Enhancing Fault Localization Through Ordered Code Analysis with LLM Agents and Self-Reflection

MD NAKHLA RAFI, Concordia University, Canada
DONG JAE KIM, DePaul University, USA
TSE-HSUN (PETER) CHEN, Concordia University, Canada
SHAOWEI WANG, University of Manitoba, Canada

Locating and fixing software faults is a time-consuming and resource-intensive task in software development. Traditional fault localization methods, such as Spectrum-Based Fault Localization (SBFL), rely on statistical analysis of test coverage data but often suffer from lower accuracy. Learning-based techniques, while more effective, require extensive training data and can be computationally expensive. Recent advancements in Large Language Models (LLMs) offer promising improvements in fault localization by enhancing code comprehension and reasoning. However, these LLM-based techniques still face challenges, including token limitations, degraded performance with long inputs, and difficulties managing large-scale projects with complex systems involving multiple interacting components. To address these issues, we introduce *LLM4FL*, a novel LLM-agent-based fault localization approach that integrates SBFL rankings with a divide-and-conquer strategy. By dividing large coverage data into manageable groups and employing multiple LLM agents through prompt chaining, *LLM4FL* navigates the codebase and localizes faults more effectively. The approach also incorporates self-reflection and chain-of-thought reasoning, enabling agents to iteratively generate fixes and re-rank suspicious methods. We evaluated *LLM4FL* on the Defects4J (V2.0.0) benchmark, comprising 675 real-world faults from 14 open-source Java projects. Our results demonstrate that *LLM4FL* outperforms AutoFL by 19.27% in Top-1 accuracy and surpasses state-of-the-art supervised techniques such as DeepFL and Grace, all without task-specific training. Additionally, we highlight the impact of coverage splitting and prompt chaining on fault localization performance and show that different method ordering can improve Top-1 accuracy by up to 22%.

1 Introduction

The process of locating and fixing software faults requires significant time and effort. Research shows that software development teams allocate more than half of their budgets to testing and debugging activities [5, 19]. As software systems become increasingly complex, the demand for more efficient and accurate fault localization techniques continues to grow. Hence, to assist developers and reduce debugging costs, researchers have developed various fault localization techniques [3, 32, 35, 38, 46, 54]. These techniques analyze code coverage and program execution to identify the most likely faulty components, assisting developers in finding the fault.

However, despite the advances in fault localization techniques, many existing approaches still struggle with scalability and precision. Traditional methods, such as Spectrum-Based Fault Localization (SBFL), use statistical analysis to analyze coverage data from passing and failing test cases to rank suspicious code elements [2]. While these techniques provide valuable insights, their accuracy is lower. Their reliance on statistical correlations between test failures and code coverage does not always capture the deeper semantic relationships needed for more accurate fault localization [29, 62, 66]. To address these issues, recent techniques applied machine learning and deep learning models to improve fault localization [32, 35, 54, 74]. These methods enhance the ranking of suspicious code elements by incorporating additional information like code complexity, text similarity, and historical fault data. Researchers also leveraged models like Graph Neural

Authors' Contact Information: Md Nakhla Rafi, Concordia University, Montreal, Canada, mdnakhla.rafi@mail.concordia.ca; Dong Jae Kim, DePaul University, Chicago, USA, dkim121@depaul.edu; Tse-Hsun (Peter) Chen, Concordia University, Montreal, Canada, peterc@encs.concordia.ca; Shaowei Wang, University of Manitoba, Winnipeg, Canada, Shaowei.Wang@umanitoba.ca.

Networks (GNNs) to represent code structures and achieved state-of-the-art fault localization accuracy [38, 38, 46, 49]. However, these techniques often require extensive training data and significant training time.

Recent advances in Large Language Models (LLMs) have shown great potential for fault localization due to their strong language comprehension and generation capabilities [1, 36]. LLMs trained on extensive programming datasets can understand code structure, interpret error messages, and even suggest fixes for common software bugs [25, 31, 44, 64]. These models, with their ability to analyze and process both natural language and code, present an opportunity to significantly improve traditional fault localization methods by incorporating deeper semantic analysis and context-aware reasoning. Wu et al. [64], directly present LLMs with faulty methods or classes with test failure information and ask to locate the issue, which provides valuable insights. Kang et al. [25] which operates as an automated fault localization technique that leverages LLMs to localize fault given a single failing test. It focuses on method-level fault localization and provides bug location and also a natural language explanation of why a particular code location is likely to be faulty.

Despite their potential, existing LLM-based fault localization techniques face several challenges. The token limitations of LLMs restrict their ability to effectively process long code files or large sets of code coverage data, which is often required when dealing with large-scale software systems [18, 21, 64]. Additionally, the performance of LLMs can degrade when applied to complex systems that require the model to reason over multiple interacting components, making it difficult to maintain accuracy and consistency across broader projects [30, 37]. Furthermore, current LLM-based techniques have yet to fully explore how these models can be effectively integrated with traditional fault localization techniques to maximize their strengths in a complementary and efficient manner [10, 25, 64, 72].

In this paper, we propose *LLM4FL*, an LLM-based fault localization. To address the challenges of analyzing large-scale software projects, where code coverage and complexity often exceed LLM token limits [37, 64], *LLM4FL* implements a divide-and-conquer strategy. Before applying the divide-and-conquer strategy, we use an SBFL technique to sort the covered methods, building on findings from prior research [15] that demonstrate LLM performance improves when the order of instructions is carefully considered. Then, we divide the coverage data into manageable groups that the LLM can process within its token limits. Note that, regardless of the order, LLM4FL eventually analyzes every covered method.

In addition to using a divide-and-conquer strategy, *LLM4FL* takes inspiration from how human developers debug software. Developers typically analyze multiple types of information, including error messages, stack traces, and code snippets, to incrementally narrow down potential faulty components [5, 11]. *LLM4FL* emulates this process by utilizing two LLM agents collaborating to iteratively and autonomously navigate the code to locate the faults. *LLM4FL* implements a Tester and a Debugger Agent, each tasked with specialized tools to assist in the fault localization process. The Tester Agent identifies and prioritizes suspicious methods by analyzing the failing test, and stack traces in different groups of covered methods to list a list of highly suspicious methods. It mimics the developer's process of investigating suspicious areas in the code by understanding the context of error messages. Meanwhile, the Debugger Agent thoroughly evaluates the given list of candidate methods by navigating the code and ranks them based on its analysis. These two agents communicate through a prompt chaining mechanism that allows them to share insights and build upon each other's findings.

We evaluated *LLM4FL* using the Defects4J (V2.0.0) benchmark [24], which contains 675 real-world faults from 14 open-source Java projects. Our results demonstrate that *LLM4FL* surpasses LLM-based technique AutoFL [25], by achieving 19.27% higher Top-1 accuracy. Additionally, *LLM4FL* outperforms supervised techniques such as *DeepFL* [32] and *Grace* [38], even without task-specific

training. We also analyzed the impact of individual components within *LLM4FL* on fault localization accuracy. Our findings indicate that each component plays a significant role in its performance, with coverage splitting and prompt chaining contributing the most. When these components are removed, accuracy drops considerably, underscoring their importance in managing token limitations and facilitating efficient multi-agent collaboration. Moreover, we examined whether the initial ordering of methods provided to the LLM influences performance. The results reveal that method ordering is important, with a Top-1 accuracy difference of up to 22% when comparing an execution-based ordering and the order provided by *DepGraph* [49].

The paper makes the following contributions:

- We introduce *LLM4FL*, a novel LLM-based fault localization technique that employs a divide-and-conquer strategy. This technique groups large coverage data and ranks the covered methods using an SBFL formula. By utilizing multiple agents and prompt chaining, *LLM4FL* navigates the code iteratively to effectively identify and localize faults.
- *LLM4FL* demonstrates superior performance, surpassing AutoFL [25] by 19.27% in Top-1 accuracy. It also outperforms state-of-the-art supervised techniques like DeepFL and Grace, achieving these results without requiring task-specific training.
- Our analysis of *LLM4FL*'s components shows that key features like coverage splitting and prompt chaining are essential to its fault localization accuracy. Removing these features leads to significant performance declines, emphasizing their importance in handling token limitations and enabling effective agent collaboration.
- We further investigate the effect of method ordering on LLM performance. The study reveals that different ordering can enhance fault localization accuracy by up to 22% in Top-1 scores. While $LLM4FL_{DepGraph}$ achieves the highest overall accuracy, $LLM4FL_{Ochiai}$ offers a more efficient solution, balancing accuracy gains with lower computational overhead, making it practical for broader use.

In short, we provide a strategy to mitigate the token limitation issues in LLM-based fault localization and highlight the impact of initial method ordering. The findings may help inspire future research to improve LLM-based fault localization for large-scale software projects.

