Order Matters! Unveiling Large Language Models' Input Order Bias in Software Fault Localization

Anonymous ACL submission

Abstract

Large Language Models (LLMs) show great promise in software engineering tasks like Fault Localization (FL) and Automatic Program Repair (APR). This study examines how input order and context size affect LLM performance in FL, a key step for many downstream software engineering tasks. We test different orders for methods using Kendall Tau distances, including "perfect" (where ground truths come first) and "worst" (where ground truths come last). Our results show a strong bias in order, with Top-1 accuracy falling from 57% to 20% when we reverse the code order. Breaking down inputs into smaller contexts helps reduce this bias, narrowing the performance gap between perfect and worst orders from 22% to just 1%. We also look at ordering methods based on traditional FL techniques and metrics. Ordering using *DepGraph*'s ranking achieves 48% Top-1 accuracy, which is better than more straightforward ordering approaches like $CallGraph_{DFS}$. These findings underscore the importance of how we structure inputs, manage contexts, and choose ordering methods to improve LLM performance in FL and other software engineering tasks.

1 Introduction

Software development has significantly transformed with the emergence of Large Language Models (LLMs) like ChatGPT (OpenAI, 2023). These tools have revolutionized how developers code, debug, and maintain software systems (Zhang et al., 2023). LLMs are widely adopted for their ability to simplify and accelerate development workflows, providing insights into complex tasks such as code generation and comprehension (Abedu et al., 2024; Lin et al., 2024).

Recent research has explored the use of LLMs in various software engineering tasks, including Fault Localization (FL) (Kang et al., 2024; Yang et al., 2024) and Automatic Program Repair (APR) (Zhang et al., 2024; Xia et al., 2024), which show great potential for automatically resolving real-world issues in large code bases. In particular, FL is a foundational step in the process, where the LLM processes structured lists to locate potential faulty code that requires fixing. Hence, high FL accuracy is instrumental to APR and automatic issue resolution.

While LLMs have demonstrated strong reasoning capabilities, prior research from other domains highlights a sensitivity to the order of input information. Studies have shown that LLMs perform better when information is presented in a sequence aligned with logical steps, with accuracy dropping significantly when the order is randomized (Chen et al., 2024). Additionally, LLMs exhibit a primacy effect, often prioritizing earlier information in prompts (Wang et al., 2023). Although these findings are well-documented in reasoning tasks, whether such sensitivities extend to software engineering scenarios like FL is unclear. Since FL involves analyzing ordered lists of methods or elements, the sequence in which this information is presented may influence the model's ability to identify faults.

This paper investigates how input order and context size affect the performance of large language models (LLMs) in Fault Localization (FL). We used Defects4J (Just et al., 2014) benchmark, a widely used dataset in software engineering for evaluating FL techniques. First, we evaluate whether the order of methods impacts the LLM's ability to rank and identify faults by generating various input orders using Kendall Tau distance (Cicirello, 2019), including *perfect* (ground truth methods first) and *worst* (ground truth methods last) orders. We found that the LLM's performance is significantly influenced by input order, with Top-1 accuracy dropping from 57% to 20% when the perfect method list is reversed, indicating a strong order bias. Next, we explore segmenting large inputs into smaller contexts to address observed order biases. We observed that segmenting input sequences into smaller contexts reduces this bias; for example, the Top-1 gap between *Perfect-Order* and *Worst-Order* rankings decreased from 22% at a segment size of 50 to just 1% at a size of 10. Finally, we tested traditional FL and metrics-based ordering methods and found that using FL techniques improved results, with *DepGraph* outperforming *Ochiai* by 16% in Top-1 accuracy, while simpler strategies like *CallGraph* and *LOC* produced similar outcomes. These results highlight the importance of input order, context size, and effective ordering methods for enhancing LLM-based fault localization.

In summary, our contributions are as follows:

- Method order significantly impacts LLM performance, with Top-1 accuracy dropping from 57% in *Perfect-Order* (ground truths first) to 20% in *Worst-Order* (ground truths last).
- We demonstrate that dividing input sequences into smaller segments effectively mitigates order bias, reducing the Top-1 performance gap between *perfect* and *worst* orders from 22% to just 1%.
- Ordering with different metrics and FL strategies significantly impacts outcomes. Ordering based on *DepGraph* achieves 48% Top-1 accuracy, 13.4% higher than $CallGraph_{BFS}$. However, simpler methods like $CallGraph_{DFS}$ reach 70.1% Top-10 accuracy, highlighting their practicality in resource-constrained environments.

2 Background and Related Works

2.1 Fault Localization

Fault Localization (FL) (Wong et al., 2016) is a critical software engineering task identifying specific program parts responsible for a failure. It is particularly essential in large and complex codebases, where manually finding faults can be time-consuming and error-prone. FL saves significant developer effort and serves as a cornerstone for many downstream software engineering tasks such as Automatic Program Repair (APR) (Le Goues et al., 2021), debugging automation (Zamfir and Candea, 2010), and performance optimization (Woodside et al., 2007). The process begins with some indication of a fault, typically indicated by a failing test, which serves as the starting point. The input for FL often consists of a set of methods or code elements executed during the failing test case. FL aims to produce a ranked list of the most likely fault locations, providing developers with a focused starting point for investigation and resolution. Its significance lies in facilitating effective debugging and establishing the groundwork for workflows that automate and optimize the software development lifecycle.

2.2 Related Works

Spectrum-based and Supervised Fault Localization. Traditional methods such as Spectrum-Based Fault Localization (SBFL) use statistical techniques to assess the suspiciousness of individual code elements. (Abreu et al., 2007). The intuition is that the code elements covered by more failing tests and fewer passing tests are more suspicious. While simple and lightweight, these techniques, such as Ochiai (Abreu et al., 2009), often struggle with achieving high accuracy in complex systems. To improve accuracy, supervised techniques like DeepFL (Li et al., 2019) and Grace (Lou et al., 2021) incorporate features such as code complexity, historical fault data, and structural relationships using machine learning and Graph Neural Networks (GNNs). DepGraph (Rafi et al., 2024) further refines this by leveraging code dependencies and changing history for better fault ranking.

