2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)

Explainable Fault Localization for Programming Assignments via LLM-Guided Annotation

Fang Liu*, Tianze Wang*, Li Zhang, Zheyu Yang, Jing Jiang†, Zian Sun
State Key Laboratory of Complex & Critical Software Environment, School of Computer Science and Engineering
Beihang University, Beijing, China
Email: {fangliu, wangtz, lily, jiangjing, sza}@buaa.edu.cn, yangzheyu00@hotmail.com

***Abstract*—Providing timely and personalized guidance for students' programming assignments, particularly by indicating fine-grained error locations with explanations, offers significant practical value for helping students complete assignments and enhance their learning outcomes. In recent years, various automated Fault Localization (FL) techniques, particularly those leveraging Large Language Models (LLMs), have demonstrated promising results in identifying errors in programs. However, existing fault localization techniques face challenges when applied to educational contexts. Most approaches operate at the method level without explanatory feedback, resulting in granularity too coarse for students who need actionable insights to identify and fix their errors. While some approaches attempt line-level fault localization, they often depend on predicting line numbers directly in numerical form, which is ill-suited to LLMs. To address these challenges, we propose FLAME, a fine-grained, explainable Fault Localization method tailored for programming assignments via LLM-guided Annotation and Model Ensemble. FLAME leverages rich contextual information specific to programming assignments to guide LLMs in identifying faulty code lines. Instead of directly predicting line numbers, we prompt the LLM to annotate faulty code lines with detailed explanations, enhancing both localization accuracy and educational value. To further improve reliability, we introduce a weighted multi-model voting strategy that aggregates results from multiple LLMs to determine the suspiciousness of each code line. Extensive experimental results demonstrate that FLAME outperforms state-of-the-art fault localization baselines on programming assignments, successfully localizing 207 more faults at top-1 over the best-performing baseline. Beyond educational contexts, FLAME also generalizes effectively to general-purpose software codebases, outperforming all baselines on the Defects4J benchmark.**

***Index Terms*—automated fault localization, programming education, large language models**

I. Introduction

Programming assignments are essential for students to develop coding skills. Since manual grading of these assignments is a time-consuming and error-prone process, Online Judge (OJ) systems are widely employed for automated evaluation [10, 14, 5, 17]. To complete an assignment, students submit their programs to an online judge system, which automatically executes the programs against a suite of predefined tests. If the program fails any test, students are expected to independently identify and fix the errors in their code before resubmitting. However, the feedback from such systems is often overly simplistic—commonly limited to verdicts such as "Wrong Answer" or "Runtime Error" [40, 41]—which poses significant challenges for less skilled students, who may struggle with identifying the errors, let alone fixing them [36]. Some students may even become discouraged and give up on the assignment altogether, putting them at risk of failing the course. Given that instructors are often responsible for hundreds or even thousands of students [26], it is impractical for them to provide timely and personalized guidance to each individual. Therefore, offering automated feedback with more informative insights—particularly by indicating fine-grained error locations with explanations—offers significant practical value for helping students complete assignments and enhance their learning outcomes [24].

Automated fault localization techniques offer promising avenues for addressing this challenge [7, 15, 16, 28]. *Fault Localization (FL)* is a critical activity within the software debugging process, aiming to identify the specific program elements (*e.g.*, lines, methods, classes, or files) that cause observed program failures [44]. Over the years, various automated fault localization techniques have been proposed. One prominent paradigm is *Spectrum-Based Fault Localization (SBFL)* [3, 4, 43], which postulates that program elements executed more frequently by failing tests than by passing tests are more likely to be faulty. Another notable paradigm is *Learning-Based Fault Localization (LBFL)* [8, 16, 25, 27], which leverages machine learning and deep learning techniques to improve fault localization. Most recently, the rise of Large Language Models (LLMs) has further advanced this field. Several studies have explored fault localization using LLMs, demonstrating promising results [47, 35, 42, 46].

Among the existing approaches, most operate at the method level and are evaluated on the Defects4J [18] benchmark. For example, Xu et al. [46] present a two-stage LLM-based fault localization framework, FlexFL, which leverages existing fault localization techniques for search space reduction and LLMs for localization refinement. Only a few approaches attempt line-level fault localization. LLMAO [47] fine-tunes lightweight adapters on top of LLMs to predict buggy lines. FuseFL [42] combines SBFL results, test outcomes, and code descriptions to guide LLMs in predicting buggy lines and generating explanations.

However, despite the existence of various fault localization

*Fang Liu and Tianze Wang contributed equally to this paper.
†Corresponding author.

techniques, most are ineffective when applied to educational contexts due to the following challenges: ❶ **Coarse localization granularity**: Most existing techniques operate at the method level [35, 46], with relatively few supporting line-level localization. Such coarse granularity is often insufficient for students, who need more actionable insights to identify and fix their errors. ❷ **Inaccurate faulty line prediction**: While some techniques attempt line-level fault localization, most of them depend on predicting line numbers directly in numerical form [45, 42], which is particularly ill-suited to LLMs. Due to the inherent limitations of LLMs in numerical understanding and processing [48], line numbers generated in this way are frequently inaccurate, making such techniques unreliable in practice. ❸ **Lack of explainability**: Most existing techniques simply identify fault locations without providing explanations, limiting their educational value. ❹ **Artifact availability gap**: Some techniques depend on software development artifacts such as version control histories or bug reports [37, 46], which are generally unavailable in educational contexts. Some others utilize documentation or code comments [19], which are often missing, incomplete, or unreliable in student programs due to their highly inconsistent code quality and limited development maturity [20, 33], especially among less skilled students. In contrast, artifacts specific to programming assignments, such as problem statements and reference programs, remain largely underutilized despite their potential to enhance fault localization effectiveness.

In this paper, we propose FLAME, a *fine-grained*, *explainable* fault localization method tailored for programming assignments. Specifically, FLAME empowers LLMs to identify faults in programming assignments by leveraging rich contextual information, including problem statements, test outcomes, and reference programs retrieved from historical submissions. To enhance fault localization accuracy and educational value, rather than predicting line numbers directly in numerical form, we prompt the LLM to annotate faulty lines in the program with detailed natural language explanations. To further improve the reliability of localization results, we perform multiple rounds of fault location annotation using different LLMs, and then aggregate the results to determine the suspiciousness of each line via a weighted voting strategy.

To assess the effectiveness of FLAME on programming assignments, we introduce PADEFECTS, a dataset encompassing both single-file and project-level submissions from two programming courses. Experimental results show that FLAME outperforms previous state-of-the-art fault localization baselines on PADEFECTS, localizing 207 more faults at top-1 over the best-performing baseline [45], demonstrating its effectiveness in educational contexts. We further evaluate FLAME on the Defects4J [18] benchmark, where it achieves state-of-the-art performance, confirming its generalization capability to general-purpose software codebases beyond educational contexts. Additionally, experiments reveal that FLAME's fault localization results can provide valuable guidance for automated program repair.

Our main contributions are as follows:

- We introduce PADEFECTS, a new dataset consisting of 1,932 faulty-fixed pairs of student submissions from two programming courses collected from an online judge system, encompassing both single-file and project-level submissions.
- We propose FLAME, a fine-grained, explainable fault localization method tailored for programming assignments, which supports line-level fault localization with detailed natural language explanation for each identified faulty line.
- We conduct a comprehensive evaluation of FLAME, demonstrating its superior performance compared to state-of-the-art fault localization methods across both programming assignments and general-purpose software codebases. The results highlight FLAME's strong generalization capability and practical applicability. To facilitate further research, the code and data are available at https://github.com/FLAME-FL/FLAME.