Paper Organization. Section 2 discusses related work. Section 3 describes our technique, *LLM4FL*. Section 4 presents the experimental results. Section 5 discusses the threats to validity. Section 6 concludes the paper.

2 Background and Related Work

2.1 Background

Large Language Models. Large Language Models (LLMs), primarily built on the transformer architecture [12, 39, 52], have significantly advanced the field of natural language processing (NLP). These LLMs, such as the widely recognized GPT3 model with its 175 billion parameters [12], are trained on diverse text data from various sources, including source code. The training involves self-supervised learning objectives that enable these models to develop a deep understanding of language and generate contextually relevant and semantically coherent text. LLMs have shown substantial capability in tasks that involve complex language comprehension and generation [1, 36], such as code recognition and generation. Recent research has leveraged LLMs in software engineering tasks, particularly in fault localization [25, 48, 72], where they assist in identifying the faulty code groups responsible for software errors. One of the key advantages of using LLMs in fault localization is their ability to process both natural language and code without re-training, allowing them to analyze error messages, stack traces, and test case information to infer suspicious methods or code sections in an unsupervised zero-shot setting.

LLM Agents. LLM agents leverage LLMs to autonomously execute tasks described in natural language, making them versatile tools across various domains. LLM agents are artificial intelligence systems that utilize LLMs as their core computational engines to understand questions and generate human-like responses. They leverage functionalities like memory management [76] and tool integration [51, 65] to handle multi-step and complex operations seamlessly. The agents can refine their responses based on feedback, learn from new information, and even interact with other AI agents to collaboratively solve complex tasks [20, 36, 45, 70]. Through prompting, agents can be assigned different roles (e.g., a developer or a tester), providing more domain-specific responses that help improve the answer [20, 53, 59]. As their potential expands, LLM agents play a crucial role in advancing automation and boosting productivity in software development. In this paper, we explore using LLM agents to improve fault localization by emulating developers' debugging process.

2.2 Related Work

Spectrum-based Fault Localization. Spectrum-based fault localization (SBFL) [2, 3, 23, 60] employs statistical techniques to evaluate the suspiciousness of individual code elements, such as methods, by analyzing test outcomes and execution traces. The core idea of SBFL is that code components that are executed more frequently in failing tests and less frequently in passing tests are more likely to contain faults. Despite its widespread study, SBFL's practical effectiveness remains limited [26, 67]. To enhance SBFL's accuracy, recent research [14, 16, 58, 69] has suggested incorporating additional data, such as code changes [14, 58] or mutation analysis [16, 69]. However, SBFL's reliance on code coverage metrics still poses challenges, as its suspiciousness scores may not generalize effectively to different faults or systems.

Learning-based fault localization. Recent efforts have focused on improving SBFL with learning-based methods [32, 33, 35, 54, 74, 75]. These approaches use machine learning models like radial basis function networks [61], back-propagation networks [63], and convolutional neural networks [6, 35, 75] to estimate suspiciousness scores based on historical faults. Some techniques, such as *FLUCCS* [54], combine SBFL scores with metrics like code complexity, while others, like *DeepFL* [32] and *CombineFL* [77], merge multiple sources such as spectrum-based and mutation-based data [17, 40, 42]. Graph neural networks (GNNs) have also been applied to fault localization [38, 46, 47, 68]. Techniques like *Grace* [38] and *GNET4FL* [46] utilize test coverage and source code structure for improved accuracy, while *DepGraph* [49] refines these approaches by graph pruning and incorporating code change information, resulting in higher performance with reduced computational demands. Although these learning-based techniques show improved results, they require training data that may not be available to every project, and the training process can be expensive due to model complexity.

LLM-Based Fault Localization. Large Language Models (LLMs), such as GPT-4o [41], LLaMA [39], and ChatGPT [4], demonstrated remarkable abilities in processing both natural and programming languages. LLMs have shown potential in identifying and fixing errors using program code and error logs [4]. However, one of the major challenges that LLMs face in fault localization is the token limitation issue. LLMs are restricted to a fixed number of tokens, ranging from 2,000 to 128,000 [39, 41], which poses difficulties in handling large-scale software projects with long stack traces and extensive codebases. This limitation can lead to incomplete analyses, as critical context might be truncated or lost, forcing models to work with fragmented information, ultimately affecting the overall fault localization performance [18, 21].

Moreover, LLM-based fault localization techniques often focus on localizing faults within small code snippets due to these context limitations. For example, LLMAO [72] uses lightweight bidirectional adapters on LLMs to generate suspiciousness scores for code lines, but only within a

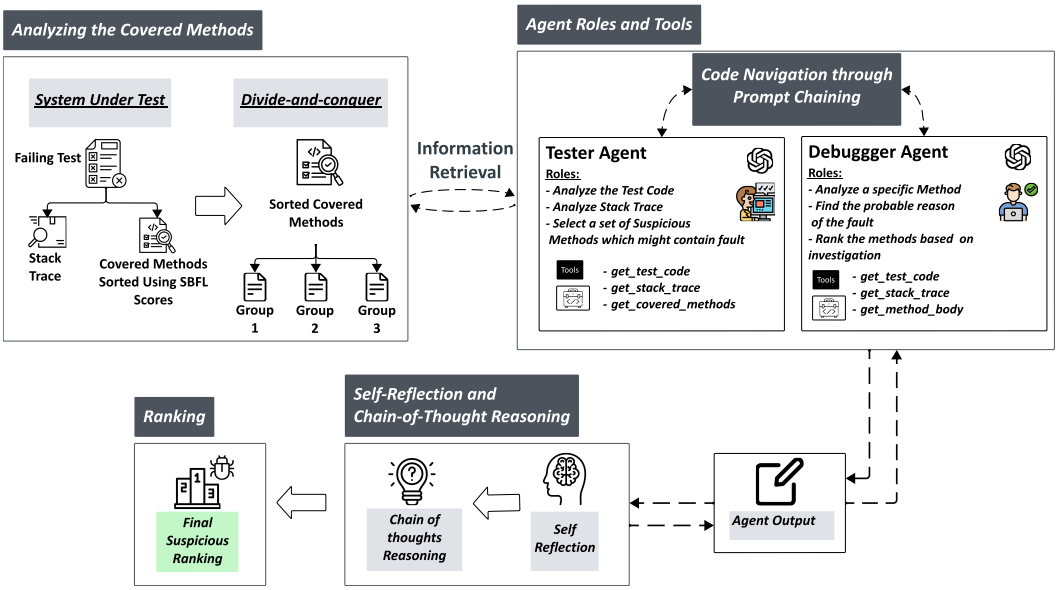

Fig. 1. An overview of *LLM4FL*.

limited context of 128 lines of code. Similarly, Wu et al. [64] prompt ChatGPT with code and error logs to identify faulty lines, but these methods struggle to scale to larger software projects [37]. AutoFL [25] is an LLM-based fault localization technique that detects faulty code locations and generates natural language explanations for bugs. However, it also faces difficulty handling large-scale projects due to LLM context length constraints and reduced performance on complex bugs requiring deeper repository exploration. To address these challenges, there is a growing need for approaches that enable LLMs to localize faults across entire software projects, ensuring they can handle larger inputs while maintaining accuracy and effectiveness. Thus, we propose *LLM4FL*, that leverages a divide-and-conquer approach to navigate through codebases and identify faults through multiple-agent collaboration.

3 Methodology

Applying Large Language Models (LLMs) for fault localization presents specific challenges, such as token size limitations and performance issues with longer input contexts [30, 37, 64]. Processing large datasets like test coverage and source code in a single query is impractical due to these constraints. Moreover, there are limited studies on effective prompting techniques for fault localization. To address these issues, we propose *LLM4FL*, an LLM-agent-based fault localization system. Figure 1 illustrates an overview of our technique. *LLM4FL* operates in four stages: 1) Analyzing the Covered Methods Using a Divide-and-Conquer Strategy, 2) Defining LLM Agent Roles and Tools, 3) Code Navigation through Agent Collaboration and Prompt Chaining, and 4) Self-Reflection and Chain-of-Thought Reasoning. First, the coverage data is divided into smaller, manageable groups to fit within the LLM's token limits. Second, LLM agents are assigned specific roles and equipped with tools to analyze these groups. Third, LLM agents collaborate through prompt chaining to navigate the code autonomously and identify potential faults. Finally, the agents refine their analysis using self-reflection and chain-of-thought reasoning. Below, we provide more details on each of these stages.