LLM-Based Fault Localization. Recent advances in Large Language Models (LLMs) have demonstrated significant potential for FL by leveraging their ability to analyze both code and natural language. Trained on extensive programming datasets, LLMs can understand code structure, interpret test failures, and even suggest fixes (Kang et al., 2024; Wu et al., 2023; Pu et al., 2023). Building on these capabilities, LLM agents extend LLM functionalities by incorporating features like memory management (Zhou et al., 2023) and tool integration (Roy et al., 2024), enabling them to autonomously execute tasks described in natural language. These agents can also adopt specialized roles, such as developers or testers, to enhance their domain-specific reasoning and improve problem-solving workflows (Hong et al., 2024; White et al., 2024).

Several recent works have leveraged LLMs for FL. Wu et al.(Wu et al., 2023) leverage test failure data to identify faulty methods or classes, enabling

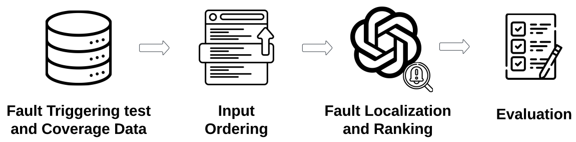

Figure 1: An overview of our overall approach

context-aware reasoning directly from the input. AutoFL(Kang et al., 2024) enhances LLM capabilities by integrating tools to fetch and analyze covered classes and methods, providing additional insights for FL. AgentFL (Qin et al., 2024) takes a more specialized approach, using agents with a Document-Guided Search method to navigate codebases, locate faults, and prioritize suspicious methods. In contrast, Agentless (Xia et al., 2024) simplifies FL with a three-phase workflow—localization, repair, and validation—eliminating the need for agents or complex tools. These tasks often involve handling large contexts, as LLMs process extensive lists of methods or code snippets, making the structure of input data a critical factor.

Prior research shows that LLMs are sensitive to input order, impacting their reasoning and decision-making. For example, studies have shown that LLMs perform better when premises are presented in a sequence aligned with logical reasoning steps, with accuracy dropping substantially when the order is randomized (Chen et al., 2024). Additionally, LLMs exhibit a primacy effect, prioritizing earlier information in prompts, influencing their outputs (Wang et al., 2023). In fault localization (FL), where code is analyzed as ordered lists of methods, the presentation order may affect the ranking of suspicious methods. This paper explores how the sequence of code elements and context window size influence LLM performance in fault localization tasks.

3 Methodology and Experiment Design

This section describes our overall approach to conducting experiments, summarized in Figure 1. First, we collect coverage information, including details about failing tests, stack traces, and the methods covered. Next, we generate various input orderings using Kendall Tau distance and different metrics. We pass this information along with the prompt to the LLMs for fault localization. Finally, we evaluate the model's bias by calculating the Top-K accuracy across different orderings. Below, we discuss more in detail.

Project	#Faults	LOC	#Tests	Fault-triggering Tests
Cli	39	4K	94	66
Codec	18	7K	206	43
Collections	4	65K	1,286	4
Compress	47	9K	73	72
Csv	16	2K	54	24
Gson	18	14K	720	34
JacksonCore	26	22K	206	53
JacksonXml	6	9K	138	12
Jsoup	93	8K	139	144
Lang	64	22K	2,291	121
Math	106	85K	4,378	176
Mockito	38	11K	1,379	118
Time	26	28K	4,041	74
Total	501	490	15,302	901

Table 1: The studied projects from Defects4J.

3.1 Methodology

Prompt Design. We use LLMs to rank the most suspicious methods in fault localization tasks by analyzing *failing tests*, *stack traces*, and *covered methods*. We designed the prompts to be simple so we could better study the order bias. Our prompts consist of two primary components: a 1) *System Message* and a 2) *User Message*, to guide the LLM in ranking suspicious methods. The *system message* establishes the task by instructing the LLM to analyze a *failing test*, its *stack trace*, and a list of *covered methods* during execution. The LLM ranks the top ten methods in descending order based on its analysis of suspicion. To ensure consistency in the generated output, the *system message* specifies the required output format as a JSON structure, which includes method identifiers and their corresponding ranks.

The *user message* provides the input data specific to a *failing test*, including the failing test code, the minimized *stack trace*, and the *covered methods*. Following prior works (Kang et al., 2024; Wu et al., 2023), we retain only the information directly relevant to fault localization for *stack traces*, discarding unrelated lines such as those from external libraries or other modules. This reduction enhances clarity and ensures that the LLM only processes essential data to identify the root cause of the failure. Covered methods are presented as an ordered list, serving as the candidate set for ranking. Additional details are provided in the appendix A.4.

3.2 Experiment Design

Benchmark Dataset. We conducted the experiment on 501 faults across 13 projects from the Defects4J benchmark (V2.0.0) (Just et al., 2014). Defects4J is a widely used benchmark in the software

Analyze the provided failing test, stack trace, and covered methods to localize faults and rank the top 10 most suspicious methods.

Test Code:
`{test_code}`
Stack Trace:
`{stack_trace}`
Covered Methods:
`{covered_methods}`

The output should follow the JSON format below:
JSON Format:

```
{
  "methodB": "rank",
  "methodA": "rank",
  ...
}
```

Figure 2: Prompt for Fault Localization.

engineering community for fault localization (Lou et al., 2021; Sohn and Yoo, 2017; Chen et al., 2022; Zhang et al., 2017; Rafi et al., 2024). It provides a controlled environment for reproducing real-world bugs from a variety of projects, which differ in type and size. The benchmark includes both faulty and fixed project versions, along with associated test cases (including failing ones), metadata, and automation scripts, which facilitate research in FL, testing, and program repair.

Table 1 gives detailed information on the projects and faults we use in our study. We excluded a few projects from Defects4J due to compilation errors that limited test coverage for most bugs. In total, we studied 501 faults and over 1.4K fault-triggering tests (i.e., failing tests that cover the fault). Note that since a fault may have multiple fault-triggering tests, there are more fault-triggering tests than faults.

