

Systematically Understanding of Code Semantic Interpretation for LLMs

Jia Zhao

*School of Information Science and Technology, Beijing Forestry University, Beijing, China
finmzhao@163.com*

Abstract: Understanding and interpreting code is a crucial task in intelligent software engineering, aiding developers and users in adjusting code for correctness and robustness. The emergence of large language models (LLMs) provides new perspectives for code interpretation tasks. However, current LLM-based code interpretation remains restricted to limited dimensions, lacks a unified evaluation standard, and is missing a comprehensive and systematic assessment methodology. To address this issue, this paper proposes an LLM code understanding evaluation method based on a multi-granularity voting mechanism, aiming to systematically investigate and analyze LLMs' performance in code interpretation tasks. First, we carefully select code snippets from open-source GitHub projects and preprocess them for LLM analysis. Second, we use identical prompts and inputs to test three popular LLMs, recording their output. During this process, we apply prompt engineering techniques to specific target code snippets and conduct repeated experiments to explore the impact of prompt engineering on LLM-generated code explanations. Next, we design evaluation metrics to quantify the LLM outputs and assess their effectiveness based on the obtained scores. Experimental results demonstrate significant differences in code analysis and generation capabilities among the evaluated general-purpose LLMs from different vendors when given identical prompts and inputs. When multiple dimensions are considered in evaluating the generated content, different LLMs exhibit varying strengths in different aspects. Additionally, applying specific prompt engineering techniques can moderate the discrepancies in code analysis and generation capabilities among different LLMs.

Keywords: LLM, Code Interpretation, Java, C++, Code Understanding, Software Development

1. Introduction

General-purpose large language models (LLMs) have demonstrated strong application potential in code generation and code comprehension helping software engineering practitioners improve work efficiency and enhance the user experience of software applications. With the emergence of large-scale language models, many researchers are focusing on the development of LLM-driven code interpretation tools or investigating LLMs' capabilities in generating code explanations—both of which have yielded significant results.

Despite these advancements, existing research on LLM-based code interpretation remains limited in scope, making it difficult to comprehensively assess an LLM's ability to understand and analyze code across different dimensions. Additionally, there is no clear consensus on how LLMs perform

when analyzing code at varying granularities. Furthermore, prior studies have limitations in evaluating the impact of prompt engineering on LLMs' code explanation capabilities and in comparing differences in performance across various LLMs.

To address these issues, this study conducts a series of experiments. While ensuring coverage of different granularities and programming languages, we carefully select source code from popular open-source GitHub projects as the experimental dataset. For the tested models, we select three of the most popular LLMs: GPT-4o, Claude 3.5 Sonnet, and ERNIE 4.0 Turbo. The experiment consists of two stages. In the first stage, we employ a few-shot prompting approach, sequentially prompting each of the three LLMs with the same target code snippets and recording their generated explanations. In the second stage, we select a subset of the target code from the first stage and conduct repeated experiments using specific prompt engineering techniques to investigate their impact on LLM-generated code explanations.

We propose a multi-granularity voting-based evaluation method, where the outputs of different LLMs under the same multi-granularity input conditions are compared and rated through a voting mechanism. We design scoring metrics based on linear mapping and weighted averaging and conduct a quantitative evaluation based on the generated ratings. Finally, we analyze the score data to derive conclusions. Experimental results show that, under limited prompting conditions, different general-purpose LLMs exhibit significant differences in completeness and accuracy when generating code analysis. When LLM-generated code explanations involve multiple dimensions—including semantics, performance, and security—different LLMs exhibit strengths in different areas. After applying prompt engineering techniques, constrained inputs effectively reduce the differences in code analysis and generation capabilities among various LLMs. Prompt engineering, by precisely designing the structure of input language and defining constraints, enables models to better understand task requirements. Well-designed prompts are a key factor in minimizing performance differences among models in the field of code interpretation.

2. Background

2.1. LLM-based code analysis

With the rise of LLMs, LLM-based code analysis has gradually become a research hotspot in the field of software engineering.