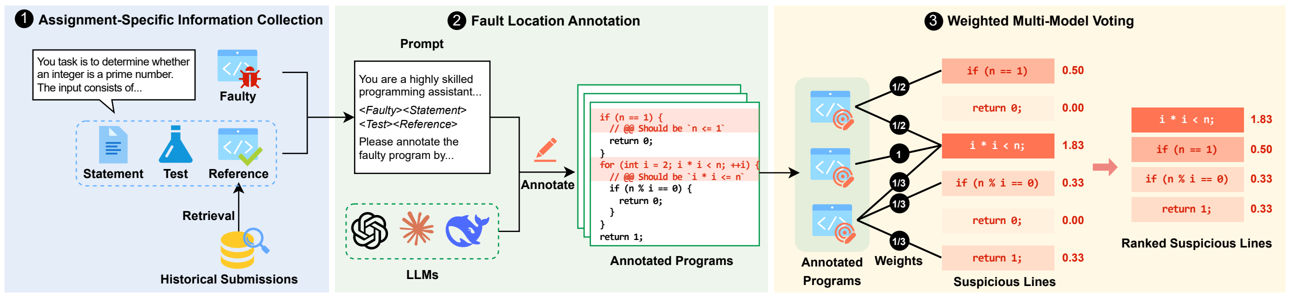
II. Related Work

A. Fault Localization Techniques for Programming Assignments

Several fault localization techniques are designed for programming assignments. Araujo et al. [7] conducted an empirical study on the application of SBFL to novice programs, highlighting its effectiveness while also showing its limitations in certain scenarios. NBL [16] combined convolutional neural networks with prediction attribution techniques to localize buggy lines in student programs. FFL [28] used graph-based syntax and semantic reasoning with graph neural networks to localize buggy statements in student programs. FuseFL [42] combined SBFL results, test results, and code descriptions to guide LLMs in localizing the fault and generating step-by-step reasoning about the fault. These techniques primarily focus on single-file submissions with no more than a few hundred lines of code, and may not scale effectively to project-level submissions with thousands of lines of code. Moreover, the two learning-based techniques (NBL [16] and FFL [28]) necessitate effort in preparing labeled datasets for supervised training, which hinders their broader adoption.

B. Fault Localization Techniques for General-Purpose Software Codebases

Most existing fault localization techniques are designed for general-purpose software codebases. Among these, Learning-Based Fault Localization (LBFL), especially deep learning-based techniques, has gained prominence over the years. CNN-FL [49] and DeepRL4FL [22] treated fault localization as a pattern recognition task, using convolutional neural networks to learn discriminative representations of code coverage matrices. DeepFL [21] incorporated various suspiciousness-value-based, fault-proneness-based, and textual-similarity-based features to train deep models. GRACE [25] used graph neural networks to learn from graph-based coverage representations, enabling better utilization of coverage information. LLMAO [47]



You task is to determine whether an integer is a prime number. The input consists of...

You are a highly skilled programming assistant... *<Faulty><Statement> <Test><Reference>* Please annotate the faulty program by...

```
if (n == 1) {
  // @@ Should be `n <= 1`
  return 0;
}
for (int i = 2; i * i < n; ++i) {
  // @@ Should be `i * i <= n`
  if (n % i == 0) {
    return 0;
  }
}
return 1;
```

```
if (n == 1)          0.50
return 0;            0.00
i * i < n;           1.83
if (n % i == 0)      0.33
return 0;            0.00
return 1;            0.33
```

```
i * i < n;           1.83
if (n == 1)          0.50
if (n % i == 0)      0.33
return 1;            0.33
```

Fig. 1. Overview of FLAME, including ❶ Assignment-Specific Information Collection, ❷ Fault Location Annotation, and ❸ Weighted Multi-Model Voting.

fine-tuned lightweight bidirectional adapters on top of pre-trained left-to-right language models to directly predict faulty lines without relying on test coverage information. Due to the nature of the learning algorithms used in these techniques, their fault localization results are not explainable.

Most recently, the rise of Large Language Models (LLMs) has further advanced this field. Several studies have explored fault localization using LLMs, demonstrating promising results. Wu et al. [45] conducted a comprehensive evaluation of LLMs (specifically ChatGPT-3.5 and ChatGPT-4) on fault localization, examining their sensitivity to prompt design and context granularity. AgentFL [35] scaled LLM-based fault localization to project-level contexts using a multi-agent system that simulates human debugging behavior. AutoFL [19] equipped LLMs with function-calling capabilities to autonomously navigate large repositories and generate explanations along with suggested fault locations. FlexFL [46] enabled flexible and effective fault localization through a two-stage LLM-based framework leveraging existing fault localization techniques for search space reduction and open-source LLMs for localization refinement. All these techniques operate at the method-level, and none of them address the explainability of fault localization except AutoFL [19].

To sum up, most existing techniques are ineffective when applied to educational contexts due to their coarse localization granularity and lack of explainability. Moreover, these techniques are primarily designed for general-purpose software codebases and do not take advantage of artifacts specific to programming assignments, further limiting their effectiveness.

III. Methodology

We propose FLAME, a *fine-grained*, *explainable* fault localization method for programming assignments based on large language models. An overview of FLAME is illustrated in Figure 1. Given a faulty program $P = (\ell_1, \ell_2, \ldots, \ell_n)$, where ℓ_i is the i-th line of the program, FLAME begins by collecting assignment-specific auxiliary information I, such as the problem statement, test outcomes, and reference program. Then both the faulty program P and the assignment-specific auxiliary information I are used to prompt the LLM to determine the suspiciousness of each line in P, finally producing a ranked sequence of suspicious lines $S = (\ell'_1, \ell'_2, \ldots, \ell'_n)$ with our proposed weighted voting strategy, ordered by their suspiciousness scores $s(\ell)$ in descending order. Specifically, our method consists of three stages:

- **Assignment-Specific Information Collection**: To provide critical hints for understanding program semantics and narrow down potential fault candidates, we collect assignment-specific auxiliary information such as problem statements, test outcomes, and reference programs, and incorporate them into the prompt to enhance LLM's fault localization effectiveness.
- **Fault Location Annotation**: Instead of predicting line numbers directly in numerical form, we prompt the LLM to annotate faulty lines in the program with detailed natural language explanations, ensuring both localization accuracy and educational value.
- **Weighted Multi-Model Voting**: To mitigate single-model biases and improve reliability, we perform multiple rounds of fault location annotation using different LLMs, then aggregate the results via a weighted voting strategy that prioritizes consensus predictions, producing a more robust final ranking of suspicious lines.

Below we present the details of each stage.

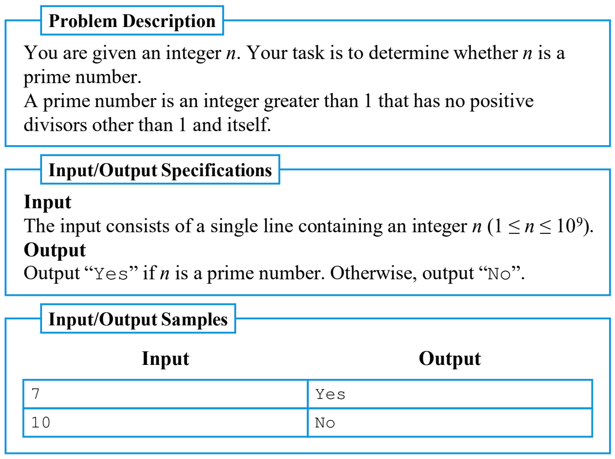
A. Assignment-Specific Information Collection

We collect the following assignment-specific auxiliary information I to enhance LLM's fault localization effectiveness, which provides critical insights into program semantics and narrows the scope of potential fault candidates.