Evaluation Metrics. We perform our fault localization process at the method level in keeping with prior work (Li et al., 2019; Lou et al., 2021; Vancsics et al., 2021; Rafi et al., 2024; Kang et al., 2024). Namely, we aim to identify the source code methods that cause the fault. We apply the following commonly-used metrics for evaluation:

Accuracy at Top-N. The Top-N metric measures the number of faults with at least one faulty program element (in this paper, methods) ranked in the top N. The results are a ranked list based on the suspiciousness score. Prior research (Parnin and Orso, 2011) indicates that developers typically only scrutinize a limited number of top-ranked faulty elements. Therefore, our study focuses on Top-N, where N is set to 1, 3, 5, and 10.

Following prior LLM-based FL studies (Kang et al., 2024; Wu et al., 2023), we did not use metrics like Mean First Rank (MFR) and Mean Average Rank (MAR) to measure how early faulty methods are ranked and their average position (Lou et al., 2021; Li et al., 2019). These metrics are unsuitable for LLM-based approaches, which makes it difficult to provide a specific score because LLM is a language model.

Implementation and Environment. To collect test coverage data and compute results for baseline techniques, we utilized Gzoltar (Campos et al., 2012), an automated tool that executes tests and gathers coverage information. For the LLM-based components, we employed OpenAI's GPT-4o mini, which currently points to gpt-4o-mini-2024-07-18, which has a context window of 128,000 tokens and can output 16,384 tokens at once (OpenAI, 2024). We used LangChain v0.2 to streamline the process of our experiment (Langchain, 2024). To minimize the variations in the output, we set the temperature parameter to 0.

4 Experiment Results

4.1 RQ1: Does the order in which the model processes code elements impact its performance?

Motivation. LLMs often struggle to reason over long input sequences, known as order bias, where the model prioritizes input tokens at the beginning or end of the sequence (Wang et al., 2023). While order bias has been studied in NLP tasks, such as deductive and mathematical reasoning (Chen et al., 2024), its impact on software engineering tasks remains under-explored. Order is crucial in software engineering tasks, such as fault localization and program repair, where the model must reason over a long code sequence. Therefore, in this RQ, we investigate how code sequence order affects LLM accuracy in fault localization.

Approach. To study how code order sequences affect LLM-based fault localization, we create baselines with varying orderings. The first baseline, *Perfect-Order*, places faulty methods (ground truths) at the top, followed by non-faulty methods, ordered by their *call-graph* (see Appendix A.1 for details) to minimize arbitrariness. Our intuition is that *Perfect-Order* serves as an idealized benchmark to test the hypothesis that prioritizing faulty methods should yield the highest accuracy if LLMs favor earlier orders due to their sequential processing nature. We then generate four additional baselines by adjusting the order using Kendall Tau distance (Cicirello, 2019), which measures the correlation between two lists (i.e., 1 = perfect alignment with the *Perfect-Order*, -1 = complete misalignment with the *Perfect-Order*). From *Perfect-Order*, we derive: ① *Random-Order* ($\tau = 0$; methods shuffled randomly), ② *Worst-Order* ($\tau = -1$; faulty methods last), ③ *Moderately Perfect-Order* ($\tau = 0.5$; partial alignment), and ④ *Moderately Inverted-Order* ($\tau = -0.5$; partial misalignment). Comparing these baselines to *Perfect-Order* allows us to assess how deviations from the *Perfect-Order* affect FL results. Finally, we evaluate the model's FL performance by ranking methods based on suspiciousness and measuring Top-K accuracy. For instance, a Top-1 score of 50% indicates that 50% of 501 faults' faulty methods were ranked first.

Results. ***LLMs exhibit a bias toward the initial input order, achieving approximately 38% higher Top-1 accuracy for Perfect-Order compared to Worst-Order.*** Figure 3 shows the results of the experiments. For *Perfect-Order*, the model identifies 57.4% of faults in the Top-1 accuracy, while *Moderately Perfect-Order* reduces the model's fault detection to 26.1% (Δ 31.1%). As Kendall Tau decreases, the accuracy declines further, reaching the lowest (19.4%) for *Worst-Order*, despite the ***code context remaining identical, except for the code order***. These results highlight key limitations in how LLMs process code, suggesting they may ***rely more on surface-level patterns than on a deep understanding of code semantics***.

This trend persists across all Top-K metrics. For Top-3, the model detects 70.9% of bugs in the *Perfect-Order*, decreasing to 38.6% for *Moderately Perfect-Order*, which then stabilizes to 33% for both *Random-Order* and *Moderately Inverted-Order*, then decreasing further to 26.3% with the *Worst-Order*. We see similar trends for Top-5 and Top-10, with *Perfect-Order* detecting the most faults, with 78% and 86% faults, respectively, compared to the lowest fault detection of 30.5% and 35.2% for *Worst-Order*. ***These findings suggest that LLMs are biased toward methods listed earlier in the input, indicating a potential order bias when analyzing code sequences.***

The low variability in standard deviation (STDEV) across multiple runs suggests consistent order bias. To ensure the reliability of our findings on order bias, we conducted the experiments three times. Across all Top-K results, the STDEV remains stable, ranging from 0.00 to 2.49. For instance, the highest STDEV of 2.49 for Top-3 indicates minimal variations, with only two methods changing position across runs. This consistency demonstrates that ***order bias is not an artifact of randomness but a persistent limitation in how LLMs process code sequences.***

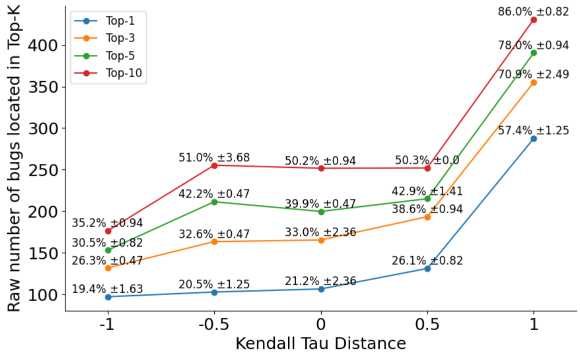

Figure 3: Top-K fault localization. The x-axis represents the number of bugs located, and the data points on the lines indicate the percentage of bugs identified out of the total (with standard deviation) at each Top-K position for various Kendall Tau (τ) values.