Fang et al. [1] evaluate the effectiveness of LLMs in code analysis, particularly in relation to obfuscated code. Szabó Z. and Bilicki V. explore methods of leveraging GPT-3.5 for static analysis of frontend application source code to detect and prevent security vulnerabilities [2]. Leinonen et al. [3] examine the application of LLMs, such as Codex, to improve the clarity and helpfulness of programming error messages for novice programmers.

Several efforts aim to develop code analysis tools or methods based on LLMs. Deng et al. [4] introduced TitanFuzz, which achieved improvements in code coverage and uncovered bugs. Jain et al. [5] introduced the Jigsaw system, which enhances code synthesis capabilities by integrating LLMs with program analysis and synthesis techniques.

Some of these studies are limited to specific dimensions or struggle to integrate LLM outputs across different dimensions for a comprehensive evaluation. Additionally, they have not reached clear conclusions on how LLMs perform when handling target code at varying granularities.

2.2. Code interpretation

Code interpretation is an important research direction in software engineering and has received increasing attention in recent years with the integration of large language model (LLM) technology.

Pan et al. [6] proposed that while LLMs show promise in code translation, significant challenges remain in achieving accuracy and reliability. Nam et al. [7] introduced an IDE plugin that leverages LLMs to assist developers in understanding code. Leinonen et al. [8] suggest the potential applications of LLMs in programming education by comparing code explanations generated by students and LLMs. Stephen et al. [9] found that GPT-3 can produce diverse and contextually relevant explanations. In their 2023 study [10], they explored integrating LLMs into an interactive e-book on web software development, with the majority of students finding them helpful. Sami et al. [11] find that LLMs can effectively create diverse and challenging programming exercises along with comprehensive explanations, aiding in educational contexts.

These studies have not explored the impact of prompt engineering techniques on LLM-generated content. Additionally, there are still limitations in the scientific validity of human evaluation in assessing LLM-generated code analysis, as well as in the methodologies used to compare and generalize the code analysis capabilities of different LLMs.

3. Approach

3.1. Workflow

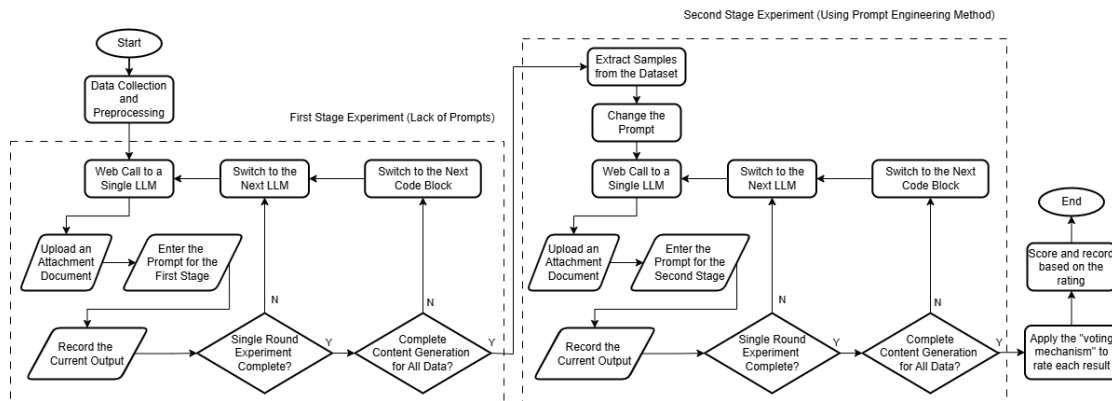


Figure 1: The workflow of LLM interpretation evaluation

At the beginning of the study, data collection and preprocessing were conducted. (see Section 4.1)

In the first phase, a minimal-prompt approach was employed, where the input was limited to basic output requirements and documents containing the code segments. Each of the three selected LLMs was sequentially tasked with generating content for each target code segment.

