Measure Twice, Locate Once: Mitigating Hallucinations in LLM-based Agents for Repository-Scale Fault Localization

FEIYUE CHEN, School of Information Science and Technology, ShanghaiTech University, China
GUOWEI YANG, School of Information Science and Technology, ShanghaiTech University, China
CHERYL LEE, Department of Computer Science and Engineering, The Chinese University of Hong Kong, China
ZHANMING JIE, ByteDance, Singapore
YUQI CHEN*, School of Information Science and Technology, ShanghaiTech University, China

Fault Localization (FL) is a critical yet inherently complex phase in the software debugging process. Over the years, numerous automated FL techniques have been developed to alleviate the time and effort involved. More recently, the emergence of Large Language Models (LLMs) has marked a new era for FL. However, existing LLM-based approaches often arrive at premature conclusions due to both extrinsic and intrinsic hallucinations. To address these challenges, we propose FaultLens, a novel FL technique that equips an LLM-based agent with a fine-grained feedback mechanism for repository-scale fault localization. Specifically, the decision-making stage of our approach starts with identifying FL candidates through an LLM-based agent. Here, location extraction validation detects extrinsic hallucinations, triggering further investigation. A defined rule determines investigation completion, while a self-check mechanism mitigates intrinsic hallucinations arising from incomplete investigation. The advanced location identification stage further minimizes intrinsic hallucinations caused by faulty reasoning. We demonstrate the effectiveness of our approach through a comprehensive evaluation on the Defects4J benchmark. Our results show that FaultLens outperforms several fault localization techniques across multiple categories, including spectrum-based methods, mutation analysis, machine learning approaches, and LLM-based systems. Specifically, FaultLens achieves a 43.35% improvement over SoapFL and a 24.24% improvement over AutoFL in the Top-1 metric, surpassing state-of-the-art LLM-based agent methods. Additional experiments further indicate that FaultLens generalizes across different programming languages and LLM backends, and that its hallucination-mitigation mechanisms are transferable to an external localization workflow.

CCS Concepts: • **Software and its engineering** → **Software defect analysis**; **Software testing and debugging**; • **Computing methodologies** → *Natural language processing*;

Additional Key Words and Phrases: fault localization, large language models

*Yuqi Chen is the corresponding author.

Authors' Contact Information: Feiyue Chen, chenfy2023@shanghaitech.edu.cn, School of Information Science and Technology, ShanghaiTech University, Shanghai, China; Guowei Yang, yanggw2022@shanghaitech.edu.cn, School of Information Science and Technology, ShanghaiTech University, Shanghai, China; Cheryl Lee, cheryllee@link.cuhk.edu.hk, Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China; Zhanming Jie, allan.jie@bytedance.com, ByteDance, Singapore, Singapore; Yuqi Chen, chenyq@shanghaitech.edu.cn, School of Information Science and Technology, ShanghaiTech University, Shanghai, China.

ACM 1557-7392/2026/7-ART
https://doi.org/10.1145/3831589

1 Introduction

Fault Localization (FL), the process of pinpointing specific locations in code or a system where faults or errors cause incorrect or unexpected behavior, is a crucial and popular topic in software engineering. Recently, with the advent of Large Language Models (LLMs), FL techniques have evolved from traditional methods (e.g., spectrum-based FL [3, 19] and mutation-based FL [40, 43]) to LLM-based approaches. For instance, Yang et al. [64] propose a statement-level FL technique by fine-tuning bidirectional adapter layers on top of LLMs' learned representations. Additionally, Widyasari et al. [55] employ LLMs to generate step-by-step reasoning, achieving explainable FL. However, due to token limitations and the complex nature of multi-file and multi-module dependencies, this category of LLM-based approaches cannot be applied to address repository-scale FL. To address this issue, current research has shifted towards utilizing LLM-based agents. These approaches typically convert FL into a series of progressive tasks. As the agent progresses through each task, it adjusts its approach based on intermediate feedback, leveraging tools and LLM capabilities throughout the process to gather insights and refine its plan for achieving the desired outcomes. For example, Kang et al. [21] demonstrate how LLMs use tools to extract information from the codebase. These tools help LLMs extract class structures and method contents relevant to FL.

However, since the AI agent for fault localization typically narrows down the range of FL candidates by progressively collecting information from the codebase, an error in the early stages can drastically influence the results. Unfortunately, current research [38] demonstrates that LLMs suffer from hallucinations, which can be categorized into two types: extrinsic and intrinsic hallucinations. Extrinsic hallucinations refer to model generations that cannot be verified against the source content, while intrinsic hallucinations involve generations that contradict the source content [16]. In the context of repository-scale FL, we define extrinsic hallucinations as cases where the model identifies suspicious locations that do not actually exist in the codebase, typically due to prematurely ending its investigation process. Intrinsic hallucinations, on the other hand, arise when the model selects existing locations that are in fact unrelated to the observed failure. Both types of hallucinations may originate from a common underlying cause: incomplete investigation, in which the model fails to collect sufficient contextual information. In addition, intrinsic hallucinations can also result from faulty reasoning, where the model has already collected sufficient contextual information but fails to identify the actual locations responsible for the fault. Through our investigation, we observed that both types of hallucinations frequently occur during FL, significantly contributing to the low Top-1 accuracy of current approaches. This low Top-1 accuracy indicates that FL methods frequently fail to correctly identify the faulty method as the top suggestion in their prediction lists. Since FL is an upstream task for program repair, more precise fault identification can significantly enhance the efficiency of downstream tasks.

In response to these challenges, we explore various strategies to mitigate hallucinations, enhancing the accuracy of repository-scale FL tasks powered by LLM-based agents. To address extrinsic hallucinations, we implement a location extraction validation to verify whether the identified locations genuinely exist within the codebase, thereby detecting extrinsic hallucinations early. For intrinsic hallucinations, we design targeted components based on their specific causes. We require the agent to reevaluate its outputs to alleviate intrinsic hallucinations from incomplete investigation. For those arising from faulty reasoning, we divide the location identification process into two phases: preliminary location identification and advanced location identification. This approach allows the agent the opportunity to correct earlier reasoning errors. Together, these components mitigate hallucinations, thereby enhancing the accuracy of repository-scale FL, particularly in the Top-1 metric.

Overall, we propose FaultLens, a novel FL technique that employs an LLM-based agent to identify suspicious locations for repository-scale fault localization. The key insight is that a well-designed fine-grained feedback mechanism can effectively mitigate hallucinations. Building on this insight, we have designed a decision-making stage that incorporates fine-grained feedback to reduce hallucinations in FL tasks. Through our observations, we

find that the LLM-based agent occasionally identifies invalid locations that do not exist within the codebase. For example, it may flag a candidate method that cannot be located in the specified class. To address this issue, we design a location extraction validation mechanism to capture such extrinsic hallucinations. If the agent identifies non-existent locations, it is required to conduct deeper investigations using codebase exploration tools. We establish a specific rule to determine whether investigations are complete and use a self-check mechanism to address intrinsic hallucinations caused by incomplete investigation. In addition, faulty reasoning remains a challenge in FL tasks. Even when the agent has all the necessary information to pinpoint fault locations, flaws in its reasoning can still lead to incorrect conclusions. We adopt advanced location identification to mitigate such intrinsic hallucinations resulting from faulty reasoning.

We evaluate FaultLens on Defects4J version 1.2.0 [20], which comprises 395 real-world bugs from 6 open-source Java projects. FaultLens demonstrates outstanding overall performance in the Top-1 metric, surpassing various baselines, including Ochiai [3], Metallaxis [43], FLUCCS [50], AutoFL [21] and SoapFL [46]. Furthermore, we assess the transferability of our hallucination-mitigation mechanisms on an external repository-scale localization workflow, and evaluate the generalizability of FaultLens on a Python benchmark and with both proprietary and open-weight LLM backends.

Our contributions are summarized as follows:

- We design a decision-making stage with fine-grained feedback to mitigate both extrinsic and intrinsic hallucinations in LLMs for repository-scale FL tasks. This stage integrates location extraction validation, a self-check mechanism, and advanced location identification.
- We propose FaultLens, a novel LLM-based agent technique for repository-scale FL that enhances the accuracy of suspicious location identification.
- FaultLens outperforms various baselines on the Defects4J benchmark, achieving up to a 43.35% improvement in the Top-1 metric compared to LLM-based agent approaches. Additional experiments demonstrate the generalizability of FaultLens across different programming languages and model providers, as well as the transferability of the proposed hallucination-mitigation mechanisms to the fault-localization stage of AutoCodeRover, an external issue-driven program repair framework.

2 Background

2.1 Fault Localization

Fault Localization is a critical task in software debugging aimed at automatically identifying program elements that may be responsible for a program failure. Traditional FL techniques are represented by spectrum-based FL (SBFL). SBFL utilizes coverage information from test case executions to calculate suspicious scores of program elements. Most popular methods leverage both passing test cases and failing test cases to generate a ranked list of potential fault locations [58]. For each program element, several metrics are involved in calculating its suspiciousness in the context of SBFL, including the counts of failed and passed test cases that cover and do not cover the element. Various SBFL techniques leverage these metrics to assess and rank the suspiciousness of program elements [3, 4, 19]. In recent years, learning-based FL has been widely studied [26, 28, 34], aiming to leverage various fault-diagnosis features, including information from test cases traditionally used in SBFL for more precise FL.

With the advancement of LLMs, recent research has explored their application in FL [55, 60, 64]. In these approaches, LLMs are provided with code context or test case failure information to identify suspicious locations. However, these methods face limitations due to the restricted input length of LLMs, which can hinder their ability to process FL tasks in large-scale codebases effectively. Additionally, techniques like Chain-of-Thought (CoT) [54] may not effectively mitigate these issues, as a recent study indicates that CoT reasoning does not improve

tasks related to code comprehension [18]. This is a significant limitation, as code comprehension is crucial for LLMs to accurately identify root causes during FL tasks.

The use of LLM-based agents mitigates this issue, allowing the inclusion of repository-scale FL. These agents can invoke specialized tools to gather relevant information, allowing them to learn more detailed insights about the given issue. After conducting investigations, the agents identify the root cause of a failure and pinpoint the specific methods responsible. However, without proper guidance, agents may still struggle with incomplete investigation or faulty reasoning, leading to hallucinations and inaccurate results. Our work demonstrates how to guide the agent in conducting more in-depth investigations, and rectifying potential errors in early reasoning, overcoming hallucinations in LLMs to enhance accuracy for fault localization.

2.2 LLM-Based Agents

AI agents represent entities that display intelligent behavior and are characterized by attributes like autonomy, social ability, reactivity, and pro-activeness [59]. Recent developments in LLMs have given rise to a new class of AI agents known as LLM-based agents, in which LLMs serve as the core of the agent's brain [61]. Endowed with the ability to perceive and utilize external resources and tools, LLM-based agents surpass standalone LLMs in both versatility and expertise. LLM-based agents can operate as either a single agent or multiple agents working in collaboration, with human interaction seamlessly integrated into both configurations.

LLM-based agents generally consist of three key modules: perception, brain, and action [61]. The perception module is responsible for receiving and processing information from the environment, promoting the brain module's work. Acting as the core of agents, the brain handles crucial activities such as information processing and decision-making through its components of planning and memory. During the planning phase, agents decompose a complex task into smaller sub-tasks, formulating plans for each sub-task. The plans can be dynamically adjusted as the task goes on through reflection mechanisms [49, 65]. The memory component plays a pivotal role by storing historical observations, decisions, and actions, thereby enhancing agents' ability to generate effective plans and make informed decisions in the future [52]. The action module executes interactions with the environment based on the plans formulated by the brain. External tools significantly enhance the capabilities of the action module, enabling agents to perform more sophisticated tasks [45].

3 Motivating Example

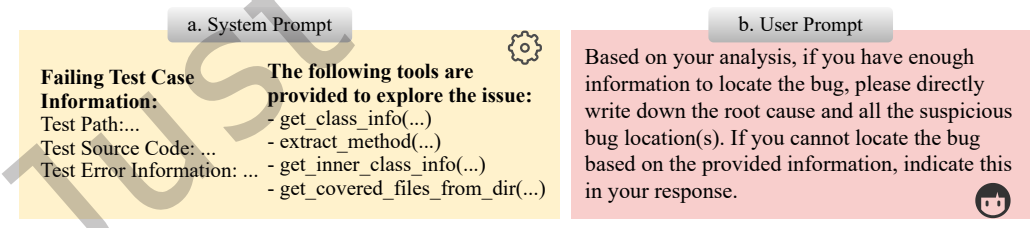

Fig. 1. Prompts used by the basic agent in the motivating example. (a) The core system prompt provides the failing-test information and the available codebase exploration tools. (b) The user prompt asks the agent to report the root cause and suspicious locations only when sufficient evidence has been collected. Otherwise, it should indicate that more investigation is needed.

