FLARE: Fine-Grained Diagnostic Feedback for LLM Code Refinement

Yinsheng Yao*
Tongji University
Shanghai, China
2251929@tongji.edu.cn

Hongxiang Zhang*
Purdue University
West Lafayette, IN, USA
hxxzhang@purdue.edu

Weixi Tong
Purdue University
West Lafayette, IN, USA
tong172@purdue.edu

Tianyi Zhang
Purdue University
West Lafayette, IN, USA
tianyi@purdue.edu

Abstract

Large language models often generate code with bugs. Existing methods rely on feedback signals such as test failures and self-critiques to iteratively refine the generated code. Such signals are either too coarse-grained or too high-level, which is not sufficient to inform the model where to fix the bug. In this work, we present FLARE, an iterative framework with a lightweight diagnostic model that predicts line-level suspiciousness signals for bug localization and code refinement. Given the inherent uncertainty of diagnostic predictions, FLARE searches over the top-k suspicious regions and selects the best candidate according to execution outcomes. Experiments on LiveCodeBench and BigCodeBench with five base LLMs show that, even without candidate search (k=1), FLARE outperforms the strongest baseline with an absolute improvement from 1.72% to 7.42%. Furthermore, searching over 10 candidates yields an average improvement of 8.50% compared with no candidate search. When evaluated in isolation, our lightweight diagnostic model achieves the best performance compared with recent fault localization methods, demonstrating that it can provide reliable fine-grained guidance for code refinement.

1 Introduction

Large Language Models (LLMs) have demonstrated remarkable capabilities in coding (Jiang et al., 2026; Achiam et al., 2023; Zhu et al., 2024). Despite this progress, LLMs remain prone to generating code with bugs (Jesse et al., 2023; Wang et al., 2025). Given the challenge of generating correct code in one pass, existing methods often perform iterative refinement, where a model generates an initial program, receives feedback, and then refines the program over multiple rounds (Chen et al., 2024; Madaan et al., 2023; Zhang et al., 2023, 2025). However, these methods usually rely on feedback signals such as test failures and self-critiques to iteratively refine the generated code. Such signals are either too coarse-grained or too high-level, which is not sufficient to inform the model where to fix the bug.

To address this limitation, we propose FLARE, an iterative framework with a lightweight diagnostic model that predicts line-level suspiciousness for targeted code refinement. Given an initial code solution, FLARE collects the language model's token-level probabilities and aligns them from subword tokens to lexical units such as identifiers, operators, and keywords. The diagnostic model scores these lexical units and aggregates the scores into line-level suspiciousness predictions. During refinement, FLARE combines the ranked suspicious lines with execution feedback to generate targeted candidate refinements. Given the inherent uncertainty of fine-grained feedback predictions, FLARE is further equipped with a search procedure over the top-k plausible fault regions at each iteration and selects the next candidate according to execution outcomes.

We evaluate FLARE on two code generation benchmarks, LiveCodeBench (Jain et al., 2025) and BigCodeBench (Zhuo et al., 2025), with five base LLMs and three existing methods, including NL-Debugging (Zhang et al., 2025), Self-Debugging (Chen et al., 2024), and Self-Refine (Madaan et al., 2023). Even without candidate search ($k = 1$), FLARE outperforms the most competitive baseline, NL-Debugging, in 9 of 10 settings with an absolute improvement from 1.72% to 7.42%. Furthermore, searching over 10 candidates ($k = 10$) yields an absolute improvement of 8.50% compared with no candidate search. Ablation studies further show that both fine-grained diagnostic feedback and candidate search contribute significantly to the overall gains. When evaluated in isolation, our diagnostic model achieves 67% Top-1 and 89% Top-10 localization accuracy, out-

*Equal contribution.

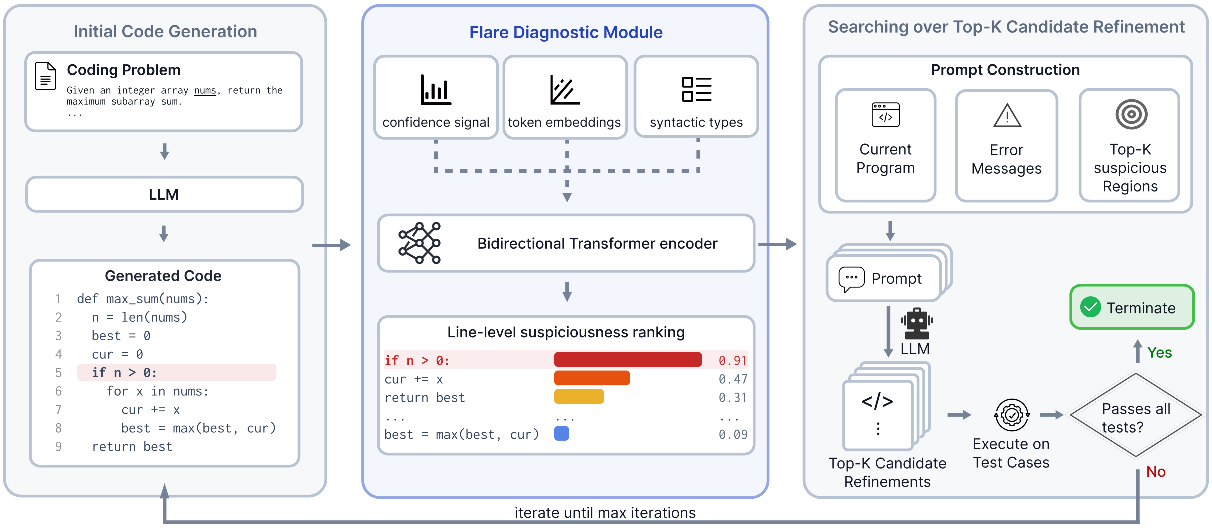

Figure 1: Overview of FLARE. The framework aligns LLMs' probability signals to predict line-level suspiciousness, combines this diagnostic signal with execution feedback, and searches over top-k candidate refinements.

performing recent fault localization methods, including FlexFL (Xu et al., 2025), LLMAO (Yang et al., 2024), and BAP (Stein et al., 2025).

2 Methodology

Figure 1 provides an overview of FLARE. Given an initial candidate program, FLARE predicts suspicious lines with a lightweight diagnostic model trained on LLM internal signals (Section 2.1). To account for diagnostic uncertainty, FLARE searches over the top-k suspicious regions and selects the best candidate according to execution outcomes (Section 2.2). The process continues until the program passes all tests or reaches a maximum iteration budget.

2.1 Fine-grained Diagnostic Feedback

Existing self-refinement methods often rely on coarse-grained or high-level feedback, which provides limited guidance about where the underlying issue resides. FLARE introduces a fine-grained diagnostic model that ranks suspicious lines for targeted refinement.[1] Formally, given a generated program $\mathcal{C}$ segmented into L lines $\mathcal{C} = [c_1, \ldots, c_L]$, we train a diagnostic model f_θ that outputs line-level suspiciousness scores:

$$\mathbf{r} = f_\theta(\mathcal{C}) \in \mathbb{R}^L, \qquad \mathbf{r} = [r_1, \ldots, r_L], \tag{1}$$

where r_i denotes the suspiciousness score assigned to line c_i.

[1] We evaluate the diagnostic model and ablate its components in Section 3.5.