1) Problem Statement: The *statement* of a problem comprehensively and clearly defines the expected behavior of a program, which is valuable for fault localization. As shown in Figure 2, it consists of *problem description*, *input/output specifications*, and *input/output samples*:

- The *problem description* defines the problem that needs to be solved.
- The *input/output specifications* define the expected input and output format.
- The *input/output samples* provide one or more sample inputs and their expected outputs, possibly with explanations, aiding in understanding the problem.

2) Test Outcomes: Online judge systems evaluate programs by executing them against a suite of predefined *tests*. For each test, the system issues a *verdict*, which is typically one of *Accepted (AC)*, *Wrong Answer (WA)*, *Runtime Error (RE)*,



Problem Description

You are given an integer n. Your task is to determine whether n is a prime number.
A prime number is an integer greater than 1 that has no positive divisors other than 1 and itself.

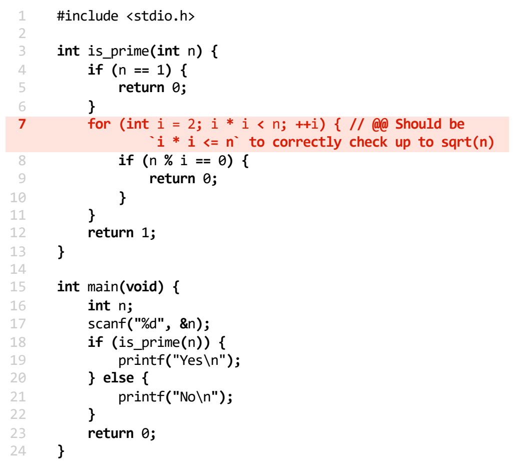
Input/Output Specifications

Input
The input consists of a single line containing an integer n ($1 \leq n \leq 10^9$).
Output
Output "`Yes`" if n is a prime number. Otherwise, output "`No`".

Input/Output Samples

Input	Output
`7`	`Yes`
`10`	`No`

Fig. 2. An example of a problem statement, consisting of the problem description, input/output specifications, and input/output samples.



```
1   #include <stdio.h>
2
3   int is_prime(int n) {
4       if (n == 1) {
5           return 0;
6       }
7       for (int i = 2; i * i < n; ++i) { // @@ Should be
            `i * i <= n` to correctly check up to sqrt(n)
8           if (n % i == 0) {
9               return 0;
10          }
11      }
12      return 1;
13  }
14
15  int main(void) {
16      int n;
17      scanf("%d", &n);
18      if (is_prime(n)) {
19          printf("Yes\n");
20      } else {
21          printf("No\n");
22      }
23      return 0;
24  }
```

Fig. 3. An example of an annotated program. Line 7 is marked as faulty by appending a marker `// @@` at the end of the line, followed by an explanation stating that `i * i < n` should be corrected to `i * i <= n`.

Time Limit Exceeded (TLE), and *Memory Limit Exceeded (MLE)*. A program is accepted if and only if it receives AC for all tests. If a program is rejected, it must have received a non-AC verdict on at least one test (or, in some cases, failed to compile and thus could not be tested). We utilize these failing tests as follows:

- If the program fails to compile, we use the diagnostic messages from the compiler.
- Otherwise, we extract the input, expected output, and verdict of one failing test, prioritizing tests with a verdict of WA, followed by RE, TLE, and MLE. If the verdict is WA, the actual output is also included.

3) Reference Programs: Online judge systems typically maintain a repository of historical submissions. If a previously accepted submission to the current problem exhibits similarity to the faulty program, it may offer critical guidance for fault localization. To identify such submissions, we use a text embedding model to generate embeddings for the faulty program and all accepted historical submissions to the current problem. The most semantically similar submission based on cosine similarity is then retrieved as the reference program.

B. Fault Location Annotation

Next, we prompt an LLM to annotate faulty lines based on the faulty program $P = (\ell_1, \ell_2, \ldots, \ell_n)$ and the collected auxiliary information I. Ideally, the LLM should directly output the line numbers of faulty lines, possibly with explanations, as done in prior work [45, 42]. However, while identifying line numbers is trivial for humans, LLMs struggle with this task due to their inherent limitations in numerical understanding and processing [48]. As a result, line number predictions with LLMs are frequently inaccurate, making such techniques unreliable in practice. Our preliminary experiments further confirm that directly prompting the LLM to output line numbers often yields suboptimal results, even when line numbers are explicitly prepended to each line in the input faulty program.

To address this, instead of predicting line numbers directly, we prompt the LLM to return an *annotated* version of the program, in which each faulty line is marked by appending a marker `// @@` at the end, followed by a detailed natural language explanation, resulting in a set of annotated lines $A \subseteq \{\ell_1, \ell_2, \ldots, \ell_n\}$ and their explanations $e(\ell)$ ($\ell \in A$). An example is shown in Figure 3.

To accurately identify the annotated lines and extract their explanations, we perform fuzzy matching between the lines in the annotated and original programs, instead of relying solely on the positions of the markers, which can be misaligned due to code formatting changes introduced by the LLM, such as inserting blank lines, adjusting indentation, or adding comments. Specifically, we consider two lines to match if the cosine similarity of their vector representations (computed using the `HashingVectorizer` from the *scikit-learn* [34] library) exceeds a threshold of 0.9, after stripping out any appended annotations. To align an annotated line with its original counterpart, we perform a bidirectional linear search (upward and downward) starting from the line with the same line number, stopping once a match is found.

For project-level submissions from advanced courses (such as software engineering, compiler technology, and operating systems), where multiple source files are typically involved, we concatenate the file paths and their contents into a single plain-text document, which serves as the faulty program to be annotated. However, unlike single-file submissions, project-level submissions are often much longer and may exceed the output token limits of proprietary inference services, which are typically lower than the input token limits. To mitigate this, we instruct the LLM to collapse non-essential code sections into `···` when deemed appropriate, thereby reducing the number of output tokens while preserving critical contextual information for analysis.

C. Weighted Multi-Model Voting

A single annotation result generated by a single model may be suboptimal due to inherent model biases and the stochastic nature of sampling. Different models exhibit varying strengths and weaknesses stemming from differences in model architecture, parameter size, training data, training methodology, *etc*. LLM ensemble has shown promise in leveraging the individual strengths of diverse models through strategic combination [11]. Additionally, techniques such as Self-Consistency [39] have shown that aggregating multiple generations from a single model can also improve performance. Motivated by this, we employ m state-of-the-art LLMs, each performing r rounds of fault location annotation. We then aggregate the annotation results to determine the suspiciousness of each line via a weighted voting strategy inspired by prior work [19]. Specifically, given a faulty program $P = (\ell_1, \ell_2, \ldots, \ell_n)$, we formalize the process as follows:

(1) Model Selection and Annotation

- **Step 1:** Select m state-of-the-art LLMs.
- **Step 2:** For each LLM, perform r rounds of fault location annotation, resulting in a total of $m \cdot r$ annotations:

$$\mathscr{A} = \{A_1, A_2, \ldots, A_{m \cdot r}\}. \tag{1}$$

(2) Weight Assignment

- **Step 3:** For each annotation A, we assign a normalized weight to each annotated line ℓ:

$$w_A(\ell) = \frac{1}{|A|} \quad (\ell \in A), \tag{2}$$

where $|A|$ is the number of annotated lines in A. For lines not annotated, we assign a weight of 0:

$$w_A(\ell) = 0 \quad (\ell \notin A). \tag{3}$$

(3) Suspiciousness Score Computation and Ranking

- **Step 4:** Compute the suspiciousness score of each line ℓ in P by summing the weights across all annotations:

$$s(\ell) = \sum_{A \in \mathscr{A}} w_A(\ell). \tag{4}$$

- **Step 5:** Rank suspicious lines in descending order of their suspiciousness scores $s(\ell)$ (breaking ties by ascending line numbers) to get the final ranked sequence:

$$S = (\ell'_1, \ell'_2, \ldots, \ell'_n), \tag{5}$$

where S is a permutation of $(\ell_1, \ell_2, \ldots, \ell_n)$, satisfying

$$s(\ell'_1) \geq s(\ell'_2) \geq \cdots \geq s(\ell'_n). \tag{6}$$

For each suspicious line ℓ with a non-zero suspiciousness score $s(\ell)$, we choose the explanation generated by the model that assigned the highest weight $w_A(\ell)$ to the line as the final explanation. We discard explanations generated by other models to avoid overwhelming the user with excessive information.

