Let the Code Speak: Incorporating Program Dynamic State for Better Method-Level Fault Localization

Yihao Qin, Shangwen Wang*, Bo Lin, Xin Peng, Sheng Ouyang, Liqian Chen, Xiaoguang Mao
College of Computer Science and Technology
National University of Defense Technology, China
{yihaoqin, wangshangwen13, linbo19, xinpeng, ouyangsheng23, lqchen, xgmao}@nudt.edu.cn

***Abstract*—Fault localization (FL) is a critical but time-consuming part of software debugging. With the improvement of the Large Language Models (LLMs) in their code capabilities, the increasing demand for automated software development has encouraged more research on building LLM-based Fault Localization (LLMFL) systems. However, existing LLMFL techniques are typically restricted to predicting bug locations by analyzing *static code*, while overlooking crucial *dynamic program state* of the software. This lack of context makes LLMs prone to generating "hallucinations", incorrectly identifying bug-free code as suspicious. To address this, this paper introduces PINGFL, the LLMFL system that incorporates program dynamic information for more accurate automatic fault localization. PINGFL comprises a Fault Localization (FL) agent and a Print Debugging (PD) agent. The FL agent is tasked with understanding the root cause through a set of callable tools. When the FL agent nominates a location as suspicious, it would entrust the PD agent to verify the suspected issue through multiple rounds of print debugging. In particular, these two agents communicate efficiently by conveying the textual thought generated by the LLM. The evaluation on 812 real-world bugs from the Defects4J benchmark shows that PINGFL can localize 450 bugs within Top-1, which significantly outperforms other LLM-based approaches by 41% to 122%. A deeper dive into PINGFL's performance reveals that it exhibits specific FL strategies and tool usage patterns even without explicit instructions. Finally, PINGFL proves to be cost-effective, spending an average of $0.23 and 104.62 seconds per bug, with the print debugging mechanism accounting for only $0.07 and 48.14 seconds.**

***Index Terms*—Large Language Model, Fault Localization, Print Debugging**

I. INTRODUCTION

Fault localization (FL) is a critical step in the software debugging process, aiming to pinpoint the problematic program elements within a codebase. The FL process is usually time-consuming, as developers can spend up to half of their debugging time on identifying the root cause of failures [1]. To alleviate the burden of manual fault localization, researchers have proposed various automated FL techniques. Mainstream FL techniques primarily include spectrum-based fault localization (SBFL) and learning-based fault localization (LBFL). SBFL works on the assumption that faulty code elements tend to be covered by more failing tests and fewer passing tests. It utilizes formulas like Ochiai [2] to compute suspiciousness values for program elements. LBFL goes a step further by integrating a wider array of software features, such as code complexity, textual similarity, and graph representations of code structures. It then trains models with diverse architectures, such as recurrent neural networks (RNNs) [3] or graph neural networks (GNNs) [4], to learn implicit bug patterns. However, these traditional FL techniques often face complaints from practitioners due to the lack of bug explanation or complicated training processes [5], which results in their rare deployment in real-world production scenarios.

The rise of large language models (LLMs) has profoundly influenced software engineering development and maintenance paradigms. Automated programming assistants such as GitHub Copilot [6] and Cursor [7] have become integral to many developers' daily workflows. As one of the foundational technologies for more advanced automated programming, LLM-based fault localization (LLMFL) has recently attracted increasing research attention. Existing LLMFL techniques can be broadly categorized into two types. The approaches like SoapFL [8] involve manually decomposing the FL task into multiple steps and then instructing the LLM to perform inference step-by-step according to a predefined workflow. The second approach, represented by AutoFL [9], provides the LLM with various callable tools that allow it to autonomously retrieve desired information from code repositories. However, a significant challenge faced by current LLMFL techniques is the "hallucination" problem, where LLMs may misidentify bug-free code as faulty locations. We attribute it to the fact that **these approaches limit LLMs to analyze root cause through only examining source code, without access to the program's internal runtime states**.

To overcome the above issue, our key insight is that **dynamic software information may be an important evidence to help LLM verify its hypotheses from static analysis**. For obtaining dynamic information, we turn to print debugging [10], a simple yet effective debugging technique widely adopted by developers. Our observation from motivating cases shows that LLMs can acquire crucial evidence for

*Shangwen Wang is the corresponding author.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00072

confirming the root cause by strategically printing variables within the suspicious code. Based on the above idea, we introduce PINGFL, the first LLM-based FL agent capable of performing Print debuggING. Specifically, PINGFL consists of two collaborative LLM agents: a Fault Localization (FL) agent and a Print Debugging (PD) agent. The FL agent coordinates the entire FL task, utilizing a set of predefined tools to obtain coverage information, retrieve source code, or nominate suspicious methods. In particular, when the FL agent suspects a method as erroneous, it delegates the verification task to the PD agent through a tool call. The PD agent then performs multiple rounds of print debugging to analyze the software's runtime state, providing verification feedback based on actual program outputs. To seamlessly integrate print debugging into the FL process, we've designed two effective mechanisms: (1) Unlike existing LLMFL that regard FL as a "Information Collection - Summarization" process, PINGFL explicitly prompts the LLM to provide a thought before each tool invocation. This design allows the LLM to always keep aware of the task completion status, which aligns more with the "Thought-Action" pattern seen in ReAct [11] agents; (2) We use the "suspected issue" raised by the FL agent as a communication bridge between the FL and PD agents. This not only prevents overwhelming context from historical messages but also helps the PD agent to clarify its debugging objective.

We comprehensively evaluate PINGFL on the widely used Defects4J [12] benchmark, which includes 812 real-world bugs from 16 Java projects. We compare PINGFL with various traditional FL techniques (e.g., Ochiai [2], DeepFL [3], and GRACE [4]) and the state-of-the-art FL systems (e.g., AutoFL [9], SoapFL [8], and Agentless [13]) driven by LLMs. The results show that PINGFL can localize 450 out of 812 bugs within Top-1, which significantly surpasses other LLM-based approaches by 41% to 122%. PINGFL also consistently outperforms the traditional FL techniques on the cross-project scenario. We conducted a detailed investigation into the state transition and tool usage of PINGFL during the FL task. We discovered that even without any special instructions, PINGFL is able to exhibit specific FL strategies and tool usage patterns. Finally, our cost analysis shows that PINGFL spends an average of 0.23 dollars and 104.62 seconds to localize a fault, while the print debugging mechanism we introduced in this paper takes only 0.07 dollars and 48.14 seconds. The main contributions of this paper are:

- **Perspective.** We explore the feasibility of using program dynamic information to enhance LLM-based FL.
- **Methodology.** We present PINGFL, which incorporates the print debugging technique for more accurate method-level FL. PINGFL is publicly available at: *https://github.com/IntHelloWorld/PingFL*.
- **Experiment.** We comprehensively evaluate PINGFL on 812 real-world bugs. The result shows that PINGFL significantly outperforms the other LLM-based approaches. Additionally, we observe that PINGFL exhibits specific tool usage strategies during the FL process.

II. BACKGROUND & RELATED WORK

A. Print Debugging

Print debugging is a straightforward yet highly effective method for identifying and resolving code issues. The core idea behind this technique is to strategically insert print statements (e.g., `System.out.println()` in Java) at various points within the codebase. With the program outputs, the developers can trace the program's execution flow and understand software state during runtime, thereby facilitating the localization and repair of bugs. Interestingly, research by Beller et al. [10] has discovered that although more advanced debugging tools like breakpoint debuggers are now available, print debugging remains a commonly used method for rapidly pinpointing issues. Its intuitiveness and lack of complex configuration make it a favored choice for many developers, particularly in circumstances when a quick verification of a small logical point is needed.