To investigate the extent of hallucinations exhibited by LLMs in repository-scale FL tasks, we implement a basic agent capable of invoking codebase exploration tools, based on the GPT-4o-mini model. By randomly selecting 200 bugs from Defects4J dataset [20], we find that during the FL process for 32.5% (65/200) of the bugs, the agent

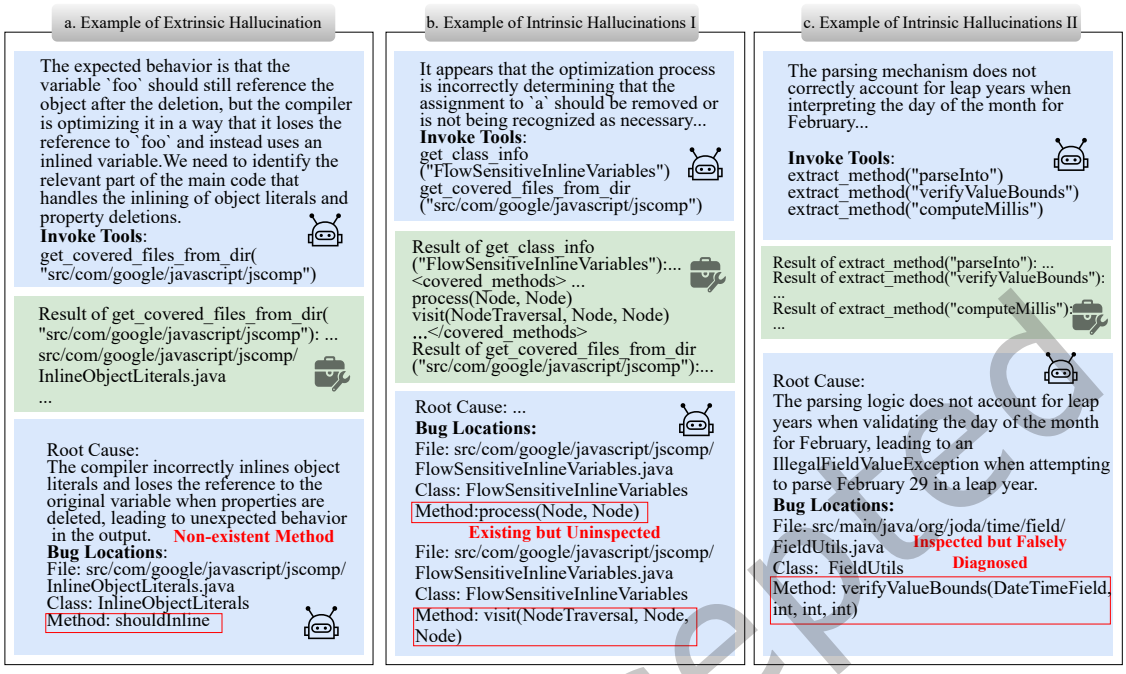

Fig. 2. Examples of hallucinations in LLMs during fault localization: (a) Extrinsic hallucinations in the fault localization, (b) Intrinsic hallucinations in the fault localization due to incomplete investigation, and (c) Intrinsic hallucinations in the fault localization due to faulty reasoning.

identifies at least one non-existent method, demonstrating that extrinsic hallucinations significantly impact FL accuracy. Moreover, we observe two distinct types of intrinsic hallucinations that further contribute to FL failures. To effectively illustrate extrinsic and intrinsic hallucinations in LLMs during repository-scale FL tasks, we use three real-world bugs from the Defects4J dataset to present motivating examples. Following prior LLM literature [16], we use extrinsic hallucination and intrinsic hallucination in a task-specific sense for repository-scale FL. Specifically, extrinsic hallucinations refer to predicted suspicious locations that do not actually exist in the codebase, whereas intrinsic hallucinations refer to predicted locations that exist but are unrelated to the observed failure. In the motivating examples below, incomplete investigation and faulty reasoning are treated as two underlying causes that may lead to such incorrect localization outputs.

To make the annotation of Figure 2 reproducible, we further operationalize the motivating examples at the predicted-location level. We first identify extrinsic hallucinations using an observable repository-scale criterion. A predicted location is labeled as an extrinsic hallucination if it cannot be resolved to an extractable method in the checked-out repository. For predicted locations that do exist in the repository but are still incorrect, we observed two recurring intrinsic-hallucination patterns in our motivating traces. In Figure 2, code inspection is defined as follows: a method is considered inspected only if its body has been exposed to the agent through `extract_method` or `get_inner_class_info` before the localization output. Intrinsic hallucination due to incomplete investigation occurs when the model identifies an existing but incorrect location without inspecting the body of the predicted method before the localization output. Intrinsic hallucination due to faulty reasoning occurs when the model

produces an existing but incorrect localization output, even though it has already inspected the body of at least one ground-truth faulty method before the localization output. These criteria are used to annotate the motivating examples in Figure 2 in our repository-scale fault-localization setting.

The core components of the system prompt, depicted in Figure 1(a), include details about a failing test case and a list of available tools. The failing test case information encompasses the file path, code, and error information. The agent is equipped with a set of tools to explore the codebase. The provided tools enable the agent to retrieve information about classes, inner classes, methods, and files from specific directories. After each tool invocation, the agent analyzes the output to determine whether sufficient information has been collected for FL, with the key elements of the user prompt shown in Figure 1(b). To reduce redundancy, these two prompts are omitted from Figure 2, which directly illustrates the agent invoking tools, processing the results, and identifying FL candidates.

According to the above criteria, Figure 2(a) is classified as an extrinsic hallucination. While the agent's analysis regarding a wrong inlining action by the compiler is plausible, after identifying the covered files from the relevant directory, it prematurely reports FL candidates without conducting further investigation. The predicted method `shouldInline` cannot be resolved to an extractable method in the checked-out repository, and is therefore labeled as an extrinsic hallucination.

According to the above criteria, Figure 2(b) and Figure 2(c) are classified as intrinsic hallucinations due to incomplete investigation and faulty reasoning, respectively. In Figure 2(b), the agent investigates the bug by utilizing tools to search for classes and files. It discovers the methods `process(Node, Node)` and `visit(NodeTraversal, Node, Node)` within the class `FlowSensitiveInlineVariables`. However, instead of inspecting the bodies of these predicted methods using `extract_method`, the LLM prematurely identifies them as suspicious locations. In reality, these two methods are not defective in this scenario and could have been excluded if inspected properly. Therefore, this case is classified as an intrinsic hallucination due to incomplete investigation. In Figure 2(c), the LLM invokes `extract_method` multiple times and inspects the defective method `parseInto(ReadWritableInstant, String, int)` along with two other methods before producing its localization output. However, it still incorrectly concludes that the method `verifyValueBounds(DateTimeField, int, int, int)` is responsible for the bug. Since the true faulty location has already been inspected before the incorrect prediction is made, this case is classified as an intrinsic hallucination due to faulty reasoning.

To complement the motivating traces in Figure 2, Figure 3 presents condensed correction traces for the same three bugs after applying FaultLens. Because the full agent trajectories and raw tool outputs are lengthy, we retain only the intermediate feedback that is directly involved in correcting the hallucinated prediction.

Figure 3(a) shows how FaultLens corrects the extrinsic hallucination in Figure 2(a). The initial hallucinated output predicts `InlineObjectLiterals.shouldInline`, which cannot be resolved to an extractable method in the checked-out repository. FaultLens first captures this error through location extraction validation and returns the feedback. The agent is then required to continue the investigation. In the condensed trace, the follow-up investigation queries class-level information for `InliningBehavior` in `InlineObjectLiterals.java` and `ReferenceCollectingCallback` in `ReferenceCollectingCallback.java`. Based on the newly collected information, the agent proposes two extractable candidate methods, `process(Node, Node)` in ReferenceCollectingCallback and `isInlinableObject(List<Reference>)` in InliningBehavior. Since these candidates are identified from class-level exploration rather than prior method-body inspection, FaultLens invokes conditional self-check. The self-check rejects `ReferenceCollectingCallback.process(Node, Node)` and retains `InliningBehavior.isInlinableObject(List<Reference>)`. The blue box marks this ground-truth faulty method, which remains in the final corrected output.

Figure 3(b) shows how FaultLens corrects the intrinsic hallucination caused by incomplete investigation in Figure 2(b). In the original trace, the agent outputs existing but incorrect suspicious methods before inspecting the relevant method bodies, so the error is not one of repository existence but of insufficient investigation. FaultLens therefore applies conditional self-check to the previously proposed but uninspected candidates and

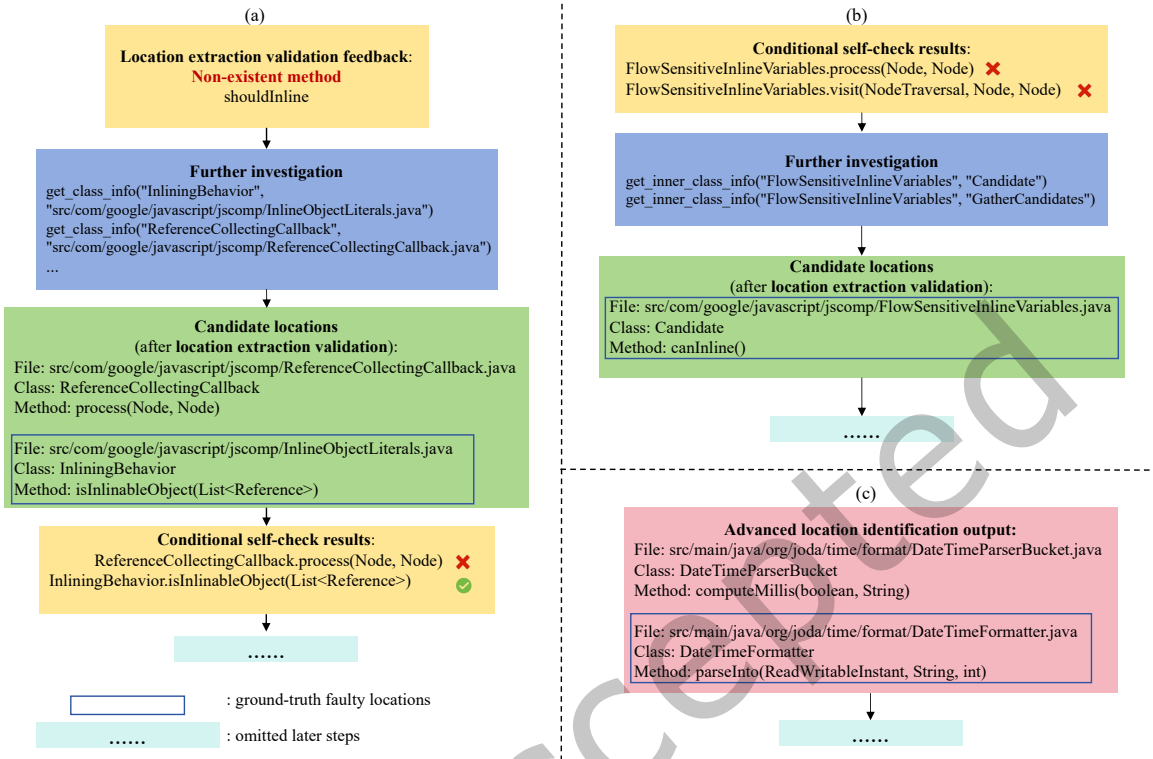

Fig. 3. FaultLens corrects the three motivating hallucinations shown in Figure 2: (a) extrinsic hallucination, (b) intrinsic hallucination due to incomplete investigation, and (c) intrinsic hallucination due to faulty reasoning. For readability, we omit steps that are not directly involved in correcting the hallucinated prediction. Blue-boxed methods denote ground-truth faulty methods that are retained in the final corrected output.

rules them out. The agent is then sent back for further investigation. In the condensed trace, the follow-up calls to `get_inner_class_info("FlowSensitiveInlineVariables", "Candidate")` and `get_inner_class_info("FlowSensitiveInlineVariables", "GatherCandidates")` expose inner-class methods together with their code content. With this additional method-level evidence, the subsequent localization output identifies `Candidate.canInline()` as a candidate that passes location extraction validation. The blue box marks `Candidate.canInline()` as the ground-truth faulty method, and this method is retained in the final corrected output.

Figure 3(c) shows how FaultLens corrects the intrinsic hallucination caused by faulty reasoning in Figure 2(c). In the original motivating trace, the agent had already inspected the true faulty method `parseInto(ReadWritableInstant, String, int)` but still incorrectly blamed `verifyValueBounds(DateTimeField, int, int, int)`. This case reflects faulty reasoning rather than missing code evidence. In FaultLens, this type of error is addressed by advanced location identification, which revisits the earlier investigation context after validation and self-check. In the condensed trace, advanced location identification produces `DateTimeParserBucket.computeMillis(`

`boolean, String)` and `DateTimeFormatter.parseInto(ReadWritableInstant, String, int)` as candidate methods. Among them, the blue-boxed `parseInto(ReadWritableInstant, String, int)` is the ground-truth faulty method.

Taken together, Figures 2 and 3 show both how the three motivating hallucinations are operationally classified and how FaultLens corrects them in practice: by rejecting non-existent predicted methods through location extraction validation, reducing uninspected false positives through conditional self-check, and recovering previously overlooked true faulty methods through advanced location identification.

4 Methodology

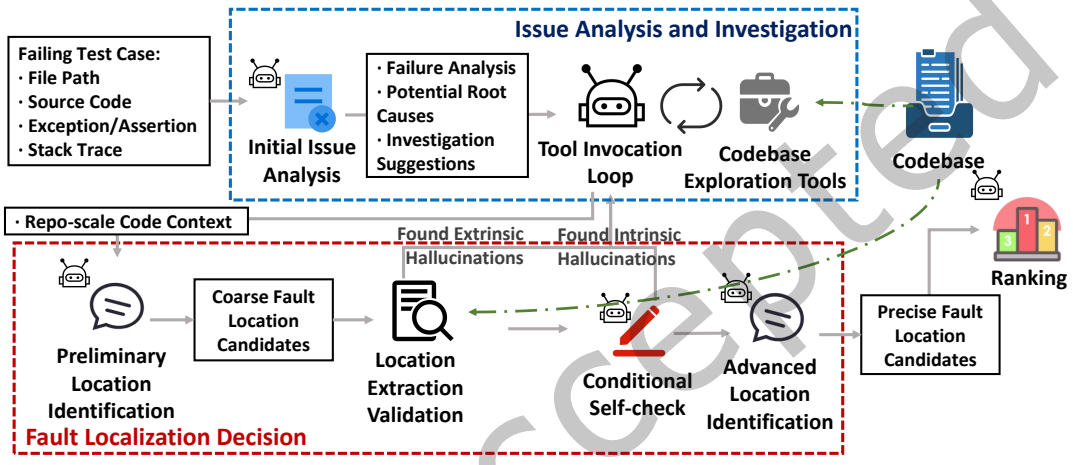

Fig. 4. Fault localization pipeline of FaultLens. After initial issue analysis and iterative tool-based investigation, FaultLens performs preliminary location identification, validates extracted locations to catch extrinsic hallucinations, applies conditional self-check to re-inspect unexamined candidates, conducts advanced location identification to mitigate faulty reasoning, and finally ranks the remaining candidates.

In this section, we introduce our proposed approach, FaultLens (as shown in Figure 4), which aims to precisely localize faults while mitigating hallucination effects in LLM-based fault localization. In repository-scale fault localization, the agent often needs to collect information across multiple files and classes before identifying suspicious locations. In each run, FaultLens takes exactly one selected failing test case as input and performs fault localization using an LLM-based agent configured to act as a software maintenance engineer. We adopt this single-test setting because fault investigation is often triggered by an individual observed failing test, and it may be non-trivial to determine a priori whether multiple failing tests stem from the same underlying root cause. Each run consists of two stages: (1) **Issue Analysis and Investigation**: upon receiving the failing test case information, the agent begins by analyzing the bug to identify initial investigation points and then uses codebase exploration tools to conduct a detailed examination. (2) **Fault Localization Decision**: the agent performs preliminary location identification and provides root cause analysis. It employs *location extraction validation* to mitigate extrinsic hallucinations, while the *conditional self-check mechanism* and *advanced location identification* are designed to address intrinsic hallucinations. To improve robustness against the stochasticity of LLM outputs, FaultLens can be executed multiple times and aggregate the ranked results across runs, as described in Section 4.3.