IV. Experimental Setup

A. Research Questions

We aim to answer the following research questions in our evaluation:

- **RQ1: How effective is FLAME compared to baseline methods?** This question evaluates the performance of FLAME on programming assignments and compare it against state-of-the-art fault localization techniques.
- **RQ2: How effective are the design choices of FLAME?** This question examines the contribution of each component within FLAME to overall performance.
- **RQ3: How well does FLAME generalize to general-purpose software codebases?** This question assesses the generalization capability of FLAME to general-purpose software codebases beyond educational contexts by evaluating its performance on the Defects4J [18] benchmark.
- **RQ4: How helpful are the results of FLAME in assisting automated program repair?** This question investigates whether FLAME's fault localization results can enhance the bug-fixing process when integrated with APR tools.

B. Datasets

To evaluate the effectiveness of FLAME in fault localization for programming assignments on both single-file and project-level submissions, we introduce PADEFECTS, a new dataset consisting of student submissions from two programming courses, *Data Structures* and *Compiler Technology*, collected from the online judge system of Beihang University. The detailed information of PADEFECTS is as follows:

Data Structures (PADEFECTS$_{\text{DS}}$): We collect student submissions from the *Data Structures* course, a foundational course for second-year undergraduate computer science students. In this course, students are tasked with solving programming problems involving various data structures. Each submission is a single-file C program. We select 10 programming problems covering a diverse range of topics, including linked lists, stacks, queues, trees, graphs, and strings. Submissions to these problems typically contain 50–200 lines of code. For each problem, we randomly select 150 students from all students enrolled in the 2024 academic year who made at least one failing attempt before passing all tests. For each student, we pair their final accepted submission with their last rejected submission. This results in 1,500 submission pairs in total. Additionally, we collect submissions to the same problems from the 2023 academic year to serve as reference programs.

Compiler Technology (PADEFECTS$_{\text{CT}}$): We also collect student submissions from the *Compiler Technology* course, a specialized course for third-year undergraduate computer science students. In this course, students are tasked with building a compiler that translates source programs into assembly. The development process is divided into several incremental stages, such as lexical analysis, syntax analysis, semantic analysis, and code generation. Each stage is formulated as a programming problem, with a statement and a suite of tests. Different from Data Structures, each submission in Compiler Technology is

TABLE I

STATISTICS OF PADEFECTS, INCLUDING THE NUMBER OF SUBMISSION PAIRS (# SUBMISSION PAIRS) IN EACH SUBSET, THE PROGRAMMING LANGUAGE OF THE SUBMISSIONS (PL), THE NUMBER OF SOURCE FILES PER SUBMISSION (# SOURCE FILES), AND THE NUMBER OF LINES OF CODE PER SUBMISSION (# LOC).

Dataset	# Submission Pairs	PL	# Source Files		# LoC	
			Avg.	Med.	Avg.	Med.
PADEFECTS$_{DS}$	1,500	C	1	1	72	61
PADEFECTS$_{CT}$	432	C++, Java	20	7	1,371	620

a full C++ or Java project consisting of multiple source files. Our study focuses on the first two critical stages, "lexical analysis" and "syntax analysis." Submissions to "lexical analysis" and "syntax analysis" typically contain about 500 and 2,000 lines of code, respectively. For each stage, we select all students enrolled in the 2024 academic year who made at least one failing attempt before passing all tests. For each student, we pair their final accepted submission with their last rejected submission. We exclude submission pairs involving file creation or deletion between the two submissions. This results in 239 submission pairs for "lexical analysis" and 193 submission pairs for "syntax analysis." Due to more complex requirements and flexible implementations, obtaining similar reference programs in Compiler Technology is challenging. Therefore, we exclude reference programs from this dataset.

The detailed statistics of PADEFECTS are summarized in Table I. For all submission pairs, following prior work [38, 47], we treat lines that differ between the two submissions as the ground truth of faulty lines.

Moreover, to assess the generalization capability of our method to general-purpose software codebases, we also evaluate it on *Defects4J* [18], a widely used dataset containing real bugs from real-world open-source projects. Following the empirical study by Wu et al. [45], we use a representative subset of Defects4J v1.2.0 containing 408 bugs for our evaluation.

C. Metrics

We adopt the *top-k* metric to evaluate the effectiveness of FLAME and baselines, which is widely adopted in fault localization tasks [35, 42, 46]. This metric is defined as the number of successfully localized programs, where a localization is considered successful if the faulty line appears within the top k most suspicious lines. In scenarios where a program contains multiple faulty lines, a localization is considered successful if at least one of the faulty lines appears within the top-k most suspicious lines, which is consistent with established practice [42]. Moreover, we also report the *top-k accuracy*, defined as the proportion of successfully localized programs across all programs.

D. Baseline Methods

We compare FLAME with the following baseline methods, including several state-of-the-art LLM-based fault localization methods and a spectrum-based fault localization method[1]:

[1]We also compare FLAME with single-model baselines (*e.g.*, Claude-only). For convenience of presentation, these baselines are treated as variants of FLAME and are discussed in RQ2.

- **LLMAO** [47]: A learning-based fault localization method that fine-tunes bidirectional adapters on top of pre-trained models to predict faulty lines. We fine-tune it using submissions from the 2023 academic year of the Data Structures course for evaluation on PADEFECTS$_{DS}$. We exclude it from evaluation on PADEFECTS$_{CT}$ due to its insufficient input length limit, which is shorter than most programs in PADEFECTS$_{CT}$.
- **FuseFL** [42]: An LLM-based fault localization method that combines SBFL results, test results, and code descriptions to guide LLMs in localizing the fault and generating step-by-step reasoning about the fault. This method was originally implemented using GPT-3.5 [9] via the "September 11, 2023" release of ChatGPT [29], which is no longer accessible. As a replacement, we implement it using the more advanced GPT-4.1 [32] (`gpt-4.1-2025-04-14`) in our evaluation.
- **ChatGPT-4 (Log)** [45]: A method used in the empirical study by Wu et al. [45] that directly prompts ChatGPT-4 with the faulty program and basic instructions, followed by a failing test and its corresponding error log. This method was originally implemented using GPT-4 [31] via the ChatGPT Web interface [29], which is no longer accessible. As a replacement, we implement it using the more advanced GPT-4.1 [32] (`gpt-4.1-2025-04-14`) in our evaluation.
- **Ochiai** [3]: The commonly used spectrum-based fault localization method using the Ochiai coefficient.