Training Data Construction. We train the diagnostic model on the training split of `Collu-Bench` (Jiang et al., 2024), which contains 10504 buggy code solutions generated by 11 diverse LLMs on five datasets. In `Collu-Bench`, each buggy solution is paired with token-level probabilities from the generator LLM and ground-truth bug locations. Previous empirical studies reveal two key insights (Jiang et al., 2024; Spiess et al., 2025): LLMs tend to exhibit lower confidence when generating buggy code, and their errors are concentrated in certain token categories, such as `Keyword`, `Identifier`, and `Type`. Motivated by these findings, we use the LLM's confidence signals, token embeddings, and syntactic token types as inputs to the diagnostic model.

To construct these inputs, we first align the LLM's confidence signals with the code's lexical-unit representation. While the LLM assigns probabilities to subword tokens, program semantics are organized around syntactic units such as identifiers, operators, and statements. This mismatch is particularly problematic for code, as standard Byte-Pair Encoding (BPE) tokenization often splits a single lexical unit into multiple subword pieces, e.g., `max_retries` $\rightarrow$ {`max`, `_`, `retries`}.

Specifically, we parse $\mathcal{C}$ and extract the AST-aligned lexical sequence

$$\mathcal{T}(\mathcal{C}) = \{t_1, \ldots, t_M\}, \tag{2}$$

where t_j corresponds to a lexical unit. Let $\{p_n\}_{n=1}^{N}$ denote the probabilities of the N subword tokens.

Given a lexical unit t_j, let $\mathcal{S}(j) \subseteq \{1, \ldots, N\}$ denote the subword token indices aligned with t_j. Then, we compute the confidence of t_j as the probability of its aligned subword sequence:

$$p'_j = \prod_{n \in \mathcal{S}(j)} p_n. \quad (3)$$

For each lexical unit t_j in the program $\mathcal{C}$, we represent it by the tuple

$$\mathbf{x}_j = \big(t_j,\ \mathrm{Type}(t_j),\ p'_j\big), \quad (4)$$

where $\mathrm{Type}(t_j)$ denotes the syntactic category of t_j, such as `Keyword`, `Identifier`, or `Type`.[2]

Training Objective. Diagnosing bugs in generated code often requires reasoning beyond the local token or line. For example, an incorrect variable update or branch condition may only become suspicious when compared with definitions, uses, or constraints that appear elsewhere in the program. To model these dependencies, we encode the AST-aligned sequence with a bidirectional transformer encoder, which allows each code unit to attend to the full program context (Guo et al., 2021).

Given the program $\mathcal{C}$, we first embed each lexical unit t_j in $\mathcal{C}$ into an input vector by concatenating its token embedding, syntactic type, and projected confidence:

$$\mathbf{e}_j = \big[\mathrm{Emb}_{tok}(t_j);\ \mathrm{Emb}_{type}(\mathrm{Type}(t_j));\ \mathrm{Proj}(p'_j)\big], \quad (5)$$

where Emb_{tok} and Emb_{type} are learned embedding layers, and Proj is a learned linear projection from the scalar p'_j to a vector representation.[3]

We add positional encodings and feed the resulting sequence into the transformer encoder to obtain contextual representations:

$$\mathbf{h}_1, \ldots, \mathbf{h}_M = \mathrm{TransformerEnc}\big(\mathrm{PosEnc}(\mathbf{e}_1, \ldots, \mathbf{e}_M)\big), \quad (6)$$

where PosEnc adds sinusoidal positional encodings to each input vector as in Vaswani et al. (2017).

We then apply a linear classifier to each contextualized unit representation $\mathbf{h}_j$:

$$s_j = \mathbf{w}^\top \mathbf{h}_j + b. \quad (7)$$

We map the annotated buggy token in the training data to its aligned lexical unit, and let $j^\star \in \{1, \ldots, M\}$ denote the index of the ground-truth buggy lexical unit in the AST-aligned sequence $\mathcal{T}(\mathcal{C})$. We train the model with position-level cross-entropy over lexical-unit logits:

$$\mathcal{L}_{\mathrm{FL}} = -\log \frac{\exp(s_{j^\star})}{\sum_{m=1}^{M} \exp(s_m)}. \quad (8)$$

All trainable components of the diagnostic model are optimized end-to-end with $\mathcal{L}_{\mathrm{FL}}$.

At inference time, we map lexical-unit scores back to the line-level task defined above. Let $\mathcal{T}(c_i)$ denote the set of lexical units appearing on line c_i. The line-level suspiciousness score is defined as[4]

$$r_i = \max_{t_j \in \mathcal{T}(c_i)} s_j. \quad (9)$$

During inference, the diagnostic model ranks lines by their suspiciousness scores $\mathcal{R} = \mathrm{Rank}(\{(c_i, r_i)\}_{i=1}^{L})$. The refine loop uses the top-k lines from $\mathcal{R}$ for targeted refinement.

2.2 Top-k Candidate Search

During inference, FLARE refines a candidate program iteratively. Starting from an initial program $\mathcal{C}^{(0)}$, FLARE executes the current program in a sandboxed environment and obtain execution feedback $\xi^{(t)}$ at iteration t. If the program passes all test cases, the loop terminates and returns the current candidate as the solution. Otherwise, FLARE applies the diagnostic model to obtain a ranked list of suspicious lines $\mathcal{R}^{(t)}$.

Given the inherent uncertainty of fine-grained feedback predictions, FLARE searches over the top-k suspicious lines instead of committing to a single localization prediction. For each selected line, FLARE constructs a targeted refinement prompt containing the current program, the execution feedback, and that suspicious line. The language model generates one candidate revision for each prompt, producing k candidate revisions at the current iteration. Thus, $k = 1$ uses only the most suspicious line, while larger k values explore multiple plausible fault locations.

We execute all candidate revisions and select the best candidate according to their execution outcomes. If any candidate passes all tests, the loop terminates and returns a passing candidate as the final solution. Otherwise, FLARE prefers the candidate that passes more tests or resolves more fail-

[2]We list type details in Appendix B.

[3]The implementation details are provided in Appendix A.3

[4]We experimented with alternative designs in Section 3.4.

ures.[5] When candidates pass the same number of tests, we prefer the shortest candidate for conciseness. The loop repeats until all tests pass or the maximum iteration budget is reached.

3 Experiments

3.1 Experimental Setup

Benchmarks. We evaluate FLARE on two representative code generation benchmarks. First, LiveCodeBench (Jain et al., 2025) is a state-of-the-art benchmark that dynamically selects recent programming problems while mitigating contamination risks. BigCodeBench (Zhuo et al., 2025) contains practical programming tasks involving diverse library dependencies and intricate logic. We report Pass@1 as the main evaluation metric, measuring the percentage of tasks whose final generated solution passes all test cases.

Base LLMs. We evaluate FLARE with five base LLMs spanning different model families and scales, including Qwen3-8B (Yang et al., 2025), Qwen2.5-Coder-7B (Team, 2024), Llama-3.1-8B (Grattafiori et al., 2024), Llama-3.3-70B (Grattafiori et al., 2024), and GPT-4o-mini (Hurst et al., 2024).

Comparison Baselines. We compare FLARE with three existing iterative refinement methods. First, Self-Debugging (Chen et al., 2024) generates an explanation of the original code and combines it with the execution results as feedback for code refinement. Self-Refine (Madaan et al., 2023) prompts the model to generate self-feedback for code refinement. NL-Debugging (Zhang et al., 2025) translates code solutions back to natural language and performs debugging in the natural language space to guide code refinement. Following prior work (Zhang et al., 2025; Chen et al., 2024), we also include the original LLM performance as a weak baseline.