> The LLM's fault localization performance is significantly impacted by input order, with Top-1 accuracy dropping from 57% to 20% when the method list is reversed, indicating a bias toward early-presented data.

4.2 RQ2: Does limiting context window help reduce the bias towards order?

Motivation. In RQ1, we identified order bias in the sequence in which methods are presented in the zero-shot prompt. We hypothesize that a larger context window might amplify the bias toward method order, as the LLM processes all methods simultaneously and may weigh their order more heavily when generating responses. In this RQ, we investigate how the context window influences order bias. Specifically, we examine how segmenting the input sequence into smaller independent segments affects LLM's performance in software engineering tasks, such as fault localization, where both context size and order play a crucial role in reasoning.

Approach. We investigate whether a divide-and-conquer approach, where the input sequence is split into smaller contexts and each subset is reasoned individually, can reduce this bias. We divide an ordered list of N methods, $M = \{m_1, m_2, \ldots, m_N\}$, into $\max(\lceil N/S \rceil, 1)$ contiguous segments. Each segment $M_i \subseteq M$ (for $i =$

$\{1, 2, \ldots, J\}$) contains up to S methods, ensuring $|M_i| \leq S$. If $S > N$, the entire list M forms a single segment ($J = 1$). For this study, we experiment with five segment sizes $S \in \{10, 20, 30, 40, 50\}$. In each segment M_i, the model ranks the Top-K suspicious methods R_i, and the results R_i are summarized into G_i. For the subsequent segment S_{i+1}, the prompt includes both the M_{i+1} and the contextual information from G_i. This iterative context-passing approach allows the model to re-rank methods based on combined information from previous segments. By incrementally varying the segment size (S), we analyze how the context window impacts order bias in fault localization. Specifically, we compare performance across two extreme ordering sequences: *Perfect-Order* ($\tau = 1$) and *Worst-Order* ($\tau = -1$) (defined in RQ1) to assess whether the iterative context-passing effectively mitigates order bias, improving reliability across diverse ordering sequences.

Results. ***The size of the context window impacts fault localization results, with larger context windows exhibiting a stronger order bias.*** Table 2 presents the Top-K scores across different context segments for *Perfect-Order* and *Worst-Order*. When the context is provided in larger segments (e.g., segment size 50), the model detects 278 bugs (55.5%) in Top-1 with the *Perfect-Order*, while the *Worst-Order* identifies only 170 bugs (33.9%), around 22% fewer bugs than the *Perfect-Order*. The large difference in Top-1 shows a significant order bias towards the order of the input method list. This is also evident in the Top 3, 5, and 10. For example, in the Top-10, the Perfect ranking reaches 408 (81.4%) compared to 292 (58.3%) for the Worst ranking, detecting around 23% more bugs.

As the segment size decreases, the difference between the *Perfect-Order* and *Worst-Order* becomes smaller across all Top-K. For a segment size of 40, the model detects approximately 54% of bugs in Top-1 with the *Perfect-Order*, which is 20% more than the 34% bugs detected with the *Worst-Order* ranking. This pattern holds for Top-3, Top-5, and Top-10 as well. At a segment size of 30, the difference in Top-1 narrows further to 17%, with *Perfect-Order* identifying 51% of bugs compared to 34% for *Worst-Order*. When the segment size is reduced to 20, the *Perfect-Order* detects around 49% of bugs in Top-1, while *Worst-Order* detects 37%, shrinking the difference to 12%. This trend continues for Top-3, Top-5, and Top-10, where the performance gap between the two rankings becomes progressively smaller.

Ordering	Seg. Size	Top-1	Top-3	Top-5	Top-10
Perfect	10	217 (43.3%)	295 (58.9%)	313 (62.5%)	338 (67.5%)
Worst	10	211 (42.1%)	298 (59.5%)	330 (65.9%)	362 (72.3%)
Perfect	20	247 (49.3%)	311 (62.1%)	343 (68.5%)	374 (74.7%)
Worst	20	186 (37.1%)	265 (52.9%)	288 (57.5%)	335 (66.9%)
Perfect	30	261 (52.1%)	326 (65.1%)	347 (69.3%)	382 (76.2%)
Worst	30	175 (34.9%)	236 (47.1%)	266 (53.1%)	309 (61.7%)
Perfect	40	270 (53.9%)	328 (65.5%)	347 (69.3%)	390 (77.8%)
Worst	40	171 (34.1%)	223 (44.5%)	249 (49.7%)	284 (56.7%)
Perfect	50	278 (55.5%)	338 (67.5%)	368 (73.5%)	408 (81.4%)
Worst	50	170 (33.9%)	224 (44.7%)	248 (49.5%)	292 (58.3%)

Table 2: A comparison of fault localization performance across techniques and segments. The table shows bugs detected in the Top-1, Top-3, Top-5, and Top-10 positions using *Perfect-Order* and *Worst-Order* orders across various segments.

At the smallest segment size of 10, there is nearly no difference in Top-1 (only a 1% gap between *Perfect-Order* and *Worst-Order*). Interestingly, for Top-3, Top-5, and Top-10, the model performs slightly better using the *Worst-Order* compared to the *Perfect-Order*. These findings suggest that ***as segment sizes decrease, the order bias toward the input order diminishes***.

> As the context window size decreases, the order bias diminishes significantly, with the Top-1 gap between *Perfect-Order* and *Worst-Order* rankings reducing from around 22% at a segment size of 50 to just 1% at a size of 10. Larger context windows tend to increase bias, whereas smaller context windows help reduce it.

4.3 RQ3: How do different ordering strategies influence fault localization performance?

Motivation. We find that LLMs may have order biases toward *Perfect-Order* when investigating a list of methods for FL. However, in practice, such ground truth ordering is unknown. Hence, in this RQ, we investigate whether ordering methods based on the static or dynamic nature of the code or using existing FL techniques can help LLMs achieve better FL results.