3.1 Analyzing the Covered Methods Using Divide-and-Conquer

Prior studies in fault localization utilized dynamic execution data, such as coverage information from both failing and passing test cases, to identify faulty code locations [3, 32, 38, 46, 49]. However, Large Language Models (LLMs) face challenges when processing large-scale coverage data due to inherent token size limitations [25]. These limitations lead to performance degradation, extended processing times, and reduced accuracy when input tokens are truncated. To address these constraints, we propose a divide-and-conquer strategy, dividing the coverage data (i.e., containing the covered methods and the corresponding code) into manageable groups that the LLM can process within its token limits.

We generate a list of covered methods using tools like GZoltar [13], focusing only on methods executed by failing tests. To manage the token limitation issue, we include only the specific method statements that were executed by the failing tests, excluding any uncovered portions of the code. Before doing a divide-and-conquer, we use a Spectrum-Based Fault Localization (SBFL) technique to sort the covered methods so that those more likely to be faulty can be grouped together. This approach is inspired by a recent study [55] that demonstrates LLMs perform better when the order of instruction is carefully considered. Note that, regardless of the order, *LLM4FL* eventually analyzes every covered method. Specifically, we use *Ochiai* [2] to sort the methods before the divide-and-conquer process. *Ochiai* is a widely used unsupervised spectrum-based fault localization technique because of its high efficiency and good fault localization accuracy [2, 16, 35, 38, 47, 58]. Intuitively, *Ochiai* assigns a higher suspiciousness score to a statement that is executed more frequently by failing cases and less frequently by passing test cases. *Ochiai* is defined as:

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

where a_{ef} is the number of failed test cases that execute a statement, a_{nf} is the number of failed test cases that do not execute the statement, and a_{ep} is the number of passed test cases that execute the statement. The result of *Ochiai* is a value between 0 and 1, with higher values indicating a higher likelihood that a particular statement is faulty. We then aggregate the scores for every statement in a method to get the method-level score.

Once the covered methods are sorted, we divide the methods into smaller groups, ensuring that each group fits within the token limitation of the LLM. More formally, let C represent the sorted covered methods, containing a sequence of pairs (m, s), where m denotes a covered source method and s denotes the corresponding code statements. The group length is determined using the token limitation specified in the official documentation of GPT-4o-mini (the LLM that we use in this study), which has an input context window of 128,000 tokens [41]. The length is calculated as $TotalTokenLength/TokenLimitation$, ensuring each group remains within the model's token limitation. Specifically, we divide C into to sequence of $C_1, C_2, \ldots, C_n$ each C_i contains the subset of pairs (m_i, s_i), and satisfies the constraint $|C_i| <= TokenLimitation$.

3.2 LLM Agents and the Available Tools

LLM4FL emulates developers' debugging process by implementing two LLM agents: ***1) Tester*** and ***2) Debugger***. Below we discuss the details of these agents:

Tester Agent. The tester agent's main goal is to identify and prioritize suspicious methods. More formally, it navigates through each group C_i of the covered methods. The Tester Agent's input includes the sorted covered methods in each group C_i and the corresponding test code, stack traces, and related test information from failing executions. To further reduce the input token size and allow LLM to focus on the relevant code, we only provide the portion of the test code leading to the

Prompt Template for Initial Candidate Selection with Advanced Tools

As an Intelligent Tester Agent, utilize the following tools to identify methods most likely related to the fault. Focus on analyzing the test code, stack trace, and covered methods to pinpoint suspicious or fault-inducing methods. Ensure the exact `method_id` is included in the final JSON output.

Available Tools:
`get_test_code()` – Retrieves the complete test code.
`get_stacktrace()` – Access the stack trace of the failed test.
`get_covered_methods()` – Lists all methods executed during the test.

Your Task: Use the insights from the tools and provide the potential suspicious methods in the following JSON format:

```
{
    "method_ids": [1, 2, 3, ...]  // The potential suspicious method's ids
}
```

Fig. 2. Enhanced Prompt for Initial Candidate Selection.

failure as input. We analyze the stack trace to identify where the test failed and extract the code up to that point, omitting the rest.

Figure 2 gives an example of the Tester Agent's prompt. The Tester Agent is responsible for identifying suspicious methods likely related to faults in the system. To achieve this, it needs to 1) analyze methods executed by failing tests, 2) review relevant test code, and 3) examine stack trace data to pinpoint where failures occurred. To facilitate this process, we provide the Tester Agent with three tools. The tool `get_covered_methods` allows the agent to retrieve the source code of the methods executed during failing tests, `get_test_code` fetches the relevant test code, and `get_stacktrace` gathers stack trace data. Each tool has a unique name and description, which the agent uses to determine when to apply the tool during its analysis. The agent decides autonomously, based on its task, which tool to invoke by interpreting the tool's description and relevance to the current stage of fault localization. Additionally, each method within the grouped data is assigned a unique ID during the dividing process, allowing the agent to reference and retrieve specific methods. After analyzing all groups, the Tester Agent generates a list of suspicious methods, denoted as $S_i \subseteq M_i$, where M_i represents the methods in group C_i. This list of suspicious methods is then consolidated into a final list of method IDs for further analysis.

Debugger Agent. Figure 3 gives an example of the Debugger Agent's prompt. The Debugger Agent is responsible for thoroughly evaluating and ranking a given list of candidate methods S. To perform its analysis, the Debugger Agent relies on three tools designed to extract relevant information. `get_test_code` and `get_stacktrace` provide LLM's requested information by retrieving the relevant test code and stack trace data, helping the Debugger Agent understand the failure scenarios associated with each method. Additionally, the `get_method_body` tool allows the agent to fetch the full source code of a method using its unique method ID, enabling the Debugger Agent to analyze each method's source code.

Once the Debugger Agent completes its analysis, it assigns a rank score σ_j to each method m_j, representing the likelihood that the method is contributing to the fault. The candidate methods in S are then ranked based on these scores, producing a ranked list R that orders the methods from most to least suspicious:

$$R = \text{sort}(\{(m_j, \sigma_j) : m_j \in S\})$$

This final ranked list R is output in JSON format for further evaluation and parsing.

Prompt Template for Code Understanding and Fault Analysis

As an Advanced Debugger Agent, use the following tools to analyze the provided method bodies based on the Tester Agent's insights from the test and stack traces. The fault may propagate across multiple methods or be indirectly related to the original issue.

Available Tools:
`get_test_code()` – Retrieve the entire test code for detailed analysis.
`get_stacktrace()` – Access the stack trace to trace method calls and identify potential faults.
`get_method_body()` – Retrieve the body of specific methods for further inspection.

Task: Using the provided tools, analyze and focus on the following ***Methods***:

`{suspicious methods}`

Rank these methods from most to least suspicious, and provide brief reasoning for each based on its behavior and potential involvement in the fault.

IMPORTANT: Provide your output in the following JSON format:

```
[
    {
        "method_id": int,  // The most suspicious method's id
        "reasoning": string,  // Reasoning for its suspiciousness
        "rank": int  // Rank of suspiciousness
    }
]
```

Fig. 3. Enhanced Prompt for Code Understanding and Fault Analysis.

3.3 Code Navigation Through Prompt Chaining

Code navigation is a critical part of how developers trace faults in real-world scenarios. When debugging, developers often start by analyzing a specific method and then explore its caller or callee methods to better understand the overall logic and pinpoint where the fault might lie. Our approach emulates this process by implementing code navigation through prompt chaining, allowing the Tester and Debugger Agents to collaborate and progressively focus their analysis on the most relevant parts of the codebase.

The process begins with the Tester Agent, which works through each group of the coverage data $C = \{C_1, C_2, \ldots, C_n\}$, where each group C_i contains a subset of the covered methods M_i. The Tester Agent identifies suspicious methods within each group, producing a list $S_i \subseteq M_i$ of candidate methods. This step mimics how a developer might flag areas of interest in the code that seem related to the fault. After analyzing all the groups, the Tester Agent compiles the suspicious methods into a consolidated list:

$$S = \bigcup_{i=1}^{n} S_i$$

This list S is passed to the Debugger Agent, initiating the prompt chaining process.

The Debugger Agent takes the output from the Tester Agent and dives deeper into the suspicious methods. In this stage, the Debugger Agent navigates through each method in S, inspecting not only the method itself but also the related methods, such as caller and callee methods, if the agent believes further analysis can help identify faulty code location. The Debugger Agent emulates the manual process of tracing how faults propagate through different parts of the code. By leveraging

Table 1. An overview of our studied projects from Defects4J v2.0.0. *#Faults*, *LOC*, and *#Tests* show the number of faults, lines of code, and tests in each system. *Fault-triggering Tests* shows the number of failing tests that trigger the fault.