Algorithm 1 End-to-end workflow of FaultLens for a single run

Require: failing test case information x, codebase $\mathcal{R}$, coverage information $\text{cov}(t_f)$, tool set $\mathcal{T}$, maximum investigation rounds N_{iter}
Ensure: root-cause description rc, ranked fault-location candidates $\mathcal{C}_{\text{rank}}$

1: $\mathcal{A} \leftarrow$ InitAgent()
2: $ctx \leftarrow [\,]$; $\mathcal{M}_{chk} \leftarrow \emptyset$; $iter \leftarrow 0$
3: $rc \leftarrow \epsilon$; $\mathcal{C} \leftarrow [\,]$; $\mathcal{C}_{\text{val}} \leftarrow [\,]$; $\mathcal{C}_{\text{rank}} \leftarrow [\,]$
4: $h_0 \leftarrow \mathcal{A}$.InitialIssueAnalysis$(x)$
5: $ctx \leftarrow ctx \,\|\, (x, h_0)$
6: **while true do**
7: **while** $iter < N_{iter}$ **do**
8: $iter \leftarrow iter + 1$
9: $\mathcal{K} \leftarrow \mathcal{A}$.SelectToolCalls$(ctx, \mathcal{T})$
10: **for all** $a \in \mathcal{K}$ **do**
11: $r_a \leftarrow$ Execute$(a; \mathcal{R}, \text{cov}(t_f))$
12: $ctx \leftarrow ctx \,\|\, (a, r_a)$
13: **if** $a.name \in \{\texttt{extract\_method}, \texttt{get\_inner\_class\_info}\}$ **then**
14: $\mathcal{M}_{chk} \leftarrow \mathcal{M}_{chk} \cup$ MethodsFrom(r_a)
15: **end if**
16: **end for**
17: $ctx \leftarrow \mathcal{A}$.AnalyzeToolResult$(ctx)$
18: **if** $\mathcal{A}$.StopInvestigation(ctx) **then**
19: **break**
20: **end if**
21: **end while**
22: $(rc, \mathcal{C}, ctx) \leftarrow \mathcal{A}$.PreliminaryIdentify$(ctx)$
23: $(ctx, \mathcal{C}_{\text{val}}, \mathcal{E}) \leftarrow$ ValidateLocations$(\mathcal{C}, \mathcal{R}, ctx)$
24: **if** $\mathcal{E} \neq [\,]$ **and** $iter < N_{iter}$ **then**
25: **continue**
26: **end if**
27: $(ctx, \mathcal{C}_{\text{val}}) \leftarrow \mathcal{A}$.ConditionalSelfCheck$(\mathcal{C}_{\text{val}}, \mathcal{M}_{chk}, ctx)$
28: **if** $\mathcal{C}_{\text{val}} = [\,]$ **and** $iter < N_{iter}$ **then**
29: **continue**
30: **end if**
31: $(ctx, \mathcal{C}_{\text{val}}) \leftarrow \mathcal{A}$.AdvancedIdentify$(\mathcal{C}_{\text{val}}, ctx)$
32: $\mathcal{C}_{\text{rank}} \leftarrow \mathcal{A}$.RankBySuspiciousness$(\mathcal{C}_{\text{val}}, ctx)$
33: **break**
34: **end while**
35: **return** $(rc, \mathcal{C}_{\text{rank}})$

Algorithm 1 summarizes the end-to-end control flow of FaultLens for a single run. Given a failing test case, the agent first performs InitialIssueAnalysis to build an initial understanding of the failure (Line 4), and appends both the input test information and the analysis result to the running context ctx (Line 5). Throughout the workflow, ctx stores the interaction history available to the agent, including the failing-test information,

tool calls, tool outputs, intermediate analyses, and subsequent decision-stage feedback. The framework also maintains the set of inspected methods $\mathcal{M}_{chk}$, the investigation-round counter *iter*, the preliminary candidate list $\mathcal{C}$, the validated candidate list $\mathcal{C}_{\text{val}}$, the final ranked list $\mathcal{C}_{\text{rank}}$, and the root-cause description *rc* (Lines 2–3). The inner loop of the algorithm corresponds to codebase investigation (Lines 7–21). In each round, the agent selects one or more exploration tools according to the current context (Line 9), executes them on the codebase, appends the returned evidence to *ctx* (Lines 10–12), and analyzes the accumulated evidence to guide subsequent actions (Line 17). For tool calls such as `extract_method` and `get_inner_class_info`, which expose method contents, FaultLens additionally updates $\mathcal{M}_{chk}$ through MethodsFrom(r_a) (Lines 13–14). The investigation loop terminates when the agent judges that the collected evidence is sufficient for preliminary localization (Lines 18–19), or when the maximum round budget N_{iter} is reached (Line 7).

After the investigation loop, the agent performs PreliminaryIdentify to produce a root-cause description *rc* and a preliminary candidate list $\mathcal{C}$ (Line 22). These candidates are then passed to ValidateLocations, which returns a normalized list $\mathcal{C}_{\text{val}}$ together with a feedback list $\mathcal{E}$ (Line 23). Here, $\mathcal{E}$ records the candidates that fail extraction validation, such as cases where the predicted file, class, or method cannot be matched to the codebase. If $\mathcal{E} \neq [\,]$ and the investigation budget has not been exhausted, the workflow returns to the investigation stage, allowing the agent to gather additional evidence and revise its localization hypothesis (Lines 24–25). Once validation succeeds, FaultLens invokes ConditionalSelfCheck and uses $\mathcal{M}_{chk}$ to determine whether candidate locations satisfy the trigger condition for subsequent self-check (Line 27). If no suspicious candidate remains after self-check while there are still remaining investigation rounds, the workflow again returns to codebase investigation, since the current evidence is considered insufficient to support a reliable localization decision (Lines 28–29). Otherwise, the remaining candidates are further refined by AdvancedIdentify (Line 31) and then ranked by suspiciousness to produce the final output $\mathcal{C}_{\text{rank}}$ (Line 32).

4.1 Issue Analysis and Investigation

4.1.1 Initial Issue Analysis. Given the information about a bug, the agent first analyzes the issue to gain an initial understanding of the situation, aligning with the common practice of human debugging. The complete prompt is shown in Figure 5. The agent is provided with a failing test case that includes the file path, test source code, and error information, which encompasses assertions or exceptions.

To facilitate the agent's further investigation, the error information also contains the stack traces of the bug [8]. Since full stack traces can be lengthy and often contain calls from libraries or testing infrastructure, we keep only the trace entries whose reported source locations resolve to files in the subject repository under analysis. Concretely, we parse the stack trace entry by entry and retain those with file-and-line information that can be matched to a file in the checked-out repository, while omitting the remaining entries. However, when stack traces are provided only as textual input to the agent, the file names and line numbers in trace entries mainly serve as references to source locations rather than as direct code evidence. Unless the corresponding code is explicitly retrieved, the LLM receives only indirect location cues and may have difficulty using them for fault reasoning. To make the retained trace entries more informative, for each repository-local stack-trace entry, we append the corresponding source statement after the trace. For example, given `MathArrays.linearCombination:846`, we resolve the referenced file in the checked-out repository and recover the statement containing line 846 from `MathArrays.java`. Specifically, we parse the file with the tree-sitter library [1], locate the smallest named syntax node whose span covers the reported line, and then climb to the nearest enclosing statement or declaration. The text span of that syntactic unit is appended to the prompt. If the extracted unit spans multiple lines, we serialize it into a single line only for concise presentation in the prompt. Since we apply this augmentation only to stack-trace entries whose source locations can be resolved to files in the subject repository, this step avoids

Failing Test Case Information
Test Path:
{ path of the file containing the failing test }
Test Source Code:
{ code of the failing test }
Test Error Information:
{assertion or exception information and stack traces}

Note: Each stack trace entry in Test Error Information is followed by the corresponding source code.
For example, the entry `at org.jfree.chart.plot.XYPlot.getDataRange(XYPlot.java:4493) Collection c = r.getAnnotations();` shows:
`org.jfree.chart.plot.XYPlot.getDataRange`: The method where the error occurred.
`(XYPlot.java:4493)`: The file and line number.
`Collection c = r.getAnnotations();`: The specific line of code at this location, indicating what was executing when the error was triggered.
Please analyse the issue based on the provided test case error information. Address the analysis in three steps:
Analysis of the Test Failure
Potential Cause of the Issue
Suggested Starting Points for Root Cause Investigation

Fig. 5. Prompt for the initial issue analysis. The prompt provides the failing-test path, test code, and error information and asks the agent to produce three outputs: failure analysis, potential causes, and starting points for root-cause investigation.

searching the entire codebase and incurs only modest overhead. A clear note is included in the prompt in Figure 5, indicating that the stack trace format has been modified to incorporate the exact line content from the code.

The analysis is structured in three sequential parts to enhance the agent's understanding of the issue: analyzing the test failure, identifying potential root causes, and suggesting starting points for root cause investigation. Failure analysis encourages the agent to consider the direct cause of the problem while inferring potential root causes helps the agent prioritize key areas for further investigation. By suggesting starting points, the agent establishes an initial plan for subsequent investigations. This comprehensive analysis enables the agent to build a foundational understanding of the issue, laying the groundwork for the FL task.

4.1.2 Tool Invocation Loop. While the initial analysis provides the agent with a foundational understanding of the issue, it cannot ensure accurate FL when relying solely on the limited information from a failing test case. This is due to the complexity of repository-scale FL, which often involves numerous interdependencies and cross-file interactions, unlike localizing issues within a short code snippet. Codebase exploration tools equip the agent with the ability to navigate the codebase, allowing it to gradually uncover FL candidates through a detailed examination.

We provide a set of codebase exploration tools to facilitate the agent's code investigation, as outlined in Table 1. The tool set is designed from the perspective of repository-scale fault localization to support flexible investigation paths, rather than enforcing a fixed exploration order. Accordingly, these tools expose complementary information at different granularities, including file-level structure, class-level overviews, inner-class details, method contents, and import information. This applies whether the agent starts by investigating files and gradually narrows the

Table 1. List of Codebase Exploration Tools

Tool	Description
get_class_info(*c*, [*f*=None])	Get an overview of the class *c* in the file *f*, including package information, inheritance and implementation, class-level comments, fields, covered inner classes, and a list of covered methods.
get_covered_files_from_dir(d)	Get covered files under the directory *d*.
extract_method(*m*, [*c*=None], [*f*=None])	Extract the code and comment of the method *m* within class *c* from the file *f*.
get_inner_class_info(*c*, *ic*, [*f*=None])	Retrieve the fields and covered methods of the inner class *ic* within class *c* from the file *f*.
get_imports(*f*)	Retrieve a list of libraries imported in the file *f*.

scope to identify FL candidates, or directly examines a class, inner class, or method, depending on its current hypothesis, with each tool providing contextual information useful for FL.

To balance effectiveness and efficiency, we aim for each tool's output to be informative without being overly detailed, carefully dividing the responsibilities among tools. Specifically, the `get_class_info` tool provides a concise overview of a class, displaying information such as the names of covered inner classes and the signatures of covered methods, while omitting detailed content. Correspondingly, we offer `get_inner_class_info` and `extract_method` to access their respective code content on demand. Statistical analysis across the real-world open-source projects comprising the Defects4J-V1.2.0 benchmark reveals that inner classes are significantly smaller in scale, containing an average of 3.53 methods (median: 2), compared to an average of 14.48 methods (median: 6) in top-level classes. Given this compact nature, `get_inner_class_info` displays the code content of all covered methods within an inner class when invoked. Additionally, we provide the tool `get_imports` to supply imported-library information for a file when such dependency context is needed. These tools are not tied to benchmark-specific metadata. In our implementation, codebase navigation is generally repository-agnostic, and the syntax-aware structure extraction is built on tree-sitter, making the same tool design applicable to other repositories written in supported languages.

Since the input to FaultLens is the information of one failing test case, the tools are designed to focus information extraction on the actual coverage provided by this failing test case. For instance, in the code content provided by `extract_method`, lines covered by the test are marked as covered, distinguishing them from uncovered lines. This allows the agent to concentrate on the lines directly relevant to the failing test. In the `get_covered_files_from_dir` tool, line coverage data is utilized to refine the results. Since the agent may provide broad directories such as `src/` that include many files, the tool filters and returns only those files that are actually covered at the line level.

During this loop, tool invocation is demand-driven rather than exhaustive. As formalized in Algorithm 1, in each iteration the agent selects a subset of tools via SelectToolCalls($ctx, \mathcal{T}$) according to the current context, which includes the failing-test information, previously retrieved code context, and the agent's intermediate analyses. Therefore, not all tools are invoked for every bug, while multiple tools may still be invoked within one iteration when they provide complementary evidence. The order of tool invocation is unrestricted, ensuring both flexibility and efficiency.

After receiving the tools' outputs, the agent is prompted to analyze the newly obtained evidence, mirroring the actions a human would take after gathering information. In each iteration, the tools' outputs and the agent's analysis are saved as contextual information. Based on this accumulated context, the agent determines whether sufficient information has been collected to identify suspicious locations or whether further investigation is still needed. This decision is formalized as StopInvestigation(ctx) in Algorithm 1. If the agent decides to proceed,

the saved context forms the basis for the next iteration, enabling the construction of subsequent tool calls. This accumulated context also helps reduce redundant exploration. When the tool list is provided to the agent in each investigation round, the prompt explicitly instructs it not to repeat a tool call with identical arguments, because the returned result would remain unchanged. Combined with the retained context of prior tool outputs, this encourages the agent to build subsequent actions on previously collected evidence rather than issuing duplicate calls.

To balance thoroughness with efficiency, the tool invocation loop is bounded. The agent stops the loop either when the accumulated context is deemed sufficient for preliminary localization or when the predefined maximum number of investigation rounds, denoted by N_{iter} in Algorithm 1, is reached. Moreover, as shown in Algorithm 1, even when later stages request additional investigation, control returns to the tool loop only if the remaining iteration budget is still available. This design guarantees termination and prevents infinite tool-invocation loops.

4.2 Fault Localization Decision

When the agent enters the fault localization decision stage, it conducts preliminary location identification at the method level, identifying FL candidates and generating root cause analysis. The decision stage provides fine-grained feedback to mitigate hallucinations and enhance the accuracy of FL. Potential hallucinations caused by incomplete investigation are captured through location extraction validation and a conditional self-check mechanism. Additionally, faulty reasoning poses a challenge during FL tasks. Even when the agent has access to all the relevant information needed to identify fault locations, it may still make incorrect conclusions due to errors in reasoning. To address this, we design the advanced location identification phase to rectify potential errors in early reasoning.

4.2.1 Location Extraction Validation. Location extraction validation is designed specifically to capture extrinsic hallucinations, where the agent identifies invalid locations that do not exist in the codebase. Since the locations are at the method level, each location includes the file, class, and method name provided by the agent. Through interaction with the codebase, these locations are verified to determine whether they are extractable or non-existent. Validation is considered successful only if all locations are confirmed as extractable.