Implementation Details. The diagnostic model in FLARE is a bidirectional transformer encoder with 6 layers and 8 attention heads. We train the full diagnostic model from scratch, including the input embeddings, transformer encoder, and linear prediction head. Following Zhang et al. (2025); Amini et al. (2025), we use a maximum refinement iteration budget of N=5 and a candidate-search budget of k=10 unless otherwise noted. We also report k=1 variant, which uses only the top-ranked suspicious region and therefore isolates the effect of diagnostic localization without expanded candidate search. Additional details are provided in Appendix A.

[5] During refinement, execution feedback and candidate selection use only visible/public tests. Hidden tests are never used during refinement and are only used for final evaluation.

3.2 Main Results

Table 1 shows the comparison results. FLARE with expanded candidate search (k=10) consistently achieves the best pass@1 across all settings. Even without expanded search, FLARE (k=1) still outperforms Self-Debugging and Self-Refine in all settings and surpasses NL-Debugging in 9 of the 10 settings.

Specifically, FLARE (k=1) achieves absolute pass@1 gains of 1.72% to 7.42% over the strongest baseline NL-Debugging, in four of five settings on LiveCodeBench. On BigCodeBench, FLARE (k=1) also provides consistent gains over NL-Debugging across all five models. These consistent gains show that diagnostic localization improves iterative refinement even before expanded candidate search is applied.

Expanded candidate search allows FLARE to explore multiple plausible fault regions instead of committing to the highest-ranked one. Compared with the k=1 variant, FLARE (k=10) further improves Pass@1 by 8.50% on average, with gains in all settings. On BigCodeBench, expanded candidate search improves FLARE by 5.08 to 13.42 points across models. On LiveCodeBench, the gains range from 4.57% to 12.00%. These results show that precise diagnostic localization and candidate search play complementary roles. Localization identifies likely refinement regions, while expanded search mitigates uncertainty in the predicted fault location.

3.3 Ablation Study

The main results already show that ablating the search procedure by setting k to 1 outperforms other methods. In this section, we therefore focus the ablation study on the contribution of each feedback source. Using GPT-4o-mini on LiveCodeBench and BigCodeBench, we compare FLARE with variants that keep only execution feedback or only fine-grained feedback. We report both k=1 and k=10 results in Table 2 to separate the effect of feedback quality from the effect of the candidate searching procedure. A more detailed analysis of N and k is provided in Section 3.6.

LiveCodeBench					
Method	**Qwen3-8B**	**Qwen2.5-Coder-7B**	**Llama-3.1-8B**	**Llama-3.3-70B**	**GPT-4o-mini**
LLM alone	30.86	26.86	20.57	46.86	44.00
Self-Debugging	38.29$_{\uparrow 7.43}$	32.57$_{\uparrow 5.71}$	27.43$_{\uparrow 6.86}$	58.86$_{\uparrow 12.00}$	51.43$_{\uparrow 7.43}$
Self-Refine	36.00$_{\uparrow 5.14}$	33.71$_{\uparrow 6.85}$	24.57$_{\uparrow 4.00}$	50.29$_{\uparrow 3.43}$	48.00$_{\uparrow 4.00}$
NL-Debugging	44.57$_{\uparrow 13.71}$	30.29$_{\uparrow 3.43}$	22.86$_{\uparrow 2.29}$	61.71$_{\uparrow 14.85}$	53.14$_{\uparrow 9.14}$
FLARE (k=1)	40.00$_{\uparrow 9.14}$	37.71$_{\uparrow 10.85}$	29.14$_{\uparrow 8.57}$	64.57$_{\uparrow 17.71}$	54.86$_{\uparrow 10.86}$
FLARE (k=10)	**46.86**$_{\uparrow 16.00}$	**44.57**$_{\uparrow 17.71}$	**33.71**$_{\uparrow 13.14}$	**76.57**$_{\uparrow 29.71}$	**65.14**$_{\uparrow 21.14}$
BigCodeBench					
LLM alone	34.04	31.58	28.68	37.63	36.23
Self-Debugging	52.89$_{\uparrow 18.85}$	44.12$_{\uparrow 12.54}$	38.42$_{\uparrow 9.74}$	56.58$_{\uparrow 18.95}$	55.00$_{\uparrow 18.77}$
Self-Refine	54.12$_{\uparrow 20.08}$	47.46$_{\uparrow 15.88}$	30.44$_{\uparrow 1.76}$	57.46$_{\uparrow 19.83}$	55.53$_{\uparrow 19.30}$
NL-Debugging	55.00$_{\uparrow 20.96}$	47.02$_{\uparrow 15.44}$	40.35$_{\uparrow 11.67}$	58.33$_{\uparrow 20.70}$	56.67$_{\uparrow 20.44}$
FLARE (k=1)	57.63$_{\uparrow 23.59}$	47.81$_{\uparrow 16.23}$	40.53$_{\uparrow 11.85}$	59.39$_{\uparrow 21.76}$	58.51$_{\uparrow 22.28}$
FLARE (k=10)	**64.47**$_{\uparrow 30.43}$	**55.79**$_{\uparrow 24.21}$	**45.61**$_{\uparrow 16.93}$	**72.81**$_{\uparrow 35.18}$	**69.65**$_{\uparrow 33.42}$

Table 1: Pass@1 (%) comparison between FLARE and iterative refinement baselines. For FLARE, k=1 uses a single candidate revision, whereas k=10 denotes the candidate searching setting. Arrows indicate gains or drops relative to the LLM alone. The best result in each column is highlighted in bold.

Method	LCB	BCB
Execution feedback only (k=1)	50.27	53.81
Fine-grained feedback only (k=1)	50.86	50.35
FLARE (k=1)	54.86	58.51
Execution feedback only (k=10)	58.29	59.47
Fine-grained feedback only (k=10)	55.43	58.42
FLARE (k=10)	**65.14**	**69.65**

Table 2: Ablation results on LiveCodeBench (LCB) and BigCodeBench (BCB). k denotes the candidate-search budget. "Execution feedback only" removes fine-grained diagnostic feedback, while "Fine-grained feedback only" removes execution feedback.

Fine-Grained Feedback. Removing fine-grained feedback indicates that the refinement prompt contains only execution feedback and no predicted suspicious lines. At k=1, execution-only refinement reaches 50.27% Pass@1 on LiveCodeBench and 53.81% on BigCodeBench, while full FLARE reaches 54.86% and 58.51%. At k=10, execution-only refinement reaches 58.29% and 59.47%, while full FLARE reaches 65.14% and 69.65%. The consistent gaps show that fine-grained feedback provides useful localization guidance on top of execution feedback in both the single-candidate and expanded-search settings.

Execution Feedback. Removing execution feedback leaves a diagnostic-only variant that prompts the language model with suspicious regions but no execution feedback. At k=1, diagnostic-only refinement reaches 50.86% pass@1 on LiveCodeBench and 50.35% on BigCodeBench, below the full FLARE scores of 54.86% and 58.51%. At k=10, diagnostic-only refinement reaches 55.43% and 58.42%, while full FLARE reaches 65.14% and 69.65%. These gaps show that execution feedback remains necessary for candidate refinements. While fine-grained feedback provides localization for revision, execution feedback supplies behavioral evidence.