In the second phase, a subset of the target code segments from the first phase was selected, and prompt engineering techniques were applied. The input prompts differed from those in the first phase, while the code files remained the same. The three LLMs were again sequentially tasked.

After completing both experimental phases, a “multi-granularity voting mechanism” (Section 3.2) was applied to evaluate and rate the LLM responses. Finally, results were summarized, and conclusions were drawn.

3.2. LLM-based code evaluation via adaptive voting

The experiment utilized three different LLMs (Section 4.3). In each round, a text document containing the target code was inputted identically into all three models. Their responses were treated as votes, and their results were compared to evaluate each model's response quality. To facilitate evaluation, an inclusion relationship was defined for LLM outputs: if the output of model A is entirely found within the output of model B, and model B contains additional content not found in model A, then

model A's output is considered covered by model B's output. Based on these relationships, the LLM responses in each experiment round were categorized into five levels, as detailed in Table 1:

Table 1: Rating criteria for LLM output

No.	Response Evaluation	Detailed Explanation
1	Most Limited	The response is entirely covered by another model's output, and it does not fully cover any other model's output.
2	Limited	The response is entirely covered by another model's output but also fully covers another model's output.
3	Average	The response is not fully covered by any other model, but it lacks content that appears in both of the other models' responses.
4	Fairly Complete	The response is not fully covered by any other model and does not miss content mentioned in both other models' responses.
5	Most Complete	The response includes all content found in the outputs of the other two models.

3.3. Multi-dimensional and multi-granularity evaluation

LLM evaluation of code comprehension was conducted along three dimensions: semantic understanding, performance, and security.

- Semantic Understanding refers to the meaning or intent of the code, reflecting its function and purpose—i.e., what the code is designed to accomplish.
- Performance assesses how efficiently the code executes under specific load conditions, including factors like execution speed and resource consumption.
- Security evaluates whether the code is designed to resist malicious attacks and unauthorized access, ensuring the protection of user data and system functionality.

Using the criteria in Section 3.2, each LLM's responses were rated for these three aspects. ①To quantify the overall analysis capability of each LLM, a weighted average score was calculated across semantic understanding, performance, and security (see Section 4.2 for details). ②The target code segments were categorized by granularity into two levels: function and file/class. ③Restricting LLM responses to a single paragraph was intended to evaluate their ability to provide concise yet comprehensive summaries. The effectiveness of this approach improves as input code segments increase in size and complexity. To highlight this, when assessing each LLM's ability to analyze different granularities of code, the overall score was weighted accordingly (see Section 4.2).

3.4. Input and prompt engineering methods

To ensure fairness, the experiment used identical prompts for all LLMs. To assess the impact of prompt engineering techniques on LLMs' ability to explain code, the experiment was divided into two phases based on different prompt styles:

In the first phase, for each source code segment, all tested LLMs followed these steps:

①Upload the target code as a .txt document.

②Provide the following prompt: "Summarize the semantics of the code in the document in a single paragraph. Evaluate the code's performance in a single paragraph. Evaluate the code's security in a single paragraph."

Restricting responses to a single paragraph facilitated direct comparison of response quality while assessing LLMs' capability to concisely summarize and analyze different lengths of code.

In the second phase, employed prompt engineering techniques, including few-shot prompting, chain-of-thought reasoning, and role-playing. A subset of 25% of the dataset (four code segments) was selected for this phase. The detailed prompt inputs for this stage are illustrated in Figure 2.



Figure 2: Experiment prompts in the second phase

4. Evaluation

4.1. Datasets

The data used in the experiment consists of 16 selected source code segments from popular open-source projects on GitHub. The minimum star count for the selected projects is 19.3k, and the maximum is 195k (as of November 21, 2024). All selected projects adhere to good programming practices, feature clear functionality, a well-structured codebase, and a modular design. Additionally, the source files containing the selected code segments include detailed comments. The dataset covers two granularity levels: "functions" and "files/classes," with a significant difference in the number of lines between code segments at different granularities. The dataset encompasses four programming languages: C, C++, Python, and Java, with four code segments for each language.