Project	#Faults	LOC	#Tests	Fault-triggering Tests
Cli	39	4K	94	66
Closure	174	90K	7,911	545
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	675	380K	24,302	1,486

its tools, the Debugger Agent retrieves information on how each method interacts with others and how it may contribute to the fault. If the Debugger Agent identifies that a method m_{j+1} is called within m_j and might require further examination, it can request the code for m_{j+1} if it is part of the candidate list, ensuring deeper analysis where necessary. *LLM4FL* implements an LLM-based fault navigation system that allows the Debugger Agent to dynamically traverse the codebase and retrieve additional relevant methods for further inspection. Through this process, the Debugger Agent traces potential fault propagation pathways and evaluates how faults spread across methods.

3.4 Self-Reflection and Chain-of-Thought for Improved Fault Localization Results

LLM4FL uses two additional prompting techniques to further improve the fault localization results.
Self-Reflection. In our approach, both the Tester and Debugger Agents engage in self-reflection to enhance the quality of their outputs. After completing their initial analysis, each agent operates in two phases: a generator phase and a reflector phase. During the generator phase, the agents produce their primary results, such as identifying suspicious methods or ranking them by suspiciousness. In the reflector phase, the agents critique their own work, offering feedback to refine their results and correct potential errors. While it has not yet been studied in the fault localization context, a recent study [50] found that self-reflection improves LLM's performance in problem-solving tasks.
Chain-of-Thought for Refining Fault Localization. We incorporate a chain-of-thought reasoning process to enhance fault localization. After generating a ranked list of suspicious methods, the LLMs propose potential fixes for the top-ranked methods. The agents then re-rank the methods by considering the generated fixes. This process allows the agents to think more deeply about the faults and reassess the ranking based on the insights gained from considering these fixes [8, 57].

4 STUDY DESIGN AND RESULTS

In this section, we first describe the study design and setup. Then, we present the motivation, approach, and results of the research questions.
Benchmark Dataset. To answer the RQs, we conducted the experiment on 675 faults across 14 projects from the Defects4J benchmark (V2.0.0) [24]. Defects4J provides a controlled environment to reproduce faults collected from projects of various types and sizes. Defects4J is widely used in prior automated fault localization research [14, 38, 54, 73]. We excluded three projects, JacksonDatabind,

JxPath, and Chart, from Defects4J in our study because we encountered many execution errors and were not able to collect test coverage information for them. Table 1 gives detailed information on the projects and faults we use in our study. In total, the faults have over 1.4K fault-triggering tests (i.e., failing tests that cover the fault). The sizes of the studied projects range from 2K to 90K lines of code. Note that since a fault may have multiple fault-triggering tests, there are more fault-triggering tests than faults.

Evaluation Metrics. According to prior findings, debugging faults at the class level lacks precision for effective location [26]. Alternatively, pinpointing them at the statement level might be overly detailed, omitting important context [43]. Hence, in keeping with prior work [7, 9, 32, 38, 56], we perform our fault localization process at the method level. We apply the following commonly-used metrics for evaluation:

Recall 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 result from *LLM4FL* is a ranked list based on the suspiciousness score. Prior research [43] 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.

Implementation and Environment. To collect test coverage data and compute results for baseline techniques, we utilized Gzoltar [13], an automated tool that executes tests and gathers coverage information. For the LLM-based components, we employed OpenAI's GPT-4o mini, a more cost-effective and capable alternative to GPT-3.5 Turbo [41]. LangChain v0.2 was used to streamline the development of *LLM4FL*, facilitating the integration of language models with external tools and enhancing the system's overall functionality [27]. To implement the self-reflection technique, we leveraged the LangGraph framework, which enabled graph-based reasoning and decision-making processes [28]. To minimize the variations in the output, we set the temperature parameter to 0 during model inference.

4.1 RQ1: How does *LLM4FL* perform compared with other fault localization techniques?

Motivation. In this RQ we evaluate the fault localization accuracy of *LLM4FL* by comparing with various baseline techniques.

Approach. We compare *LLM4FL*'s fault localization accuracy with five baselines: ***Ochiai*** [2], ***DeepFL*** [32], ***Grace*** [38], **DepGraph** [49], and **AutoFL [25]**.

Ochiai [2] is widely recognized in fault localization research for its high efficiency and accuracy, making it a common baseline for comparison [16, 35, 38, 47, 49, 58]. As such, we use *Ochiai* to rank the methods during the segmentation process and include it as a baseline for accuracy comparison. *DeepFL* [32] is a deep-learning-based fault localization technique that integrates spectrum-based and other metrics such as code complexity, and textual similarity features to locate faults. It utilizes a Multi-layer Perceptron (MLP) model to analyze these varied feature dimensions. We follow the study [32] to implement *DeepFL* and include the SBFL scores from 34 techniques, code complexity, and textual similarities as part of the features for the deep learning model. *Grace [38]* is one of the first fault localization techniques based on graph neural networks (GNN) that represents code as a graph and uses a gated graph neural network to rank the faulty methods. Since then, GNN-based techniques have shown superior fault localization accuracy compared to traditional techniques. *DepGraph* [49] is a GNN-based technique that further improves *Grace* by enhancing code representation in a graph using interprocedural call graph analysis for graph pruning and integrating historical code change information in the graph.

Table 2. Fault localization accuracy in terms of Top-1, 3, 5, and 10. The numbers in the parenthesis show the number of faults in each project. ***The result continues in Table 3.***

Project (# faults)	Techniques	Top-1	Top-3	Top-5	Top-10
Cli (39)	Ochiai	3.0	5.0	10.0	18.0
	DeepFL	11.0	21.0	24.0	28.0
	Grace	14.0	24.0	26.0	30.0
	DepGraph	17.0	24.0	27.0	34.0
	AutoFL	12.0	19.0	19.0	20.0
	LLM4FL	16.0	21.0	23.0	24.0
Closure (174)	Ochiai	20.0	39.0	70.0	72.0
	DeepFL	46.0	61.0	92.0	99.0
	Grace	51.0	78.0	102.0	121.0
	DepGraph	60.0	99.0	126.0	148.0
	AutoFL	45.0	58.0	65.0	82.0
	LLM4FL	52.0	77.0	102.0	118.0
Codec (18)	Ochiai	3.0	12.0	17.0	17.0
	DeepFL	5.0	10.0	12.0	16.0
	Grace	6.0	11.0	13.0	17.0
	DepGraph	7.0	10.0	14.0	16.0
	AutoFL	12.0	14.0	14.0	16.0
	LLM4FL	9.0	13.0	13.0	13.0
Collections (4)	Ochiai	1.0	1.0	2.0	2.0
	DeepFL	1.0	1.0	2.0	2.0
	Grace	1.0	1.0	2.0	2.0
	DepGraph	1.0	2.0	2.0	2.0
	AutoFL	1.0	1.0	1.0	1.0
	LLM4FL	1.0	1.0	1.0	1.0
Compress (47)	Ochiai	5.0	12.0	17.0	29.0
	DeepFL	22.0	27.0	31.0	38.0
	Grace	23.0	29.0	34.0	42.0
	DepGraph	25.0	33.0	36.0	45.0
	AutoFL	23.0	33.0	34.0	35.0
	LLM4FL	23.0	32.0	34.0	34.0
Csv (16)	Ochiai	3.0	8.0	10.0	12.0
	DeepFL	7.0	8.0	9.0	11.0
	Grace	6.0	8.0	10.0	12.0
	DepGraph	8.0	9.0	12.0	13.0
	AutoFL	5.0	11.0	12.0	14.0
	LLM4FL	8.0	10.0	10.0	10.0
Gson (16)	Ochiai	4.0	9.0	9.0	12.0
	DeepFL	8.0	11.0	12.0	12.0
	Grace	11.0	13.0	14.0	15.0
	DepGraph	14.0	15.0	16.0	16.0
	AutoFL	5.0	7.0	10.0	10.0
	LLM4FL	11.0	14.0	14.0	14.0

AutoFL is a LLM-based fault localization technique. It begins by providing the LLM with information about a failing test and descriptions of specific methods that can be used to navigate the codebase. The LLM then interacts with these methods to gather relevant information, such as covered classes, methods, and code snippets. Scoring and ranking candidate methods (depicted as black rectangles) based on five AutoFL prediction outcomes. The methods are ranked by assigning scores based on multiple runs of AutoFL. In each run, a method's score is the inverse of the total number of predicted methods, and these scores are averaged across all runs. Methods are then ranked in descending order of their average scores, with earlier predictions used to break any ties. While the original paper used OpenAI's GPT-3.5-turbo-0613 model for their experiments, for our evaluation, we are using the latest lightweight GPT-4o mini model to perform AutoFL's experiments.