3.4 Alternative Aggregation Design

Method	LCB	BCB
FLARE (Average pooling)	59.43	62.54
FLARE (Max pooling)	**65.14**	**69.65**

Table 3: Alternative line-level aggregation designs.

The diagnostic model predicts suspiciousness scores for lexical units, while the refinement prompt uses line-level feedback. In the main method, we aggregate lexical-unit scores on each line with max pooling. We also evaluate an alternative average-pooling design:

$$r_i = \frac{1}{|\mathcal{T}(c_i)|} \sum_{t_j \in \mathcal{T}(c_i)} s_j. \qquad (10)$$

Table 3 compares average pooling with the max-pooling design used in FLARE on GPT-4o-mini. On LiveCodeBench, average pooling reaches 59.43% Pass@1, below the 65.14% achieved by max pooling. Similarly, average pooling achieves 62.54% Pass@1 on BigCodeBench, below the 69.65% achieved by max pooling. This is because max

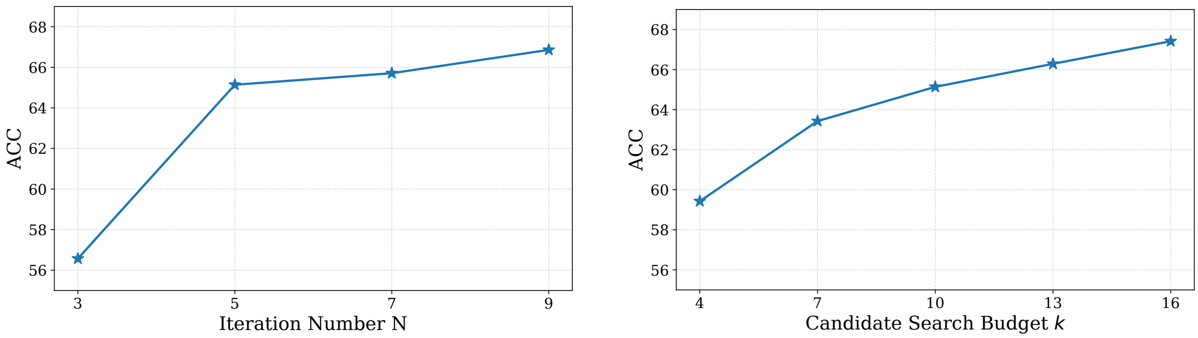

Figure 2: Effect of the iteration budget N and candidate-search budget k on GPT-4o-mini on LiveCodeBench.

pooling preserves a strong suspiciousness signal from a single faulty unit, whereas average pooling can dilute that signal with many non-suspicious units on the same line.

3.5 Diagnostic Model Performance

In this section, we evaluate the performance of the diagnostic model alone on 100 randomly selected tasks from LiveCodeBench. We evaluate performance in terms of top-k localization accuracy. We compare against direct prompting, which asks an LLM to localize suspicious lines from the code directly.[6] We also compare with recent fault localization methods from two related lines: FlexFL (Xu et al., 2025), which uses LLMs to refine results from existing FL techniques, and LLMAO (Yang et al., 2024) and BAP (Stein et al., 2025), which use model-internal signals to localize likely faulty regions. Table 4 reports Top-1, Top-3, and Top-10 accuracy.

Method	Top 1	Top 3	Top 10
Direct Prompting	51	70	79
FlexFL	38	55	76
LLMAO	63	74	87
BAP	54	62	84
FLARE	**67**	**75**	**89**

Table 4: Performance comparsion of the diagnostic module in FLARE with fault localization baselines on LiveCodeBench.

FLARE achieves the strongest localization performance across all settings. Specifically, it outperforms the strongest baseline, LLMAO, with an absolute improvement of 4% in Top-1 accuracy.

3.6 Effect of Search Budget

We analyze how FLARE's performance varies with two search budgets: the maximum number of refinement iterations N and the number of top-ranked suspicious regions k explored at each iteration. We vary one budget at a time while holding the other fixed on LiveCodeBench with GPT-4o-mini. Figure 2 shows the results.

Increasing the iteration number from 3 to 5 improves Pass@1 from 56.57% to 65.14%. Further increasing the budget yields smaller gains, reaching 65.71% at 7 iterations and 66.86% at 9 iterations. Similarly, increasing the scaling factor k from 4 to 10 improves Pass@1 from 59.43% to 65.14%, while increasing k beyond 10 yields smaller returns relative to the added refinement cost, reaching 66.29% at $k = 13$ and 67.42% at $k = 16$. The default setting ($N = 5$ and $k = 10$) provides the best balance between computational cost and refinement performance.

3.7 Efficiency Analysis

To measure runtime cost, we report the average seconds per task on LiveCodeBench with in Table 5. Compared with Self-Debugging and Self-Refine, FLARE (k=1) reduces the average runtime from 95.96 and 91.94 seconds per task to 73.76 seconds per task. This efficiency arises because FLARE does not require another LLM pass to generate self-debugging feedback or self-critique. Instead, our lightweight diagnostic model provides fine-grained localization guidance. NL-Debugging incurs the highest latency because it adds an intermediate natural-language debugging step before refinement, introducing an additional LLM generation step for each debugging iteration.

Searching over 10 candidate refinements (k=10) increases the runtime cost to 610.19 seconds per task because each iteration generates multiple candidate revisions and evaluates them with test cases. Language model inference accounts for approximately 87% of this runtime cost, while test execution accounts for most of the remaining cost. No-

[6] We provide the prompting details in Appendix A.4.

Method	Seconds/task
Self-Debugging	95.96
Self-Refine	91.94
NL-Debugging	211.68
FLARE (k=1)	73.76
FLARE (k=10)	610.19

Table 5: Runtime comparison on LiveCodeBench.

tably, the lightweight diagnostic model contributes less than 1% of the total runtime. As a result, the additional cost mainly comes from expanded generation and validation, reflecting the intended trade-off between computation and refinement quality. Expanded search is more expensive, but it yields the strongest Pass@1 results in Table 1.

3.8 Case Study

To understand how FLARE refines programs, we present an example from LiveCodeBench. The problem gives N foods and a calorie budget X. Each food has a vitamin type, a vitamin amount, and a calorie cost. The objective is to maximize the smallest total amount among the three vitamins. Thus, a correct solution must use one shared food subset that both stays within budget and balances the three vitamin totals. We provide complete examples in Table 16 and Table 17.

The initial solution incorrectly defines the dynamic programming (DP) state as `dp = [[0] * (X + 1) for _ in range(4)]`, using rows 1 to 3 for the three vitamin types and optimizes each vitamin separately. This design treats each row as an independent knapsack table, which is the core mistake. Each vitamin row can spend the full calorie budget on a different subset of foods, so `dp[1][X]`, `dp[2][X]`, and `dp[3][X]` may not come from the same food selection. However, the task requires one shared subset whose total calories are within X. The final `min` computation therefore treats three separately optimized values as if they were achieved by the same selection, so the reported answer can be too large.

Execution feedback can show that the output is wrong, but it does not identify that the abstraction of the DP state is the root cause. FLARE's diagnostic model ranks the DP initialization line as the most suspicious. This feedback points directly to the flawed state abstraction.

Guided by this diagnostic feedback, FLARE produces a refined solution with a different strategy. This solution uses binary search over a target value m and checks whether one subset of foods can make all three vitamin totals at least m within budget X. This formulation enforces the shared budget constraint and avoids the independent-DP mistake.

