Li, Hao Wu, and Jin Liu. 2022b. Code-mvp: learning to represent source code from multiple views with contrastive pre-training. *arXiv preprint arXiv:2205.02029*.

Chunqiu Steven Xia and Lingming Zhang. 2023. Keep the conversation going: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. *arXiv preprint arXiv:2304.00385*.

Zhou Yang, Jieke Shi, Junda He, and David Lo. 2022. Natural attack for pre-trained models of code. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1482–1493.

He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th International Conference on Software Engineering*, ICSE '22, page 1506–1518, New York, NY, USA. Association for Computing Machinery.

Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32.

A Prompt Design

GPT-based models utilize the prompt-based learning paradigm (Liu et al., 2023a). The design of the prompt can significantly impact the performance of the model. To design better prompts, iteratively, we employ ChatGPT to optimize our initial prompts and then evaluate these optimized prompts by some trial queries. Specifically, we prompt ChatGPT with the message, "*Act as a prompt optimizer and optimize the following prompt for [TASK DESCRIPTION]. The prompt is [PROMPT]*", to help us generate the prompts. [TASK DESCRIPTION] is the task description placeholder and [PROMPT] is the draft prompt placeholder. For each task, we have multiple draft prompts, and then we manually evaluate them using some task data to observe their differences. Finally, according to the experience obtained from the trials, we summarize our prompt templates as *role prompt* and *instruction prompt*. Through our continuous optimizations, our prompt may not be the best, but it is excellent.

Role prompt assigns a specific role to LLM, providing a task context for the model to effectively generate the desired output. Its template is shown below,

```
You are [ROLE] for [LANG]. [TASK
DESCRIPTION]. [OUTPUT FORMAT].
The input is [INPUT].
```

where the placeholder [ROLE] denotes the specific role assigned to LLM. We define six roles: AST

parser, expression tree matcher, control flow graph analyzer, call graph analyzer, code static analyzer, and pointer analyzer. [LANG] refers to the programming language used for the code analyzed. [TASK DESCRIPTION] outlines the expected task for LLM to perform. [OUTPUT FORMAT] provides the output specification. [INPUT] serves as a placeholder for the code under analysis.

Instruction prompt does not assign a specific role to LLM, instead, they provide a command. These prompts are typically useful for tasks involving multiple roles or those without any applicable roles. The template for the instruction prompt is defined as follows:

```
Please analyze [LANG]. [DOMAIN
KNOWLEDGE]. Here are some
examples [EXAMPLE CODE]. Please
identify if [TASK DESCRIPTION].
[OUTPUT FORMAT]. The input is
[INPUT].
```

where [LANG] specifies the used programming language for the analyzed code. [DOMAIN KNOWLEDGE] explains the domain knowledge relevant to the task. [EXAMPLE CODE] provides sample code related to domain knowledge for task demonstration. [TASK DESCRIPTION] describes the task instruction. [OUTPUT FORMAT] outlines the output specification. [INPUT] serves as a placeholder for the code under analysis. Notice that different LLMs may need different prompt formats. For StarCoder (Li et al., 2023) and CodeLlama (Roziere et al., 2023), we also refer to their original papers and adapt the prompt design for both through adding special tokens. For StarCoder, its format is "<|system|>\n<|end|>\n<|user|>\n{query}<|end|>\n<|assistant|>." The placeholder {query} will be replaced by our prompts. For CodeLlama-13b-instruct, we insert the special tokens into our prompt with this format "<s>[INST]{query}[/INST]". For the manual evaluation tasks, we created GPTs tools based on GPT4 to maximize optimization prompt for GPT4.

In our study, we employ the role-based prompt for RQ1 and RQ2 where the prompt does not include examples. Owing to the complexity of these tasks, which entail numerous patterns, prompting the model to generate outputs using illustrated examples might potentially downgrade the performance. For example, by presenting the data-dependence example (Figure 4 ①), the model may hyperfocus on this specific instance, consequently overlooking other situations of data-dependence due to in-context learning (Liu et al., 2023b). Wang et al. (2023) report that few examples in the prompt may not help improve the performance. In contrast, for tasks under RQ3, we resort to instruction prompts given that this type of prompt works a little better during our prompt-designed trials due to their unfitness for resolution by a single role. We present all used prompts on our website (Anonymous, 2024).