B. Traditional Fault Localization Techniques

Among various FL techniques, spectrum-based FL (SBFL) and learning-based FL (LBFL) have been extensively studied and achieve prominent performance on the Defects4J [12] benchmark. The SBFL is built on a basic statistical assumption that the fault locations tend to be executed by more failing tests and less passing tests. Given a defective software and the test suite including both passing and failing test cases, SBFL first collects coverage information for each program element e by recording the number of passed tests $T_p(e)$ and failed tests $T_f(e)$ that cover e. It then utilizes different formulas such as Ochiai [2], and DStar [14] to calculate a suspiciousness score for each element e.

To combine various information for better FL, the learning-based fault localization (LBFL) has arisen with the rapid development of deep learning techniques. DeepFL [3] collects various kinds of code features including coverage information, code complexity, and text similarity. It then trains RNN [15] and MLP [16] models to identify the implicit patterns of the bug. GRACE [4] proposes to model each buggy method as a unified graph that integrates both coverage information and fine-grained code structures. Next, it utilizes a Gated Graph Sequence Neural Network (GGNN) [17] for learning the representation of the buggy method. There are also many other LBFL techniques such as CNNFL [18], TraPT [19], CombineFL [20], etc.

C. LLM-Based Fault Localization

The emergence of Large Language Models (LLMs) has unlocked a new era for fault localization (FL) technology. Current mainstream LLM-based FL techniques (LLMFL) generally fall into two categories. One category designs fixed workflows for LLMs. For instance, SoapFL [8] decomposes the FL task into three steps: Fault Comprehension, Codebase Navigation, and Fault Confirmation. It leverages the code documentation to guide the LLM to progressively pinpoint finer-grained suspicious code locations. The second category grants LLMs to make decisions on their own. For example,

AutoFL [9] provides LLMs with multiple callable tools, allowing them to actively retrieve bug-related context from the codebase before predicting the suspicious locations. Besides these two representative works, some other approaches [21], [22] or program repair systems [13], [23] are also involved in leveraging LLMs for FL. Agentless [13] introduces an FL pipeline based on several fixed steps. It first identifies potentially problematic files through project directory structure and retrieval techniques. Then, it guides an LLM to pinpoint buggy program elements by examining the code skeleton (i.e., a list of declaration headers for classes and functions).

D. LLM-Powered Debugger

The debuggers (such as JDB for Java and Pdb for Python) allow programmers to identify and fix bugs by investigating program state and navigating program execution. To enhance the user-friendliness of conventional debuggers, Levin et al. has recently presented a LLM-driven debugging assistant ChatDBG [24]. Given a user query like *"why is variable x null?"*, ChatDBG is granted to autonomously investigate the program state and source code for answering the query. Nevertheless, we note that while both ChatDBG and PINGFL leverage dynamic program information to enhance fault localization, they differ significantly in two key aspects: **(1) Automation Level:** PINGFL aims to automate the entire fault localization process; In contrast, ChatDBG is a semi-automated approach, focusing more on optimizing the interaction experience between traditional debuggers and users. **(2) LLM's Role:** ChatDBG uses LLMs to interpret information from debuggers and provide more user-friendly answers; Conversely, PINGFL explores whether LLMs themselves possess the capability to perform debugging like human developers.

III. MOTIVATING EXAMPLE

Figure 1 illustrates a bug extracted from the JavaScript code optimization software *closure-compiler* [1]. As shown in the bug information, the failing test expects to receive the optimized expression `(Math.random()>0.5"1":2)+3+4`, but the software incorrectly transforms `"1"` to `1` and folds the add operation (`+3+4` to `+7`). The root cause of this bug resides in method `mayBeString`, which fails to identify `"1"` as a string, thus incorrectly triggers the subsequent folding process of the method `tryFoldAdd`.

For this case, we observe that when provided with the bug information and the general code navigation tools [9], the LLM can easily nominate the `tryFoldAdd()` method as suspicious, as it notices that this method *"pertains directly to addition folding"*. However, due to the lack of software runtime information, the LLM was unable to realize that the fault is not caused by the `tryFoldAdd()` method itself. The actual problem resides in its callee method `mayBeString()`, which returns an unexpected variable `mayBeString`, blocking the program from entering the if branch on line 4. This raises a critical problem: **To move beyond surface-level suspicion, the LLM must be equipped with a way to test its hypotheses and perform reflexion**, just as human developers may rely on runtime feedback. To address this problem, we draw inspiration from the widely used debugging technique: print debugging [10]. Specifically, we first instrument the `tryFoldAdd()` method at line 3 by printing the variable `mayBeString`. We then re-execute the method to obtain the software output *"mayBeString: false"*. This output indicates that the root cause is resident in `mayBeString()`, as it produces the wrong value for `mayBeString` variable that mutes the correct functionality (Lines 5 to 8) of `tryFoldAdd()`. Finally, we prompt the LLM to reflect on its former response based on the software output. We found that such a simple mechanism works surprisingly well: the LLM correctly concludes that *"the issue lies earlier in the `mayBeString()` method"*, not in the folding logic itself. This example demonstrates a key insight: **LLMs, when equipped with the ability to perform print debugging, may more accurately pinpoint the root cause through understanding the dynamic program states.**

Motivated by this observation, our work investigates how an LLM-driven print debugging mechanism can significantly enhance FL. By integrating runtime signals into the LLM's reasoning loop, we aim to improve its diagnostic precision and reliability, thereby advancing automated FL techniques toward practical and deployable software maintenance tools.

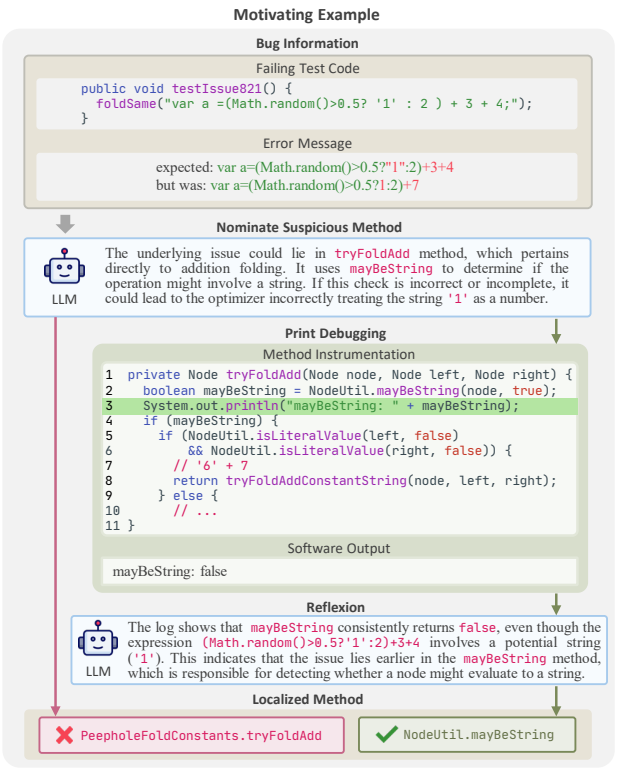


Fig. 1: Motivating Example.