For cases where validation fails, suppose a location identified by the LLM at the method level is represented as a tuple (F, C, M), where F denotes the file name containing the identified fault, C denotes the class or inner class name where the faulty method resides, and M denotes the method name identified as faulty. When validation fails, feedback indicates which part of the identified location is imprecise, guiding further investigations to avoid such hallucinations. There are three types of feedback in such cases:

- F cannot be located.
- F is located, but C cannot be found within F.
- C is located within F, but M cannot be found within C.

It is possible for the agent to identify a method with the correct name but the wrong class or file name. The rationale behind providing feedback and prompting further investigation, rather than extracting candidates with the same method name from the codebase, is that identifying a location with the correct file, class, and method name is a fundamental requirement for FL. If this requirement is not met, there is a high likelihood of hallucinations in the response, compromising the quality of the content. Offering clear feedback to prompt a deeper investigation is more effective than continuing with imprecise results. Once the location extraction validation captures extrinsic hallucinations, the agent is sent back to the tool invocation loop with specific feedback. It then uses the tools to gather new information to rectify the imprecise locations.

4.2.2 Conditional Self-check Mechanism. If the agent identifies fault-localization candidates solely from method signatures, class-level summaries, or test-case clues without examining the method body, it cannot confidently

Here is the code of the bug locations:
Bug Location 1:
{The content of the location including comment and code.}
...
Please inspect the code of each location above and answer whether it is buggy or not.
You should give reasons for your judgment. Return your answer in **JSON format** as follows:

```
{
  "recheck": [
    {
      "file": "path/to/file",
      "class": "class_name",
      "signature": "method_signature",
      "buggy": true or false,
      "reason": "reason for judgment"
    },
    ...
  ]
}
```

Fig. 6. Prompt for the conditional self-check mechanism. Given the code of the candidate bug locations, the agent re-inspects each location and returns a JSON object indicating whether the location is buggy, together with the rationale for the judgment.

determine whether the location is genuinely responsible for the bug. A suspicious method inferred in this way may only be an entry point in the execution flow, while the actual fault resides in one of its callees or in another related method. To mitigate such intrinsic hallucinations caused by incomplete investigation, we introduce a conditional self-check mechanism that prompts further code-level verification when needed.

More specifically, the framework first determines whether candidate locations satisfy the trigger condition for self-check, rather than applying this process indiscriminately to all methods that may be relevant to the bug. For each candidate location, we determine whether additional inspection is required by consulting an inspected-method list maintained throughout the tool invocation loop. A method is added to this list only when its code content has actually been returned by content-revealing tools, such as `extract_method` and `get_inner_class_info`. Therefore, when a candidate is inferred from higher-level evidence without prior examination of its method body, the framework treats it as requiring supplementary inspection and organizes the subsequent self-check accordingly.

Figure 6 shows the prompt used in this process. The framework provides the full context, together with the code and comments of the locations to be checked, and asks the agent to assess whether each location is genuinely buggy and to justify its judgment. The self-check results are returned in the JSON format shown in Figure 6. Candidates that are rechecked and judged not buggy are removed from the FL candidate list. If no suspicious candidate remains after self-check, the agent returns to the tool invocation loop for further investigation.

4.2.3 Advanced Location Identification. To mitigate intrinsic hallucinations caused by faulty reasoning, we introduce an advanced location identification step in the decision stage. As shown in Figure 4, this step is performed after location extraction validation and the conditional self-check mechanism. As illustrated by the motivating example in Figure 2(c), even after relatively thorough code inspection, the agent may still incorrectly

include innocent locations in the candidate list due to flawed reasoning. To address this issue, FaultLens performs an additional round of candidate identification based on the full task context, aiming to supplement suspicious locations that may have been missed in preliminary location identification.

```
Based on your previous investigations and interaction results, carefully reconsider the bug and
identify additional suspicious locations that may have been missed earlier.

Return results in JSON:
{
  "more_suspicious_locations": [
    {
      "file": "...",
      "class": "...",
      "method": "..."
    }
  ]
}
```

Fig. 7. Prompt for advanced location identification. Based on the previous investigation and interaction results, the agent reconsiders the bug and returns additional suspicious locations that may have been missed earlier in JSON format.

Advanced location identification takes the full context ctx as input. This context includes the entire interaction history accumulated during fault localization, including previous investigations, feedback from location extraction validation, and the self-check process. With this more comprehensive context, the agent is better positioned to identify additional suspicious locations beyond those found in the preliminary stage. Figure 7 presents the prompt used for advanced location identification. Following this prompt, the agent is required to return newly identified candidate locations in a structured format.

Algorithm 2 Workflow of Advanced Location Identification

Require: Existing validated candidate list $\mathcal{C}_{\text{val}}$, full context ctx
Ensure: Updated validated candidate list $\mathcal{C}_{\text{val}}$, updated full context ctx
1: Predefined prompt Q for advanced location identification
2: $\Delta\mathcal{C} \leftarrow \textsc{Identify}(ctx, Q)$
3: $\Delta\mathcal{C}_{\text{val}} \leftarrow \{\, c \in \Delta\mathcal{C} \mid \textsc{Extractable}(c) = \text{true} \,\}$
4: $\mathcal{C}_{\text{val}} \leftarrow \textsc{AppendUnique}(\mathcal{C}_{\text{val}}, \Delta\mathcal{C}_{\text{val}})$
5: $ctx \leftarrow \textsc{UpdateCtx}(ctx, Q, \Delta\mathcal{C}, \Delta\mathcal{C}_{\text{val}})$
6: **return** $(ctx, \mathcal{C}_{\text{val}})$

Algorithm 2 summarizes the workflow of advanced location identification. Specifically, given the full context ctx and a predefined prompt Q, the framework first generates an additional candidate list $\Delta\mathcal{C}$ (Line 2). It then performs location extraction on $\Delta\mathcal{C}$ and directly retains only the locations that can be successfully validated, yielding $\Delta\mathcal{C}_{\text{val}}$ (Line 3). Different from the earlier location extraction validation step, this stage does not preserve error feedback for failed extractions. Instead, it simply filters them out. Next, $\Delta\mathcal{C}_{\text{val}}$ is appended to the end of the existing validated candidate list $\mathcal{C}_{\text{val}}$ in the order of identification, while duplicate entries are removed (Line 4). Finally, the prompt and identification results are written back to the context for subsequent ranking, and the updated context ctx and candidate list $\mathcal{C}_{\text{val}}$ are returned (Lines 5–6).

Since extrinsic hallucinations caused by insufficient investigation have already been reduced by the preceding location extraction validation step, advanced location identification typically introduces fewer non-existent locations. Any remaining non-existent locations are directly filtered out if they appear.

4.2.4 Ranking of Fault Localization Candidates. All FL candidates are aggregated for final ranking. The agent receives a list of identified FL candidates, excluding those filtered out during location extraction validation and the self-check process. Based on the conversation context throughout the task, the agent ranks all the remaining candidate methods according to their suspiciousness.

4.3 Multi-run Rank-Based Voting

Algorithm 3 Rank-Based Voting Scheme for Fault Localization Candidate Ranking

Require: N FL candidate lists from independent runs: $\mathcal{C}_{rank}^{(1)}, \mathcal{C}_{rank}^{(2)}, \ldots, \mathcal{C}_{rank}^{(N)}$
Ensure: Final ranked list of FL candidates
1: $L_{\max} \leftarrow \max\{|\mathcal{C}_{rank}^{(i)}| \mid 1 \leq i \leq N\}$
2: Score $\leftarrow \emptyset$
3: **for** $i \leftarrow 1$ **to** N **do**
4: **for** each location l in $\mathcal{C}_{rank}^{(i)}$ with rank k **do**
5: **if** $l \notin$ Score **then**
6: Score(l) $\leftarrow 0$
7: **end if**
8: Score(l) $\leftarrow$ Score(l) + ($L_{\max} - k + 1$)
9: **end for**
10: **end for**
11: **Sort** all locations based on descending Score(l)
12: **Return** the ranked list of FL candidates

Although low-temperature or greedy decoding reduces stochasticity, repeated runs can still produce slightly different results because LLM inference is subject to implementation-level non-determinism, such as hardware-dependent floating-point computations [66]. Even tiny numerical deviations may accumulate and be amplified during autoregressive generation, causing variations in candidate assessments and rankings for the same input. To reduce the influence of nondeterminism while improving reliability and decision quality, FaultLens allows the number of runs to be configurable and employs a rank-based voting mechanism to aggregate the ranked lists of FL candidates from all runs. In each run, FaultLens independently performs fault localization and ranks the identified candidates. The rank-based voting approach is outlined in Algorithm 3. It begins by initializing $L_{\max}$ (Line 1) as the maximum list length across all ranked lists and creating an empty score dictionary (Line 2). Next, for each run's list, the algorithm iterates through each location (Lines 3-10). The location's score is then incremented based on its rank k, using the formula ($L_{\max}$ - k + 1) (Line 8), which gives higher increments to locations ranked higher in each run. Finally, the algorithm sorts all locations based on their accumulated scores in descending order (Line 11) and returns the ranked list of FL candidates (Line 12). This rank-based voting scheme promotes locations that are consistently identified and highly ranked across multiple runs, improving the reliability of FL.

5 Evaluation

We design the following research questions to assess the performance of our technique.

- **RQ1:** How does the performance of FaultLens compare with other FL techniques?

Table 2. Overview of the Defects4J-V1.2.0 Benchmark

Chart		Closure		Lang		Math		Mockito		Time		Sum	
#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.
26	92	133	348	65	124	106	176	38	118	27	76	395	934

Table 3. Overview of the BugsInPy Projects Used in RQ5

ansible		black		fastapi		httpie		matplotlib		pandas		PySnooper	
#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.
15	19	19	20	3	10	5	5	27	28	167	211	1	1
scrapy		**spacy**		**thefuck**		**tornado**		**tqdm**		**youtube-dl**		**Sum**	
#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.	#Bugs	F.T.
21	33	2	2	30	38	14	16	8	9	42	42	354	434

- **RQ2:** What impact do individual components of FaultLens have on the accuracy?
- **RQ3:** To what extent do the proposed mechanisms provide evidence of transferability beyond FaultLens in an external repository-scale localization workflow?
- **RQ4:** How do different hyperparameter configurations influence FaultLens?
- **RQ5:** How effectively does FaultLens generalize to different programming languages?
- **RQ6:** How effectively does FaultLens generalize to different LLMs?

The six research questions address distinct aspects of our evaluation. For RQ1, we compare the performance of FaultLens against state-of-the-art FL techniques on the Defects4J [20] benchmark. For RQ2, we investigate the significance of the location extraction validation, conditional self-check mechanism, and advanced location identification, analyzing their impact on accuracy. In addition, we also examine the contribution of the ranking component introduced in Section 4.2.4 by conducting an ablation study that removes this step. For RQ3, we conduct an external case study to assess whether the proposed hallucination-mitigation mechanisms provide evidence of transferability beyond FaultLens by adapting them to the fault-localization stage of AutoCodeRover [71], an issue-driven program repair framework. For RQ4, we adjust the hyperparameter configurations to observe and conclude the trends in FaultLens's behavior.

For RQ5, we assess the cross-language generalizability of FaultLens by applying the same overall framework and default hyperparameters to a Python benchmark, with language-specific adaptations to the codebase exploration tools. For RQ6, we evaluate the model-level adaptability of FaultLens by testing it with different foundation models. Finally, we provide a supplementary failure analysis of FaultLens by examining representative individual-run failure modes, highlighting residual cases where hallucinations persist or later decision stages introduce new ranking errors.

In line with previous works [21, 26, 34, 50], we evaluate the performance of FaultLens using the Top-N metric. Specifically, we evaluate for N = 1, 3, and 5. This metric measures the number of bugs that have at least one defective element ranked within the top N locations of the list. In our evaluation, defective elements are identified at the method level.

5.1 Experiment Setup

5.1.1 Benchmark. For each defect, we use exactly one selected failing test case as the input to fault localization. When multiple failing tests are available for a defect, we deterministically select the first failing test listed in the benchmark-provided failing-test list. We evaluate FaultLens on the Defects4J benchmark version 1.2.0 [20], which consists of 395 bugs from real-world projects. Defects4J has been extensively used in prior studies on

software debugging, including FL [21, 50] and automatic program repair (APR) [31, 35, 37]. Table 2 presents an overview of the projects in Defects4J v1.2.0, outlining the number of bugs, and the total number of failing tests (F.T.) for each project. Each bug is associated with at least one failing test that can reproduce the issue. Additionally, to investigate the generalizability of FaultLens across different programming languages (RQ5), we evaluate its performance on BugsInPy [56], a benchmark consisting of 501 real bugs from Python projects. Leveraging the improved BugsInPy dataset provided by [5], we successfully reproduced 354 bugs. The remaining bugs could not be reproduced due to various challenges encountered during setup, mainly involving environment configuration, as well as issues during execution. The reproducible bugs were subsequently included in our evaluation. Table 3 summarizes the number of bugs and failing tests across the 13 BugsInPy projects used in our study.

5.1.2 Implementation. FaultLens is implemented in Python. We utilize Gzoltar [9] to gather coverage information for the Java benchmark, Coverage.py [7] for the Python benchmark, and the tree-sitter library [1] to parse the code, enabling effective codebase exploration tools. Considering both cost and effectiveness, we adopt GPT-4o-mini (gpt-4o-mini-2024-07-18) as the default foundation model in our framework, unless explicitly stated otherwise.

5.1.3 Hyperparameters for RQ1, RQ2, RQ5, and RQ6. We configure FaultLens with 5 independent runs. In all runs of the same defect, the selected failing test case is fixed. The temperature is set to 0.2, and the upper limit of the tool invocation loop is 10.

5.2 Experiments and Discussion

5.2.1 RQ1: How does the performance of FaultLens compare with other FL techniques? FaultLens is compared against representative FL techniques on the Defects4J benchmark using Top-N metrics, including Ochiai [3], Metallaxis [43], FLUCCS [50], AutoFL [21], and SoapFL [46]. These baselines cover spectrum-based, mutation-based, learning-based, and LLM-based FL techniques. In particular, AutoFL and SoapFL are the closest repository-scale LLM-based comparators to FaultLens, while Ochiai, Metallaxis, and FLUCCS provide representative non-LLM references that are widely used on Defects4J.

Baseline provenance and rerun protocol. We collect Ochiai results with Gzoltar [9]. The evaluation results for Metallaxis (using the Ochiai formula) and FLUCCS are sourced from Li et al. [26]. AutoFL is an LLM-based agent FL technique with configurable repetition. For a fair comparison with FaultLens, we use GPT-4o-mini as the foundation model for AutoFL and rerun it 5 times, aligning with the settings of FaultLens. SoapFL [46] is a multi-agent FL framework that follows a standard operating procedure consisting of three phases. Since FaultLens employs rank-based voting across 5 runs to mitigate the nondeterminism in LLM outputs, while SoapFL produces a single-run result, we rerun SoapFL 5 times and apply rank-based voting in the same manner as described in Section 4.3 for FaultLens, ensuring a fair one-to-one comparison. Similarly, we use GPT-4o-mini as the foundation model for SoapFL.