E. Implementation Details

For reference program retrieval, we use OpenAI's `text-embedding-3-small` [30] model to generate program embeddings. Regarding the weighted multi-model voting stage, we employ $m = 3$ state-of-the-art LLMs, *i.e.*, Claude 3.7 Sonnet [6], DeepSeek-V3 [13], and DeepSeek-R1 [12], all of which have demonstrated strong performance in code-related tasks [6, 13, 12, 1, 2], and our preliminary experiments also show that using these models strikes a good balance between performance and inference cost. Each model performs $r = 2$ rounds of fault location annotation for voting. For all models, we set the temperature to 0.1 after evaluating several candidate values in preliminary experiments, which can better balance output diversity and reliability.

V. RESULTS AND ANALYSIS

A. RQ1: How effective is FLAME compared to baseline methods?

To address this research question, we evaluate FLAME against four baseline methods, *i.e.*, LLMAO [47], FuseFL [42], ChatGPT-4 (Log) [45], and Ochiai [3], on PADEFECTS. The results are shown in Table II. Overall, FLAME outperforms all baselines, localizing 207 more faults at top-1 over the best-performing baseline (ChatGPT-4 (Log) [45]), demonstrating its effectiveness.

On PADEFECTS$_{DS}$, FLAME consistently outperforms all baselines across all top-k metrics, especially in top-1. Specifically, FLAME achieves a top-1 accuracy of 62.7%, substan-

TABLE II
FAULT LOCALIZATION RESULTS ON PADEFECTS. THE NUMBER INSIDE THE PARENTHESES IS THE *top-k accuracy* COMPUTED WITHIN EACH SUBSET.

Dataset	Method	Top-1	Top-3	Top-5	Top-10
PADEFECTS$_{DS}$	LLMAO	232 (15.5%)	471 (31.4%)	644 (42.9%)	858 (57.2%)
	FuseFL	363 (24.2%)	515 (34.3%)	579 (38.6%)	635 (42.3%)
	ChatGPT-4 (Log)	792 (52.8%)	1054 (70.3%)	1141 (76.1%)	1183 (78.9%)
	Ochiai	159 (10.6%)	305 (20.3%)	457 (30.5%)	705 (47.0%)
	FLAME	**941 (62.7%)**	**1195 (79.7%)**	**1269 (84.6%)**	**1309 (87.3%)**
PADEFECTS$_{CT}$	FuseFL	87 (20.1%)	124 (28.7%)	150 (34.7%)	150 (34.7%)
	ChatGPT-4 (Log)	125 (28.9%)	205 (47.5%)	218 (50.5%)	222 (51.4%)
	Ochiai	68 (15.7%)	117 (27.1%)	146 (33.8%)	177 (41.0%)
	FLAME	**183 (42.4%)**	**259 (60.0%)**	**285 (66.0%)**	**292 (67.6%)**

tially higher than the best-performing baseline, ChatGPT-4 (Log), which achieves only 52.8%. Since a substantial number of the faulty programs in the dataset fail all tests, Ochiai, which is based on SBFL, becomes ineffective due to its dependence on both passing and failing tests. This also explains the poor performance of FuseFL, which takes the SBFL results as an important input, and may be misled when those results are unreliable. The suboptimal performance of LLMAO can be attributed to the diverse error patterns across student programs. These patterns may not be well represented in the fine-tuning dataset and pose significant challenges for model learning during fine-tuning, resulting in limited generalization capability. While ChatGPT-4 (Log) outperforms other baselines, its performance remains inferior to FLAME. This gap may stem from the inherent biases of a single model, which could consistently overlook certain error patterns.

Even on the more challenging PADEFECTS$_{CT}$, FLAME still outperforms all baselines, achieving a top-1 accuracy of 42.4%. These results demonstrate FLAME's effectiveness on both single-file and project-level submissions.

Figure 4 and Figure 5 further illustrate the set of commonly and uniquely identified faults within top-1 and top-3 by different methods. As seen from the results, FLAME uniquely identified 189 and 72 faults within top-1 on PADEFECTS$_{DS}$ and FLAME$_{CT}$, respectively, substantially outperforming other methods. This indicates that FLAME complements existing techniques by detecting error patterns overlooked by other methods, thereby improving overall performance.

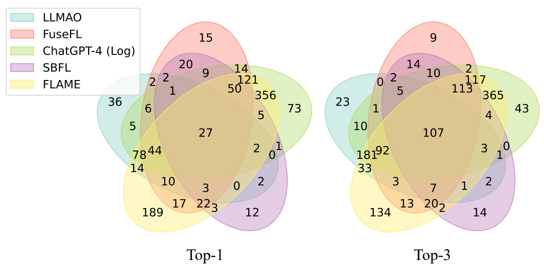


Fig. 4. Distribution of localized faults on PADEFECTS$_{DS}$.

Answer to RQ1: FLAME consistently outperforms all baseline methods across all top-k metrics on both datasets, demonstrating its effectiveness in fault localization on both single-file and project-level submissions.

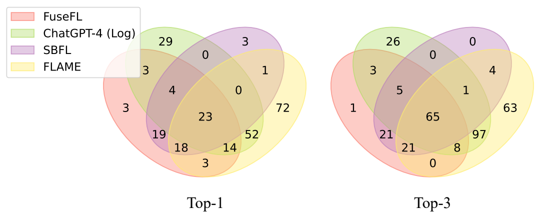


Fig. 5. Distribution of localized faults on PADEFECTS$_{CT}$.

B. RQ2: How effective are the design choices of FLAME?

To address this research question, we conduct an ablation study on PADEFECTS by removing each component from FLAME across its three stages, resulting in several variants.

- **Assignment-Specific Information Collection (*Collection*)**: FLAME leverages problem statements (*stmt*), test outcomes (*test*), and reference programs (*ref*) to enhance LLM's fault localization effectiveness. To assess the contribution of each type of information, we create three variants: FLAME$_{\text{w/o stmt}}$, FLAME$_{\text{w/o test}}$, and FLAME$_{\text{w/o ref}}$, each excluding one type of information.
- **Fault Location Annotation (*Annotation*)**: FLAME prompts the LLM to annotate faulty lines instead of predicting line numbers. To examine the effectiveness of this strategy, we create a variant, FLAME$_{\text{num}}$, which predicts line numbers directly.
- **Weighted Multi-Model Voting (*Voting*)**: FLAME determines the suspiciousness of each line by aggregating annotation results from Claude 3.7 Sonnet (*Claude*), DeepSeek-V3 (*DS-V3*), and DeepSeek-R1 (*DS-R1*) via a weighted voting strategy. To examine the effectiveness of this strategy, we create three single-model variants (FLAME$_{\text{Claude}}$, FLAME$_{\text{DS-V3}}$, and FLAME$_{\text{DS-R1}}$) that replace the multi-model setup with each individual model, and one unweighted variant (FLAME$_{\text{unweighted}}$) that assigns a weight of 1 to all annotated lines.

The results are shown in Table III, indicating that all components within FLAME contribute positively to overall performance. Specifically:

1) Assignment-Specific Information Collection: Excluding any type of information results in drops across all top-k

TABLE III
ABLATION STUDY RESULTS OF FLAME ON PADEFECTS.