During preprocessing, all comments were removed. The selected code segments were then saved as "txt" format files for ease of use as input for the LLM. Specific details of the dataset are presented in Table 2.

Table 2: Details of test dataset

No.	Repository URL	Language	Granularity	Source File Name	Number of Lines
1	https://github.com/Genymobile/scrcpy/blob/master/app/src/adb/adb_parser.c	C	file/class	adb_parser.c	156
2	https://github.com/git/git/blob/master/builtin/am.c	C	file/class	am.c	1770
3	https://github.com/raysan5/raylib/blob/master/examples/audio/audio_mixed_processor.c	C	function	audio_mixed_processor.c	58

Table 2: (continued)

4	C/searching/pattern_search/rabin_karp_search.c at master · TheAlgorithms/C · GitHub	C	function	rabin_karp_search.c	48
5	https://github.com/TheAlgorithms/C/blob/master/searching/pattern_search/rabin_karp_search.c	C++	file/class	calc.cpp	151
6	https://github.com/microsoft/calculator/blob/main/src/CalcManager/CEngine/scicomm.cpp	C++	file/class	scicomm.cpp	828
7	https://github.com/microsoft/calculator/blob/main/src/CalcManager/Header%20Files/CalcEngine.h	C++	file/class	CalcEngine.h	164
8	https://github.com/TheAlgorithms/C-Plus-Plus/blob/master/search/jump_search.cpp	C++	function	jump_search.cpp	29
9	https://github.com/pallets/flask/blob/main/src/flask/app.py	Python	file/class	app.py	864
10	https://github.com/pallets/flask/blob/main/src/flask/cli.py	Python	file/class	cli.py	991
11	https://github.com/scrapy/scrapy/blob/master/scrapy/contracts/default.py	Python	file/class	default.py	105
12	https://github.com/liuweicss/Algorithms-Python/blob/master/searches/hill_climbing.py	Python	function	hill_climbing.py	61
13	https://github.com/macrozheng/mall/blob/master/mall-search/src/main/java/com/macro/mall/search/service/impl/EsProductServiceImpl.java	Java	file/class	EsProductServiceImpl.java	251
14	https://github.com/Stirling-Tools/Stirling-PDF/blob/main/src/main/java/org/apache/pdfbox/examples/signature/ValidationTimeStamp.java	Java	file/class	ValidationTimeStamp.java	63
15	https://github.com/PhilJay/MPAndroidChart/blob/master/MPChartLib/src/main/java/com/github/mikephil/charting/animation/ChartAnimator.java	Java	file/class	ChartAnimator.java	102
16	https://github.com/TheAlgorithms/Java/blob/master/src/main/java/com/thealgorithms/strings/HorspoolSearch.java	Java	function	HorspoolSearch.java	25

4.2. Metrics

1)Score Setting

To facilitate quantitative evaluation and comparison, the five levels in Section 3.2 are linearly mapped to scores ranging from 0 to 10, as follows:

$$S = \left(\frac{I - I_{min}}{I_{max} - I_{min}} \right) \times (S_{max} - S_{min}) + S_{min}$$

The scores corresponding to the five levels are shown in Table 3.

Table 3: Scores corresponding to each level

No.	Score
1	0
2	2.5
3	5
4	7.5
5	10

Using the above standards, scores were assigned to each large model's responses in the three aspects of "semantics," "performance," and "security" for each round of the experiment.

2)Multidimensional Integrated Evaluation

To quantitatively represent the overall ability of different large models to analyze code, the experiment calculates the weighted average score of the large models' responses in the three aspects of "semantics," "performance," and "security."

The ISO/IEC 25010 quality model provides eight major characteristics and 31 sub-characteristics for evaluating software quality [12]. Based on the statistics, we can derive the number of sub-characteristics involved in different code analysis dimensions in the large models' output, as shown in Table 4.