Table 4 presents the performance of FaultLens compared with other FL techniques. For the 395 bugs in the benchmark, FaultLens successfully identifies 205, 259, and 278 defective methods in the Top-1, Top-3, and Top-5 metrics, respectively. Notably, FaultLens achieves outstanding overall performance in the Top-1 metric, surpassing all baselines across the benchmark, while demonstrating superior or comparable performance for individual projects.

FaultLens demonstrates significant improvement over Ochiai and Metallaxis across the Top-1, Top-3, and Top-5 metrics. Specifically, FaultLens shows increases of 57.69%, 56.02%, and 49.46% over Ochiai in the Top-1, Top-3, and Top-5 metrics, respectively, and an increase of 118.08% in the Top-1 metric compared to Metallaxis. Against the machine learning-based technique FLUCCS, FaultLens achieves a 28.12% increase in the Top-1

Table 4. Performance of FaultLens and other FL techniques on the Defects4J benchmark.

Techniques	Chart			Closure			Lang		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Ochiai	10	16	18	25	37	41	37	42	46
Metallaxis	7	15	17	19	47	64	32	51	56
FLUCCS	15	19	20	**42**	**66**	**77**	40	53	55
AutoFL	16	**23**	**23**	27	50	57	38	**57**	**60**
SoapFL$_{5runs}$	12	19	20	24	50	60	37	51	54
FaultLens	**19**	22	**23**	39	57	63	**48**	54	59

Math			Mockito			Time			Overall		
Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
41	51	60	7	7	8	10	13	13	130	166	186
20	51	71	9	15	21	7	12	15	94	191	244
48	77	83	7	19	22	8	15	18	160	249	275
56	81	84	15	22	24	13	**19**	**20**	165	252	268
46	71	73	13	22	22	11	16	16	143	229	245
63	**83**	**88**	**20**	**24**	**25**	**16**	**19**	**20**	**205**	**259**	**278**

metric and outperforms FLUCCS by 4.01% in the Top-3 metric, while delivering comparable performance in the Top-5 metric. Note that FLUCCS is trained on specific projects using 10-fold cross-validation, whereas FaultLens does not require any specialized training.

When compared to AutoFL, which is also based on LLMs, FaultLens achieves a 24.24% improvement in the Top-1 metric, along with additional gains in the Top-3 and Top-5 metrics. In FaultLens, location extraction validation and conditional self-check mechanisms enable a more thorough investigation during the preliminary location identification phase, leading to more accurate fault localization. The advanced location identification process helps uncover previously overlooked suspicious locations. Together, these strategies contribute to overall improvements in Top-N performance, with a particularly notable gain in Top-1 accuracy. Although the hallucination mitigation strategies employed by FaultLens introduce a slight increase in token usage due to extended interactions with the model, this overhead is negligible in light of the continuously decreasing cost of mainstream large language models.

When compared to SoapFL, which is rerun five times and aggregated using the same rank-based voting scheme (denoted as SoapFL$_{5runs}$ in Table 4), FaultLens achieves improvements of 43.35%, 13.10%, and 13.46% in the Top-1, Top-3, and Top-5 metrics, respectively. SoapFL narrows the faulty code space in a step-by-step manner, initially identifying suspicious classes and then selecting methods within those classes. While this structured pipeline helps to focus the search, it lacks dynamic adaptability. Once a class is flagged as suspicious, the agent is restricted to investigating methods within that class, even when further inspection shows that those methods are unrelated to the actual root cause of the bug. This rigidity limits the agent's ability to explore other potentially relevant classes that may have been overlooked in earlier stages. In contrast, FaultLens adopts a dynamic tool-calling loop, allowing the agent to independently determine when to continue or terminate the investigation. More importantly, the hallucination mitigation strategies integrated into FaultLens further enhance the investigation process and location identification, resulting in improved FL performance.

These results demonstrate that FaultLens exhibits a strong capability in identifying defective locations compared to different kinds of FL techniques using only the information from failing tests, without the need for additional training. The enhanced hallucination-mitigation capability of FaultLens enables it to fully leverage LLMs' code comprehension strengths and minimize mislocalization caused by hallucinations, thereby achieving performance improvements that surpass various baselines.

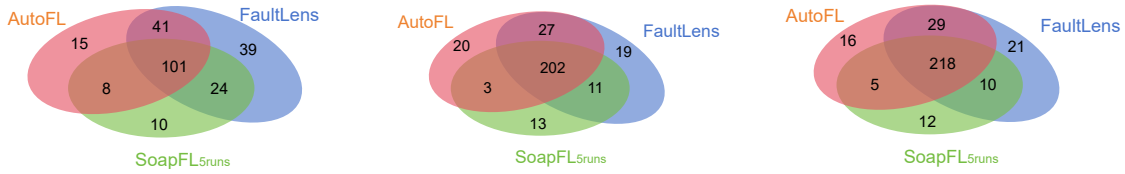

Fig. 8. Overlap of AutoFL, SoapFL$_{5runs}$, and FaultLens under Top-1, Top-3, and Top-5. The three Venn diagrams show the overlaps for Top-1 (left), Top-3 (middle), and Top-5 (right), respectively.

Overlap Analysis. To further understand the noticeably larger Top-1 improvement of FaultLens over other LLM-based approaches, we analyze the overlap of successfully localized bugs shown in Figure 8. At Top-1, FaultLens localizes 39 bugs that are not localized by either AutoFL or SoapFL$_{5runs}$, whereas AutoFL and SoapFL$_{5runs}$ localize only 15 and 10 unique bugs, respectively. In contrast, the overlap among the three approaches becomes much larger at Top-3 and Top-5, where all three methods jointly localize 202 and 218 bugs. This indicates that, once multiple candidates are allowed, these LLM-based techniques often converge to similar shortlists, which naturally narrows the gap at Top-3 and Top-5. The main advantage of FaultLens therefore lies in identifying the correct faulty method more reliably during the localization process.

Importantly, this advantage is not merely a consequence of the final ranking step. Before the final ranking stage, FaultLens still localizes 195 bugs at Top-1, including 35 bugs that are not localized by either AutoFL or SoapFL$_{5runs}$. This observation is also consistent with the ablation results in Table 5, where removing the ranking step causes only very limited changes in performance. Therefore, the contribution of the final ranking is minor, and the gain of FaultLens mainly comes from the hallucination-mitigation strategies in the decision stage, which improve candidate identification before final ranking. A small-scale manual inspection of FaultLens-only Top-1 cases suggests that, under rank-based voting across five runs, FaultLens more often preserves the correct fault location across runs, whereas competing approaches may identify the correct location in some runs but fail to do so consistently enough for it to remain at the top after aggregation.

Hallucination-Related Event Analysis. To complement the Top-N results, we further analyze hallucination-related events observed during FaultLens execution on Defects4J. For tractability, this analysis is conducted on one sampled trajectory for each bug under the default five-run setting. We observe that extrinsic hallucinations are not rare in repository-scale FL: 154 out of 395 bugs (38.98%) trigger location extraction validation because at least one predicted location cannot be extracted from the repository. This phenomenon also shows a clear project-level skew. The rate is especially high in Closure (97/133, 72.93%) and Mockito (22/38, 57.89%), but much lower in Chart (2/26, 7.69%) and Lang (4/65, 6.15%). Closure is a JavaScript checker and optimizer, while Mockito is a Java mocking framework centered on mock creation, verification, and stubbing. Compared with smaller utility-style libraries, such projects may require broader context integration and make premature location decisions more likely.

We further find that insufficiently deep investigation is even more widespread than non-existent location prediction. Conditional self-check is triggered for 287 of 395 bugs, indicating that preliminary localization often proposes at least one candidate whose method body has not yet been inspected. This tendency is again visible across projects, with particularly high trigger rates in Closure (117/133, 87.96%) and Mockito (30/38, 78.94%), but it remains common in the other projects as well, including Chart (18/26, 69.23%), Math (73/106, 68.86%), and Time (20/27, 74.07%). Compared with extrinsic hallucinations, these recheck events are more prevalent, suggesting that when equipped with codebase exploration tools, the agent is usually able to identify relevant regions of code, but still tends to terminate investigation before sufficiently examining all candidate method bodies. Across the

triggered cases, self-check removes 184 candidates, among which 177 are non-faulty according to the benchmark ground truth, while 7 correspond to fault locations, yielding a removal precision of 96.19%.

Finally, advanced location identification further improves candidate recovery after the earlier checks. We observe that this stage corrects 29 and 35 previously unsuccessful localization stages under Top-3 and Top-5, respectively. Although its effect on Top-1 is smaller, this result suggests that revisiting earlier reasoning can still help recover true faulty methods that were missed in earlier stages and bring them into the final candidate list.

Case Study. To highlight the significance of FaultLens's enhanced performance, particularly in the Top-1 metric, we demonstrate its FL capabilities on a real-world bug. Figure 9 illustrates FaultLens's pipeline for bug `Closure-59` from the Defects4J [20] dataset, showing a single independent run of the framework. `Closure-59` represents the 59th documented bug in the Google Closure Compiler [13], included in Defects4J. Due to space limitations, some details, such as the agent's analysis of tool outputs, are omitted. In Figure 9, the system comprises the user, the agent, codebase exploration tools, and the environment. The user only needs to supply the failing test case information and receive the final ranked list of FL candidates, while the FL process involves interactions between the agent, tools, and environment, with the agent driving the process of identifying suspicious locations.

The top portion of Figure 9 provides a brief description of the failing test case. This bug arises because the `Compiler` class does not correctly handle the `--jscomp_off=globalThis` option, which leads to an unintended `JSC_USED_GLOBAL_THIS` warning. During the initial issue analysis, the agent identifies a potential cause of the failure, but further investigation is required to determine the responsible location.

In the first iteration, the agent invokes two tools to retrieve information on the class `CommandLineRunnerTest` and the imported libraries in `CommandLineRunner`. After analyzing the tools' outputs, the agent attempts to exit the loop but incorrectly identifies two non-existent methods, `compile` and `createCommandLineRunner`, within the class `CommandLineRunner`. These extrinsic hallucinations are captured by the environment through location extraction validation, prompting the agent to continue its investigation. Upon receiving feedback indicating that `compile` is a non-existent method in the class `CommandLineRunner`, the agent proceeds to gather additional compilation-related information. It then uses tools to obtain information on the `Compiler` class and the files under `src/com/google/javascript/jscomp`. Based on the newly collected information, the agent identifies two actual methods, `compile` and `initOptions`, but does not inspect their contents, which triggers the self-check mechanism. The agent rechecks these methods, confirming their defectiveness. Through advanced location identification, it produces two additional locations and ultimately ranks all four candidates, outputting a final ranked list in which `initOptions`, the true fault location, is ranked highest.

In contrast, the LLM-based agent approaches, AutoFL and SoapFL, both face challenges in accurately locating the truly defective location for `Closure-59`. In the case of AutoFL, the method `initOptions` is ranked well outside the Top-5, indicating that its prioritization may not effectively support downstream repair tasks, particularly if the repair process adheres to the ranking provided by the FL results. SoapFL, which follows a fixed workflow by first identifying suspicious classes and then selecting suspicious methods within those classes, also struggles in this case. Specifically, for `Closure-59`, SoapFL fails to identify the class containing `initOptions` as suspicious, thereby preventing accurate fault localization.

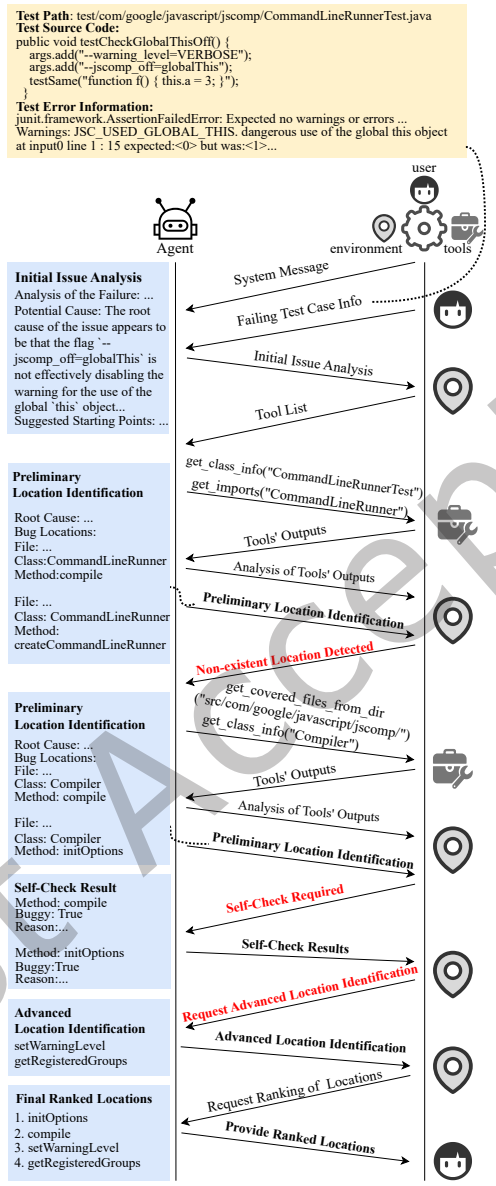

Fig. 9. FaultLens's pipeline on the bug Closure-59. The figure shows how initial issue analysis, location extraction validation, conditional self-check, and advanced location identification interact during fault localization.

Answer to RQ1: FaultLens outperforms various baselines on the Defects4J benchmark. The overlap analysis shows that although FaultLens shares many localized bugs with existing LLM-based agents under Top-3 and Top-5, it identifies substantially more unique bugs under Top-1. The hallucination-related event analysis further indicates that repository-scale FL agents frequently suffer from invalid locations, insufficiently inspected candidates, and missed fault locations, while FaultLens's location extraction validation, conditional self-check, and advanced location identification mechanisms help mitigate these issues. The case study on `Closure-59` illustrates how the overall workflow helps recover from misleading intermediate localization results and ultimately identify the true faulty method.