Table 3. ***Continued from Table 2***. Fault localization accuracy in terms of Top-1, 3, 5, and 10. The numbers in the parenthesis show the number of faults in each project.

Project (# faults)	Techniques	Top-1	Top-3	Top-5	Top-10
JacksonCore (26)	Ochiai	6.0	11.0	13.0	14.0
	DeepFL	5.0	5.0	9.0	10.0
	Grace	9.0	13.0	14.0	15.0
	DepGraph	12.0	15.0	15.0	16.0
	AutoFL	10.0	17.0	17.0	17.0
	LLM4FL	12.0	13.0	15.0	15.0
JacksonXml (6)	Ochiai	0.0	0.0	0.0	0.0
	DeepFL	3.0	3.0	4.0	5.0
	Grace	3.0	3.0	4.0	5.0
	DepGraph	4.0	5.0	5.0	5.0
	AutoFL	2.0	2.0	2.0	3.0
	LLM4FL	4.0	4.0	4.0	4.0
Jsoup (93)	Ochiai	15.0	40.0	48.0	57.0
	DeepFL	33.0	39.0	46.0	49.0
	Grace	40.0	64.0	72.0	77.0
	DepGraph	53.0	73.0	78.0	83.0
	AutoFL	36.0	52.0	52.0	54.0
	LLM4FL	41.0	56.0	60.0	60.0
Lang (64)	Ochiai	25.0	45.0	51.0	59.0
	DeepFL	42.0	53.0	55.0	57.0
	Grace	43.0	53.0	57.0	58.0
	DepGraph	48.0	55.0	60.0	61.0
	AutoFL	40.0	57.0	60.0	60.0
	LLM4FL	48.0	55.0	58.0	58.0
Math (106)	Ochiai	23.0	52.0	62.0	82.0
	DeepFL	52.0	81.0	90.0	95.0
	Grace	64.0	79.0	92.0	97.0
	DepGraph	72.0	92.0	97.0	102.0
	AutoFL	53.0	81.0	87.0	94.0
	LLM4FL	68.0	87.0	92.0	94.0
Time (26)	Ochiai	6.0	12.0	13.0	16.0
	DeepFL	12.0	15.0	18.0	20.0
	Grace	11.0	16.0	20.0	21.0
	DepGraph	17.0	20.0	21.0	22.0
	AutoFL	13.0	18.0	21.0	22.0
	LLM4FL	14.0	21.0	22.0	23.0
Mockito (38)	Ochiai	7.0	14.0	18.0	23.0
	DeepFL	10.0	18.0	23.0	26.0
	Grace	16.0	24.0	26.0	29.0
	DepGraph	21.0	29.0	32.0	34.0
	AutoFL	18.0	23.0	29.0	29.0
	LLM4FL	21.0	22.0	26.0	27.0
Total (675)	Ochiai	121.0	260.0	340.0	413.0
	DeepFL	257.0	353.0	427.0	468.0
	Grace	298.0	416.0	486.0	541.0
	DepGraph	359.0	481.0	541.0	597.0
	AutoFL	275.0	393.0	423.0	457.0
	LLM4FL	328.0	426.0	474.0	495.0

Results. ***LLM4FL outperforms the LLM-based baseline, AutoFL, by achieving a much higher Pass@1 (328 v.s. 275).*** Table 2 and 3 show the fault localization results of *LLM4FL* and the baseline techniques. Between the two LLM-based techniques, *LLM4FL* achieves a better Top@N than AutoFL when N is 1, 3, 5, and 10. In the Top-1 metric, *LLM4FL* locates the correct fault in 328 cases, compared to AutoFL's 275, representing a 19.27% improvement. Similarly, in the Top-3 and Top-5 metrics, *LLM4FL* achieves scores of 426 and 474, respectively, outperforming AutoFL's results of 393 and 423. Even in the broader Top-10 metric, *LLM4FL* shows an 8.32% advantage, with a score of 495 compared

to AutoFL's 457. These numbers highlight *LLM4FL*'s enhanced ability to pinpoint faulty methods more accurately and efficiently, reinforcing its superiority over AutoFL for fault localization tasks.
LLM4FL shows higher Top-1 and Top-3 compared to all other non-LLM-based techniques, except for DepGraph. For the Top-1 metric, *LLM4FL* scores 328, which is 171.07% higher than Ochai's score of 121, 27.63% higher than DeepFL's score of 257, and 10.07% better than Grace's result of 298. One exception is DepGraph, which achieves a Top-1 of 359, 8.64% higher than *LLM4FL*. As the range expands to Top-3 and beyond, *LLM4FL* continues to demonstrate its robustness, significantly outperforming DeepFL and maintaining strong performance alongside Grace. It is important to note that DeepFL, Grace, and DepGraph are supervised methods trained and evaluated using leave-one-out cross-validation, following the original study protocols. Despite *LLM4FL* being a zero-shot approach without task-specific training, its ability to perform competitively with these supervised techniques further highlights its effectiveness. The fact that *LLM4FL* achieves strong results without the need for extensive training data or cross-validation underscores its potential and sheds light on future research directions, especially when training models on specific data sets may not always be feasible.

LLM4FL outperforms the other LLM-based technique, AutoFL, by achieving a 19.27% higher Top-1 and delivering competitive results compared to supervised techniques like DeepFL and Grace, even without task-specific training. The findings highlight the potential of using *LLM4FL* for zero-shot fault localization.

4.2 RQ2: How do different components affect *LLM4FL*'s fault localization accuracy?

Motivation. In this research question, we study the influence of each individual component within *LLM4FL* on its fault localization performance. *LLM4FL* employs a combination of advanced techniques, including coverage splitting to handle token limitations, prompt chaining for fault navigation, self-reflection for iterative refinement of the agents' decisions, and chain-of-thought reasoning for re-ranking suspicious methods. Each of these components plays a distinct role in the overall process. To study their effects, we investigate how the removal of any one of these components affects the *LLM4FL*'s ability to accurately localize faults. By isolating the effects of each technique, we can better understand which components are critical to maintaining or improving the overall accuracy of the *LLM4FL* and provide insights to improving future LLM-based fault localization techniques.
Approach. To evaluate the impact of each component, we designed four experimental configurations: $\textbf{\textit{LLM4FL}}_{\textit{w/o CodeNavigation}}$, $\textbf{\textit{LLM4FL}}_{\textit{w/o Divide\&Conquer}}$, $\textbf{\textit{LLM4FL}}_{\textit{w/o Reflection}}$, and $\textbf{\textit{LLM4FL}}_{\textit{w/o CoT}}$. For each configuration, one component is systematically removed, and we compare the result with the full-featured baseline of *LLM4FL*. This allows us to directly measure how much each component contributes to the fault localization process.

$\textbf{\textit{LLM4FL}}_{\textit{w/o CodeNavigation}}$ removes the prompt chaining mechanism, which essentially means the agents no longer collaborate do fault navigation through multiple rounds of reasoning. Instead, the *LLM4FL* uses a single prompt to perform fault localization without iterative agent communication and fault navigation. This configuration tests whether the multi-step agent collaboration improves the ranking and selection of faulty methods or if a single round of communication is sufficient.

$\textbf{\textit{LLM4FL}}_{\textit{w/o Divide\&Conquer}}$ removes the grouping of the covered methods, giving the agents the entire coverage data at once instead of dividing it into smaller, manageable groups. Coverage segmentation addresses token limitations in LLMs, so removing it explores the impact of feeding the full dataset to the agents in one step. We aim to see how handling large amounts of data in a single input influences the fault localization result, as it may overwhelm the model or reduce precision.

LLM4FL$_{w/o\ Reflection}$ removes the self-reflection technique, which is used to allow agents to review and refine their initial candidate selection and ranking. Without this self-reflection step, the agents rely solely on their initial assessments without iterative improvements.

LLM4FL$_{w/o\ CoT}$ disables the chain-of-thought reasoning process, which asks the agents to think deeper by asking LLMs to generate potential fixes and re-rank the suspicious methods. The chain-of-thought approach is designed to enhance logical reasoning and ensure that intermediate results are critically evaluated and refined.

Results. ***While all components help improve the results, including coverage grouping and prompt chaining provide the largest improvement to fault localization results (23% and 17% in Top-1).*** Table 4 shows the Top-1, 3, 5, and 10 scores when each component is removed. Removing coverage splitting has the largest overall impact across all scores, reducing Top-N by 19% to 23%. Removing prompt chaining has the second largest impact (11% to 17%). At the individual project level, these two components also have the largest impact in Top-1 in 9/13 studied projects. Our finding shows that employing sorted coverage grouping following the divide and conquer technique and agent communication significantly improves fault localization results. Future research should consider these techniques when designing fault localization techniques.