By contrast, Self-Debugging keeps the three independent DP tables and adds a binary-search check over a single calorie value. Self-Refine only makes a minor revision, preserving the original per-vitamin DP tables and the final `min` over independently optimized values. NL-Debugging rewrites the table as `dp[c][v]`, but each entry still records only the best value for one vitamin rather than a joint food selection. All revisions continue to combine incompatible choices across vitamin types, reflecting a common refinement failure mode in which the model applies superficial patches without correcting the underlying abstraction. By localizing the suspicious DP state, FLARE helps refinement to address the root cause and produce a correct solution strategy.

3.9 Failure Case Analysis

To understand the limitations of FLARE, we manually inspected 418 failure cases from BigCodeBench and LiveCodeBench. We identify three failure patterns. We provide concrete examples for each category in Appendix D.

Language Model Reasoning Failure (60.53%). Most failures reflect cases where localization identifies useful regions, but the language model still lacks the capability to perform the required reasoning or algorithmic change. For example, in the purchasing problem in Table 13, where buying k units of product i costs $k^2 P_i$, the model falsely uses binary search on the target number of units and validates it with a naive loop. A correct solution instead needs to reason about marginal costs and purchase units in increasing marginal-cost order. In such cases, the bottleneck is not only where to edit the program, but also whether the language model can correctly select the right algorithm or problem solving strategy.

Misinterpretation of Task Requirements. (21.05%) Some failures stem from an incorrect understanding of the problem specification. The model may implement a plausible algorithmic component but apply the wrong constraint, formula, or objective in the final computation. For example, in the stone-merging problem in Table 14, the model applies a linear-basis solution for subset

XOR, but the task requires reasoning about XOR values after whole-bag merge operations. These errors are difficult to refine through localization alone because the faulty code often reflects a mistaken interpretation of the task rather than a localized implementation bug.

Long-Range State and Simulation Errors. (18.42%) The remaining failures often involve complex state transitions, multi-stage simulations, or long-range variable dependencies. In these cases, the model may fix one local transition while breaking a later dependency, or confuse a global state with a local increment. For example, in the simulation problem in Table 15, the model collapses the history of which adults still have stones into a single global counter, losing information needed for later years. This suggests that future work may need richer intermediate-state feedback in addition to line-level localization.

4 Related Work

Iterative Code Refinement. Iterative code refinement typically follows a refinement loop in which a model generates code, receives feedback, and then revises the generated code. Prior work has explored different forms of feedback for this process. Self-Debugging (Chen et al., 2024), Self-Edit (Zhang et al., 2023), Cycle (Ding et al., 2024), and OpenCodeInterpreter (Zheng et al., 2024) use execution feedback to guide code revision. By contrast, Self-Refine relies on the model's own critique to iteratively improve its output (Madaan et al., 2023). Welleck et al. (2023) studies self-correction with model-generated feedback. More recently, NL-Debugging introduces the idea of natural language debugging, where it translates the code solution back to natural language and performs debugging in the natural language space to guide code refinement (Zhang et al., 2025). However, these feedback signals are often coarse-grained or high-level, which is not sufficient to inform the model where to fix the bug. FLARE addresses this limitation by augmenting iterative refinement with line-level diagnostic feedback.

Fault Localization. Traditional fault localization (FL) methods often rely on test coverage information to predict the fault location. For instance, spectrum-based FL analyzes runtime spectra from passing and failing test cases and ranks program entities with suspiciousness metrics such as Jaccard (Abreu et al., 2006), Ochiai (Abreu et al., 2006), and DStar (Wong et al., 2014). Learning-based FL extends this paradigm by learning ranking or prediction models from execution spectra and other program features such as code complexity metrics and error messages (Li et al., 2019; Sohn and Yoo, 2017).

More recently, LLM-based FL methods have used language models to reason about suspicious code regions. AgentFL (Qin et al., 2024) prompts LLMs with diversified debugging information from failing tests. FlexFL (Xu et al., 2025) decomposes localization into two stages that use LLMs to refine localization results obtained from one or more existing FL techniques. SOAPFL (Qin et al., 2025) further standardizes LLM-based localization into structured reasoning stages, including comprehension, navigation, and confirmation. Another line of work probes model-internal signals to predict buggy regions. LLMAO (Yang et al., 2024) fine-tunes bidirectional adapters to predict line-level bugginess, while BAP (Stein et al., 2025) uses weak supervision to interpret attention patterns for scalable localization.

Compared to these FL methods, FLARE trains a lightweight diagnostic model to localize suspicious code regions from the language model's probabilistic signals. Our evaluation shows that this diagnostic model outperforms recent methods including FlexFL (Xu et al., 2025), LLMAO (Yang et al., 2024), and BAP (Stein et al., 2025).

5 Conclusion

We presented FLARE, an iterative code refinement framework with a lightweight diagnostic model that predicts line-level suspiciousness signals for bug localization and code refinement. To account for the uncertainty of diagnostic predictions, FLARE further searches over the top-k suspicious regions and selects candidate revisions using execution outcomes. Experiments on LiveCodeBench and BigCodeBench with five base LLMs show that FLARE consistently outperforms existing iterative refinement methods in most settings, even without candidate search. Ablations confirm that both fine-grained diagnostic feedback and candidate search contribute significantly to the overall gains. These results suggest that fine-grained localization signals serve as reliable guidance for code refinement.

Limitations

Our framework has two main limitations. First, the iterative refinement process introduces overhead. In our efficiency analysis, we show that this overhead is moderate compared with other iterative refinement methods. Nevertheless, we acknowledge that the additional cost may become non-negligible when the number of refinement rounds or the Top-K candidate budget is large.

Second, as discussed in our failure-case analysis, the capability of the language model affects the performance. For example, failures arise from an incorrect understanding of the problem specification. These errors are difficult to resolve through localization alone because the faulty code often reflects a mistaken interpretation of the task rather than a localized implementation bug.

References

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

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Afra Amini, Tim Vieira, Elliott Ash, and Ryan Cotterell. 2025. Variational best-of-n alignment. In *International Conference on Learning Representations*, volume 2025, pages 25717–25737.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *International Conference on Learning Representations*, volume 2024, pages 8746–8825.

Yangruibo Ding, Marcus J Min, Gail Kaiser, and Baishakhi Ray. 2024. Cycle: Learning to self-refine the code generation. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):392–418.

Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, and 1 others. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, and 1 others. 2021. Graphcodebert: Pre-training code representations with data flow. In *International Conference on Learning Representations, ICLR*.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, and 1 others. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*.

Naman Jain, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. 2025. Livecodebench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, volume 2025, pages 58791–58831.

Kevin Jesse, Toufique Ahmed, Premkumar T Devanbu, and Emily Morgan. 2023. Large language models and simple, stupid bugs. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, pages 563–575. IEEE.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. 2026. A survey on large language models for code generation. *ACM Trans. Softw. Eng. Methodol.*, 35(2).

Nan Jiang, Qi Li, Lin Tan, and Tianyi Zhang. 2024. Collu-bench: A benchmark for predicting language model hallucinations in code. *arXiv preprint arXiv:2410.09997*.

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

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-refine: iterative refinement with self-feedback. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

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

Yihao Qin, Shangwen Wang, Yiling Lou, Jinhao Dong, Kaixin Wang, Xiaoling Li, and Xiaoguang Mao. 2025. Soapfl: A standard operating procedure for llm-based method-level fault localization. *IEEE Transactions on Software Engineering*, 51(4):1173–1187.