[1] https://storage.googleapis.com/google-code-archive/v2/code.google.com/closure-compiler/issues/issue-821.json

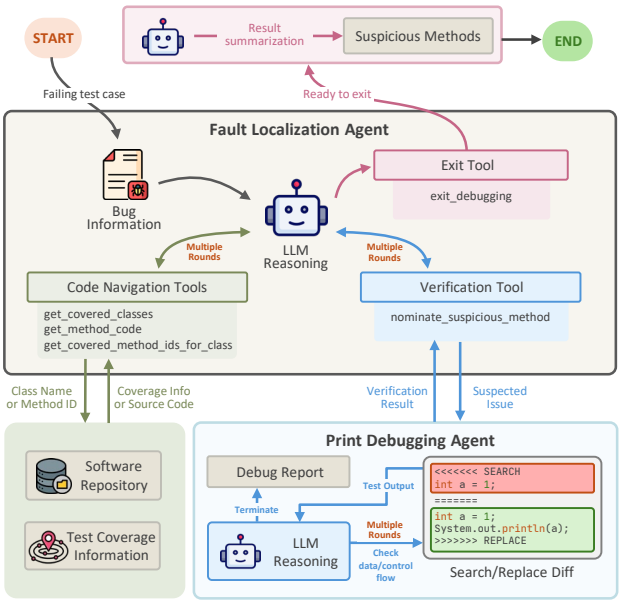


Fig. 2: Overview of PINGFL.

IV. APPROACH

A. Overview

Figure 2 presents an overview of PINGFL, which consists of two agents: the Fault Localization (FL) Agent and the Print Debugging (PD) Agent. Given the bug information from a single failing test case, the FL Agent is responsible for understanding the root cause through multiple rounds of interaction with a set of debugging tools. Specifically, we have designed three types of tools: (1) Code Navigation Tools: These tools retrieve bug-related context from the Software Repository and Test Coverage Information. (2) Verification Tool: This tool nominates and verifies a suspicious method through print debugging. (3) Exit Tool: This tool enables the agent to terminate the FL process. Notably, the Verification Tool is delegated to the PD Agent, which strategically inserts print statements to analyze the runtime behavior of suspicious methods and returns a Debug Report as the verification result. Finally, after the FL Agent invokes the Exit Tool, an LLM is employed to review the entire FL procedure and enumerate all identified suspicious methods.

B. Problem Definition

In this work, we define the FL process as an LLM-driven decision process, where the state reflects the LLM's accumulated memory of prior interactions, and each action corresponds to invoking a specific tool to gather or analyze code information. This formulation enables systematic reasoning over sequential decisions and highlights the interaction between the LLM-based agent and the environment (i.e., the codebase and the execution framework). Formally, an FL process (S, E, H, A, T, π) driven by LLM agent is as follows:

1) Action Space. In PINGFL, the FL agent performs specific actions by using the function call [25] ability of LLM.

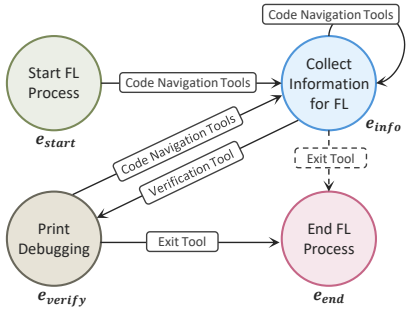


Fig. 3: Migration relationship between debugging episodes.

For example, to perform an action $a \in A$, the LLM would provide the name and arguments of the function it wants to call. Consequently, the predefined function will be executed and return the textual result o. We define the action space as $A = \{a_{\text{gcc}}, a_{\text{gcm}}, a_{\text{gmc}}, a_{\text{nsm}}, a_{\text{exit}}\}$. These actions include retrieving classes covered by the failing test (a_{gcc}), retrieving methods covered in a class (a_{gcm}), fetching method source code (a_{gmc}), nominating a suspicious method and verifying it through print debugging (a_{nsm}), and exiting debugging (a_{exit}). We refer to Section IV-D for more details of the tools.

2) State Space. The LLM-based agent's state incorporates the working memory accumulated so far. We represent the working memory at reasoning round t as a historical message queue $h_t = (u_0, a_0, o_0) \rightarrow \cdots \rightarrow (u_{t-1}, a_{t-1}, o_{t-1})$, where u is the agent's thought before it intends to perform an action, $a \in A$ represents the actions taken by the agent, and o denotes the observation obtained after each action.

We use four distinct episodes $E = \{e_{\text{start}}, e_{\text{info}}, e_{\text{verify}}, e_{\text{end}}\}$ to represent the sub-steps that the FL Agent is currently addressing. As illustrated in Figure 3, e_{start} represents the initial episode, where the agent is informed about the bug information and prepares to initiate the FL task. The agent then transitions to e_{info} using code navigation tools $\{a_{\text{gcc}}, a_{\text{gcm}}, a_{\text{gmc}}\}$, indicating that the agent is attempting to retrieve bug-related information from the code repository and test coverage. The agent can collect information for several rounds until it identifies a suspicious code location. It then nominates a suspicious method and moves to the e_{verify} episode by calling the verification tool a_{nsm}. In the e_{verify} episode, the Print Debugging Agent is employed to further investigate the suspicious method and verify whether it is the authentic root cause. To localize a fault, the agent may switch its episode between e_{info} and e_{verify} multiple times. Finally, when the FL agent's hypothesis about the root cause has been verified through print debugging, it transitions to e_{end} via the exit tool a_{exit}, signaling the completion of the FL process. In some edge cases, the agent might directly ensure the root cause by reviewing the code and terminating the FL process without going through the e_{verify} episode. Consequently, we define the state space as $S = E \times H$, where the state at round t can be represented as (e, h_t).

3) State Transition Function. After a function (tool) is invoked, the agent would add the new interaction information to its memory and go to the next debugging episode. For

TABLE I: Tools for Fault Localization Agent

Tool Type	Tool Name	Input	Description
Code Navigation Tools	get_covered_classes	None	Get the names of all classes covered by the failing test.
	get_covered_method_ids_for_class	(Inner) Class Name	Get the method IDs of all covered methods within the class.
	get_method_code	Method ID	Get the source code for the specific method ID.
Verification Tool	nominate_suspicious_method	Method ID, Thought	Nominate and verify a suspicious method through print debugging.
Exit Tool	exit_debugging	None	Terminate the fault localization process.

example, if the agent at reasoning step t decides to gather more information from the code repository, it will use one of the Code Navigation Tools (i.e., take one action a_t among $\{a_{\text{gcc}}, a_{\text{gcm}}, a_{\text{gmc}}\}$). After that, the agent will transition to the next state s_{t+1} by setting its new episode as e_{info} and expanding the memory to $h_t \rightarrow (u_t, a_t, o_t)$. Here, u_t is the LLM's thought and $o_t = \text{ExecuteTool}(a_t)$ represents the outcome of calling the function a_t. Generally, the state transition function $T : S \times A \rightarrow S$ can be formalized as:

$$s_{t+1} = \begin{cases} (e_{\text{info}}, h_t \rightarrow (u_t, a_t, o_t)) & \text{if } a_t \in \{a_{\text{gcc}}, a_{\text{gcm}}, a_{\text{gmc}}\} \\ (e_{\text{verify}}, h_t \rightarrow (u_t, a_t, o_t)) & \text{if } a_t = a_{\text{nsm}} \\ (e_{\text{end}}, h_t \rightarrow (u_t, a_t, o_t)) & \text{if } a_t = a_{\text{exit}} \end{cases} \tag{1}$$

4) Policy Function. For the FL agent in PINGFL, the policy function $\pi : S \rightarrow \Delta(A)$ is implicitly generated by the LLM. Specifically, $\pi(u, a|(e_t, h_t)) = \mathbb{P}_{\text{LLM}}(u, a|h_t)$, where the LLM provides a thought u based on the history messages h_t and determines the next tool a to invoke.