Table 5. Ablation study results. Among them, LEV stands for *Location Extraction Validation*, CSC stands for *Conditional Self-check*, and ALI stands for *Advanced Location Identification*. The entry "w/o All" refers to a variant where all three components are removed. "w/o Ranking" disables the ranking step introduced in Section 4.2.4

Top-N	w/o LEV & CSC	w/o CSC	w/o ALI	w/o All	w/o Ranking	FaultLens
Top-1	167	190	193	161	195	**205**
Top-3	207	243	233	203	256	**259**
Top-5	221	261	237	207	274	**278**

5.2.2 RQ2: What impact do individual components of FaultLens have on the accuracy? We evaluate the effectiveness of our design by investigating the impact of location extraction validation, conditional self-check mechanism, and advanced location identification on performance. We create four variants of the original design: one without the location extraction validation and self-check mechanism, one without the self-check mechanism, one without advanced location identification, and one without all three components. In addition, to examine the contribution of the ranking component introduced in Section 4.2.4, we include an extra variant that disables the ranking step. Table 5 presents the results of the ablation study.

When the conditional self-check mechanism is removed, the performance drops by 7.31%, 6.17%, and 6.11% in the Top-1, Top-3, and Top-5 metrics, respectively. Removing both the location extraction validation and the self-check mechanism together leads to declines of 18.53%, 20.07%, and 20.50% in the Top-1, Top-3, and Top-5 metrics. The location extraction validation helps identify non-existent locations, providing feedback for further investigation. Without it, premature conclusions caused by extrinsic hallucinations go undetected. The conditional self-check mechanism offers the agent a second opportunity to thoroughly inspect previously unexamined code. These results indicate that the location extraction validation is crucial for FaultLens's performance, while the conditional self-check mechanism also plays a key role in addressing intrinsic hallucinations arising from incomplete investigation.

When the advanced location identification is removed, there is only a slight degradation in the Top-1 metric. However, the decline becomes more pronounced for the Top-3 and Top-5 metrics, with decreases of 10.03% and 14.74%, respectively. Although the agent can only identify FL candidates through preliminary location identification, the location extraction validation compels it to perform an in-depth inspection of the codebase, maintaining an advantage in the Top-1 metric. However, without advanced location identification, the agent struggles to identify less obvious but still suspicious locations. These locations may be missed in preliminary location identification due to faulty reasoning, but the combination of comprehensive investigation, location extraction validation, and the self-check process helps correct the initial reasoning errors. Although the FL candidates identified in advanced location identification may be ranked lower by the agent, including them in the list is significantly better than excluding them entirely. Without advanced location identification, these locations,

which are typically ranked beyond the first location, now have no chance of appearing in the list, contributing to the decline in Top-3 and Top-5 performance.

Notably, the variant without all three components shows declines of 21.46%, 21.62%, and 25.53% in the Top-1, Top-3, and Top-5 metrics, respectively, highlighting FaultLens's superior performance over the LLM-based agent that can only use tools. This is because these three components work together to construct a more comprehensive decision stage, mitigating both extrinsic and intrinsic hallucinations while leveraging the LLMs' advantage in code comprehension, thus leading to better performance.

Finally, we analyze the impact of the ranking mechanism. Removing the ranking step results in only minor declines of 4.87%, 1.15%, and 1.43% in the Top-1, Top-3, and Top-5 metrics, respectively. This indicates that while the ranking component can modestly enhance the final output by reordering the collected candidates, its contribution is limited compared to the core components. In particular, accurately identifying fault locations, especially those obscured by hallucinations, primarily relies on the earlier components rather than on post-hoc ranking. The ranking step serves to refine the candidate list based on already discovered information, but it cannot compensate for candidates that were missed or misidentified during the initial investigation.

Table 6. Comparison between FaultLens and a deterministic-feedback variant. This variant retains *Location Extraction Validation*, removes *Advanced Location Identification*, and replaces the LLM-based self-check with rule-based filtering based on whether a candidate method has been inspected.

Top-N	Det. Feedback	FaultLens
Top-1	183	**205**
Top-3	235	**259**
Top-5	246	**278**

Motivated by the possibility that additional LLM-based checking may be replaced by deterministic feedback, we further implement a deterministic-feedback variant. This variant retains location extraction validation, removes advanced location identification, and replaces the LLM-based self-check with a rule-based strategy: candidates whose contents have not been inspected are removed directly. If no candidate remains after filtering, the agent is sent back to the tool invocation loop with feedback indicating that these locations have not been inspected and recommending the use of `extract_method` to examine their contents in the subsequent investigation.

Table 6 presents the results. Compared with FaultLens, the deterministic-feedback variant achieves lower performance across all Top-N metrics, with declines of 10.73%, 9.26%, and 11.51% in Top-1, Top-3, and Top-5, respectively. These results suggest that deterministic feedback is helpful for reducing premature conclusions, but it cannot fully replace the complete decision-stage design of FaultLens. In particular, replacing self-check with rule-based filtering weakens the agent's dedicated re-examination of previously uninspected candidates and removes an important source of feedback from the interaction history. Moreover, without advanced location identification, the agent is less able to correct earlier reasoning errors and recover suspicious locations that were missed during preliminary location identification.

Answer to RQ2: The location extraction validation, conditional self-check, and advanced location identification significantly contribute to the accuracy of FaultLens. Removing these components results in varying degrees of performance degradation. Compared to FaultLens, the variant lacking all three components shows performance declines of 21.46%, 21.62%, and 25.53% in the Top-1, Top-3, and Top-5 metrics, respectively. In contrast, removing the ranking step leads to only minor drops, indicating its comparatively limited impact. We further find that a deterministic-feedback alternative improves over weaker ablated variants, but still remains inferior to the full framework, indicating that rule-based filtering alone cannot substitute for the combination of conditional self-check and advanced location identification in FaultLens.

5.2.3 RQ3: To what extent do the proposed mechanisms provide evidence of transferability beyond FaultLens in an external repository-scale localization workflow? To examine whether the proposed hallucination-mitigation mechanisms provide evidence of transferability beyond FaultLens, we adapt them to the fault-localization stage of AutoCodeRover [71]. AutoCodeRover is an autonomous program-improvement framework rather than a standalone fault localization approach. It takes as input a GitHub issue's problem statement and the corresponding codebase, performs iterative context retrieval with code search APIs, and then generates a patch. Its default workflow is therefore issue-driven, rather than failing-test-driven as in FaultLens, although it can additionally incorporate test-based fault localization when a test suite is available.

We conduct this evaluation on SWE-bench-Lite, a curated subset of SWE-bench [17] designed for more efficient evaluation on self-contained functional bug-fixing tasks. Following prior work on repository-scale localization over SWE-bench-Lite [11], we use the patched functions as the localization targets and evaluate on the same 274 instances retained by [11], which excludes examples whose patches do not modify any existing functions. To evaluate transferability beyond FaultLens, we adapt AutoCodeRover by incorporating our hallucination-mitigation mechanisms into its localization process while keeping its original methodology otherwise unchanged. Specifically, we evaluate four configurations: the original AutoCodeRover localization process, AutoCodeRover+LEV, AutoCodeRover+LEV+CSC, and AutoCodeRover+LEV+CSC+ALI, where the abbreviations follow those used in RQ2. For all configurations, we use GPT-4o-mini as the foundation model, follow AutoCodeRover's default temperature setting of 0.2, and report the average numbers of correctly localized instances under Top-1, Top-3, and Top-5 over five independent runs, rounded to the nearest integer.

Table 7. Transferability of the proposed hallucination-mitigation mechanisms on AutoCodeRover. Results are averaged over five independent runs and rounded to the nearest integer.

Top-N	AutoCodeRover	+LEV	+LEV+CSC	+LEV+CSC+ALI
Top-1	67	86	91	**93**
Top-3	83	112	114	**124**
Top-5	85	114	116	**131**

Table 7 presents the transferability results of our hallucination-mitigation mechanisms on AutoCodeRover. We observe that all three Top-N metrics improve consistently as more components are introduced. Compared with the original AutoCodeRover localization process, the full variant with LEV, CSC, and ALI increases the number of correctly localized instances from 67 to 93 in Top-1, from 83 to 124 in Top-3, and from 85 to 131 in Top-5. These correspond to relative improvements of 38.80%, 49.39%, and 54.11%, respectively, indicating that the benefits of our design are not limited to FaultLens itself.

Among the three components, LEV yields the largest immediate gain over the original AutoCodeRover configuration, increasing the number of correctly localized instances by 19, 29, and 29 in Top-1, Top-3, and Top-5,

respectively. This result suggests that invalid locations are a non-trivial source of error in AutoCodeRover's localization process, and that explicitly validating extracted locations is effective in reducing such failures.

After LEV is introduced, CSC brings additional but relatively modest improvements. Our manual inspection of the detailed results shows that, although CSC is also triggered frequently in AutoCodeRover, it rarely rejects all currently identified locations and therefore seldom sends the workflow back for substantially deeper investigation. In most cases, its role is to more carefully re-examine locations that were not sufficiently inspected earlier, which still improves localization quality but with a smaller overall effect.

Finally, adding ALI further improves the results, especially for Top-3 and Top-5, where it increases the number of correctly localized instances by 10 and 15, respectively, over AutoCodeRover+LEV+CSC. This observation is consistent with our findings in RQ2: once invalid and insufficiently inspected locations are better controlled, an additional identification step can further recover suspicious locations that may have been missed due to earlier reasoning bias.

Notably, these improvements are obtained on an external workflow. AutoCodeRover is embedded in an issue-driven repair pipeline and does not take failing test error information as its primary input, yet our hallucination-mitigation mechanisms remain effective after adaptation. This result provides evidence that the proposed mechanisms are not tied to a single failing-test-centered FL framework, but can transfer to another LLM-based repository-scale localization workflow with a different task setting.

> Answer to RQ3: The proposed hallucination-mitigation mechanisms also improve AutoCodeRover, an external repository-scale localization workflow embedded in an issue-driven repair pipeline. Compared with the original AutoCodeRover localization process, the full adapted variant improves Top-1, Top-3, and Top-5 by 38.80%, 49.39%, and 54.11%, respectively. These results provide evidence of transferability beyond FaultLens in one external issue-driven repository-scale localization workflow.

5.2.4 RQ4: How do different hyperparameter configurations influence FaultLens? In this RQ, we examine the impact of hyperparameter configurations on FaultLens. We focus on three types of hyperparameters: model temperature t, the upper limit of the tool invocation loop l, and the number of runs n. Starting with the original configuration of FaultLens as t=0.2, l=10, n=5, we experiment with variations by modifying only one hyperparameter at a time while keeping the others unchanged. For the model temperature t, we evaluate three values: t = 0.0, 0.2, and 0.8. Compared to the original setting, a temperature of 0.0 results in more stable model outputs, while a temperature of 0.8 introduces greater variability. Based on our previous observations, in over 95% of cases, FaultLens autonomously concludes investigations without reaching the upper limit of 10 iterations. Therefore, for the upper limit of the tool invocation loop l, we test values of l = 5 and l = 10. To analyze the impact of the number of runs, we compare the performance for n = 1, 3, and 5. Table 8 shows the performance of FaultLens across various hyperparameter settings.

Table 8. Performance of FaultLens across various hyperparameter settings

Settings			Results		
t	l	n	**Top-1**	**Top-3**	**Top-5**
0.0	10	5	204	253	270
0.2	10	5	205	259	278
0.8	10	5	197	256	281
0.2	5	5	200	254	271
0.2	10	1	195	244	246
0.2	10	3	200	258	270

Starting with the original configuration, we find that modifying the model temperature or the upper limit of the tool invocation loop has minimal impact on FaultLens's performance, with fluctuations not exceeding 3.91% across all metrics. Increasing the number of runs generally results in stable or improved performance. Notably, the improvement from 1 to 3 runs is more evident in the Top-3 and Top-5 metrics, while further increasing the number of runs to 5 yields diminishing returns. This suggests that FaultLens is capable of maintaining strong performance even with a limited number of runs, making it effective when constraints restrict the number of runs to 3. Moreover, even with just one run, FaultLens achieves remarkable Top-1 performance.

> Answer to RQ4: Changes to temperature and tool loops have minimal impact on FaultLens (<=3.91%). Performance improves notably from 1 to 3 runs in the Top-3 and Top-5 metrics, with diminishing gains beyond that. Even with a single run, strong Top-1 results are achieved.

5.2.5 RQ5: How effectively does FaultLens generalize to different programming languages? To evaluate the cross-language generalizability of FaultLens, we apply the original framework to a dataset of real-world Python bugs, using 354 successfully reproduced bugs from the BugsInPy benchmark. The distribution of bugs across projects is presented in Table 3.

While the overall framework of FaultLens remains unchanged, we make several adjustments to the codebase exploration tools to accommodate differences in programming paradigms: Java is strictly object-oriented, whereas Python supports both object-oriented and procedural styles. To handle cases where Python files do not contain any classes, we introduce a tool, `get_file_info(`*f*`)`, which provides an overview of a file *f*, including its package and import statements, the classes it contains (if any) along with their covered methods, and a list of standalone functions not belonging to any class. In addition, based on our prior investigation, inner classes are rarely used in Python. As a result, we do not provide a dedicated `get_inner_class_info(`*c*`, `*ic*`, [`*f*`=None])` tool for Python projects. In the **Fault Localization Decision stage**, FaultLens identifies suspicious locations by specifying the file, class, and method. However, since suspicious functions in Python may be standalone and not associated with any class, we ensure that all tools presenting method or function signatures also include their start and end lines within the file. For consistency, suspicious locations are identified using a unified format: file, method/function name, start line, and end line.

Table 9. Performance of AutoFL and FaultLens on the BugsInPy benchmark.

Top-N	AutoFL	FaultLens
Top-1	159	**172**
Top-3	220	**230**
Top-5	231	**237**

Table 9 presents the performance of AutoFL and FaultLens on the reproduced bugs from the BugsInPy benchmark. Compared to AutoFL, FaultLens improves the Top-1, Top-3, and Top-5 metrics by 8.17%, 4.54%, and 2.59%, respectively. These results indicate that FaultLens maintains performance that is better than or comparable to the state-of-the-art when applied to a different programming language.

To further examine whether the hallucination-related trends remain similar on Python bugs, we also analyze the execution traces of FaultLens on BugsInPy. The rates are not exactly the same as those on Defects4J. In particular, extrinsic-hallucination-triggering events remain non-negligible on BugsInPy, occurring in 65 out of 354 bugs (18.36%), although this rate is lower than the 154 out of 395 bugs (38.98%) observed on Defects4J. By contrast, conditional self-check is still triggered in 250 out of 354 bugs (70.62%), which is close to the 287 out of 395 bugs (72.65%) observed on Defects4J. This suggests that when transferred to Python, the agent is still prone to

identifying non-existent locations in a noticeable portion of cases, while premature termination of investigation before sufficiently inspecting all candidate method bodies remains common. Across the triggered cases, self-check removes 106 candidates, 89 of which are non-faulty according to the benchmark ground truth, yielding a precision of 83.96%. Advanced location identification further corrects 29 and 33 previously unsuccessful localization stages under Top-3 and Top-5, respectively, indicating that revisiting earlier reasoning remains beneficial beyond the Java benchmark.