Approach. We explore four types of ordering: (1) *Statistical-based* and (2) *Learning-based*, which are directly derived from FL techniques, and (3) *Metric-based* and (4) *Structure-based*, which are grounded in static code features and not specifically tied to FL. The first two approaches leverage dynamic execution data or advanced models trained on FL tasks, making them more targeted for identifying faults. In contrast, the latter two approaches are agnostic to FL techniques. Hence, they may lack the specificity needed for accurately identifying faults, as they do not directly utilize FL data. By integrating ordering strategies with the rich contextual information in the prompt template (see Figure 2), including test code, stack traces, and coverage data, we aim to strengthen LLMs' reasoning about the most relevant areas of the program, ultimately improving fault localization.

For *Metric-based* ordering, we use *Lines of Code (LOC)*, ranking methods in descending order of their lines of code. Longer methods are often more complex and fault-prone (Herraiz and Hassan, 2010), making LOC a simple yet effective heuristic for prioritization. For *Structure-based* ordering, we consider the structure of the call graph associated with each failing test. Specifically, we use *Call Graph*$_{DFS}$, which prioritizes deeper methods by traversing the call graph using depth-first search (DFS), and *Call Graph*$_{BFS}$, which highlights immediate dependencies by traversing the call graph using breadth-first search (BFS). By explicitly encoding dependency relationships, we evaluate whether these structural insights can help LLMs reason about fault propagation within the program and improve fault localization.

Statistical-based ordering relies on dynamic execution data. For this, we use *Ochiai*, which prioritize methods most likely to contain faults, offering insights beyond static metrics or structural heuristics. *Ochiai* is a lightweight unsupervised Spectrum-Based Fault Localization (SBFL) technique (Sasaki et al., 2020) based on the intuition that methods covered by more failing tests and fewer passing tests are considered more suspicious (e.g., faulty). Its suspiciousness score is computed as:

$$Ochiai(a_{ef}, a_{nf}, a_{ep}) = \frac{a_{ef}}{\sqrt{(a_{ef} + a_{nf}) \times (a_{ef} + a_{ep})}}$$

Here, a_{ef}, a_{nf}, and a_{ep} denote the number of failed and passed test cases that execute or do not execute a code statement. Scores range from 0 to 1, with higher values indicating higher fault likelihood. We order methods by aggregating their statement-level scores.

Finally, for *Learning-based* ordering, we use *DepGraph*, which is the state-of-the-art supervised FL technique based on graph neural network (Rafi et al., 2024) that transforms the rich static and dynamic code information into a graph structure. It trains a graph neural network to rank faulty methods by analyzing structural code dependencies and code change history (see Appendix A.2).

Technique	Top-1	Top-3	Top-5	Top-10
Learning-based				
DepGraph	242.0 (48.3%)	338.0 (67.5%)	386.0 (77.0%)	419.0 (83.6%)
Structure-based				
CallGraph$_{BFS}$	175.0 (34.9%)	252.0 (50.3%)	294.0 (58.7%)	343.0 (68.5%)
CallGraph$_{DFS}$	173.0 (34.5%)	253.0 (50.5%)	305.0 (60.9%)	351.0 (70.1%)
Statistical-based				
Ochiai	164.0 (32.7%)	252.0 (50.3%)	293.0 (58.5%)	342.0 (68.3%)
Metric-based				
LOC	163.0 (32.5%)	256.0 (51.1%)	289.0 (57.7%)	351.0 (70.1%)

Table 3: Comparison of fault localization performance using different ordering strategies with the percentage of bugs found across 501 total faults.

Results. ***The choice of ordering strategy is critical in LLM's ability to localize faults, with FL-derived ordering using DepGraph detecting almost 13.4% more faults in Top-1 compared to the next highest Top-1, achieved by Call Graph$_{BFS}$***. Table 3 highlights the model's effectiveness across different ordering strategies. *DepGraph* identifies 13.4% more faults in the Top-1 rank. We see similar trends among other Top-K, where *DepGraph* identifies 16% more faults in Top-3, 16.1% in Top-5, and 13.5% in Top-10. This performance difference is expected, as *DepGraph* excels at ranking faulty methods higher on the list through its advanced fault localization capabilities. The additional faults localized by *DepGraph* across all Top-K ranks reinforce our earlier observation that ***improved ordering strategies enable the model to prioritize the most suspicious methods earlier***.

Despite the importance of order bias, actual FL methods like *DepGraph* provide significantly better fault localization than techniques that do not use LLMs. For instance, *DepGraph*'s Top-K results are higher: 299 (Top-1), 382 (Top-3), 415 (Top-5), and 449 (Top-10) compared to the results from LLM-based methods (see Appendix A.3). The results suggest that ***while LLMs can help with tasks such as ranking faulty methods, domain-specific methods (like DepGraph) are still superior for accurate results***.

Interestingly, *Ochiai* reveals an interesting trend: ***leveraging simple statistical metrics can enable LLMs to improve fault localization by better prioritizing fault-prone methods***. This finding indicates that while *Ochiai*, a simpler *Statistical-based* method, does not match the accuracy of *DepGraph*, it can still effectively assist in fault localization, particularly when computational efficiency or simplicity is a priority. *Ochiai* offers LLMs a simpler way to rank methods based on test outcomes, which aligns well with their ability to process observable patterns. In contrast, *DepGraph* relies on complex code structures like dependencies and execution traces, which require a ***deeper understanding that LLMs may not possess, making it harder for them to reason effectively***.