Table 4: The correspondence between sub-characteristics and code analysis dimensions

Code Analysis Dimension	Number of Involved Sub-characteristics	Proportion of Occurrences in Sub-characteristics
Semantics	13	54.2%
Performance	3	12.5%
Security	8	33.3%

Based on this, the score weight values for each evaluation dimension are derived, as shown in Table 5.

Table 5: Score weight values for each evaluation dimension

Attribute	Weight w
Semantics	0.542
Performance	0.125
Security	0.333

The experimental score for a single code segment in a given large model is as follows:

$$s_t = \sum s_i w_i$$

3) Multi-granularity Integrated Evaluation

Limiting the output of large models to "a full paragraph of text" aims to evaluate the model's overall ability to analyze and generalize code. The experimental effect is better when the granularity and scale of the input code segment are sufficiently large. To reflect this, the experiment calculates the weighted average of the comprehensive test scores for all code segments based on the granularity. The weights are determined by the average number of lines in code segments at different granularities. In the five code segments with a "function" granularity, the average number of lines is 44.2; in the eleven code segments with a "file/class" granularity, the average number of lines is 495. The ratio of the average number of lines between the two code segment types is used as the ratio of their weight values, as shown in Table 6 (rounded to three decimal places).

Table 6: Weight values of the two types of code segment

Code Segment Type	Weight p
file/class	0.918
function	0.082

The comprehensive analysis ability score for a single large model is the weighted average of the experimental scores for all code segments tested on that model, as follows:

$$G_t = \frac{\sum s_{tj} p_i}{\sum n_i p_i}$$

4.3. Experimental setup

The experiment utilized three LLMs from different providers: GPT-4o, Claude 3.5 Sonnet, and ERNIE 4.0 Turbo, all accessed via web invocation method.

Verification process: The content of each response generated by the large models is manually summarized, and the voting mechanism from Section 3.2 is applied to compare the different attributes (semantics, performance, security) of each code segment in the outputs of the three large models. Scores are assigned based on the levels and metrics (Section 4.2).

4.4. Results

The results of the first phase of the experiment are shown in Table 7.

Table 7: Experimental results of phase 1

No.	Granularity	Semantics			Performance			Security			Total Score		
		GPT	Claude	ERNIE	GPT	Claude	ERNIE	GPT	Claude	ERNIE	GPT	Claude	ERNIE
1	file/class	10	0	2.5	7.5	5	0	7.5	5	7.5	8.86	2.29	3.85
2	file/class	7.5	7.5	7.5	7.5	7.5	5	7.5	7.5	7.5	7.50	7.50	7.19
3	function	10	0	0	7.5	7.5	5	7.5	5	7.5	8.86	2.60	3.12
4	function	7.5	7.5	0	7.5	7.5	0	7.5	5	5	7.50	6.67	1.67
5	file/class	5	7.5	7.5	7.5	7.5	5	7.5	7.5	5	6.15	7.50	6.36
6	file/class	10	0	0	5	5	5	5	7.5	5	7.71	3.12	2.29
7	file/class	7.5	7.5	7.5	7.5	7.5	0	5	7.5	7.5	6.67	7.50	6.56
8	function	5	5	5	5	5	5	2.5	10	0	4.17	6.67	3.34
9	file/class	7.5	5	5	7.5	5	7.5	5	7.5	0	6.67	5.83	3.65
10	file/class	7.5	5	0	7.5	7.5	5	5	7.5	5	6.67	6.15	2.29
11	file/class	10	0	0	7.5	5	0	7.5	7.5	0	8.86	3.12	0.00
12	function	0	10	0	7.5	7.5	5	0	10	0	0.94	9.69	0.63
13	file/class	7.5	7.5	7.5	7.5	7.5	0	7.5	7.5	7.5	7.50	7.50	6.56
14	file/class	7.5	5	7.5	7.5	7.5	0	7.5	5	0	7.50	5.31	4.07
15	file/class	7.5	7.5	7.5	0	10	0	7.5	7.5	5	6.56	7.81	5.73
16	function	5	7.5	7.5	7.5	5	7.5	7.5	7.5	5	6.15	7.19	6.67
Total Score		7.86	4.82	4.68	6.61	6.81	2.58	6.53	7.06	4.50	7.26	5.82	4.36

The results of the second phase of the experiment (applying prompt engineering methods) are shown in Table 8. (The numbers in parentheses in the "No." column represent the correspondence between the code segments used in this phase and those in the first phase of the experiment.)