C. Prompt Design of FL Agent

The objective of FL agent is to interact with the debugging environment through multiple rounds of tool invocation, ultimately identifying suspicious methods. We will introduce the prompt for the FL agent in the following content, the complete prompt can be found in our online repository.

1) System Prompt. The FL agent autonomously performs incremental reasoning through repeated invocation of tools. We assign the agent the role of a *Software Debugging Assistant*, whose task is to *"understand the root cause of the bug step-by-step using the callable functions"*. We do not impose constraints on the agent's tool-use strategies, thereby allowing it to autonomously decide how to leverage the available tools. It is worth noting that previous FL methods [9], [22] based on LLM agents only emphasize collecting information through multiple rounds of tool calls, but fail to provide chances for the agent to think explicitly before calling each tool. Inspired by the ReAct Agent [11] paradigm, we introduced two simple yet effective designs to ensure the agent always provides a thought before each tool invocation: (1) We explicitly instruct the agent to *"explain your analysis and thoughts before each function call you initiate"*; (2) During the initialization of the default invoked tool *get_covered_classes*, we manually set the thought as *"First, let's look at all the classes covered by the failing test to understand the debugging scope."*

2) User Prompt. The user prompt section provides the FL agent with initial bug information, which guides the subsequent FL process. Note that a bug in the software may trigger multiple test cases to fail. Due to the differences in the purpose and program coverage of different failing tests, we use the bug information of a single failing test as the input for the FL agent. Specifically, we provide three types of key information: *TEST_NAME*, *TEST_CODE*, and *ERROR_MESSAGE*, which have been proven to be effective in previous studies [21].

3) Historical Message Queue. In the interaction round t_n, the historical message queue h_t encompasses all tool invocation messages initiated by the FL agent up to t_{n-1}, along with the corresponding tool invocation results. Each tool invocation message includes the thought of the LLM before using the tool, the tool name, and relevant parameters. This can be regarded as a Thought-Action step of the FL agent. The tool invocation result contains the textual feedback returned after the PINGFL executes the corresponding tool, which can be interpreted as the LLM's observation of the environment after it performs an action.

D. Tools for FL Agent

An excessive number of tools may lead to more failures in LLM function calls [26]. To strike a balance between simplicity and sufficiency, we have designed 3 types of tools as shown in Table I.

1) Code Navigation Tools. A prerequisite for the FL agent to find the buggy location is the ability to browse and analyze the source code within a software repository. Inspired by prior research [9], [23], we have designed three code navigation tools that enable the FL agent to access any covered method (i.e., methods executed by the failing test) from the codebase. The *get_covered_classes* tool enumerates all class names covered by the failing test, which enables the FL agent to understand the software structure from a high level. Given the foundational role of this tool in the overall FL process, we implement it as a default tool that should be invoked at the beginning. The *get_covered_method_ids_for_class* tool takes a class name as input and outputs the IDs of all covered methods within the specified class. As shown in Figure 4, the method ID is an identifier used to uniquely mark a method in the code repository. The above two tools allow the FL agent to iteratively expand its exploration of the code repository without overconsuming the LLM's context window. When the agent requires an in-depth inspection of the source code, it can invoke the *get_method_code* tool, which returns the complete source code of the method with the specified ID.

2) Verification Tool. The verification tool intends to investigate the runtime behavior of suspicious methods through print debugging, thereby assisting the FL agent in reflecting on whether the nominated method is indeed the actual fault

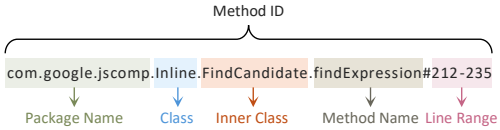


Fig. 4: The structure of the method ID.

location. Specifically, the *nominate_suspicious_method* tool takes the method ID to be verified and the corresponding thought generated by the LLM as inputs, where the thought typically contains critical information such as the suspected issue of the method and the items it expects to validate through print debugging. We implement this tool as a Print Debugging Agent, which is able to inspect key data/control flow by adaptively inserting print statements into the suspicious method. Please refer to Section IV-E for more details.

3) Exit Tool. The exit tool *exit_debugging* serves as a standardized way to end the FL process. The design maintains the consistency of the FL agent's behavior, which can perform any action through tool calls without generating responses in other formats to terminate the FL process.

E. Implementation of Print Debugging Agent

The primary objective for the Print Debugging (PD) Agent is to verify the suspicious method by editing the source code and analyzing the output logs. Specifically, for each failing test, we create a temporary "shadow project" to run all debugging and tests, which prevents side effects on the original project. We describe each component of the prompt in Figure 5 as follows:

1) System Prompt. The goal of the PD agent is *"provide an explanation about whether the suspicious method's functionality matches the suspected issue by performing Print Debugging"*. To perform print debugging, we ask the PD agent to insert print statements by generating a SEARCH/REPLACE edit [27]: a simple diff format to accurately replace the code snippet. As shown in Figure 5, a SEARCH/REPLACE edit contains two parts separated by equal signs: (1) the original code snippet needs to be replaced, and (2) the new code snippet to replace it with. This simple diff format enables small modifications to long code snippets, which not only reduces costs but also helps alleviate the hallucination of LLM.

In the Action Protocol, we specify the detailed behavioral guidelines for the PD agent as follows: (1) The agent should analyze the provided information and purposefully generate SEARCH/REPLACE edits to insert print statements into the method; (2) The agent then analyzes the program output of the modified method. It is allowed to *"re-edit the suspicious method from scratch"* if it has not collected sufficient evidence yet; (3) The aforementioned Edit-Analysis process can be repeated multiple times until the agent has gathered enough evidence or the *MAX_EDIT_COUNT* is reached. Ultimately, the agent is required to explain *"whether the suspicious method's functionality matches the suspected issue"*.

It is worth noting that guiding LLM to perform stable code editing is non-trivial. To solve this, we have adopted three main

Prompt for Print Debugging Agent

System Prompt

You are a Software Debugging Assistant. You will be provided with the test failure information, the suspicious Java method, and the suspected issue. Your goal is to provide an explanation about whether the suspicious method's functionality matches the suspected issue by performing **Print Debugging**.

Print Debugging
To perform print debugging, you should insert `System.out.println` statements into the suspicious Java method by generating a single *SEARCH/REPLACE* edit.

Here is an example:
We need to check the value of variable 'x'.

```
<<<<<<< SEARCH
    x = y + z;
=======
    x = y + z;
    System.out.println("Value of x: " + x);
>>>>>>> REPLACE
```

Action Protocol
1. Analyze the given information and generate a single *SEARCH/REPLACE* edit to purposefully debug the suspicious Java method, focus on a certain part of the logic at a time if the method code is long.
2. Judge whether you have collected enough evidence based on the program output, if not, please **re-edit the suspicious Java method from scratch** by generating a new *SEARCH/REPLACE* edit.
3. You can make up to **{MAX_EDIT_COUND}** edits. Once you have collected enough evidence or the maximum number of edits is reached, you should stop generating more edits and provide an explanation of whether the suspicious method's functionality matches the suspected issue.