Although there is no oracle during the fault localization process, asking LLMs to self-reflection still helps improve the overall Top-1 by 11%. LLMs often suffer from hallucinations, especially when there is a lack of feedback from external oracles [22, 71]. Even though we did not provide any groundtruth or external feedback to LLM, we find that self-reflection is still effective in improving fault localization results. Self-reflect brings 6% to 11% improvement across the Top-N metrics. Our finding suggests that future studies should consider self-reflection even if there is no external feedback. Interestingly, we did not see large differences after removing chain-of-thought reasoning via test generation. In some cases, removing it even slightly improves the fault localization results. One reason may be that *LLM4FL* already includes self-reflection to make LLM think deeply about the result, and adding a chain-of-thought did not further improve the result. Our finding highlights the effectiveness of self-reflection, which should be considered in future fault localization results.

The results show that each component of *LLM4FL* contributes to its overall fault localization performance, with coverage splitting and prompt chaining having the largest positive impact. Removing these components leads to significant declines in accuracy, confirming their crucial role in handling token limitations and enabling more effective multi-agent collaboration.

4.3 RQ3: Does order matter in the initial list of methods provided to the LLM?

Motivation. *LLM4FL* divides the coverage data into segments to address the token size limitation of LLMs. We sort the methods using the *Ochiai* scores before segmentation, though different sorting mechanisms may affect the final fault localization result. Although *LLM4FL* eventually visits and assesses every method, a recent study [15] observes that the order of premises affects LLM's results. However, whether this effect extends to software engineering tasks, particularly fault localization, remains unclear. Hence, in this research question, we investigate whether the order of methods within the segments affects the LLM's fault localization performance.

Approach. To test the effect of method ordering, we experiment with three distinct sorting strategies: ***LLM4FL$_{Execution}$***, ***LLM4FL$_{Ochiai}$*** (the default sorting in *LLM4FL*), and ***LLM4FL$_{DepGraph}$***. Each strategy provides a different way to sort the methods before they are segmented for further analysis.

LLM4FL$_{Execution}$: In this baseline approach, we use the unsorted list of methods executed during testing, as generated by Gzoltar [13]. This default list represents the natural execution order of

Table 4. Impacts of removing different components in *LLM4FL* on Top-1, 3, 5, and 10. The numbers in the parenthesis show the percentage changes compared to *LLM4FL* with all the components.

Project (# faults)	Techniques	Top-1	Top-3	Top-5	Top-10
Cli (39)	LLM4FL	16	21	23	24
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	11 (-31.25%)	19 (-9.52%)	21 (-8.7%)	21 (-12.5%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	11 (-31.25%)	17 (-19.05%)	17 (-26.09%)	19 (-20.83%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	11 (-31.25%)	19 (-9.52%)	21 (-8.7%)	21 (-12.5%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	16 (0.0%)	21 (0.0%)	23 (0.0%)	24 (0.0%)
Closure (174)	LLM4FL	52	77	102	118
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	32 (-38.46%)	45 (-41.56%)	53 (-48.04%)	53 (-55.08%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	27 (-48.08%)	40 (-48.05%)	49 (-51.96%)	50 (-57.63%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	50 (-3.85%)	74 (-3.9%)	92 (-9.8%)	109 (-7.63%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	52 (0.0%)	77 (0.0%)	102 (0.0%)	118 (0.0%)
Codec (18)	LLM4FL	9	13	13	13
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	8 (-11.11%)	12 (-7.69%)	13 (0.0%)	13 (0.0%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	8 (-11.11%)	11 (-15.38%)	12 (-7.69%)	13 (0.0%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	9 (0.0%)	12 (-7.69%)	13 (0.0%)	13 (0.0%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	9 (0.0%)	13 (0.0%)	13 (0.0%)	13 (0.0%)
Compress (47)	LLM4FL	23	32	34	34
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	23 (0.0%)	32 (0.0%)	34 (0.0%)	34 (0.0%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	23 (0.0%)	28 (-12.5%)	30 (-11.76%)	31 (-8.82%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	23 (0.0%)	32 (0.0%)	35 (2.94%)	35 (2.94%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	23 (0.0%)	32 (0.0%)	34 (0.0%)	34 (0.0%)
Csv (16)	LLM4FL	8	10	10	10
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	8 (0.0%)	8 (-20.0%)	9 (-10.0%)	9 (-10.0%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	8 (0.0%)	8 (-20.0%)	9 (-10.0%)	9 (-10.0%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	7 (-12.5%)	10 (0.0%)	10 (0.0%)	10 (0.0%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	8 (0.0%)	9 (-10.0%)	10 (0.0%)	10 (0.0%)
Gson (16)	LLM4FL	11	14	14	14
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	9 (-18.18%)	15 (7.14%)	15 (7.14%)	15 (7.14%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	10 (-9.09%)	14 (0.0%)	15 (7.14%)	15 (7.14%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	11 (0.0%)	12 (-14.29%)	13 (-7.14%)	14 (0.0%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	11 (0.0%)	14 (0.0%)	14 (0.0%)	14 (0.0%)
JacksonCore (26)	LLM4FL	12	13	15	15
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	9 (-25.0%)	13 (0.0%)	14 (-6.67%)	14 (-6.67%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	9 (-25.0%)	13 (0.0%)	13 (-13.33%)	13 (-13.33%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	10 (-16.67%)	13 (0.0%)	13 (-13.33%)	13 (-13.33%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	12 (0.0%)	13 (0.0%)	15 (0.0%)	15 (0.0%)
JacksonXml (6)	LLM4FL	4	4	4	4
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	2 (-50.0%)	3 (-25.0%)	3 (-25.0%)	3 (-25.0%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	2 (-50.0%)	2 (-50.0%)	2 (-50.0%)	2 (-50.0%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	2 (-50.0%)	2 (-50.0%)	2 (-50.0%)	2 (-50.0%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	3 (-25.0%)	4 (0.0%)	4 (0.0%)	4 (0.0%)
Jsoup (93)	LLM4FL	41	56	60	60
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	40 (-2.44%)	55 (-1.79%)	58 (-3.33%)	58 (-3.33%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	36 (-12.2%)	50 (-10.71%)	51 (-15.0%)	51 (-15.0%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	38 (-7.32%)	53 (-5.36%)	54 (-10.0%)	54 (-10.0%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	40 (-2.44%)	55 (-1.79%)	59 (-1.67%)	60 (0.0%)
Lang (64)	LLM4FL	48	55	58	58
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	42 (-12.5%)	54 (-1.82%)	59 (1.72%)	59 (1.72%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	43 (-10.42%)	54 (-1.82%)	59 (1.72%)	59 (1.72%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	42 (-12.5%)	54 (-1.82%)	59 (1.72%)	59 (1.72%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	48 (0.0%)	55 (0.0%)	59 (1.72%)	59 (1.72%)
Math (106)	LLM4FL	68	87	92	94
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	59 (-13.24%)	79 (-9.2%)	85 (-7.61%)	85 (-9.57%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	49 (-27.94%)	68 (-21.84%)	70 (-23.91%)	78 (-17.02%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	57 (-16.18%)	80 (-8.05%)	85 (-7.61%)	85 (-9.57%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	68 (0.0%)	87 (0.0%)	91 (-1.09%)	94 (0.0%)
Mockito (38)	LLM4FL	21	22	26	27
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	18 (-14.29%)	23 (4.55%)	23 (-11.54%)	23 (-14.81%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	15 (-28.57%)	20 (-9.09%)	21 (-19.23%)	21 (-22.22%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	18 (-14.29%)	20 (-9.09%)	20 (-23.08%)	22 (-18.52%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	21 (0.0%)	22 (0.0%)	26 (0.0%)	27 (0.0%)
Time (26)	LLM4FL	14	21	22	23
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	12 (-14.29%)	20 (-4.76%)	22 (0.0%)	22 (-4.35%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	10 (-28.57%)	16 (-23.81%)	17 (-22.73%)	20 (-13.04%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	12 (-14.29%)	19 (-9.52%)	19 (-13.64%)	22 (-4.35%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	14 (0.0%)	22 (4.76%)	23 (4.55%)	23 (0.0%)
Total (675)	LLM4FL	327	425	473	494
	$\text{LLM4FL}_{\text{w/o CodeNavigation}}$	273 (-16.51%)	378 (-11.06%)	409 (-13.53%)	409 (-17.21%)
	$\text{LLM4FL}_{\text{w/o Divide\&Conquer}}$	251 (-23.24%)	341 (-19.76%)	365 (-22.83%)	381 (-22.87%)
	$\text{LLM4FL}_{\text{w/o Reflection}}$	290 (-11.31%)	400 (-5.88%)	436 (-7.82%)	459 (-7.09%)
	$\text{LLM4FL}_{\text{w/o CoT}}$	325 (-0.61%)	424 (-0.24%)	473 (0.0%)	495 (0.2%)