Jeongju Sohn and Shin Yoo. 2017. Fluccs: Using code and change metrics to improve fault localization. In *Proceedings of the 26th ACM SIGSOFT international symposium on software testing and analysis*, pages 273–283.

Claudio Spiess, David Gros, Kunal Suresh Pai, Michael Pradel, Md Rafiqul Islam Rabin, Amin Alipour, Susmit Jha, Prem Devanbu, and Toufique Ahmed. 2025. Calibration and correctness of language models for code. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 540–552. IEEE.

Adam Stein, Arthur Wayne, Aaditya Naik, Mayur Naik, and Eric Wong. 2025. Where's the bug? attention probing for scalable fault localization. *arXiv preprint arXiv:2502.13966*.

Qwen Team. 2024. Qwen2.5: A party of foundation models.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, NIPS'17, page 6000–6010, Red Hook, NY, USA. Curran Associates Inc.

Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. 2025. Towards understanding the characteristics of code generation errors made by large language models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, pages 2587–2599. IEEE.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2023. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*.

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):290–308.

Chuyang Xu, Zhongxin Liu, Xiaoxue Ren, Gehao Zhang, Ming Liang, and David Lo. 2025. Flexfl: Flexible and effective fault localization with open-source large language models. *IEEE Trans. Softw. Eng.*, 51(5):1455–1471.

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

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

Kechi Zhang, Zhuo Li, Jia Li, Ge Li, and Zhi Jin. 2023. Self-edit: Fault-aware code editor for code generation. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 769–787, Toronto, Canada. Association for Computational Linguistics.

Weiming Zhang, Qingyao Li, Xinyi Dai, Jizheng Chen, Kounianhua Du, Weiwen Liu, Yasheng Wang, Ruiming Tang, Yong Yu, and Weinan Zhang. 2025. NL-debugging: Exploiting natural language as an intermediate representation for code debugging. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1549, Suzhou, China. Association for Computational Linguistics.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. 2024. Opencodeinterpreter: Integrating code generation with execution and refinement. In *Findings of the Association for Computational Linguistics: ACL 2024*, pages 12834–12859.

Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, and 1 others. 2024. Deepseek-coder-v2: Breaking the barrier of closed-source models in code intelligence. *arXiv preprint arXiv:2406.11931*.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, and 1 others. 2025. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. In *International Conference on Learning Representations (ICLR)*.

A Implementation Details

A.1 Model Architecture

Embeddings and projection. We use three learnable modules to embed each lexical unit: (i) a token embedding layer Emb_{tok}, (ii) a type embedding layer Emb_{type}, and (iii) a linear projection Proj for the scalar confidence value. In our implementation, the token embedding dimension is 256, the syntactic-type embedding dimension is 128, and the probability projection dimension is 128. The concatenated vector has dimension 512.

A.2 Training Setup

We optimize the diagnostic model using AdamW with a learning rate of 10^{-4} for 15 epochs and a batch size of 32. Given the ground-truth buggy unit index $j^{\star}$, we minimize a position-level cross-entropy loss over the logits $\{s_j\}_{j=1}^{M}$. We select the best checkpoint according to validation accuracy (exact match of the predicted index).

A.3 Experimental Details

We trained our diagnostic model on a single NVIDIA GeForce RTX 4090 GPU. For training and evaluating the diagnostic model on Collu-Bench, the dataset contains 13130 samples in total, split into 10504 training samples and 2626 test samples. The model was trained for 15 epochs, with a total training time of approximately 10 minutes under this hardware configuration.

A.4 Direct Prompting

We include the prompt in Table 6 for our direct prompting baseline, where the model is asked to directly identify and rank the most suspicious lines in the given code. The input code is formatted with explicit line numbers. `<LINE_NUMBERED_CODE>` is the full program text where each line is prefixed with its 1-indexed line number (e.g., `1: ...`, `2: ...`).

B Syntactic Type

Table 7 lists the syntactic type vocabulary used by the diagnostic model. The vocabulary includes standard code categories, such as `Identifier`, `Keyword`, `Operator`, `Delimiter`, and `Constant`, together with preprocessing symbols. `<PAD>` marks padded positions, `<UNK>` marks unseen or unrecognized types, and `EOS`, `Space`, and `Error` are retained when they appear in the lexical-unit sequence.

```
System message
You are a helpful code analysis assistant.

Direct Prompting
You are an expert code reviewer. Your task is
to identify lines that may contain
hallucinations (factual errors, non-existent
APIs, incorrect function usage, or fabricated
information) in the following code.

The code with line numbers:
<LINE_NUMBERED_CODE>
Instructions:

  1. Analyze the code carefully

  2. Identify the lines that most likely
     contain hallucinations

  3. Rank them from most suspicious to least
     suspicious

  4. Return 10 line numbers, ordered by
     suspicion level (most suspicious first)

Your response should be in the following
format (one line number per line, most
suspicious first):
1. Line <number>: <brief reason>
2. Line <number>: <brief reason>
...
```

Table 6: Direct Prompting LLM for fault localization.

C Prompt Templates

This section lists the prompt templates used for initial generation, refinement, ablations, and top-k candidate search. Templates use placeholders for task text, failed code, execution feedback, and diagnostic feedback.

C.1 Initial Code Generation Prompt

The initial prompt in Table 8 asks the model to produce a complete executable solution from the benchmark-provided task scaffold.

Here, `<STARTER_CODE>` denotes the benchmark-provided starter code with the problem description.

C.2 Iterative Refinement Prompt

For each failed attempt, the refinement prompt in Table 9 includes the original task, the failed code, execution feedback, and optionally a suspicious line predicted by the diagnostic model.

`<EXECUTION_ERROR_OR_TRACEBACK>` contains the observed assertion failure, runtime exception, timeout, or other execution result. Test input and output placeholders are filled when available.

Type	ID
<PAD>	0
<UNK>	1
Constant	2
Delimiter	3
EOS	4
Error	5
Identifier	6
Keyword	7
Operator	8
Space	9
Type	10

Table 7: Syntactic type vocabulary used by the diagnostic model.

```
Initial Code Generation Prompt

Please complete the following Python code based on the
problem description and the provided function/class
structure. Only provide the complete, executable code block
required to solve the problem.

<STARTER_CODE>
```

Table 8: Initial code generation prompt.

C.3 Execution Feedback Only Refinement Prompt

The prompt in Table 10 is used in the ablation study that removes diagnostic feedback and keeps only execution feedback.

C.4 Fine-grained feedback only Refinement Prompt

The prompt in Table 11 is used in the ablation study that removes detailed execution feedback and keeps only the diagnostic suspicious-line signal.

C.5 top-k Candidate Search Prompt

For top-k candidate search, each branch uses the same diagnostic-guided refinement prompt in Table 12 with a different suspicious line from the top-k list.

Each branch generates one candidate revision. Passing candidates terminate the loop; otherwise, failed candidates are ranked by execution outcomes to choose the next program state.