User Prompt

Bug Information
The test **{TEST_NAME}** failed.
The source code of the failing test method is: **{TEST_CODE}**
It failed with the following test output and stack trace: **{ERROR_MESSAGE}**

Suspicious Method
The method **{METHOD_ID}** was considered suspicious.
The source code of the suspicious method is: **{METHOD_CODE}**

Suspected Issue
{SUSPECTED_ISSUE}

Fig. 5: The prompt template of the Print Debugging Agent.

stability optimizations: (1) We set the LLM temperature to 0, which improves edit stability and success rates in practice; (2) We built an error-handling mechanism that feeds specific edit errors back to the LLM, guiding it to make corrections; (3) There is no Fallback. To improve editing success rate, PINGFL is designed to always edit the original code from scratch in each attempt.

2) User Prompt. The user prompt provides the necessary information for the PD agent to perform print debugging. The bug information contains *TEST_NAME*, *TEST_CODE*, and *ERROR_MESSAGE*, which facilitates the PD agent in linking the functionality of suspicious methods to the root causes of test failures. The suspicious method provided the *METHOD_ID* and *METHOD_CODE* of the method under verification. As we mentioned in Section IV-C, the suspected issue is extracted from the FL agent's thought before it calls the verification tool. Specifically, the FL agent typically describes in its thought why the suspicious method may have problems and the expected behavior it hopes to verify through print debugging, which helps to seamlessly connect the workflow of the FL agent and the PD agent.

F. Results Synthesis

The final output of PINGFL is a ranked list of suspicious methods. To achieve this, we synthesize FL results from different failing test cases through a two-step process:

1) LLM-based Result Summarization. As presented in Figure 2, for each failing test case c, we follow the previous

work [9] to incorporate a result summarization operation after the FL agent terminates. We employ an LLM to review the entire FL process and enumerate all suspicious methods where bugs may reside. The prompt for the LLM is:

> Based on the available information, provide the IDs of the most likely culprit Java methods for the bug. Your answer will be processed automatically, so make sure to only answer with the accurate IDs of all likely culprits without commentary.

2) Suspicious Method Ranking. For all failing tests $C = \{c_1, c_2, \ldots, c_{|C|}\}$ triggered by a bug, we apply a scoring mechanism to produce the final ranked list. Assume that all suspicious methods predicted for a failing test c is $M^c = \{m_1^c, m_2^c, \ldots, m_{|M^c|}^c\}$, the suspicious score of a method m is calculated as:

$$\text{score}(m) = \frac{1}{|C|} \sum_{k=1}^{|C|} \left(\frac{1}{|M^k|} \cdot \mathbb{1}_{M_k}(m) \right) \tag{2}$$

where $\mathbb{1}_{M_k}(m)$ is an indicator function which returns 1 when method m emerges in the predicted methods set M_k, and 0 otherwise.

V. Experiment Design

A. Research Questions

To assess the effectiveness of PINGFL, we propose to answer the following research questions:

- **RQ1: How effective is PINGFL in method-level fault localization?** In this RQ, we compare PINGFL with a variety of other existing FL approaches.
- **RQ2: How do different components affect the performance of PINGFL?** This RQ helps measure the contribution of different components in PINGFL.
- **RQ3: What is the LLM-based agent's fault localization strategy?** This RQ investigates the tool usage strategies and behavior patterns of the LLM during the FL task.

B. Benchmark

We adopt the widely-used benchmark Defects4J [12] for evaluation. Defects4J consists of 835 real-world bugs from 17 real-world Java projects, including 395 bugs from 17 Java projects in Defects4J-V1.2 and extra 440 bug introduced in Defects4J-V2.0. To further eliminate irrelevant bug samples, we filter out bugs that cannot be localized within the method-level (i.e., bugs fixed by only adding methods, classes, field modifications, etc.), as these bugs are beyond the capabilities of PINGFL. **Overall, the final version of our benchmark contains 812 bugs from 16 projects.**

C. Baseline

We compare PINGFL with three state-of-the-art LLM-based FL techniques Agentless [13], SoapFL [8], and AutoFL [9]. We also include three traditional FL techniques Ochiai [2], DeepFL [3], and GRACE [4] as baselines. More details about the baseline approaches have been introduced in Section II.

D. Implementation

We use the *gpt-4o-2024-11-20* model from OpenAI [28] as the LLM backend for PINGFL and all of the other LLM-based FL baselines. Particularly, we use *gpt-4-turbo-2024-04-09* for the PD agent of PINGFL according to its stronger ability in code editing. We use tree-sitter [29] for code parsing and program element extraction. We set the *MAX_TOOL_CALLS* for the FL agent to 30 and the *MAX_EDIT_COUNT* for the PD agent to 5. The above settings are to avoid the situation where PINGFL cannot terminate the FL process. To alleviate the randomness of the LLM responses, we set the temperature of the LLM to 0, other inference parameters follow the default settings of ChatGPT API.

E. Metrics

Following prior studies [3], [4], [30], [31], we assess the effectiveness of PINGFL based on the Top-N metric. Top-N computes the number of bugs that have at least one buggy element localized within the first N positions in the ranked list. In this work, we choose $N = 1, 3, 5$ based on two concerns: (1) A previous study [5] has shown that about 74% developers inspect only the Top-5 results in the given list. (2) The LLM-based FL techniques typically pinpoint a few suspicious locations rather than provide a ranked list that contains all program elements. Take the max N as 5 is thus sufficient to represent the performance of LLMFL.

VI. Experimental Results

A. RQ1: Fault Localization Performance

Compared to LLM-based FL approaches. We compare PINGFL with three state-of-the-art LLM-based FL techniques SoapFL [8], AutoFL [9], and Agentless [13], the results are shown in Table II. Overall, PINGFL achieves the best FL performance, it localizes 64.4% (523 out of 812) bugs within Top-5, which is improved by 10.3%, 30.4%, and 65.6% compared with SoapFL, AutoFL, and Agentless, respectively. Regarding the Top-1 metric, PINGFL exhibits a more significant performance improvement, outperforming the other three techniques by 41.1%, 47.1%, and 121.7%, respectively. This suggests that PINGFL is not only good at identifying more faulty locations but can also more accurately pinpoint the buggy methods from multiple suspicious candidates. For the results on individual projects, we observe that PINGFL consistently outperforms the other techniques across nearly all projects. Notably, we find that PINGFL makes significant progress on the Closure project, where it successfully localizes 80 out of 174 bugs within Top-1, which is 36 more than SoapFL, 37 bugs more than AutoFL, and 55 more than Agentless.

We also investigated the overlaps among the bugs localized within Top-5 by different LLMFL approaches, the results are shown in Figure 6. Overall, PINGFL can localize the most unique bugs within Top-5 (77 bugs), followed by SoapFL (40 bugs), Agentless (15 bugs), and AutoFL (5 bugs). Interestingly, we observe a complementary phenomenon in the results of the approaches based on fixed LLM workflow (i.e., Agentless and SoapFL) and flexible LLM tool use (i.e., AutoFL and

TABLE II: Comparision between PINGFL and other LLM-based FL approaches on Defects4J.