Simple static-based ordering strategies can match or even outperformance more complex FL-derived ordering across all Top-K ranks. For instance, $CallGraph_{BFS}$, which prioritizes methods closer to failing tests, identifies 175 bugs in Top-1 (34.9%), slightly outperforming $CallGraph_{DFS}$ with 173 bugs (34.5%) and achieving a higher Top-1 accuracy (32.7%) compared to the more complex *Statistical-based* ordering. A similar trend is observed across the remaining Top-K ranks, where $CallGraph_{DFS}$ either matches or slightly outperforms *Statistical-based* ordering, with differences ranging from 0% to 0.02%. Additionally, when comparing *Statistical-based* ordering with *Metric-based* methods, it shows comparable performance in Top-1 ($\Delta 0.2\%$) and Top-3 ($\Delta 0.8\%$), but outperforms in Top-5 ($\Delta 0.8\%$) and Top-10 accuracy ($\Delta 1.8\%$). This suggests that static methods, which are computationally less demanding, can still be effective for fault localization. Hence, these findings emphasize the practicality of ***simpler static-based methods as viable alternatives to more complex FL techniques***.

> While ordering helps rank faults, LLMs struggle with complex relationships. Simpler static-based methods, like $CallGraph_{BFS}$, perform comparably to more complex *Statistical-based* ordering like *Ochiai* in fault localization. Our findings highlight the practicality of static-based methods as efficient alternatives to complex FL techniques, particularly in resource-constrained environments.

5 Discussion & Conclusion

5.1 Discussion of Implications

Implications of Ordering Strategies. Our findings show that the order of inputs significantly impacts the performance of large language models (LLMs) in FL. This highlights the need for thoughtful ordering strategies. Metrics-based ordering, drawn from traditional techniques like *DepGraph* and *Ochiai*, prioritizes suspicious methods and improves accuracy. For instance, *DepGraph* achieved the highest Top-1 accuracy, demonstrating the effectiveness of advanced strategies. In contrast, simpler methods like *CallGraph* and *LOC* performed well across a broader range of cases, making them suitable for resource-limited situations.

When clear ordering metrics are not available, randomizing input orders can serve as a fallback to minimize potential biases introduced by positional effects. Additionally, refining prompts to emphasize context rather than sequence and training LLMs on diverse input sequences could further reduce order bias and improve their robustness. These insights indicate that aligning ordering strategies with task requirements and model capabilities is essential for optimizing workflows in LLM-based FL.

Effectiveness of Segment-Based Strategies. The segment-based approach reduces order bias by keeping the input size small, allowing the model to reason over information step by step in smaller contexts. Specifically, we find that a context size of 10 minimizes bias, as it leads to similar Top-K results for both the *Perfect-Order* and *Worst-Order* cases, where both share the same code context. However, as the context window increases, order bias becomes more influential, affecting the LLM's ability to reason over long sequences of code. Future research could focus on identifying optimal segment sizes that adjust based on task complexity and the amount of available input.

5.2 Conclusion

This work highlights several areas for future research. Order bias may influence the performance of large language models (LLMs) in tasks beyond FL, such as program repair, test case prioritization, and code refactoring. It would be beneficial to investigate how order bias affects these tasks and whether similar solutions can be applied. Additionally, specific prompts that incorporate domain knowledge, such as code semantics and dependency graphs, could enhance contextual understanding and reduce reliance on positional hints. Lastly, exploring new evaluation metrics that take into account the significance of input order and context size will help us gain a better understanding of how LLMs operate in software engineering tasks. We have made all data and scripts related to this work publicly available (AnonymousSubmission, 2023).

Limitations

Our experiments are conducted solely on Java programs using the Defects4J benchmark (V2.0.0) (Just et al., 2014). Although this dataset is well-established and representative of real-world faults, our findings may not apply to other programming languages or ecosystems. Future research could build upon this work to investigate how language-specific features and differences in syntax affect LLM performance in FL tasks.

We do not use Mean First Rank (MFR) and Mean Average Rank (MAR), which are metrics commonly employed in traditional federated learning (FL) studies. These metrics assess how early faulty methods appear in a ranked list. However, their relevance is limited in LLM-based FL approaches because LLMs typically rank only a subset of methods, such as the Top-10, rather than providing a ranking for the entire list of methods considered. This limitation arises from the challenge LLMs face in effectively ranking extensive lists, especially when the number of methods is very large. While Top-K accuracy offers valuable insights, it may not fully capture the nuances that MFR and MAR reveal in traditional FL setups.

Ethics Statement

We affirm that all authors of this paper comply with the ACM Code of Ethics and its code of conduct. Our research aims to investigate the strengths and limitations of large language models (LLMs) in fault localization tasks. This research contributes to a better understanding of their capabilities and applications in software engineering. Our experiments utilize publicly available datasets, such as Defects4J, ensuring reproducibility and transparency in our methods. While this work does not present direct ethical risks, it may have implications for future industrial applications. Adopting LLMs in software development workflows can influence decision-making, productivity, and job roles. We encourage practitioners to use these tools responsibly, maintaining human oversight and fairness when deploying LLMs in critical environments. Furthermore, our work acknowledges LLMs' limitations and aims to inspire further research and development to improve their performance and fairness in real-world applications.

References

Samuel Abedu, Ahmad Abdellatif, and Emad Shihab. 2024. Llm-based chatbots for mining software repositories: Challenges and opportunities. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 201–210.

Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the accuracy of spectrum-based fault localization. In *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, pages 89–98. IEEE.

Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99. IEEE.

AnonymousSubmission. 2023. Public data. GitHub repository.

José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th IEEE/ACM international conference on automated software engineering*, pages 378–381.

An Ran Chen, Tse-Hsun Chen, and Junjie Chen. 2022. How useful is code change information for fault localization in continuous integration? In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12.

Xinyun Chen, Ryan A Chi, Xuezhi Wang, and Denny Zhou. 2024. Premise order matters in reasoning with large language models. *arXiv preprint arXiv:2402.08939*.

Vincent A Cicirello. 2019. Kendall tau sequence distance: Extending kendall tau from ranks to sequences. *arXiv preprint arXiv:1905.02752*.

Israel Herraiz and Ahmed E Hassan. 2010. Beyond lines of code: Do we need more complexity metrics. *Making software: what really works, and why we believe it*, pages 125–141.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*.

René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440.

Sungmin Kang, Gabin An, and Shin Yoo. 2024. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proc. ACM Softw. Eng.*, 1(FSE).

Langchain. 2024. Langchain documentation: Overview. Accessed: 2024-09-04.