Dataset	Stage	Method	Top-1	Top-3	Top-5	Top-10
PADefects$_{\text{DS}}$	-	**FLAME**	**941 (62.7%)**	**1195 (79.7%)**	**1269 (84.6%)**	**1309 (87.3%)**
	Collection	FLAME$_{\text{w/o stmt}}$	857 (57.1%) ↓5.6%	1098 (73.2%) ↓6.5%	1160 (77.3%) ↓7.3%	1079 (71.9%) ↓15.4%
		FLAME$_{\text{w/o test}}$	715 (47.7%) ↓15.0%	1046 (69.7%) ↓10.0%	1161 (77.4%) ↓7.2%	1232 (82.1%) ↓5.2%
		FLAME$_{\text{w/o ref}}$	882 (58.8%) ↓3.9%	1128 (75.2%) ↓4.5%	1188 (79.2%) ↓5.4%	1211 (80.7%) ↓6.6%
	Annotation	FLAME$_{\text{num}}$	237 (15.8%) **↓46.9%**	453 (30.2%) **↓49.5%**	589 (39.3%) **↓45.3%**	779 (51.9%) **↓35.4%**
	Voting	FLAME$_{\text{Claude}}$	605 (40.3%) ↓22.4%	1021 (68.1%) ↓11.6%	1101 (73.4%) ↓11.2%	1124 (74.9%) ↓12.4%
		FLAME$_{\text{DS-V3}}$	570 (38.0%) ↓24.7%	915 (61.0%) ↓18.7%	972 (64.8%) ↓19.8%	983 (65.5%) ↓21.8%
		FLAME$_{\text{DS-R1}}$	791 (52.7%) ↓10.0%	998 (66.5%) ↓13.2%	1020 (68.0%) ↓16.6%	1021 (68.1%) ↓19.2%
		FLAME$_{\text{unweighted}}$	912 (60.8%) ↓1.9%	1159 (77.3%) ↓2.4%	1234 (82.3%) ↓2.3%	1306 (87.1%) ↓0.2%
PADefects$_{\text{CT}}$	-	**FLAME**	**183 (42.4%)**	**259 (60.0%)**	**285 (66.0%)**	**292 (67.6%)**
	Collection	FLAME$_{\text{w/o stmt}}$	151 (35.0%) ↓7.4%	211 (48.8%) ↓11.2%	220 (50.9%) ↓15.1%	224 (51.9%) ↓15.7%
		FLAME$_{\text{w/o test}}$	83 (19.2%) ↓23.2%	131 (30.3%) ↓29.7%	138 (31.9%) **↓34.1%**	153 (35.4%) **↓32.2%**
	Annotation	FLAME$_{\text{num}}$	36 (8.3%) **↓34.1%**	104 (24.1%) **↓35.9%**	141 (32.6%) ↓33.4%	184 (42.6%) ↓25.0%
	Voting	FLAME$_{\text{Claude}}$	145 (33.6%) ↓8.8%	154 (35.6%) ↓24.4%	154 (35.6%) ↓30.4%	154 (35.6%) ↓32.0%
		FLAME$_{\text{DS-V3}}$	147 (34.0%) ↓8.4%	166 (38.4%) ↓21.6%	167 (38.7%) ↓27.3%	167 (38.7%) ↓28.9%
		FLAME$_{\text{DS-R1}}$	168 (38.9%) ↓3.5%	180 (41.7%) ↓18.3%	181 (41.9%) ↓24.1%	181 (41.9%) ↓25.7%
		FLAME$_{\text{unweighted}}$	179 (41.4%) ↓1.0%	242 (56.0%) ↓4.0%	258 (59.7%) ↓6.3%	290 (67.1%) ↓0.5%

metrics on both datasets. Specifically, when excluding test outcomes, the top-1 accuracy drops from 62.7% to 47.7% on PADefects$_{\text{DS}}$ and from 42.4% to 19.2% on PADefects$_{\text{CT}}$, representing the largest drop among all types of information. These results indicate that test outcomes are effective in improving fault localization, possibly by offering clues about the causes of test failures. This is especially valuable for programs in PADefects$_{\text{CT}}$, which are longer and more complex than those in PADefects$_{\text{DS}}$. When excluding problem statements or reference programs, the top-k metrics also drop, though less significantly, demonstrating that problem statements and reference programs are also effective, possibly by aiding in understanding the intended behavior of the program and guiding toward identifying the faults.

2) Fault Location Annotation: Predicting line numbers directly results in significant drops across all top-k metrics on both datasets. Specifically, the top-1 accuracy drops from 62.7% to 15.8% on PADefects$_{\text{DS}}$ and from 42.4% to 8.3% on PADefects$_{\text{CT}}$, representing the largest drop among all components. These results reveal that the inherent limitations of LLMs in numerical understanding and processing severely hinder their fault localization performance, and our fault location annotation strategy effectively mitigates these limitations.

3) Weighted Multi-Model Voting: Using individual models results in drops across all top-k metrics on both datasets compared to the multi-model setup. Specifically, even when using the best-performing model (DeepSeek-R1), the top-1 accuracy drops from 62.7% to 52.7% on PADefects$_{\text{DS}}$ and from 42.4% to 38.9% on PADefects$_{\text{CT}}$. After analyzing the results, we observe that individual models consistently identify fewer faulty lines compared to the multi-model setup, resulting in more missed actual faulty lines. This is further evidenced by the slower growth rate of top-k metrics as k increases ($k \geq 3$) for individual models, particularly on the more challenging PADefects$_{\text{CT}}$ dataset. These findings indicate that while individual models may be powerful, each exhibits unique strengths and weaknesses. Our voting strategy effectively combines these complementary strengths, leading to superior performance. Additionally, using the unweighted voting strategy also results in a performance drop. Specifically, the drops in top-1, top-3, and top-5 are more significant than the drop in top-10, indicating that our weighted voting strategy more effectively prioritizes actual faulty lines in higher-ranked positions.

Answer to RQ2: Removing any component from FLAME results in drops across all top-k metrics on both datasets, demonstrating each component's positive contribution to overall performance. Notably, *Fault Location Annotation* proves the most effective component, highlighting its critical contribution to accurately identifying faulty lines.

C. RQ3: How well does FLAME *generalize to general-purpose software codebases?*

To address this research question, we evaluate FLAME on the widely used Defects4J [18] benchmark. Our evaluation follows the settings of the empirical study by Wu et al. [45], which was conducted within faulty functions. Since problem statements and reference programs are not available in Defects4J, we leverage only the test outcomes as auxiliary information. Specifically, we leverage the source code of failing unit tests and their corresponding error logs.

The results are shown in Table IV. FLAME outperforms all baseline methods across all top-k metrics, with particularly notable improvements in top-1, top-3, and top-5, which are more critical for practical applications. In particular, FLAME achieves a top-1 accuracy of 53.7%, significantly higher than the best-performing baseline, ChatGPT-4 (Log). These results demonstrate that although FLAME is primarily designed

TABLE IV
FAULT LOCALIZATION RESULTS ON DEFECTS4J.

Method	Top-1	Top-3	Top-5	Top-10
FuseFL	140 (34.3%)	201 (49.3%)	235 (57.6%)	283 (69.4%)
ChatGPT-4 (Log)	171 (41.9%)	262 (64.2%)	285 (69.9%)	321 (78.7%)
Ochiai	95 (23.3%)	190 (46.6%)	251 (61.5%)	294 (72.1%)
FLAME	**219 (53.7%)**	**297 (72.8%)**	**318 (77.9%)**	**324 (79.4%)**

for programming assignments, it can effectively generalize to general-purpose software codebases beyond educational contexts.

Answer to RQ3: FLAME outperforms all baseline methods across all top-k metrics on Defects4J, demonstrating its generalization capability to general-purpose software codebases beyond educational contexts.