Project	# All	PINGFL			SoapFL			AutoFL			Agentless		
		Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Chart	25	16	17	17	17	20	20	13	16	16	10	15	16
Closure	174	80	100	102	44	74	86	43	57	57	25	39	39
Lang	64	47	51	52	48	57	58	44	50	51	34	40	48
Math	106	69	79	79	51	76	78	53	69	69	45	71	72
Mockito	38	23	26	27	19	22	24	16	22	22	21	26	26
Time	26	16	16	18	12	14	14	12	15	16	8	10	12
Cli	39	25	27	27	10	19	21	18	21	21	6	10	11
Codec	18	8	9	9	8	11	11	5	8	8	4	5	7
Collections	4	1	1	1	0	0	0	0	1	1	0	0	0
Compress	47	27	30	30	20	28	30	19	25	27	13	21	22
Csv	16	13	14	14	7	10	10	11	12	12	5	8	8
Gson	18	11	11	11	8	8	10	8	9	9	3	5	6
JacksonCore	26	18	21	21	11	18	20	12	16	17	8	10	10
JacksonDatabind	112	33	47	47	24	37	37	15	22	25	3	6	7
JacksonXml	6	3	3	3	1	1	1	2	2	2	1	1	3
Jsoup	93	60	65	65	39	53	54	35	48	48	17	26	29
Total	**812**	**450**	**517**	**523**	**319**	**448**	**474**	**306**	**393**	**401**	**203**	**293**	**316**

TABLE III: Comparision between PINGFL and other Traditional FL approaches on Defects4J.

Project	# All	PINGFL			GRACE			DeepFL			Ochiai		
		Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Chart	25	16	17	17	14	20	22	12	18	21	6	14	15
Closure	174	80	100	102	51	78	102	46	61	92	20	39	70
Lang	64	47	51	52	43	53	57	42	53	55	25	45	51
Math	106	69	79	79	64	79	92	52	81	90	23	52	62
Mockito	38	23	26	27	16	24	26	10	18	23	7	14	18
Time	26	16	16	18	11	16	20	12	15	18	6	12	13
Cli	39	25	27	27	14	24	26	11	21	24	3	5	10
Codec	18	8	9	9	6	11	13	5	10	12	3	12	17
Collections	4	1	1	1	1	1	2	1	1	2	1	1	2
Compress	47	27	30	30	23	29	34	22	27	31	5	12	17
Csv	16	13	14	14	6	8	10	7	8	9	3	8	10
Gson	18	11	11	11	11	13	14	8	11	12	4	9	9
JacksonCore	26	18	21	21	9	13	14	5	5	9	6	11	13
JacksonXml	6	3	3	3	3	3	4	3	3	4	0	0	0
Jsoup	93	60	65	65	40	64	72	33	39	46	15	40	48
Total	**700**	**417**	**470**	**476**	**312**	**436**	**508**	**269**	**371**	**448**	**127**	**274**	**355**

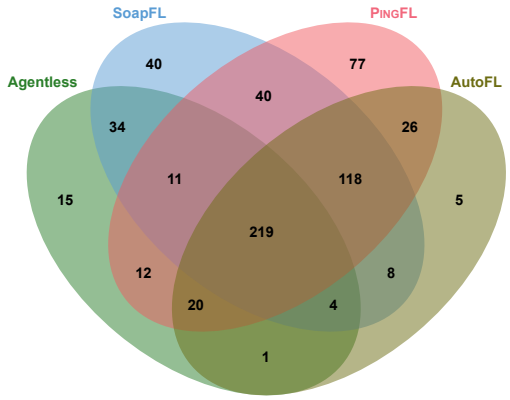


Fig. 6: Overlap results on Defects4J.

PINGFL). There are 108 bugs that can be found by PINGFL or AutoFL but not by SoapFL and Agentless, this number is 89 reversely. To understand this, we manually analyzed the logs of different types of LLM-based FL approaches. We observed that the differences in LLM autonomy may be the main reason for this phenomenon. In a fixed LLM workflow such as SoapFL, the LLM is typically provided with more structured information (e.g., the code skeleton of a file) and required to gradually find the fault program elements from global to local. However, for an FL approach based on LLM tool use capability, the LLM has more complex FL behavior since it can autonomously decide the information or source code it wants to retrieve. For example, after reviewing the source code of a method, the LLM may request to continue examining a callee method within that method. The different behavior of the LLM within these two types of LLM-based FL approaches can lead to diverse LLM exploration paths, thus resulting in complementarity in FL results.

Compared to traditional FL approaches. To investigate how PINGFL performs against the traditional FL approaches, we select three representative FL techniques based on different methodologies: DeepFL [3] (multi-layer perception), GRACE [4] (graph neural network), and Ochiai [2] (program spectrum). Note that we excluded 112 bugs from the project *JacksonDatabind* due to some technical issues in reproducing GRACE and DeepFL, the total number of bugs of the benchmark we adopted for this comparison is 700. As shown in Table III, PINGFL outperforms all of the traditional FL approaches in terms of Top-1 and Top-3 metrics. PINGFL

TABLE IV: Cross project results on Defects4J-V2.0.

Project	# Bugs	Techniques	Top1	Top3	Top5
Overall	309	Ochiai	41	100	128
		DeepFL$_{cov}$	59	122	153
		GRACE	116	163	191
		PINGFL	**185**	**205**	**206**

localizes 417 of 700 bugs within Top-1, 105 more than GRACE, 148 more than DeepFL, and 290 more than Ochiai. For the Top-3 value, it localizes 470 bugs, which is 34 more than GRACE, 99 more than DeepFL, and 196 more than Ochiai. However, when focusing on the results of Top-5 value, we find that GRACE has localized 32 more bugs than PINGFL. We attribute this phenomenon to the significant methodological difference between PINGFL and learning-based FL techniques: The input for GRACE includes all covered methods, it has a broader scope that allows it to simultaneously observe all the code within the software repository; Conversely, PINGFL explores potential fault locations within the software repository through code navigation, thus possessing a relatively narrower scope. Consequently, we can observe that the increase in the number of bugs localized by PINGFL becomes progressively smaller from Top-1 to Top-3 (417 to 470) and then from Top-3 to Top-5 (470 to 476). The similar trends can also be found in the results of AutoFL and Agentless in Table II.

To compare the generalization of PINGFL with traditional FL techniques, we have performed a cross-project evaluation (i.e., we train the LBFL approaches GRACE and DeepFL on 391 bugs from Defects4J-V1.2.0 and test them on 309 bugs from Defects4J-V2.0.0). The result in Table IV shows that PINGFL significantly outperforms the other approaches by localizing 69 more bugs within Top-1 than GRACE, 126 more than DeepFL, and 141 more than Ochiai. On the Top-5 metric, the PINGFL also consistently performs better than other traditional FL approaches. This indicates that while LBFLs are highly dependent on the training data from a specific project, the LLMFL approaches can exhibit better performance across different projects.

Answer to RQ1: PINGFL can effectively localize the bugs in Defects4J, which outperforms all of the other LLM-based approaches and performs better than learning-based techniques under the cross-project scenario.

B. RQ2: Ablation Study

To investigate the contributions of different components in PINGFL, we obtained 5 variants by separately removing different components from PINGFL:

- *w/o Test Code:* The *TEST_CODE* in the prompt for PD agent is removed (refer to Figure 5);
- *w/o Test Output:* We remove the test output within *ERROR_MESSAGE* in the prompt for PD agent;
- *w/o Stack Trace:* We remove the stack trace within *ERROR_MESSAGE* in the prompt for PD agent;
- *w/o Suspected Issue:* The *SUSPECTED_ISSUE* in the prompt for PD agent is no longer available, which blocks the communication between the PD agent and the FL agent;

TABLE V: Ablation study result.

Project	# Bugs	Variant	Top-1		Top-3		Top-5	
Overall	812	w/o Test Code	417	↓33	479	↓38	490	↓33
		w/o Test Output	417	↓33	476	↓41	489	↓34
		w/o Stack Trace	417	↓33	478	↓39	490	↓33
		w/o Suspected Issue	417	↓33	486	↓31	494	↓29
		w/o Thought	413	↓37	462	↓55	470	↓53
		w/o PD Agent	341	↓109	439	↓78	463	↓60
		PINGFL	**450**		**517**		**523**	

- *w/o Thought:* We eliminate our design that enables the FL agent to perform an iterative Thought-Action process in Section IV-C;
- *w/o PD Agent:* We remove the PD agent, which will cause PingFL to completely lose the ability to perform print debugging.