D Detailed Failure Case Analysis

This section provides representative LiveCodeBench failures for the three categories in Section 3.9. These examples illustrate what remains difficult after diagnostic feedback: the language model must still infer the right algorithm, interpret the task correctly, and maintain long-range program state.

```
Iterative Refinement Prompt

Original Task:

The original task was to complete the following Python
function:

<INITIAL_PROMPT_OR_TASK_SCAFFOLD>

Previous Attempt Analysis:

Your last code attempt failed. Here is a detailed analysis
of the failure.

Failure Context:

Input:

<FAILED_TEST_INPUT>

Expected Output:

<EXPECTED_OUTPUT>

Your Actual Output:

<ACTUAL_OUTPUT>

Execution Error or Traceback:

When tested, your code produced the following error:

<EXECUTION_ERROR_OR_TRACEBACK>

Fine-grained Diagnostic Feedback:

The diagnostic model ranks the following line as suspicious:

Suspicious Line:

<SUSPICIOUS_CODE_LINE>

This line may be relevant to the observed execution failure.

Your Failed Code:

<PREVIOUS_FAILED_CODE>

Please provide a new, complete, and corrected version of the
function that resolves all identified issues.
```

Table 9: Iterative refinement prompt. The fine-grained diagnostic feedback block is included only when diagnostic feedback is enabled.

Language Model Reasoning Failure. Table 13 shows a purchasing problem where buying k units of product i costs $k^2 P_i$. A correct solution must reason about marginal costs and purchase units in increasing marginal-cost order. The generated solution instead binary searches the target number of units and uses a naive purchasing loop that does not correctly test whether that target is achievable. This failure reflects a missing algorithmic insight rather than a local implementation mistake.

Misinterpretation of Task Requirements. Table 14 shows a problem about merging whole bags of stones and counting possible final XOR values. The generated solution builds a linear basis over the initial stone counts, which is appropriate for subset-XOR problems but not for this merge operation. The model recognizes a relevant mathematical keyword, XOR, but applies the wrong problem formulation.

Execution feedback only Refinement Prompt

```
Original Task:

The original task was to complete the following Python
function:

<INITIAL_PROMPT_OR_TASK_SCAFFOLD>

Previous Attempt Analysis:

Your last code attempt failed. Here is a detailed analysis
of the failure.

Failure Context:

Input:

<FAILED_TEST_INPUT>

Expected Output:

<EXPECTED_OUTPUT>

Your Actual Output:

<ACTUAL_OUTPUT>

Execution Error or Traceback:

When tested, your code produced the following error:

<EXECUTION_ERROR_OR_TRACEBACK>

Your Failed Code:

<PREVIOUS_FAILED_CODE>

Please provide a new, complete, and corrected version of the
function that resolves all identified issues.
```

Table 10: Execution feedback only refinement prompt.

Long-Range State and Simulation Errors. Table 15 shows a simulation problem where adults give stones over time. A correct solution must track which adults still have stones after each year. The generated solution collapses this history into a single counter, `total_stones`, so it cannot represent adults running out of stones. This illustrates failures where the code loses information needed across multiple state transitions.

Fine-grained feedback only Refinement Prompt

```
Original Task:

The original task was to complete the following Python
function:

<INITIAL_PROMPT_OR_TASK_SCAFFOLD>

Previous Attempt Analysis:

Your last code attempt failed one or more test cases.

Fine-grained Diagnostic Feedback:

The diagnostic model ranks the following line as suspicious:

Suspicious Line:

<SUSPICIOUS_CODE_LINE>

This line may be relevant to the observed failure.

Your Failed Code:

<PREVIOUS_FAILED_CODE>

Please provide a new, complete, and corrected version of the
function that resolves all identified issues.
```

Table 11: Fine-grained feedback only refinement prompt.

top-k Candidate Search Diagnostic Block

```
Fine-grained Diagnostic Feedback:

The diagnostic model ranks the following line as suspicious:

Suspicious Line:

<BRANCH_SPECIFIC_SUSPICIOUS_LINE>

This line may be relevant to the observed execution failure.
```

Table 12: Branch-specific diagnostic feedback block used in top-k Candidate Search. Each branch uses the same diagnostic-guided refinement prompt but substitutes a different top-k suspicious line.

<table>
<tr><th>Failure Pattern</th><th>Original Task Description</th><th>Generated Failure Case Snippet</th></tr>
<tr><td>Language Model Reasoning Failure</td><td>Task: There are N types of products, each having 10^{100} units in stock. You may buy any non-negative integer number of units of each product, and buying k units of the i-th product costs $k^2 P_i$ yen. Given a budget of at most M yen for the total purchase across all product types, determine the maximum possible total number of units you can buy (summing over all types).</td><td><pre><code>def main():
  import sys
  input = sys.stdin.read
  data = input().split()
  N = int(data[0])
  M = int(data[1])
  P = list(map(int, data[2:N+2]))
  def can_buy(x):
    total_cost = 0
    for price in P:
      k = 0
      while True:
        cost = (k + 1) * (k + 1) * price
        if total_cost + cost > M:
          break
        total_cost += cost
        k += 1
        if total_cost > M:
          return False
      if total_cost > M:
        return False
    return total_cost <= M
  low, high = 0, 10**9
  answer = 0
  while low <= high:
    mid = (low + high) // 2
    if can_buy(mid):
      answer = mid
      low = mid + 1
    else:
      high = mid - 1
  print(answer)
if __name__ == "__main__":
  main()</code></pre></td></tr>
</table>

Table 13: Failure case (Pattern 1): Language Model Reasoning Failure.

<table>
<tr><th>Failure Pattern</th><th>Original Task Description</th><th>Generated Failure Case Snippet</th></tr>
<tr><td>Misinterpretation of Requirements</td><td>Task: There are N bags labeled $1, 2, \ldots, N$, where bag i initially contains A_i stones. You may repeat the following operation any number of times (possibly zero): choose two bags A and B, and move all stones from bag A into bag B (so bag A becomes 0 and bag B increases accordingly). Let B_i denote the final number of stones in bag i; consider the value $B_1 \oplus B_2 \oplus \cdots \oplus B_N$, where $\oplus$ is bitwise XOR (defined by binary digits: the 2^k bit is 1 iff exactly one operand has a 1 at that bit, and XOR over multiple integers is taken associatively). Under the given setting it can be proved that only finitely many XOR values are achievable; compute how many distinct values can occur.</td><td><pre><code>def main():
  import sys
  input = sys.stdin.read
  data = input().split()
  N = int(data[0])
  A = list(map(int, data[1:N+1]))
  basis = []
  for number in A:
    for b in basis:
      number = min(number, number ^ b)
    if number > 0:
      basis.append(number)
      basis.sort(reverse=True)
  print(1 << len(basis))
if __name__ == "__main__":
  main()</code></pre></td></tr>
</table>

Table 14: Failure case (Pattern 2): Misinterpretation of Requirements.

<table>
<tr><th>Failure Pattern</th><th>Original Task Description</th><th>Generated Failure Case Snippet</th></tr>
<tr><td>Long-Range State and Simulation Errors</td><td>Task: On a certain planet there are N aliens, all initially minors; the i-th alien currently has A_i stones and will become an adult exactly i years later. Whenever an alien becomes an adult, every alien who is already an adult and has at least one stone gives exactly one stone as a congratulatory gift to the alien who has just become an adult. Assuming no new aliens will be born, determine how many stones each alien has after N years.</td><td>

```
def main():
  import sys
  input = sys.stdin.read
  data = input().split()
  N = int(data[0])
  A = list(map(int, data[1:N+1]))
  B = A[:]
  total_stones = 0
  for i in range(N):
    B[i] += total_stones
    if B[i] > 0:
      total_stones += 1
      B[i] -= 1
  print(" ".join(map(str, B)))
if __name__ == "__main__":
  main()
```

</td></tr>
</table>

Table 15: Failure case (Pattern 3): Long-Range State and Simulation Errors.