D. *RQ4: How helpful are the results of* FLAME *in assisting automated program repair?*

To address this research question, we conduct experiments on PADEFECTS$_{\text{DS}}$ to investigate whether FLAME's fault localization results, *i.e.*, identified faulty lines and their explanations, enhance the bug-fixing process when integrated with APR tools. Specifically, we implement an LLM-based APR pipeline using Claude 3.7 Sonnet [6] as the repair engine, and conduct experiments under the following two settings:

- **Plain APR**: The LLM is provided with the faulty program, problem statement, and test outcomes, and is instructed to repair the program.
- **FLAME-assisted APR**: In addition to the inputs in plain APR, the LLM is also provided with FLAME's results, including the identified faulty lines and their explanations.

We evaluate the effectiveness of APR using the following two metrics:

- **The number of fixed programs (*# Fixed*)**: A program is considered *fixed* if the repaired program *passes all tests*.
- **The number of improved programs (*# Improved*)**: A program is considered *improved* if the repaired program *passes more tests* than the original program, even if not passing all tests.

The results are shown in Table V. With the assistance of FLAME's results, the proportion of fixed programs increases from 57.2% to 60.0%, while the proportion of improved programs increases from 69.1% to 71.8%. These results demonstrate the helpfulness of FLAME's results in assisting APR.

To delve deeper into the role of FLAME's results in APR, we constructed a Sankey diagram (Figure 6) illustrating the detailed repair results with and without the assistance of FLAME's results. The results reveal that a substantial number of failed fixes are either fixed or partially improved with the assistance of FLAME's results, further demonstrating its helpfulness. However, there are also a few programs that are fixed or partially improved failed to be fixed when FLAME's results are provided. We speculate that these failures may stem from the false positives in FLAME's results, which could misdirect the LLM into introducing erroneous edits to the correct code.

TABLE V
RESULTS OF FLAME'S HELPFULNESS IN ASSISTING APR ON PADEFECTS$_{\text{DS}}$.

Setting	# Fixed	# Improved
Plain APR	858 (57.2%)	1036 (69.1%)
FLAME-assisted APR	**900 (60.0%)**	**1077 (71.8%)**

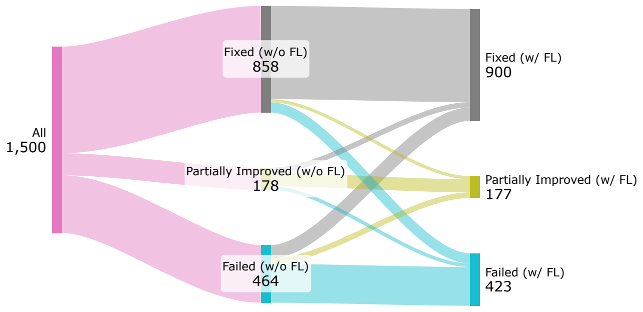


Fig. 6. APR results with and without the assistance of FLAME.

Answer to RQ4: With the assistance of FLAME's fault localization results, both the number of fixed and improved programs increase, demonstrating that FLAME's results can provide valuable guidance for automated program repair.

VI. DISCUSSION

A. *Manual Analysis*

The correctness and informativeness of the explanations greatly affect their educational value. To assess these qualities, we manually analyzed the explanations generated by FLAME. Specifically, we randomly sampled 65 cases from PADEFECTS$_{\text{DS}}$ and 55 cases from PADEFECTS$_{\text{CT}}$, excluding the cases where FLAME failed to localize the actual faulty lines within the top 5 most suspicious lines. The sample sizes were chosen to achieve a 90% confidence level with a 10% margin of error. For each sampled case, we manually checked the explanation's **correctness** (categorized as *true* or *false*) and rated its **informativeness** using a 7-point Likert scale [23] (1 = *minimally informative*, 7 = *highly informative*), with higher scores indicating greater informativeness. The analysis was carried out by one of the authors, who possesses over 4 years of C++ and Java programming experience.

Our analysis showed that FLAME generated correct explanations in 59 out of 65 PADEFECTS$_{\text{DS}}$ cases (90.8%) and in 52 out of 55 PADEFECTS$_{\text{CT}}$ cases (94.5%). The average informativeness scores were 6.47 and 6.29, respectively. These results demonstrate FLAME's ability to generate explanations that are both correct and highly informative, highlighting its educational value. Figure 7 shows an example of a highly informative explanation (score = 7). In this case, the explanation clearly indicates that *the < operator in the `if` condition*

```
    else if(p->num > n){
        p->left=inp(p->left,n);
    }
    else if(p->num < n){ // @@ The condition should be p->num <= n
        to match the problem description where right child is for
        values greater than OR EQUAL to the root
        p->right=inp(p->right,n);
    }
    return p;
}
```

Fig. 7. An example of a highly informative explanation (score = 7).

*should be changed to <= to align with the problem description: values greater than **or equal to** the current node should be in the right subtree*. This explanation is actionable for students to understand and fix the error. In contrast, Figure 8 exemplifies a less informative explanation (score = 3). Here, the explanation vaguely notes that *the handling of escape sequences is incorrect*, failing to specify the nature of the error or suggest a fix, limiting its usefulness for students.

```
} else if (curChar == '\'') {
    tokenContent += (char)curChar;
    curChar = reader.read();
    if (curChar != -1 && (char)curChar != '\\') { // @@ Incorrect
        handling of escape sequences
        tokenContent += (char)curChar;
        curChar = reader.read();
        tokenContent += (char)curChar;
        curChar = reader.read();
    }
```

Fig. 8. An example of a less informative explanation (score = 3).

B. Case Study of Failures

Despite the effectiveness of FLAME, it may still fail to accurately localize the actual faulty lines within the top 5 most suspicious lines in certain scenarios. To better understand these limitations, we conducted case studies on PADEFECTS and identified the following recurring failure patterns:

1) Correct Diagnosis but Incorrect Localization: In several cases, FLAME correctly diagnosed the root cause of the error but incorrectly identified the location. As illustrated in Figure 9, the error arises from a missing update to the `ans` variable (❶). While FLAME correctly diagnosed the root cause of the error in its explanation, it incorrectly annotated the line where `ans` is printed (❷). We believe that while these cases are classified as failures under the current strict ground-truth criteria (which prioritize precise localization), they can still guide students toward understanding the errors in their programs.

2) Failure on Complex Programs: In some scenarios, FLAME exhibited total failure in identifying the error. As shown in Figure 10, the program does not properly handle parenthesized expressions when parsing function arguments (❸). FLAME was unable to navigate the complexities of the program and failed to identify the subtle error buried within its 1,900 lines of code. As a result, it reported several incorrect faulty lines with misleading explanations (❶,❷). These cases highlight limitations in FLAME's ability to reason about semantics in large-scale programs, underscoring both the challenge and critical need for fault localization techniques to advance robustness against real-world software complexity.

```
            d[i+1]=ans;
            i++;
        }
+       else
+       {
+           ans+=d[i];                                    1
+       }
    }
    if(cnt==1)
    {
        printf("%d\n",d[0]);
    }
    else
        printf("%d\n",ans); // @@ ans may not be properly
                   updated for all cases                  2
```

Fig. 9. An example of correct diagnosis but incorrect localization.