Claire Le Goues, Michael Pradel, Abhik Roychoudhury, and Satish Chandra. 2021. Automatic program repair. *IEEE Software*, 38(4):22–27.

Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. 2019. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180.

Feng Lin, Dong Jae Kim, et al. 2024. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852*.

Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676.

OpenAI. 2023. Chatgpt. `https://chatgpt.com/`.

OpenAI. 2024. Gpt-4o mini: Advancing cost-efficient intelligence. Accessed: 2024-09-04.

Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209.

Xiao Pu, Mingqi Gao, and Xiaojun Wan. 2023. Summarization is (almost) dead. *arXiv preprint arXiv:2309.09558*.

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2024. Agentfl: Scaling llm-based fault localization to project-level context. *arXiv preprint arXiv:2403.16362*.

Md Nakhla Rafi, Dong Jae Kim, An Ran Chen, Tse-Hsun (Peter) Chen, and Shaowei Wang. 2024. Towards better graph neural network-based fault localization through enhanced code representation. *Proc. ACM Softw. Eng.*, 1(FSE).

Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. 2024. Exploring llm-based agents for root cause analysis. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, pages 208–219.

Yui Sasaki, Yoshiki Higo, Shinsuke Matsumoto, and Shinji Kusumoto. 2020. Sbfl-suitability: A software characteristic for fault localization. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 702–706. IEEE.

Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 273–283.

Béla Vancsics, Ferenc Horváth, Attila Szatmári, and Arpád Beszédes. 2021. Call frequency-based fault localization. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 365–376. IEEE.

Yiwei Wang, Yujun Cai, Muhao Chen, Yuxuan Liang, and Bryan Hooi. 2023. Primacy effect of chatgpt. *arXiv preprint arXiv:2310.13206*.

Jules White, Sam Hays, Quchen Fu, Jesse Spencer-Smith, and Douglas C Schmidt. 2024. Chatgpt prompt patterns for improving code quality, refactoring, requirements elicitation, and software design. In *Generative AI for Effective Software Development*, pages 71–108. Springer.

W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

Murray Woodside, Greg Franks, and Dorina C Petriu. 2007. The future of software performance engineering. In *Future of Software Engineering (FOSE'07)*, pages 171–187. IEEE.

Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. 2023. Large language models in fault localisation. *arXiv preprint arXiv:2308.15276*.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*.

Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. 2024. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

Cristian Zamfir and George Candea. 2010. Execution synthesis: a technique for automated software debugging. In *Proceedings of the 5th European conference on Computer systems*, pages 321–334.

Mengshi Zhang, Xia Li, Lingming Zhang, and Sarfraz Khurshid. 2017. Boosting spectrum-based fault localization using pagerank. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, pages 261–272.

Quanjun Zhang, Chunrong Fang, Yang Xie, Yaxin Zhang, Yun Yang, Weisong Sun, Shengcheng Yu, and Zhenyu Chen. 2023. A survey on large language models for software engineering. *arXiv preprint arXiv:2312.15223*.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2024, page 1592–1604, New York, NY, USA. Association for Computing Machinery.

Wangchunshu Zhou, Yuchen Eleanor Jiang, Peng Cui, Tiannan Wang, Zhenxin Xiao, Yifan Hou, Ryan Cotterell, and Mrinmaya Sachan. 2023. Recurrentgpt: Interactive generation of (arbitrarily) long text. *arXiv preprint arXiv:2305.13304*.

A Appendix

A.1 Call Graph in Programming Languages

A call graph is a structure that represents how different parts of a program call each other during execution. For example, the nodes in the graph represent functions or methods in the code, and the edges represent the "calls" or "invocations" where one function triggers another.

In NLP, ordering is crucial, whether it involves the order of words in a sentence or the sequence of steps in a pipeline because the correct sequence ensures dependencies are preserved and the process produces meaningful results. Similarly, in software, the order in which functions are called determines the program's flow of execution. For example, a call graph helps us understand this flow, enabling us to: (i) Identify which parts of the code depend on others, helping analyze dependencies or optimize performance, and (ii) focus on functions that are frequently called, which might indicate critical components in the program.

Arbitrary ordering can introduce inconsistencies in analysis. To resolve this, we adopt call graph ordering, which mirrors the program's natural execution order and ensures the ordering respects dependencies while maintaining logical consistency.

A.2 Learning-based fault localization using DepGraph

DepGraph transforms static and dynamic code information into a unified graph for fault localization. It combines the Abstract Syntax Tree (AST) with interprocedural call graphs to capture method dependencies, effectively eliminating irrelevant nodes and edges for a more streamlined graph. Dynamic test coverage connects tests to the methods and statements they cover, with pruning to retain only the most relevant connections. Additionally, the code change history, including metrics such as code churn and modification count, is incorporated as an attribute of the nodes. This provides historical insights into fault-prone areas. The enhanced graph is processed by a Gated Graph Neural Network (GGNN), which effectively ranks faulty methods. Overall, DepGraph reduces graph size, decreases GPU memory usage, and shortens training time while improving fault localization accuracy.

Techniques	Top-1	Top-3	Top-5	Top-10
DepGraph	299	382	415	449
Ochiai	101	221	270	341

Table 4: Top-K accuracy of prior FL-based techniques.

A.3 Top-K performance of traditional approaches

Table 4 presents the Top-K accuracy of two traditional fault localization techniques—*DepGraph* and *Ochiai*—evaluated on 501 bugs from the *Defects4J* dataset. *DepGraph* outperforms *Ochiai*, identifying 299 bugs in the Top-1 position compared to *Ochiai*'s 101. By Top-10, *DepGraph* detects 449 bugs, significantly higher than *Ochiai*'s 341.

However, *DepGraph*'s higher accuracy comes with significant computational overheads due to its reliance on complex GNN models and dependency graph analysis, leading to longer training and testing times. When leveraging LLMs for fault localization using ordered input derived from these techniques, *DepGraph* does not show improvement over its original performance. In contrast, *Ochiai*, despite its simpler approach and lower original accuracy, achieves notable gains when the methods are ordered using its suspiciousness scores and processed through LLMs. This demonstrates that while high-performing methods like *DepGraph* reach near-maximum accuracy and benefit less from LLM-assisted strategies, simpler techniques like *Ochiai* can substantially enhance their fault localization capabilities through optimized ordering and LLM integration. This trade-off underscores the balance between computational efficiency and the effectiveness of ordering strategies in improving fault localization outcomes.