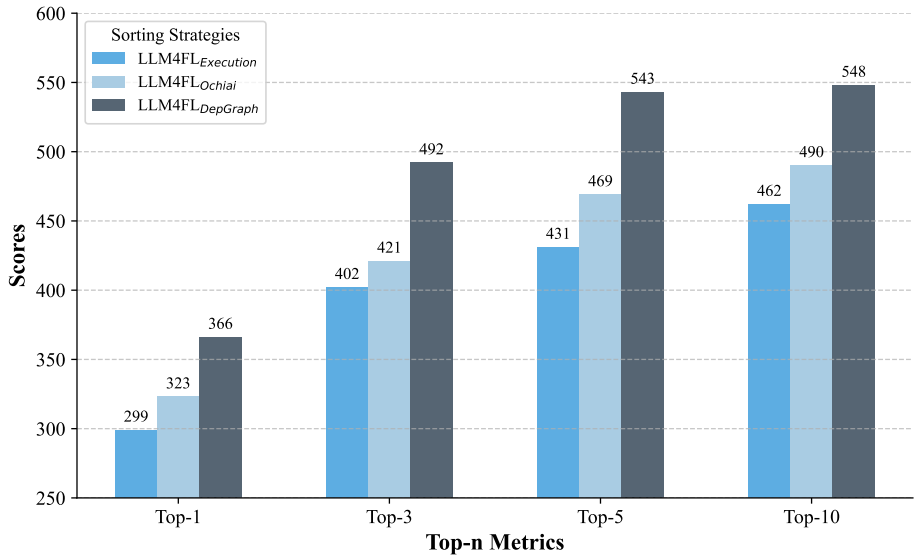

Fig. 4. Fault localization results when using different method sorting strategies during the segmentation process.

the methods, with no explicit ranking or prioritization. By providing the LLM with methods based on the execution sequence, we establish a control case to measure its performance without any ranking influence. The result shows how the order of premises in reasoning tasks can lead to varied LLM performance in software engineering context [15].

LLM4FL$_{Ochiai}$: As discussed in Section 3.1, we apply *Ochiai* to sort the methods during the segmentation process. *Ochiai* is unsupervised and is efficient to compute. We hypothesize that providing the LLM with methods sorted by their suspiciousness score will lead to more effective fault localization, as the model will focus on the most likely faulty candidates earlier in the process.

LLM4FL$_{DepGraph}$: This approach uses the ranking produced by *DepGraph*, a state-of-the-art Graph Neural Network (GNN)-based fault localization technique [34, 49], to sort the methods during the segmentation process. *DepGraph* ranks methods based on structural code dependencies and code change history. As shown in RQ1, *DepGraph* shows the highest fault localization accuracy among all the techniques, surpassing *LLM4FL$_{Ochiai}$*. By examining the fault localization result after sorting the methods using *DepGraph*'s scores, we can better study if the initial order affects LLM's results, even though LLM eventually visits all the methods.

Results. ***Method ordering has a significant impact on LLM's fault localization result, with up to 22% difference in Top-1 (from 299 to 366).*** Figure 4 shows the fault localization results when using different sorting strategies. When methods were presented in an execution-based order, *LLM4FL$_{Execution}$* achieved a Top-1 score of 299, gradually increasing to 402 for Top-3, 431 for Top-5, and reaching 462 for Top-10. This performance establishes a baseline, showing how the LLM behaves without strategic ordering. However, sorting methods with the lightweight *Ochiai* scores resulted in noticeable improvements. *LLM4FL$_{Ochiai}$* improved the Top-1 score to 323, an 8% increase over *LLM4FL$_{Execution}$*. The improvements also extended to other metrics, with the Top-3 score increasing to 421, the Top-5 reaching 469, and the Top-10 rising to 490.

LLM4FL$_{DepGraph}$ provides further improvement to the already-promising result of DepGraph, indicating method ordering is critical to LLM4FL, or LLM-based fault localization in general. *LLM4FL$_{DepGraph}$* achieved the highest Top-1 score of 366, which is significantly outperforming both LLM4FL$_{\text{Execution}}$ and LLM4FL$_{\text{Ochiai}}$. The improvement was consistent across all the metrics. For Top-3, the score rose to 492, providing a substantial boost over *LLM4FL$_{Ochiai}$*. Similarly, *LLM4FL$_{DepGraph}$* excelled in the Top-5 and Top-10 categories, reaching 543 and 548, respectively. We also see that *LLM4FL$_{DepGraph}$* has better Top-1, 3, and 5 scores compared to *DepGraph*. This consistent improvement underscores the importance of method ordering in enhancing the accuracy of LLM-based fault localization. This underscores the importance of method ordering for LLM-based fault localization, showing that lightweight techniques like *Ochiai* can achieve substantial performance gains without the heavy computational burden. Striking this balance between accuracy and efficiency makes these approaches particularly suitable for a wide range of fault localization tasks. However, it is important to recognize that *LLM4FL$_{DepGraph}$* is computationally expensive due to its reliance on graph neural networks (GNNs). While *LLM4FL$_{DepGraph}$* delivers top-tier results, *LLM4FL$_{Ochiai}$* also offers a strong, efficient alternative, delivering good localization accuracy at a fraction of the computational cost due to the unsupervised nature of *Ochiai*. In other words, *LLM4FL$_{Ochiai}$* may be more easily adapted in practice.

Nevertheless, our finding establishes a new research direction for LLM-based fault localization. It demonstrates that intelligent method ordering strategies significantly impact the result of LLM-based fault localization. This approach opens up further opportunities for optimizing LLM-based fault localization by exploring more advanced ordering techniques and how different premises of ordering affect other software engineering tasks.

The findings highlight that method ordering plays a crucial role in improving the performance of LLM-based fault localization, with a difference of up to 22% in Top-1 scores. While *LLM4FL$_{DepGraph}$* delivers the best results, the lightweight *LLM4FL$_{Ochiai}$* offers a highly efficient alternative, providing significant accuracy gains with far lower computational costs, making it more practical for real-world adoption.

5 Threats To Validity

Internal Validity. A potential threat to internal validity is the risk of data leakage in large language models (LLMs), where the model might have been exposed to data similar to the benchmark or specific fault localization cases during training. This could result in artificially inflated performance as the model may have prior knowledge of the evaluation data. Nevertheless, we mitigate this risk by preventing any content from being entered into ChatGPT that is related to the project name, the human-written bug report, or the bug ID.

External Validity. The primary threat to external validity is the generalizability of our results. Our evaluation is based on Defects4J, a well-established dataset in the software engineering community. Although this dataset includes real-world bugs, the systems studied are primarily Java-based, and it remains uncertain whether our findings will generalize to other programming languages or domains.

Construct Validity. Construct validity relates to whether the metrics we used accurately measure the performance of fault localization techniques. We used widely accepted Top-N metrics, which are commonly utilized in prior fault localization studies. However, our results are based on the assumption that developers primarily focus on the top-ranked faulty methods. Although this assumption aligns with previous research, different development practices could influence the effectiveness of our approach.

6 Conclusion

In this paper, we introduced *LLM4FL*, an LLM-agent-based fault localization technique that addresses the challenges of traditional fault localization methods and the limitations of current LLM-based approaches. By combining SBFL rankings with a divide-and-conquer strategy, *LLM4FL* groups large coverage data into manageable pieces, enabling effective analysis within the token limitations of LLMs. The use of multiple agents, prompt chaining, and self-reflection techniques allows for iterative refinement of fault localization, while chain-of-thought reasoning further enhances accuracy by generating potential fixes and re-rank the methods. Our evaluation on the Defects4J (V2.0.0) benchmark demonstrated that *LLM4FL* outperforms existing LLM-based techniques, achieving 19.27% higher Top-1 accuracy compared to AutoFL, and surpasses supervised approaches such as DeepFL and Grace without task-specific training. We also found that key components like coverage splitting and prompt chaining are essential to *LLM4FL*'s success, with method ordering playing a significant role in performance, improving Top-1 accuracy by up to 22%. Overall, *LLM4FL* presents a scalable and efficient solution for fault localization, providing improved accuracy and reducing computational costs, making it practical for real-world software projects. Future work will explore expanding *LLM4FL*'s capabilities for larger and more diverse codebases, further refining the agent collaboration and reasoning mechanisms.

References

[1] 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*. 201–210.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2006. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. IEEE, 39–46.

[3] 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*. IEEE, 88–99.

[4] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774* (2023).