> Answer to RQ5: FaultLens outperforms the LLM-based agent baseline on the BugsInPy benchmark, demonstrating its effectiveness in localizing faults in Python projects. The execution-trace analysis further shows that the hallucination-related issues targeted by FaultLens also appear in Python bugs: extrinsic hallucinations remain non-negligible, and premature termination of investigation remains common. Moreover, conditional self-check continues to remove non-faulty candidates, while advanced location identification recovers previously unsuccessful localization cases, indicating that the key mitigation mechanisms of FaultLens generalize beyond Java.

5.2.6 RQ6: How effectively does FaultLens generalize to different LLMs? To further assess the generalizability of FaultLens across different LLM backends, we evaluate it with four representative foundation models. GPT-4o (gpt-4o-2024-11-20) [15] and Qwen3-Max [47] are included as proprietary general-purpose models, while DeepSeek-V3 (DeepSeek-V3-0324) [30] and Devstral 2 [39] are included as open-weight models. Among them, Devstral 2 further represents a code-oriented backend. We keep the remaining settings unchanged so that the comparison focuses on the impact of the foundation model.

Table 10. Comparing Foundation Models for FaultLens on the Defects4J-V1.2.0 Benchmark

Settings				Results		
m	*t*	*l*	*n*	**Top-1**	**Top-3**	**Top-5**
GPT-4o	0.2	10	5	225	281	302
Qwen3-Max	0.2	10	5	269	319	325
DeepSeek-V3	0.2	10	5	237	298	319
Devstral 2	0.2	10	5	250	306	320

Table 10 shows that FaultLens remains effective across all four evaluated model backends. Replacing the default GPT-4o-mini backbone with stronger alternatives consistently improves FL performance. Among the evaluated models, Qwen3-Max achieves the best overall results, followed by Devstral 2 and DeepSeek-V3, while GPT-4o also yields clear gains over the default setting. These results suggest that FaultLens is not tied to a specific model provider or model family. Rather, it generalizes well across both proprietary and open-weight backends.

Table 11. Ablation study results with DeepSeek-V3 as the foundation model.

Top-N	w/o LEV & CSC	w/o CSC	w/o ALI	w/o All	w/o Ranking	FaultLens
Top-1	223	230	233	217	234	**237**
Top-3	266	287	287	265	292	**298**
Top-5	284	306	294	272	314	**319**

The results above show that FaultLens generalizes across different foundation models. A further question is whether the proposed decision-stage components remain beneficial after replacing the default backbone with

a stronger model. To examine this, we conduct an additional ablation study using DeepSeek-V3 as a stronger-than-default representative backbone. We choose DeepSeek-V3 for this repeated ablation because it provides a practical balance between model capability and experimental cost, while still representing a substantial upgrade over the default GPT-4o-mini setting. The ablation setting otherwise remains the same as in RQ2.

Table 11 shows that the decision-stage components of FaultLens remain effective under the stronger DeepSeek-V3 foundation model. Removing CSC leads to consistent degradation across all Top-N metrics, with performance decreasing by 2.95% in Top-1, 3.69% in Top-3, and 4.07% in Top-5. When LEV and CSC are removed together, the degradation becomes more substantial, reaching 5.90%, 10.73%, and 10.97% on Top-1, Top-3, and Top-5, respectively. These results indicate that, even with a stronger foundation model, validating extracted locations and enforcing additional self-checking still help reduce unsupported predictions and insufficient investigation.

Removing ALI also causes consistent performance degradation. The effect is limited on Top-1, where the decrease is 1.68%, but becomes more evident on Top-3 and Top-5, with reductions of 3.69% and 7.83%, respectively. This pattern is consistent with the findings in RQ2: without ALI, less obvious yet still suspicious locations are more likely to be missed, which primarily affects broader candidate sets.

When all three hallucination-mitigation components are removed, the degradation becomes considerably larger. Relative to the default configuration, performance decreases by 8.43% in Top-1, 11.07% in Top-3, and 14.73% in Top-5. This result further suggests that the three components remain complementary under DeepSeek-V3: LEV filters unsupported location predictions, CSC encourages more adequate inspection, and ALI provides an opportunity to recover from earlier reasoning errors. Overall, the benefits of the proposed hallucination-mitigation design are not confined to weaker foundation models, but remain stable after upgrading the foundation model.

Disabling the ranking step also results in consistent but smaller degradation, with reductions of 1.26%, 2.01%, and 1.56% on Top-1, Top-3, and Top-5, respectively. This observation is consistent with the ablation results in RQ2, suggesting that ranking helps refine the final output, whereas the more substantial gains come from the hallucination-mitigation components that improve candidate quality before ranking.

> Answer to RQ6: FaultLens generalizes well across diverse foundation models. It consistently benefits from stronger backbones and remains effective across both proprietary and open-weight models, including a code-oriented backend. Further ablation results on DeepSeek-V3 show that the proposed hallucination-mitigation components remain consistently beneficial even under a stronger backbone model. This indicates that the effectiveness of these components does not vanish as model capability improves, but continues to contribute to localization quality.

5.2.7 Failure Analysis of FaultLens. Although FaultLens achieves clear improvements in aggregate Top-N performance, we observed representative failure trajectories in which hallucinations still persist or in which later decision stages introduce new errors. Because LLM-based agent behavior can vary across runs, and FaultLens further reduces such variability through multi-run rank-based voting in its default setting, the following examples are representative individual-run failure modes rather than dominant outcomes of the overall framework.

Failure Trajectory 1: Conditional self-check is triggered but fails to remove a false positive. In one representative run, the agent identified a candidate whose method body had not been inspected during the tool-invocation loop, which correctly triggered the conditional self-check mechanism. However, after rechecking the code, the agent still retained this non-buggy method as suspicious, and the false positive remained in the final candidate list. This failure suggests that conditional self-check does not fully eliminate intrinsic hallucinations caused by incomplete investigation. A possible reason is that the self-check is still performed by the same model after preliminary location identification, and its judgment may therefore inherit earlier faulty assumptions formed during the initial reasoning process.

A potential mitigation is to make the self-check stage more evidence-grounded. For example, the model could be required to explicitly justify each retained candidate using concrete code evidence tied to the observed failure. Moreover, for ambiguous candidates, future work may aggregate self-check judgments across multiple agents or employ lightweight multi-agent discussion to reduce the impact of a single biased recheck.

Failure Trajectory 2: Advanced location identification introduces a false positive that outranks the true faulty method. In another representative run, the preliminary location identification had already included the true faulty method, but the advanced location identification stage introduced an additional false positive. During the final ranking stage, this newly introduced candidate was ranked above the true faulty method, thereby harming Top-1 performance even though the correct location was still present in the final list. This example shows that advanced location identification, while useful for recovering overlooked candidates, can sometimes introduce new ranking errors. A possible reason is that this stage reasons over the broader accumulated context and may overgeneralize from partial but plausible evidence.

A possible mitigation is to make the final ranking stage aware of how each candidate was produced. For instance, candidates newly introduced by advanced location identification could be assigned a lower prior confidence unless they are supported by stronger direct evidence than previously retained candidates. Future work may also introduce a secondary agent to perform external investigation and confirmation when a candidate newly introduced by advanced location identification is ranked above earlier validated candidates.

Overall, these failure trajectories indicate that the proposed hallucination-mitigation components substantially improve fault localization performance, but they do not fully eliminate residual false positives or ranking errors. At the same time, these examples should be interpreted together with the aggregate evaluation results, which show that FaultLens remains effective overall and that multi-run aggregation helps reduce the influence of unstable outcomes from individual runs.

6 Threats To Validity

One threat to internal validity is the nondeterminism of LLM inference, which can cause slight variations in results across different runs. To address this issue, we employ a rank-based voting strategy among multiple runs to reduce the impact of nondeterminism and achieve more reliable results. Additionally, our evaluation of temperature settings indicates that FaultLens remains relatively stable across different temperature values. Another internal validity concern is the potential for data leakage within LLMs, as the dataset being evaluated may be part of their training data. To minimize this risk, we avoid disclosing any information about project names, versions, bug IDs, or human-written bug reports. In addition, our ablation study reveals that an LLM-based agent lacking our designed components performs poorly compared to FaultLens, indicating that LLMs cannot directly memorize all ground-truth data. The accuracy improvements achieved by FaultLens demonstrate the effectiveness of our methodology.

The primary threat to external validity arises from the benchmarks, target workflows, and foundation models used in our evaluation. Although we assess FaultLens on two real-world benchmarks, Defects4J for Java and BugsInPy for Python, evaluate it with both proprietary and open-weight foundation models, and provide initial evidence of transferability by adapting the proposed hallucination-mitigation mechanisms to the fault-localization stage of AutoCodeRover on SWE-bench-Lite, these settings still cannot fully cover the diversity of programming languages, project types, LLM architectures, and repository-scale localization workflows encountered in practice. Evaluating additional model backends would require substantial computation and API costs, while extending FaultLens and its mitigation mechanisms to more languages, benchmarks, and target workflows would require additional engineering effort. We leave broader evaluations in these directions for future work.

Applicability and limitations. FaultLens is designed for repository-scale fault localization in settings where at least one failing test can be reproduced. In our current implementation, codebase exploration is guided by

test coverage information to focus the investigation on code that is more directly related to the observed failure. In each run, FaultLens operates on one selected failing test and currently localizes faults at the method level. The current implementation further assumes source code that can be parsed by our syntax-aware tooling and repositories that can be checked out and analyzed. Even without collected coverage information, the overall framework can still execute based on the failing test and error information alone, although such guidance helps narrow the search space and improve efficiency.

7 Related Work

7.1 Fault Localization

Spectrum-based FL (SBFL) [2–4, 19, 48, 57, 69] is one of the most prevalent traditional FL techniques. SBFL relies on coverage data gathered from test case executions to assess the likelihood of each program element being faulty. For each program element e, the number of failed test cases that cover it, $N_f(e)$, or do not cover it, $N_f(\bar{e})$, as well as the number of passed test cases that cover it, $N_p(e)$, or do not cover it, $N_p(\bar{e})$, are commonly used in suspiciousness calculations. There are numerous SBFL formulae extensively used in debugging, including Tarantula [19], Ochiai [3], Jaccard [4], and Dstar [57]. For example, in the Tarantula formula, the suspiciousness of each statement is $N_f(e)/N_f/(N_f(e)/N_f + N_p(e)/N_p)$, where N_f and N_p refer to the total number of failing and passing test cases, respectively. Mutation-based FL (MBFL) [40, 43, 70] uses mutation analysis to tackle the issue of multiple program elements receiving identical suspiciousness scores, which is caused by coarse coverage information. MBFL utilizes mutants to evaluate how program elements influence test case results in order to calculate suspiciousness values. For instance, Metallaxis [43] treats mutants that affect test outcomes as covered by the tests, while those without impact are considered uncovered. It then calculates the suspiciousness of each mutant based on SBFL formulae. However, MBFL incurs the cost of mutation testing and is ineffective for program elements where mutants cannot be generated.

In recent years, a variety of learning-based methods [6, 26, 27, 34, 50, 63, 72] have been developed to enhance SBFL by incorporating diverse features. FLUCCS [50] combines multiple code and change metrics, using Genetic Programming (GP) as its learning mechanism. CNNFL [72] leverages convolutional neural networks (CNN) [23] to process execution information for statement suspiciousness calculations, while DeepFL [26] integrates multi-dimensional features, training distinct neural networks for fault localization. GRACE [34] utilizes detailed coverage information through a Gated Graph Neural Network (GGNN) [29] model. Numerous other learning-based FL techniques [14, 28, 73] combine insights from traditional FL approaches with traditional machine-learning or deep-learning methods. Although these approaches often achieve performance gains, they typically depend on supervised training with project-specific data, which can limit scalability, adaptability, and practical use. Therefore, our evaluation includes only one representative learning-based technique, FLUCCS.

With the advent of LLMs, these models have demonstrated remarkable capabilities in programming tasks, such as code generation [12, 33, 44] and comprehension [18, 67]. Recent research [55, 60, 64] has focused on applying LLMs to FL tasks. Wu et al. [60] provide ChatGPT [42] with method-level context and test failure information. Yang et al. [64] fine-tune bidirectional adapter layers on top of CodeGen's learned representations [41]. Widyasari et al. [55] employ LLMs to generate step-by-step reasoning. However, these approaches are limited by the restricted input length in their settings, making them unsuitable for repository-scale FL tasks that require more comprehensive codebase details as input. The use of LLM-based agents, organized into perception, brain, and action modules, helps mitigate this limitation. Recent work [21, 53] has begun to employ tool-equipped agents for FL tasks. AutoFL [21] leverages tool invocations to identify failure root causes and locate defective methods, representing an early tool-augmented LLM-based approach to repository-scale fault localization. FaultLens shares this high-level setting, but differs in its technical focus. Rather than only relying on codebase investigation and direct localization output, FaultLens introduces an explicit hallucination-aware decision stage,

including location extraction validation, conditional self-check, and advanced location identification, to address invalid locations, insufficiently inspected candidates, and reasoning errors during localization. We present these differences not merely as implementation variations over AutoFL, but as a different design emphasis: improving repository-scale fault localization by explicitly modeling and mitigating hallucination-related failure modes. In addition, FaultLens does not reuse AutoFL's exploration layer. Instead, we design a set of repository exploration tools to support more flexible codebase investigation. This design allows the agent to initiate investigation from file-, class-, or method-level context as needed, instead of following a strongly pre-structured exploration path. Similar to AutoFL, RCAgent [53] uses tool invocations for FL in cloud computing environments. SoapFL [46] is a multi-agent FL system that follows a standard operating procedure guided by LLMs to localize buggy methods in a repository. It organizes the FL process into three fixed phases: fault comprehension, codebase navigation, and fault confirmation, with tools assigned to specific agents in each phase. Unlike SoapFL, FaultLens allows the agent to invoke codebase exploration tools during investigation on demand. Furthermore, FaultLens repeats the same agent configuration across multiple independent low-temperature runs and aggregates the resulting ranked candidate lists to reduce the impact of LLM nondeterminism, thereby improving result stability.

7.2 LLMs in Broader Software Engineering Applications

With the rapid advancement of LLMs, these models have been increasingly adopted across a wide range of software engineering tasks beyond fault localization. LLM-powered approaches have been applied to code generation, test case generation, program repair, and more, demonstrating their potential to automate and augment various stages of the software development lifecycle.