Table 8: Phase 2 experimental results

No.	Granularity	Semantics			Performance			Security			Total Score		
		GPT	Claude	ERNIE	GPT	Claude	ERNIE	GPT	Claude	ERNIE	GPT	Claude	ERNIE
1(1)	file/class	7.5	7.5	7.5	7.5	7.5	5	7.5	5	7.5	7.50	6.67	7.19
2(5)	file/class	7.5	7.5	7.5	7.5	7.5	7.5	7.5	5	7.5	7.50	6.67	7.50

Table 8: (continued)

3(9)	file/class	7.5	7.5	7.5	7.5	7.5	7.5	7.5	7.5	7.5	7.50	7.50	7.50
4(16)	function	7.5	7.5	7.5	7.5	7.5	7.5	7.5	5	5	7.50	6.67	6.67
Total Score		7.50	7.50	7.50	7.50	7.50	6.69	7.50	5.81	7.43	7.50	6.94	7.37

The experimental results can be illustrated in Figure 3.

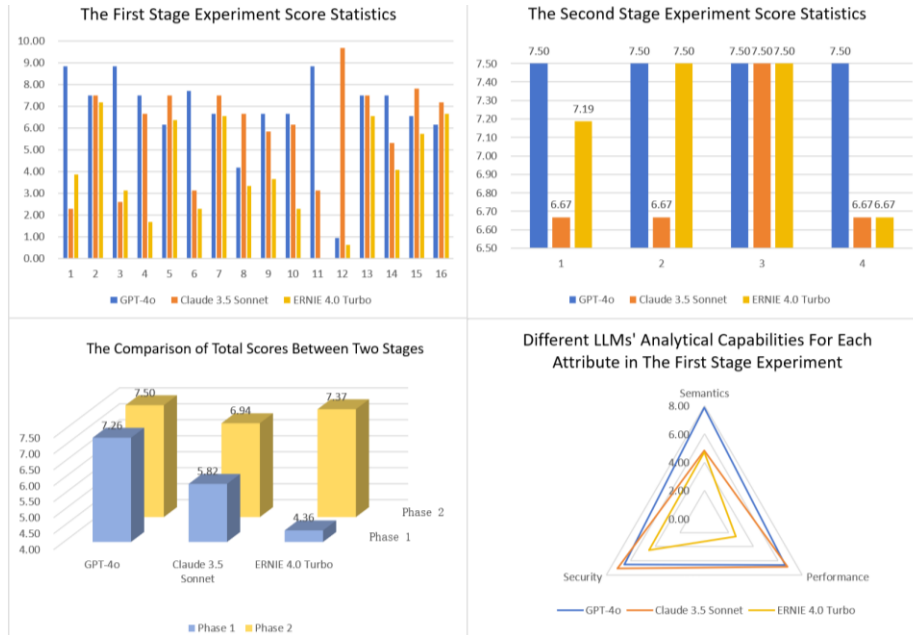


Figure 3: Experimental results

Based on the experimental results, we can achieve the following findings:

1. In the first phase of the experiment, GPT-4o achieved the highest overall score in most cases (68.75% of code segments), particularly in the "file/class" granularity experiment (72.73%). This indicates that GPT-4o has stronger generalization ability in understanding code while maintaining completeness. GPT-4o and Claude 3.5 Sonnet demonstrated relatively stable performance, with extremely low scores of "0" occurring in 6.25% and 8.33% of cases, respectively. In contrast, ERNIE 4.0 Turbo experienced "0" scores much more frequently (37.5%).
2. When analyzing the generated content separately for "semantics," "performance," and "security," GPT-4o showed the strongest semantic interpretation generation ability in the absence of prompts. Claude 3.5 Sonnet exhibited the strongest analysis generation ability for "performance" and "security" without prompts. However, when prompts were sufficient, its "security" analysis ability weakened, resulting in the lowest overall score.
3. The total score range in the second round of experiments, applying prompt engineering methods, was 0.56, which represents an 80.69% reduction compared to the 2.90 range in the first round. In the first phase, with limited prompts, GPT-4o showed a clear advantage in "semantic" interpretation generation, while ERNIE 4.0 Turbo performed inadequately in "performance" and "security" analysis generation. However, in the second phase, the aforementioned gaps (the total score range across LLMs for each attribute) significantly narrowed. This suggests that, with sufficient prompt engineering restrictions, the output differences between different large models gradually decrease.

5. Conclusion

Applying large language models (LLMs) to code interpretation and analysis generation is currently a highly popular research area. In this context, this paper addresses the limitations of existing work by proposing, for the first time, a large model-based code comprehension evaluation method based on a multi-granularity voting mechanism. It also introduces a set of quantitative metrics for comparing the code interpretation capabilities of different large models. This approach provides new ideas and insights for future research on LLMs in the field of code comprehension. Through a comparative method, this paper evaluates the ability of different large models to understand and analyze code across different programming languages, granularities, and prompt contexts. It offers ideas and references for future research in this domain. Regarding the experimental results, this paper finds significant differences in code interpretation capabilities among various large models, with varying strengths in different analytical dimensions. Proper prompt engineering can substantially reduce performance differences between models, leading to more consistent code interpretation outputs.

In the future, we will focus on ensuring the scientific validity of the experimental results by using a larger dataset. The current results still have limitations in terms of the accuracy of content correctness. Going forward, we will explore methods for evaluating the correctness of generated content in code interpretation and develop more rigorous and precise quantitative metrics.

References

- [1] Fang C, Miao N, Srivastav S, et al. Large language models for code analysis: Do {LLMs} really do their job?[C]//33rd USENIX Security Symposium (USENIX Security 24). 2024: 829-846.
- [2] Szabó Z, Bilicki V. A new approach to web application security: Utilizing gpt language models for source code inspection[J]. *Future Internet*, 2023, 15(10): 326.
- [3] Leinonen J, Hellas A, Sarsa S, et al. Using large language models to enhance programming error messages[C]//*Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023: 563-569.
- [4] Deng Y, Xia C S, Peng H, et al. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models[C]//*Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 2023: 423-435.
- [5] Jain N, Vaidyanath S, Iyer A, et al. Jigsaw: Large language models meet program synthesis[C]//*Proceedings of the 44th International Conference on Software Engineering*. 2022: 1219-1231.
- [6] Pan R, Ibrahimzada A R, Krishna R, et al. Lost in translation: A study of bugs introduced by large language models while translating code[C]//*Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024: 1-13.
- [7] Nam D, Macvean A, Hellendoorn V, et al. Using an llm to help with code understanding[C]//*Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 2024: 1-13.
- [8] Leinonen J, Denny P, MacNeil S, et al. Comparing code explanations created by students and large language models[C]//*Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*. 2023: 124-130.
- [9] MacNeil S, Tran A, Mogil D, et al. Generating diverse code explanations using the gpt-3 large language model[C]//*Proceedings of the 2022 ACM conference on international computing education research-volume 2*. 2022: 37-39.
- [10] MacNeil S, Tran A, Hellas A, et al. Experiences from using code explanations generated by large language models in a web software development e-book[C]//*Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*. 2023: 931-937.
- [11] Sarsa S, Denny P, Hellas A, et al. Automatic generation of programming exercises and code explanations using large language models[C]//*Proceedings of the 2022 ACM Conference on International Computing Education Research-Volume 1*. 2022: 27-43.
- [12] Estdale J, Georgiadou E. Applying the ISO/IEC 25010 quality models to software product[C]//*European Conference on Software Process Improvement*. Cham: Springer International Publishing, 2018: 492-503.