[5] Abdulaziz Alaboudi and Thomas D LaToza. 2021. An exploratory study of debugging episodes. *arXiv preprint arXiv:2105.02162* (2021).

[6] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. 2017. Understanding of a convolutional neural network. In *2017 international conference on engineering and technology (ICET)*. Ieee, 1–6.

[7] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th international symposium on software testing and analysis*. 177–188.

[8] Guangsheng Bao, Hongbo Zhang, Linyi Yang, Cunxiang Wang, and Yue Zhang. 2024. Llms with chain-of-thought are non-causal reasoners. *arXiv preprint arXiv:2402.16048* (2024).

[9] Samuel Benton, Xia Li, Yiling Lou, and Lingming Zhang. 2020. On the effectiveness of unified debugging: An extensive study on 16 program repair systems. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 907–918.

[10] Sardar Bin Murtaza, Aidan Mccoy, Zhiyuan Ren, Aidan Murphy, and Wolfgang Banzhaf. 2024. LLM Fault Localisation within Evolutionary Computation Based Automated Program Repair. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 1824–1829.

[11] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 2017 11th joint meeting on foundations of software engineering*. 117–128.

[12] Tom B Brown. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[13] 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*. 378–381.

[14] 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*. 1–12.

[15] 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* (2024).
[16] Zhanqi Cui, Minghua Jia, Xiang Chen, Liwei Zheng, and Xiulei Liu. 2020. Improving software fault localization by combining spectrum and mutation. *IEEE Access* 8 (2020), 172296–172307.
[17] Arpita Dutta and Sangharatna Godboley. 2021. Msfl: A model for fault localization using mutation-spectra technique. In *Lean and Agile Software Development: 5th International Conference, LASD 2021, Virtual Event, January 23, 2021, Proceedings 5*. Springer, 156–173.
[18] Muhammad Usman Hadi, Rizwan Qureshi, Abbas Shah, Muhammad Irfan, Anas Zafar, Muhammad Bilal Shaikh, Naveed Akhtar, Jia Wu, Seyedali Mirjalili, et al. 2023. Large language models: a comprehensive survey of its applications, challenges, limitations, and future prospects. *Authorea Preprints* (2023).
[19] C. Hait and G. Tassey. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. DIANE Publishing Company.
[20] 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*. https://openreview.net/forum?id=VtmBAGCN7o
[21] 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* (2023).
[22] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232* (2023).
[23] James A Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*. 467–477.
[24] 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*. 437–440.
[25] 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, Article 64 (jul 2024), 23 pages. https://doi.org/10.1145/3660771
[26] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th international symposium on software testing and analysis*. 165–176.
[27] Langchain. 2024. Langchain Documentation: Overview. https://python.langchain.com/v0.2/docs/versions/overview/ Accessed: 2024-09-04.
[28] Langchain. 2024. LangGraph: Graph-Based Extensions for LangChain. https://langchain-ai.github.io/langgraph/ Accessed: 2024-09-04.
[29] Tien-Duy B Le, Ferdian Thung, and David Lo. 2013. Theory and practice, do they match? a case with spectrum-based fault localization. In *2013 IEEE International Conference on Software Maintenance*. IEEE, 380–383.
[30] Mosh Levy, Alon Jacoby, and Yoav Goldberg. 2024. Same task, more tokens: the impact of input length on the reasoning performance of large language models. *arXiv preprint arXiv:2402.14848* (2024).
[31] Jierui Li, Szymon Tworkowski, Yingying Wu, and Raymond Mooney. 2023. Explaining competitive-level programming solutions using llms. *arXiv preprint arXiv:2307.05337* (2023).
[32] 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*. 169–180.
[33] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–30.
[34] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).
[35] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673.
[36] Feng Lin, Dong Jae Kim, et al. 2024. When llm-based code generation meets the software development process. *arXiv preprint arXiv:2403.15852* (2024).
[37] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the middle: How language models use long contexts. *Transactions of the Association for Computational Linguistics* 12 (2024), 157–173.
[38] 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. 664–676.

[39] Meta AI. 2024. Meta AI Introduces LLaMA 3: Advancing Open Foundation Models. https://ai.meta.com/blog/meta-llama-3/. Accessed: 2024-09-06.

[40] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the mutants: Mutating faulty programs for fault localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 153–162.

[41] OpenAI. 2024. GPT-4O Mini: Advancing Cost-Efficient Intelligence. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/ Accessed: 2024-09-04.

[42] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[43] 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*. 199–209.

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

[45] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924* 6 (2023).

[46] Jie Qian, Xiaolin Ju, and Xiang Chen. 2023. GNet4FL: effective fault localization via graph convolutional neural network. *Automated Software Engineering* 30, 2 (2023), 16.

[47] Jie Qian, Xiaolin Ju, Xiang Chen, Hao Shen, and Yiheng Shen. 2021. AGFL: a graph convolutional neural network-based method for fault localization. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 672–680.

[48] 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* (2024).

[49] 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, Article 86 (jul 2024), 23 pages. https://doi.org/10.1145/3660793

[50] Matthew Renze and Erhan Guven. 2024. Self-Reflection in LLM Agents: Effects on Problem-Solving Performance. *arXiv preprint arXiv:2405.06682* (2024).

[51] 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*. 208–219.

[52] 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* (2023).

[53] Yunfan Shao, Linyang Li, Junqi Dai, and Xipeng Qiu. 2023. Character-llm: A trainable agent for role-playing. *arXiv preprint arXiv:2310.10158* (2023).

[54] 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*. 273–283.

[55] Gladys Tyen, Hassan Mansoor, Victor Cărbune, Yuanzhu Peter Chen, and Tony Mak. 2024. LLMs cannot find reasoning errors, but can correct them given the error location. In *Findings of the Association for Computational Linguistics ACL 2024*. 13894–13908.

[56] 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)*. IEEE, 365–376.

[57] Yu Wang, Shiwan Zhao, Zhihu Wang, Heyuan Huang, Ming Fan, Yubo Zhang, Zhixing Wang, Haijun Wang, and Ting Liu. 2024. Strategic Chain-of-Thought: Guiding Accurate Reasoning in LLMs through Strategy Elicitation. *arXiv preprint arXiv:2409.03271* (2024).

[58] Ming Wen, Junjie Chen, Yongqiang Tian, Rongxin Wu, Dan Hao, Shi Han, and Shing-Chi Cheung. 2019. Historical spectrum based fault localization. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2348–2368.

[59] 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*. Springer, 71–108.

[60] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2013. The DStar method for effective software fault localization. *IEEE Transactions on Reliability* 63, 1 (2013), 290–308.

[61] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2011. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability* 61, 1 (2011), 149–169.

[62] 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 (2016), 707–740.

[63] W Eric Wong and Yu Qi. 2009. BP neural network-based effective fault localization. *International Journal of Software Engineering and Knowledge Engineering* 19, 04 (2009), 573–597.

[64] 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* (2023).

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

[66] Xiaoyuan Xie, Tsong Yueh Chen, Fei-Ching Kuo, and Baowen Xu. 2013. A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. *ACM Transactions on software engineering and methodology (TOSEM)* 22, 4 (2013), 1–40.

[67] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. 2016. Revisit of automatic debugging via human focus-tracking analysis. In *Proceedings of the 38th International Conference on Software Engineering*. 808–819.

[68] Jiaxi Xu, Fei Wang, and Jun Ai. 2020. Defect prediction with semantics and context features of codes based on graph representation learning. *IEEE Transactions on Reliability* 70, 2 (2020), 613–625.

[69] Xuezheng Xu, Changwei Zou, and Jingling Xue. 2020. Every mutation should be rewarded: Boosting fault localization with mutated predicates. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 196–207.

[70] Yuzhuang Xu, Shuo Wang, Peng Li, Fuwen Luo, Xiaolong Wang, Weidong Liu, and Yang Liu. 2023. Exploring large language models for communication games: An empirical study on werewolf. *arXiv preprint arXiv:2309.04658* (2023).

[71] Ziwei Xu, Sanjay Jain, and Mohan Kankanhalli. 2024. Hallucination is inevitable: An innate limitation of large language models. *arXiv preprint arXiv:2401.11817* (2024).

[72] 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* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 17, 12 pages. https://doi.org/10.1145/3597503.3623342

[73] 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*. 261–272.

[74] Mengshi Zhang, Yaoxian Li, Xia Li, Lingchao Chen, Yuqun Zhang, Lingming Zhang, and Sarfraz Khurshid. 2019. An empirical study of boosting spectrum-based fault localization via pagerank. *IEEE Transactions on Software Engineering* 47, 6 (2019), 1089–1113.

[75] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. 2019. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 445–455.

[76] 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* (2023).

[77] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering* 47, 2 (2019), 332–347.