A.4 Prompt Examples

This section outlines the prompts and examples used in our experiments, demonstrating how we design and implement prompts for Fault Localization

System: You will be given a failing test, stack trace, and the method it covers. By analyzing this information, you have to rank the methods from most suspicious to least suspicious. You should rank the top 10 most suspicious methods.

The output must be in the following JSON format:
JSON Format:

```
{
  "methodB": "rank",
  "methodA": "rank",
  ...
}
```

Human: Here are the coverage information:

```
Test Name: org.apache.commons.lang.text.StrBuilderTest.testLang412Right

Test Body:
public void testLang412Right() {
    StrBuilder sb = new StrBuilder();
    sb.appendFixedWidthPadRight(null, 10, '*');
    assertEquals( "Failed appendFixedWidthPadRight correctly", "**********", sb.toString());

StackTrace:
java.lang.NullPointerException
at org.apache.commons.lang.text.StrBuilder.appendFixedWidthPadRight(StrBuilder.java:1230)
at org.apache.commons.lang.text.StrBuilderTest.testLang412Right(StrBuilderTest.java:1755)

Covered Methods:
Method Signature:
org.apache.commons.lang.text.StrBuilder:getNullText()Ljava/lang/String;
Method Body:
public String getNullText() {
    return nullText;
}

Method Signature:
org.apache.commons.lang.text.StrBuilder:appendFixedWidthPadRight(Ljava/lang/Object;IC);
Method Body:
public StrBuilder appendFixedWidthPadRight(Object obj, int width, char padChar) {
    if (width > 0) {
    ensureCapacity(size + width);
    String str = (obj == null ? getNullText() : obj.toString());
    int strLen = str.length();
    ......
}
........ // The remaining coverage info
```

Figure 4: Prompt example for Fault Localization.

Based on the provided information, we can analyze the methods involved in the failing test and the stack trace to determine their suspiciousness. The test is failing due to a 'NullPointerException' in the 'appendFixedWidthPadRight' method when it tries to call 'str.length()' on a 'null' value. This indicates that the handling of 'null' values in the method is likely problematic.

Here's the ranking of the methods from most suspicious to least suspicious:
1. **appendFixedWidthPadRight** - This method is directly involved in the failure, as it attempts to call 'length()' on a potentially 'null' string.
2. **getNullText** - This method returns 'nullText', which is used in 'appendFixedWidthPadRight' to handle 'null' objects. If 'nullText' is 'null', it will lead to a 'NullPointerException'.
....
....

Here is the JSON output for the ranking:

````
```json
{
        "appendFixedWidthPadRight(Ljava/lang/Object;IC);": 1,
        "getNullText()Ljava/lang/String;": 2
        ...... // Remaining methods with rank
}
```
````

Figure 5: Output example from prompts.

(FL) tasks. The examples include the base prompt, the output format, and the segmentation mechanism used in our methodology. For improved clarity, we have omitted certain information from the prompt examples.

Prompt Design. The base prompt instructs the LLM to analyze failing tests, stack traces, and the covered methods to rank the top ten most suspicious methods. As illustrated in Figure 4, the system message outlines the task, while the user message provides specific coverage details, including the test name, test body, stack trace, and the covered methods. The prompt clearly specifies that the output should be in JSON format to ensure consistency and clarity in the ranking of the methods.

Output Example Figure 5 illustrates the output generated from the base prompt. The language model ranks methods based on their level of suspiciousness, which is determined by analyzing the stack trace and coverage data. In this example, the method `appendFixedWidthPadRight` is ranked highest because it is directly involved in the failure. It is followed by `getNullText`, contributing to the error by returning null. The JSON output lists the method signatures and their ranks, ensuring the results are well-structured and easy to interpret.

Segmentation for Iterative Reasoning. To manage large input contexts, we utilized a segmentation-based approach. In Figure 6, we illustrate how segmentation is implemented. Initially, the large language model (LLM) is provided with a set of covered methods, and they are ranked based on their level of suspiciousness. In the next prompt, additional covered methods and the ranked output from the previous segment are introduced. This iterative process enables the model to update and refine its rankings as it evaluates new information. The segmentation mechanism ensures we pass the previous context while maintaining continuity between prompts. The output for this prompt is also a JSON structure of ranked methods similar to what is shown in Figure 5.

System: You are provided with the remaining covered methods identified by the failing tests and the stack trace. Previously, you ranked the following methods from most suspicious to least suspicious:

```
Rank: 1
Method Signature:
org.apache.commons.lang.text.StrBuilder:appendFixedWidthPadRight(Ljava/lang/Object;IC);
Method Body:
public StrBuilder appendFixedWidthPadRight(Object obj, int width, char padChar) {
    if (width > 0) {
    ......
    ......
}
Rank: 2
Method Signature:
org.apache.commons.lang.text.StrBuilder:getNullText()Ljava/lang/String;
Method Body:
public String getNullText() {
    .....
}
```

Now, analyze the additional coverage information. Based on this new data, update the ranking of the top 10 most suspicious methods. You may adjust the existing ranking if necessary or retain it if no changes are warranted. Ensure that your final ranking reflects the latest observations.

The output must be in the following JSON format:
JSON Format:

```
{
  "methodB": "rank",
  "methodA": "rank",
  ...
}
```

Human: Here are the remaining coverage information:

```
Test Name: org.apache.commons.lang.text.StrBuilderTest.testLang412Right

Test Body:
public void testLang412Right() {
    ......
}

StackTrace:
java.lang.NullPointerException
....

Covered Methods:
Method Signature:
org.apache.commons.lang.text.StrBuilder:<init>(I)V;
Method Body:
public StrBuilder(int initialCapacity) {
    ....
}
........ // The remaining coverage info
```

Figure 6: Prompt example for segmentation experiments.