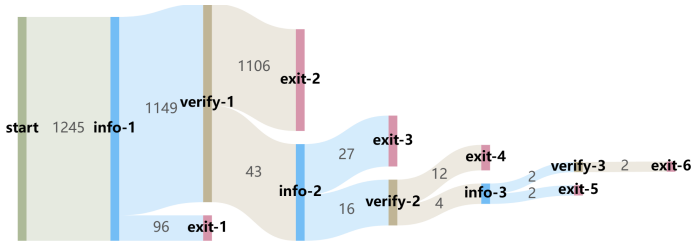
The performances of the variants are shown in Table V. We observed consistent performance degradation across all variants for every metric. *w/o PD Agent* performed the worst, identifying 109, 78, and 60 fewer bugs than PINGFL in the Top-1, Top-3, and Top-5 metrics, respectively. This result strongly confirms that the print debugging mechanism proposed in this paper can effectively improve the performance of the LLM-based FL approach. *w/o Thought* also shows a relatively obvious effect by identifying 37, 55, and 53 fewer bugs than PINGFL in the Top-1, Top-3, and Top-5 metrics, respectively. Among all variants, *w/o Suspected Issue* encounters a minor performance loss, yet still causes a reduction of 33, 31, and 29 in Top-1, Top-3, and Top-5.

Specifically, for *w/o Test Code*, *w/o Test Output*, and *w/o Stack Trace*, we observed that although the effectiveness degradation of these three variants varied across different projects, their overall performance loss remained highly consistent. This indicates that all three types of fault information (i.e., test code, test output, and stack trace) play a crucial role in the print debugging process. For *w/o Suspected Issue*, we find that the suspected issue can effectively convey the verification intent of the FL agent to the PD agent, thereby facilitating a more purposeful print debugging process. Specifically, after incorporating the suspected issue, PINGFL can localize 33 and 29 more bugs within Top-1 and Top-5, respectively. For *w/o Thought*, the result confirms that it is a finer design to ask the LLM to explicitly analyze and think about the fault localization process before initiating each tool use. This pattern is more in line with the debugging behavior of developers and the design principle of the ReAct [11] agent.

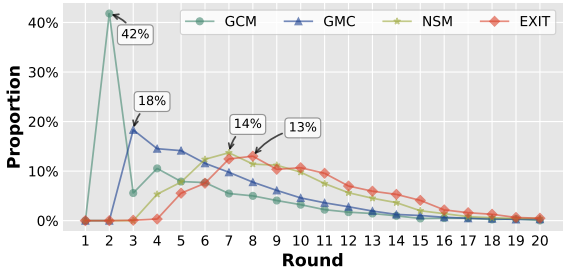
Answer to RQ2: All components in PINGFL contribute positively to the FL performance. Among them, the design of the PD agent and the thinking process of the FL agent have a relatively more significant impact.

C. RQ3: Fault Localization Strategy

PINGFL is allowed to autonomously explore the code repository or perform print debugging similar to human developers. Consequently, an interesting question emerges: *Does the LLM exhibit specific behavior patterns or strategies during the FL process?* In this RQ, we dissected the behavior of PINGFL from two dimensions: state transitions and tool usage.



(a) Sankey diagram of state transition. The name for each state is in the format of "[state]-[round]", e.g., "verify-2" means the LLM is at the second round of verify state. The amount of flow is marked on the link.



(b) The distribution of tool calls on each round. GCM: get_covered_method_ids_for_class; GMC: get_method_code; NSM: nominate_suspicious_method; EXIT: exit_debugging.

Fig. 7: State transition and tool usage during the fault localization process.

1) State Transition. The state transition of the fault localization agent is illustrated in Figure 7a. Since a single bug may involve multiple failing tests, the total amount of flow (1245) is larger than the number of bugs (812). Interestingly, we observed a specified state transition pattern of the FL agent in most cases: (start) → (info-1) → (verify-1) → (exit-2). This implies that the LLM tends to possess its own FL strategies, which involve initially navigating the codebase to gather information, followed by nominating and verifying suspicious program locations. Moreover, we found that in a small number of cases, the FL agent would autonomously decide to "simplify" or "complicate" its FL strategy. Specifically, for 96 cases, the agent exhibits high confidence in identifying the fault location during the information gathering episode, and thus chose to skip the verification state with a simpler strategy: (start) → (info-1) → (exit-1). Conversely, for 43 cases, the agent experiences up to three rounds of info and verify states before terminating the FL process. When employing a simplified strategy, PINGFL successfully localized 61.5% of bugs (59 out of 96) within Top 5. For the complicated strategy, it still maintained relatively high performance by localizing 48.8% of bugs (21 out of 43). This phenomenon reveals the advantage of LLMs over other FL techniques based on a fixed LLM workflow, i.e., the LLM is allowed to adaptively determine the length of the reasoning process according to the difficulty of the fault localization task.

2) Tool Usage. The distribution of the tool calls on each round is shown in Figure 7b. Note that we omit the *get_covered_class* tool since it is always called in the first round. In most cases, PINGFL can complete the FL process within 20 rounds of tool calls, with the highest proportion (13%) of ending in round 8. We observed a significant difference in the distribution of the various tools. Specifically, the *get_covered_method_ids_for_class* (GCM) and *get_method_code* (GMC) tools are more likely to be invoked in the early stages of the FL process. GCM has the highest proportion (42%) of being invoked in the second round, while GMC peaks in the third round with a proportion of 18%. In contrast, the *nominate_suspicious_method* (NSM) and *exit_debugging* (EXIT) tools tend to be called in the later stages of the FL process. Notably, when we shift the distribution of NSM to the right by one round, we discover that it almost overlaps with the distribution of EXIT. This suggests that the LLM typically invokes NSM just before terminating the FL process. Interestingly, we notice an exceptional increase in the proportion of GCM from rounds 3 to 4. This phenomenon indicates an inherent strategy of the LLM agent during the information gathering state: the LLM prefers to first use GCM to identify the methods covered within a certain class and then subsequently invoke GMC to examine the specific code of those methods. This behavior is somewhat analogous to a depth-first search algorithm.

Answer to RQ3: PINGFL generally employs the FL strategy that first collects information and then verifies the suspicious program locations. In a minority of cases, it demonstrates the potential to adapt its strategy based on the complexity of the FL task.

VII. DISCUSSION

A. How PD Agent Works

The process by which PINGFL localizes different bugs varies, making it difficult to describe the PD agent's role with a single, uniform pattern. In this discussion, we try to intuitively demonstrate how the PD agent leads to better fault localization performance.

We manually examined the PD agent's print debugging behavior on 50 random bugs and found that it inserted an average of 4.47 log statements per bug. This indicates that the PD agent typically does not simply print all program states, but purposefully focuses on key variable values. Next, by identifying keywords including "confirm", "indeed", or "evident" in the LLM's response, we found that the FL agent can benefit from these program logs in a majority of cases (36 out of 50), which to some extent validates the usefulness of the dynamic information summarized by the PD agent.