3) Failure in Suspicious Line Ranking: Additionally, there are cases where FLAME correctly identified ground-truth faulty lines but ranked them outside the top-5 most suspicious lines, resulting in their classification as failures. These cases underscore the need to enhance LLM's semantic understanding and reasoning capabilities, as well as to refine the voting and ranking strategies.

```
    else{
        Error error = new Error(
            ErrorType.j,wordList.get(now-1).getRow()); // @@
                Should use current token's row number
                instead of previous token                 1
        errorList.add(error);
    }
...
    if(!isEqualWordType(0, WordType.RPARENT)){
        Error error = new Error(
            ErrorType.j,wordList.get(now-1).getRow()); // @@
                Doesn't properly handle deeply nested
                function call parentheses                 2
        errorList.add(error);
    }
...
    if(isEqualWordType(0, WordType.PLUS)
            ||isEqualWordType(0,WordType.MINU)
            ||isEqualWordType(0,WordType.NOT)
            ||isEqualWordType(0,WordType.IDENFR)
            ||isEqualWordType(0,WordType.INTCON)
-           ||isEqualWordType(0,WordType.CHRCON)){
+           ||isEqualWordType(0,WordType.CHRCON)
+           ||isEqualWordType(0,WordType.LPARENT)){       3
        unaryExp.funcRParams = analyseFuncRParams();
    }
```

Fig. 10. An example of failure on a complex program.

C. Threats to Validity

Threats to internal validity relate to the reproducibility of FLAME's results, given the stochastic nature of sampling from LLMs. Although we mitigate this issue by performing multiple rounds of fault location annotation and aggregating the results via a weighted voting strategy, completely eliminating uncertainty remains challenging. As a result, the reproducibility of FLAME's results may still be affected to some extent.

Threats to external validity relate to the availability of auxiliary information leveraged by FLAME. FLAME leverages auxiliary information such as problem statements, test outcomes, and reference programs retrieved from historical submissions. However, the availability and quality of such information cannot always be guaranteed. In scenarios where the auxiliary information is limited or unreliable, FLAME's effectiveness may be adversely affected.

VII. Conclusion and Future Work

In this paper, we propose FLAME, a fine-grained, explainable fault localization method tailored for programming assignments. FLAME leverages rich contextual information specific to programming assignments to guide LLMs in identifying faulty code lines. By prompting LLMs to annotate faulty code lines with detailed natural language explanations, our method enhances both localization accuracy and educational value. To further improve reliability, we introduce a weighted multi-model voting strategy that aggregates the results of multiple LLMs. Extensive experimental results show that FLAME outperforms state-of-the-art fault localization baselines on both our newly constructed programming assignment benchmark (PADefects) and the widely used Defects4J benchmark for general-purpose codebases.

In the future, we plan to enhance FLAME's semantic understanding and reasoning capabilities, particularly for large-scale programs. We will also refine the voting and ranking strategies by better distinguishing between different types of errors. Finally, we plan to investigate our method's generalization capability across diverse programming languages and educational contexts with varying assignment types.

Acknowledgement

This research is supported by the National Natural Science Foundation of China Grants Nos. 62302021, 62177003, 62332001, the Fundamental Research Funds for the Central Universities (Grant No. JK2024-28), and Beijing Advanced Innovation Center for Future Blockchain and Privacy Computing.

References

[1] Aider llm leaderboards. URL https://aider.chat/docs/leaderboards/.

[2] Livebench. URL https://livebench.ai/.

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

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

[5] Kirsti M Ala-Mutka. A survey of automated assessment approaches for programming assignments. *Computer science education*, 15(2):83–102, 2005.

[6] Anthropic. Claude 3.7 sonnet and claude code, Feb 2025. URL https://www.anthropic.com/news/claude-3-7-sonnet.

[7] Eliane Araujo, Matheus Gaudencio, Dalton Serey, and Jorge Figueiredo. Applying spectrum-based fault localization on novice's programs. In *2016 IEEE Frontiers in Education Conference (FIE)*, pages 1–8. IEEE, 2016.

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

[9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

[10] Brenda Cheang, Andy Kurnia, Andrew Lim, and Wee-Chong Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[11] Zhijun Chen, Jingzheng Li, Pengpeng Chen, Zhuoran Li, Kai Sun, Yuankai Luo, Qianren Mao, Dingqi Yang, Hailong Sun, and Philip S Yu. Harnessing multiple large language models: A survey on llm ensemble. *arXiv preprint arXiv:2502.18036*, 2025.

[12] DeepSeek-AI. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2501.12948.

[13] DeepSeek-AI. Deepseek-v3 technical report, 2025. URL https://arxiv.org/abs/2412.19437.

[14] Christopher Douce, David Livingstone, and James Orwell. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)*, 5(3):4–es, 2005.

[15] Bob Edmison and Stephen H Edwards. Experiences using heat maps to help students find their bugs: Problems and solutions. In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, pages 260–266, 2019.

[16] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Neural attribution for semantic bug-localization in student programs. *Advances in Neural Information Processing Systems*, 32, 2019.

[17] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli calling international conference on computing education research*, pages 86–93, 2010.

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

[19] Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1(FSE):1424–1446, 2024.

[20] Hieke Keuning, Bastiaan Heeren, and Johan Jeuring.

Code quality issues in student programs. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education*, pages 110–115, 2017.

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

[22] Yi Li, Shaohua Wang, and Tien Nguyen. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 661–673. IEEE, 2021.

[23] Rensis Likert. A technique for the measurement of attitudes. *Archives of psychology*, 1932.

[24] Yuxing Liu, Zhanwen Zhang, Xuchuan Zhou, and Wentao Liu. An empirical study on spectrum-based fault localization for student programs. In *2023 3rd International Symposium on Computer Technology and Information Science (ISCTIS)*, pages 547–551. IEEE, 2023.

[25] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 664–676, 2021.

[26] Ken Masters et al. A brief guide to understanding moocs. *The Internet Journal of Medical Education*, 1(2):2, 2011.

[27] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1169–1180, 2022.

[28] Thanh-Dat Nguyen, Thanh Le-Cong, Duc-Minh Luong, Van-Hai Duong, Xuan-Bach D Le, David Lo, and Quyet-Thang Huynh. Ffl: Fine-grained fault localization for student programs via syntactic and semantic reasoning. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 151–162. IEEE, 2022.

[29] OpenAI. Chatgpt — release notes. URL https://help.openai.com/en/articles/6825453-chatgpt-release-notes.

[30] OpenAI. New embedding models and api updates, Jan 2024. URL https://openai.com/index/new-embedding-models-and-api-updates/.

[31] OpenAI. Gpt-4 technical report, 2024. URL https://arxiv.org/abs/2303.08774.

[32] OpenAI. Introducing gpt-4.1 in the api, Apr 2025. URL https://openai.com/index/gpt-4-1/.

[33] Linus Östlund, Niklas Wicklund, and Richard Glassey. It's never too early to learn about code quality: A longitudinal study of code quality in first-year computer science students. In *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, pages 792–798, 2023.

[34] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

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

[36] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, pages 15–26, 2013.

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

[38] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 180–182. IEEE, 2017.

[39] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[40] Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)*, 51(1):1–34, 2018.

[41] Yutaka Watanobe, Md Mostafizer Rahman, Taku Matsumoto, Uday Kiran Rage, and Penugonda Ravikumar. Online judge system: Requirements, architecture, and experiences. *International Journal of Software Engineering and Knowledge Engineering*, 32(06):917–946, 2022.

[42] Ratnadira Widyasari, Jia Wei Ang, Truong Giang Nguyen, Neil Sharma, and David Lo. Demystifying faulty code: Step-by-step reasoning for explainable fault localization. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 568–579. IEEE, 2024.

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

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

[45] Yonghao Wu, Zheng Li, Jie M Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models

in fault localisation. *arXiv preprint arXiv:2308.15276*, 2023.

[46] Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. Flexfl: Flexible and effective fault localization with open-source large language models. *IEEE Transactions on Software Engineering*, 2025.

[47] Aidan ZH Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

[48] Haotong Yang, Yi Hu, Shijia Kang, Zhouchen Lin, and Muhan Zhang. Number cookbook: Number understanding of language models and how to improve it. *arXiv preprint arXiv:2411.03766*, 2024.

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