<table>
<tr><th>Item</th><th>Content</th></tr>
<tr><td>Task Description</td><td>Given N foods and a maximum total calorie limit X, each food provides a specific amount of exactly one type of vitamin, Vitamin 1, 2, or 3, and costs a certain number of calories. The goal is to choose a subset of foods such that the total consumed calories do not exceed X, while maximizing the minimum total intake among the three vitamins. In other words, the solution should balance the intake of all three vitamins and make the lowest accumulated vitamin amount as large as possible.</td></tr>
<tr><td>Initial Code</td><td>

```
def main():
    import sys
    input = sys.stdin.read
    data = input().splitlines()
    N, X = map(int, data[0].split())
    foods = [tuple(map(int, line.split())) for line in data[1:N+1]]
    dp = [[0] * (X + 1) for _ in range(4)]
    for v, a, c in foods:
        for j in range(X, c - 1, -1):
            dp[v][j] = max(dp[v][j], dp[v][j - c] + a)
    max_min_intake = 0
    for calories in range(X + 1):
        min_intake = min(dp[1][calories], dp[2][calories], dp[3][calories])
        max_min_intake = max(max_min_intake, min_intake)
    print(max_min_intake)
if __name__ == "__main__":
    main()
```

</td></tr>
<tr><td>Predicted Suspicious Line</td><td>

```
dp = [[0] * (X + 1) for _ in range(4)]
```

</td></tr>
<tr><td>Final Passing Candidate</td><td>

```
def main():
    import sys
    input = sys.stdin.read
    data = input().splitlines()
    N, X = map(int, data[0].split())
    foods = [tuple(map(int, line.split())) for line in data[1:N+1]]
    vitamin_foods = {1: [], 2: [], 3: []}
    for v, a, c in foods:
        vitamin_foods[v].append((a, c))
    def can_achieve(m):
        dp = [False] * (X + 1)
        dp[0] = True
        for v in range(1, 4):
            new_dp = [False] * (X + 1)
            for a, c in vitamin_foods[v]:
                if a >= m:
                    for j in range(X, c - 1, -1):
                        if dp[j - c]:
                            new_dp[j] = True
            dp = new_dp
        return any(dp)
    low, high = 0, max(a for v, a, c in foods)
    answer = 0
    while low <= high:
        mid = (low + high) // 2
        if can_achieve(mid):
            answer = mid
            low = mid + 1
        else:
            high = mid - 1
    print(answer)
if __name__ == "__main__":
    main()
```

</td></tr>
</table>

Table 16: Example of FLARE from LiveCodeBench. FLARE correctly solves the task with refinement that uses a binary search over a target value m and checks whether one subset of foods can make all three vitamin totals at least m within budget X.

<table>
<tr><th>Method</th><th>Failed Code Snippet</th><th>Failure Analysis</th></tr>
<tr><td>Self-Debug</td><td>

```python
def main():
    import sys
    from collections import defaultdict
    input = sys.stdin.read
    data = input().splitlines()
    N, X = map(int, data[0].split())
    foods = [tuple(map(int, line.split())) for line in data[1:N+1]]
    vitamin_foods = defaultdict(list)
    for v, a, c in foods:
        vitamin_foods[v].append((a, c))
    dp = [[0] * (X + 1) for _ in range(4)]
    for v in range(1, 4):
        for a, c in vitamin_foods[v]:
            for cal in range(X, c - 1, -1):
                dp[v][cal] = max(dp[v][cal], dp[v][cal - c] + a)
    low, high = 0, min(dp[1][X], dp[2][X], dp[3][X]) + 1
    while low < high:
        mid = (low + high) // 2
        can_achieve = False
        for cal in range(X + 1):
            if dp[1][cal] >= mid and dp[2][cal] >= mid and dp[3][cal] >= mid:
                can_achieve = True
                break
        if can_achieve:
            low = mid + 1
        else:
            high = mid
    print(low - 1)
if __name__ == "__main__":
    main()
```

</td><td>The solution computes three independent knapsack tables, one for each vitamin type. However, the feasibility check is incorrect: it checks whether there exists the same calorie value <code>cal</code> such that all three independent DP tables reach the target value. This does not model the shared calorie budget. The correct condition should check whether there exist calorie allocations c_1, c_2, c_3 such that $c_1 + c_2 + c_3 \leq X$ and each vitamin reaches the target.</td></tr>
<tr><td>NL-Debugging</td><td>

```python
def main():
    import sys
    input = sys.stdin.read
    data = input().splitlines()
    N, X = map(int, data[0].split())
    foods = []
    for i in range(1, N + 1):
        V, A, C = map(int, data[i].split())
        foods.append((V, A, C))
    def can_achieve(min_vitamin):
        dp = [[0] * 4 for _ in range(X + 1)]
        for V, A, C in foods:
            for c in range(X, C - 1, -1):
                dp[c][V] = max(dp[c][V], dp[c - C][V] + A)
        return dp[X][1] >= min_vitamin and dp[X][2] >= min_vitamin and dp[X][3]
      >= min_vitamin
    low, high = 0, 200000
    answer = 0
    while low <= high:
        mid = (low + high) // 2
        if can_achieve(mid):
            answer = mid
            low = mid + 1
        else:
            high = mid - 1
    print(answer)
if __name__ == "__main__":
    main()
```

</td><td>The DP state is invalid for the problem. <code>dp[c][v]</code> only records the best amount of one vitamin under calorie budget <code>c</code>; it does not represent a joint selection of foods. Therefore, <code>dp[X][1]</code>, <code>dp[X][2]</code>, and <code>dp[X][3]</code> may correspond to three different subsets of foods, each using up to X calories independently. The algorithm may thus falsely conclude that all vitamin requirements are simultaneously feasible under the single shared budget X.</td></tr>
<tr><td>Self-Refine</td><td>

```python
def main():
    import sys
    input = sys.stdin.read
    data = input().splitlines()
    N, X = map(int, data[0].split())
    foods = []
    for i in range(1, N + 1):
        V, A, C = map(int, data[i].split())
        foods.append((V, A, C))
    dp = [[0] * (X + 1) for _ in range(4)]
    for V, A, C in foods:
        for c in range(X, C - 1, -1):
            dp[V][c] = max(dp[V][c], dp[V][c - C] + A)
    max_min_vitamin = 0
    for c in range(X + 1):
        min_vitamin = min(dp[1][c], dp[2][c], dp[3][c])
        max_min_vitamin = max(max_min_vitamin, min_vitamin)
    print(max_min_vitamin)
if __name__ == "__main__":
    main()
```

</td><td>The solution keeps the same flawed per-vitamin DP formulation after refinement. It computes <code>dp[V][c]</code> independently for each vitamin and then takes <code>min(dp[1][c], dp[2][c], dp[3][c])</code> at the same calorie value <code>c</code>. These values may come from different food subsets and do not guarantee a jointly feasible selection under the shared budget X. Thus, the code can overestimate the achievable minimum vitamin intake.</td></tr>
</table>

Table 17: Case study of failed solutions produced by Self-Debug, NL-Debugging, and Self-Refine. All failures arise from an incorrect modeling of the shared calorie budget across three vitamin categories.