Code generation is a core application of LLMs in software engineering. SYNCHROMESH [44] improves generation reliability through semantic example selection and constrained semantic decoding. Broader evaluations [68] have systematically benchmarked pre-trained models across diverse code generation tasks, revealing both their strengths and limitations. Recently, library-oriented code generation has attracted increasing attention, where models are required to generate code that adheres to specific library usage. CodeGen4Libs [32] addresses this challenge via a two-stage process that handles import prediction and code generation separately. CAPIR [36] further improves generation quality by decomposing coarse-grained requirements and retrieving relevant APIs in a compositional manner. The popularity of ChatGPT has also sparked a surge of interest in LLM-based code generation, leading to a growing body of follow-up studies [12, 24, 33].

Recent studies have explored the use of LLMs for automated test case generation. CODET [10] leverages pre-trained models to generate test cases for verifying candidate code completions and selecting the most reliable outputs, significantly improving generation accuracy. LIBRO [22] focuses on generating executable test cases directly from natural language bug reports, highlighting the potential of LLMs in bug reproduction tasks. To address challenges in reasoning and correctness, TestChain [25] introduces a multi-agent framework that decouples input/output generation and incorporates interpreter feedback, resulting in higher-quality test cases from standalone LLMs.

LLMs have also been adopted for APR, offering greater flexibility compared to traditional pattern- or rule-based approaches. ChatRepair [62] introduces a conversational APR paradigm, where the LLM iteratively generates patches based on prior attempts and test failure feedback. AutoCodeRover [71] enhances repair effectiveness by combining LLMs with structural code search over abstract syntax trees to resolve GitHub issues. MAGIS [51] further proposes a multi-agent framework that assigns specialized roles to agents for collaborative planning and patch generation, demonstrating the potential of structured coordination among LLMs in APR tasks.

Unlike the above studies that primarily focus on generation-based tasks, such as producing code, test cases, or patches, our work tackles fault localization, which aims to identify suspicious locations in the codebase that are likely responsible for observed failures. FaultLens leverages codebase exploration tools to investigate root

causes, and incorporates specially designed components for hallucination mitigation to achieve more precise localization.

8 Conclusion

In this work, we present FaultLens, an LLM-based agent FL technique with a decision-making stage that incorporates fine-grained feedback. By integrating location extraction validation, a conditional self-check mechanism, and advanced location identification, FaultLens effectively mitigates both extrinsic and intrinsic hallucinations, thereby enhancing FL accuracy. Our evaluation shows that FaultLens outperforms multiple FL techniques across different categories, with up to a 43.35% improvement in Top-1 accuracy over LLM-based agent approaches. Beyond FaultLens itself, our external workflow study further shows that the proposed hallucination-mitigation mechanisms can be transferred to the fault-localization stage of AutoCodeRover, providing additional evidence of their applicability in repository-scale localization workflows. In addition, experiments on a Python benchmark and different LLM backends show that FaultLens maintains strong performance across different programming languages and model providers, indicating its potential generalizability. A promising direction for future work is to investigate how more accurate repository-scale fault localization may benefit downstream software maintenance tasks.

Acknowledgments

We sincerely appreciate the anonymous reviewers for their valuable feedback, which significantly enhanced this paper. This research was partially supported by the National Natural Science Foundation of China (Grant No. 62302304) and the ShanghaiTech Startup Funding.

References

[1] 2026. Tree-sitter. https://tree-sitter.github.io/tree-sitter/. Accessed: 2026-04-02.

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE '09). IEEE Computer Society, USA, 88–99. doi:10.1109/ASE.2009.25

[3] Rui Abreu, Peter Zoeteweij, and Arjan J.c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). 39–46. doi:10.1109/PRDC.2006.18

[4] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007). 89–98. doi:10.1109/TAIC.PART.2007.13

[5] Faustino Aguilar, Samuel Grayson, and Darko Marinov. 2023. Reproducing and Improving the BugsInPy Dataset. In 2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM). IEEE, 260–264.

[6] 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.

[7] Ned Batchelder. 2025. coverage.py. https://coverage.readthedocs.io/en/7.8.0/. Accessed: 2025-04-02.

[8] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (Atlanta, Georgia) (SIGSOFT '08/FSE-16). Association for Computing Machinery, New York, NY, USA, 308–318. doi:10.1145/1453101.1453146

[9] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. GZoltar: an eclipse plug-in for testing and debugging. In 2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. 378–381. doi:10.1145/2351676.2351752

[10] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. arXiv preprint arXiv:2207.10397 (2022).

[11] Zhaoling Chen, Robert Tang, Gangda Deng, Fang Wu, Jialong Wu, Zhiwei Jiang, Viktor Prasanna, Arman Cohan, and Xingyao Wang. 2025. LocAgent: Graph-Guided LLM Agents for Code Localization. In Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 8697–8727. doi:10.18653/v1/2025.acl-long.426

[12] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. 2024. Self-Collaboration Code Generation via ChatGPT. ACM Trans. Softw. Eng. Methodol. 33, 7, Article 189 (Sept. 2024), 38 pages. doi:10.1145/3672459

[13] Google. 2025. Closure Compiler. https://github.com/google/closure-compiler. Accessed: 2025-04-02.

[14] Xiaodong Gou, Ao Zhang, Chengguang Wang, Yan Liu, Xue Zhao, and Shunkun Yang. 2024. Software Fault Localization Based on Network Spectrum and Graph Neural Network. IEEE Transactions on Reliability (2024), 1–15. doi:10.1109/TR.2024.3374410

[15] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. arXiv preprint arXiv:2410.21276 (2024).

[16] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. ACM Comput. Surv. 55, 12, Article 248 (March 2023), 38 pages. doi:10.1145/3571730

[17] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world GitHub Issues?. In The Twelfth International Conference on Learning Representations (ICLR). https://openreview.net/forum?id=VTF8yNQM66

[18] Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. 2023. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. arXiv preprint arXiv:2312.09601 (2023).

[19] James A Jones, Mary Jean Harrold, and John T Stasko. 2001. Visualization for fault localization. In in Proceedings of ICSE 2001 Workshop on Software Visualization. Citeseer.

[20] 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 (San Jose, CA, USA) (ISSTA 2014). Association for Computing Machinery, New York, NY, USA, 437–440. doi:10.1145/2610384.2628055

[21] 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 (July 2024), 23 pages. doi:10.1145/3660771

[22] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE). IEEE, 2312–2323.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. Advances in neural information processing systems 25 (2012).

[24] Jia Li, Ge Li, Yongmin Li, and Zhi Jin. 2024. Structured Chain-of-Thought Prompting for Code Generation. ACM Trans. Softw. Eng. Methodol. (Aug. 2024). doi:10.1145/3690635 Just Accepted.
[25] Kefan Li and Yuan Yuan. 2024. Large language models as test case generators: Performance evaluation and enhancement. arXiv preprint arXiv:2404.13340 (2024).
[26] 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 (Beijing, China) (ISSTA 2019). Association for Computing Machinery, New York, NY, USA, 169–180. doi:10.1145/3293882.3330574
[27] Xia Li and Lingming Zhang. 2017. Transforming programs and tests in tandem for fault localization. Proc. ACM Program. Lang. 1, OOPSLA, Article 92 (Oct. 2017), 30 pages. doi:10.1145/3133916
[28] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. Fault Localization with Code Coverage Representation Learning. In Proceedings of the 43rd International Conference on Software Engineering (Madrid, Spain) (ICSE '21). IEEE Press, 661–673. doi:10.1109/ICSE43902.2021.00067
[29] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. 2016. Gated Graph Sequence Neural Networks. In Proceedings of ICLR'16.
[30] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. arXiv preprint arXiv:2412.19437 (2024).
[31] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (Beijing, China) (ISSTA 2019). Association for Computing Machinery, New York, NY, USA, 31–42. doi:10.1145/3293882.3330577
[32] Mingwei Liu, Tianyong Yang, Yiling Lou, Xueying Du, Ying Wang, and Xin Peng. 2023. Codegen4libs: A two-stage approach for library-oriented code generation. In 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 434–445.
[33] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. 2024. No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. IEEE Transactions on Software Engineering 50, 6 (2024), 1548–1584. doi:10.1109/TSE.2024.3392499
[34] 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 (Athens, Greece) (ESEC/FSE 2021). Association for Computing Machinery, New York, NY, USA, 664–676. doi:10.1145/3468264.3468580
[35] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: combining context-aware neural translation models using ensemble for program repair. In Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (Virtual Event, USA) (ISSTA 2020). Association for Computing Machinery, New York, NY, USA, 101–114. doi:10.1145/3395363.3397369
[36] Zexiong Ma, Shengnan An, Bing Xie, and Zeqi Lin. 2024. Compositional API recommendation for library-oriented code generation. In Proceedings of the 32nd IEEE/ACM International Conference on Program Comprehension. 87–98.
[37] Matias Martinez and Martin Monperrus. 2016. ASTOR: a program repair library for Java (demo). In Proceedings of the 25th International Symposium on Software Testing and Analysis (Saarbrücken, Germany) (ISSTA 2016). Association for Computing Machinery, New York, NY, USA, 441–444. doi:10.1145/2931037.2948705
[38] Joshua Maynez, Shashi Narayan, Bernd Bohnet, and Ryan McDonald. 2020. On Faithfulness and Factuality in Abstractive Summarization. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, Dan Jurafsky, Joyce Chai, Natalie Schluter, and Joel Tetreault (Eds.). Association for Computational Linguistics, Online, 1906–1919. doi:10.18653/v1/2020.acl-main.173
[39] Mistral AI. 2025. Introducing: Devstral 2 and Mistral Vibe CLI. https://mistral.ai/news/devstral-2-vibe-cli. Accessed: 2025-12-22.
[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. 153–162. doi:10.1109/ICST.2014.28
[41] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint arXiv:2203.13474 (2022).
[42] OpenAI. 2025. ChatGPT. https://openai.com/chatgpt/. Accessed: 2025-04-02.
[43] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: mutation-based fault localization. Softw. Test. Verif. Reliab. 25, 5–7 (Aug. 2015), 605–628. doi:10.1002/stvr.1509
[44] Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. arXiv preprint arXiv:2201.11227 (2022).
[45] Yujia Qin, Shengding Hu, Yankai Lin, Weize Chen, Ning Ding, Ganqu Cui, Zheni Zeng, Yufei Huang, Chaojun Xiao, Chi Han, et al. 2023. Tool Learning with Foundation Models. CoRR abs/2304.08354 (2023).
[46] Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2025. S oap FL: A Standard Operating Procedure for LLM-based Method-Level Fault Localization. IEEE Transactions on Software Engineering (2025).
[47] Qwen Team. 2025. Qwen3-Max: Just Scale it. https://qwen.ai/blog?id=qwen3-max. Accessed: 2025-12-22.

[48] Sofia Reis, Rui Abreu, and Marcelo d'Amorim. 2019. Demystifying the Combination of Dynamic Slicing and Spectrum-based Fault Localization.. In IJCAI. 4760–4766.
[49] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2024. Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems 36 (2024).
[50] 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 (Santa Barbara, CA, USA) (ISSTA 2017). Association for Computing Machinery, New York, NY, USA, 273–283. doi:10.1145/3092703.3092717
[51] Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. 2024. Magis: Llm-based multi-agent framework for github issue resolution. Advances in Neural Information Processing Systems 37 (2024), 51963–51993.
[52] Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. 2024. A survey on large language model based autonomous agents. Frontiers of Computer Science 18, 6 (2024), 186345.
[53] Zefan Wang, Zichuan Liu, Yingying Zhang, Aoxiao Zhong, Jihong Wang, Fengbin Yin, Lunting Fan, Lingfei Wu, and Qingsong Wen. 2024. Rcagent: Cloud root cause analysis by autonomous agents with tool-augmented large language models. In Proceedings of the 33rd ACM International Conference on Information and Knowledge Management. 4966–4974.
[54] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2024. Chain-of-thought prompting elicits reasoning in large language models. In Proceedings of the 36th International Conference on Neural Information Processing Systems (New Orleans, LA, USA) (NIPS '22). Curran Associates Inc., Red Hook, NY, USA, Article 1800, 14 pages.
[55] Ratnadira Widyasari, Jia Wei Ang, Truong Giang Nguyen, Neil Sharma, and David Lo. 2024. Demystifying faulty code: Step-by-step reasoning for explainable fault localization. In 2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, 568–579.
[56] Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. 2020. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering. 1556–1560.
[57] W. Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. 2014. The DStar Method for Effective Software Fault Localization. IEEE Transactions on Reliability 63, 1 (2014), 290–308. doi:10.1109/TR.2013.2285319
[58] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. IEEE Trans. Softw. Eng. 42, 8 (Aug. 2016), 707–740. doi:10.1109/TSE.2016.2521368
[59] Michael Wooldridge and Nicholas R Jennings. 1995. Intelligent agents: Theory and practice. The knowledge engineering review 10, 2 (1995), 115–152.
[60] 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).
[61] Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, et al. 2023. The rise and potential of large language model based agents: A survey. arXiv preprint arXiv:2309.07864 (2023).
[62] Chunqiu Steven Xia and Lingming Zhang. 2024. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using ChatGPT. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 819–831.
[63] Jifeng Xuan and Martin Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In 2014 IEEE International Conference on Software Maintenance and Evolution. 191–200. doi:10.1109/ICSME.2014.41
[64] 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. doi:10.1145/3597503.3623342
[65] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. arXiv preprint arXiv:2210.03629 (2022).
[66] Jiayi Yuan, Hao Li, Xinheng Ding, Wenya Xie, Yu-Jhe Li, Wentian Zhao, Kun Wan, Jing Shi, Xia Hu, and Zirui Liu. 2025. Understanding and Mitigating Numerical Sources of Nondeterminism in LLM Inference. In The Thirty-ninth Annual Conference on Neural Information Processing Systems.
[67] Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. arXiv preprint arXiv:2308.01240 (2023).
[68] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In Proceedings of the 31st ACM SIGSOFT international symposium on software testing and analysis. 39–51.
[69] Lingming Zhang, Miryung Kim, and Sarfraz Khurshid. 2011. Localizing failure-inducing program edits based on spectrum information. In Proceedings of the 2011 27th IEEE International Conference on Software Maintenance (ICSM '11). IEEE Computer Society, USA, 23–32. doi:10.1109/ICSM.2011.6080769

[70] Lingming Zhang, Lu Zhang, and Sarfraz Khurshid. 2013. Injecting mechanical faults to localize developer faults for evolving software. SIGPLAN Not. 48, 10 (Oct. 2013), 765–784. doi:10.1145/2544173.2509551

[71] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. 1592–1604.

[72] 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). 445–455. doi:10.1109/SANER.2019.8668002

[73] Zhuo Zhang, Yan Lei, Xiaoguang Mao, Meng Yan, Xin Xia, and David Lo. 2023. Context-Aware Neural Fault Localization. IEEE Transactions on Software Engineering 49, 7 (2023), 3939–3954. doi:10.1109/TSE.2023.3279125