To further illustrate how the PD agent contributes to the FL task, we describe a case where PINGFL attempts to find bug *JacksonCore-15*. As shown in Figure 8, the LLM's initial analysis suggests that a program element involving "filtering logic during JSON parsing" might be problematic. The FL agent then browses the codebase to examine several

Fig. 8: A case to show how the PD agent works.

suspicious methods like `nextToken` and `_nextToken`. However, without supporting evidence from dynamic information, PINGFL is still unable to confirm which suspicious method is the true fault location. Subsequently, the FL agent utilizes the PD agent to perform print debugging on method `nextToken`, confirming that it is indeed the root cause. This determination helps the FL agent to place the authentic faulty method `nextToken` at a higher position, ultimately improving the localization accuracy. Similarly, in another case *Closure-14* mentioned in Section III, the PD agent verifies that method `tryFoldAdd` is not the root cause. This helps eliminate an incorrect location, thereby also improving the localization performance.

B. Cost Analysis

We evaluate the economic and time overhead of the FL agent and PD agent in PINGFL, respectively. The results are illustrated in Figure 9. Regarding the overall cost of PINGFL, we find it requires an average of only $0.23 and 104.62 seconds to process each bug. In 75% of cases, the cost does not exceed $0.60 and 200 seconds. Furthermore, the PD agent spends an average of merely $0.07 for print debugging each bug, demonstrating the high cost-effectiveness of PINGFL. In terms of execution time, the FL agent takes an average of 56.48 seconds to localize a bug, with more than 75% of cases finishing within 100 seconds. Since the PD agent frequently modifies and executes suspicious methods, its average time

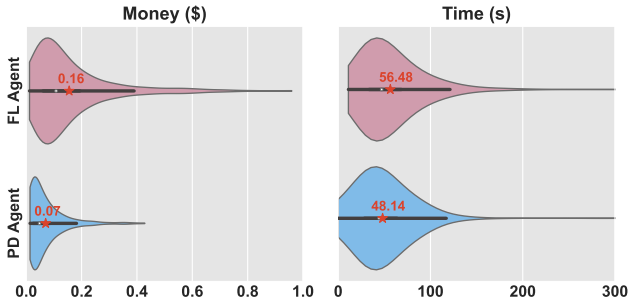


Fig. 9: Overhead analysis for PINGFL. FL Agent: fault localization agent; PD Agent: print debugging agent.

cost per bug is 48.14 seconds, which is close to that of the FL agent.

C. Threats to Validity

Internal. The main internal threat comes from the data leakage problem. Since the knowledge of the *gpt-4o-2024-11-20* model is up to Oct 2023 while the Defects4J benchmark was released in Feb 2018, the bug information may have been included in the training corpus. To mitigate this threat, we performed a fair comparison between PINGFL and other state-of-the-art LLM-based FL techniques using the same LLM. Additionally, we also conducted an ablation study to confirm the usefulness of each component within PINGFL.

External. The main external threat is that the results obtained by PINGFL may not generalize to other software systems. However, the benchmark we chose alleviates this threat to some extent. The Defects4J benchmark we utilized comprises 812 bugs from 16 projects, which is a good representation of the defective software population.

VIII. Conclusion

In this paper, we introduce PINGFL, an LLM agent system that incorporates program dynamic information for more accurate fault localization. Inspired by the print debugging technique of human developers, PINGFL consists of two collaborative LLM agents: A fault localization agent equipped with several tools to pinpoint the root cause, and a print debugging agent that aims to verify the suspicious location nominated by the FL agent. The evaluation results on the Defects4J benchmark show that PINGFL outperforms other LLM-based approaches and surpasses traditional FL techniques in the cross-project scenario. Besides, we discovered that the LLM can exhibit specific strategies and tool usage patterns for solving the fault localization task. The cost analysis shows that PINGFL is cost-effective, it takes only an average of 0.23 dollars and 104.62 seconds for localizing a fault.

Acknowledgements

This work was supported by the National Natural Science Foundation of China (Grant No.62402506, No.62474196) as well as the Research Foundation from NUDT (Grant No. ZK24-05).

REFERENCES

[1] A. Alaboudi and T. D. LaToza, "An exploratory study of debugging episodes," arXiv:2105.02162 [cs.SE], 2021.

[2] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, 2006, pp. 39–46.

[3] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[4] Y. Lou, Q. Zhu, J. Dong, X. Li, Z. Sun, D. Hao, L. Zhang, and L. Zhang, "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*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 664–676.

[5] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 165–176.

[6] "Github copilot." https://github.com/features/copilot, 2025.

[7] "Cursor: The ai code editor." https://www.cursor.com/, 2025.

[8] Y. Qin, S. Wang, Y. Lou, J. Dong, K. Wang, X. Li, and X. Mao, "Soapfl: A standard operating procedure for llm-based method-level fault localization," *IEEE Transactions on Software Engineering*, vol. 51, no. 4, pp. 1173–1187, 2025.

[9] S. Kang, G. An, and S. Yoo, "A quantitative and qualitative evaluation of llm-based explainable fault localization," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.

[10] M. Beller, N. Spruit, D. Spinellis, and A. Zaidman, "On the dichotomy of debugging behavior among programmers," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 572–583.

[11] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," in *The Eleventh International Conference on Learning Representations*, 2023.

[12] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[13] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Agentless: Demystifying llm-based software engineering agents," arXiv:2407.01489 [cs.SE], 2024.

[14] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[15] M. Tomáš, K. Martin, B. Lukáš, C. Jan, and K. Sanjeev, "Recurrent neural network based language model," *Interspeech 2010*, p. 1045, 09 2010.

[16] M.-C. Popescu, V. E. Balas, L. Perescu-Popescu, and N. Mastorakis, "Multilayer perceptron and neural networks," *WSEAS Trans. Cir. and Sys.*, vol. 8, no. 7, p. 579–588, jul 2009.

[17] Y. Li, R. Zemel, M. Brockschmidt, and D. Tarlow, "Gated graph sequence neural networks," in *Proceedings of ICLR'16*, April 2016.

[18] Z. Zhang, Y. Lei, X. Mao, and P. Li, "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, 2019, pp. 445–455.

[19] X. Li and L. Zhang, "Transforming programs and tests in tandem for fault localization," *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, oct 2017.

[20] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 332–347, 2021.

[21] Q. Feng, X. Ma, J. Sheng, Z. Feng, W. Song, and P. Liang, "Integrating various software artifacts for better llm-based bug localization and program repair," arXiv:2412.03905 [cs.SE], 2025.

[22] C. Xu, Z. Liu, X. Ren, G. Zhang, M. Liang, and D. Lo, "Flexfl: Flexible and effective fault localization with open-source large language models," *IEEE Transactions on Software Engineering*, pp. 1–17, 2025.

[23] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 1592–1604.

[24] K. Levin, N. van Kempen, E. D. Berger, and S. N. Freund, "Chatdbg: An ai-powered debugging assistant," *arXiv:2403.16354 [cs.SE]*, 2024.

[25] "Function calling." https://platform.openai.com/docs/guides/function-calling, 2025.

[26] Z. Shen, "Llm with tools: A survey," arXiv:2409.18807 [cs.AI], 2024.

[27] P. Gauthier, "Aider pair programming in your terminal," https://aider.chat/, 2025.

[28] OpenAI, "Openai," https://openai.com, 2024.

[29] treesitter, "tree-sitter," https://tree-sitter.github.io/tree-sitter, 2024.

[30] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 273–283.

[31] J. Qian, X. Ju, and X. Chen, "Gnet4fl: Effective fault localization via graph convolutional neural network," *Automated Software Engg.*, vol. 30, no. 2, apr 2023.