Prism: A Multi-Solution Reasoning and Synthesis Framework for Repository-Level Issue Resolution

Yibo Wang
Northeastern University
Shenyang, China
yibowangcz@outlook.com

Guotian Wang
Northeastern University
Shenyang, China
2471386@stu.neu.edu.cn

Zhihao Peng
Northeastern University
Shenyang, China
2471378@stu.neu.edu.cn

Ying Wang✉
Northeastern University
Shenyang, China
wangying@swc.neu.edu.cn

Chang Xu
Nanjing University
Nanjing, China
changxu@nju.edu.cn

Zhiliang Zhu
Northeastern University
Shenyang, China
ZHUZhiLiang_NEU@163.com

Abstract

Large language model (LLM)-based agents have achieved notable success in repository-level issue resolution, yet existing frameworks still face two key bottlenecks: insufficient diversity among candidate repair solutions and limited ability to combine complementary strengths across candidate solutions. To address these challenges, we introduce Prism, a multi-solution reasoning and synthesis framework that adopts a coarse-to-fine paradigm to systematically generate, refine, and integrate diverse repair solutions. Prism comprises three stages: (1) a *global exploration* mechanism with contrastive semantic constraints that broadens the search space by encouraging divergence across historical solutions; (2) an *A* algorithm*-inspired *local exploration* mechanism that identifies suitable branching points to generate divergent sub-solutions; and (3) a multi-agent collaborative *solution synthesis* process where reviewer and judge agents jointly evaluate, complement, and integrate partially correct candidate solutions. We evaluate Prism on *SWE-bench Verified* and *SWE-bench-Live Lite*. With *Claude-4.6-Sonnet* and *DeepSeek-V3.2-Reasoner*, Prism achieves *Pass@1* rates of 82.0% and 80.0% on *SWE-bench Verified*, respectively, outperforming the compared baselines. On *SWE-bench-Live Lite*, designed to reduce data leakage concerns, Prism outperforms the reported baselines. Further analyses show that Prism expands the solution space and that its solution synthesis stage uniquely resolves 15 issues unaddressed by isolated candidates.

CCS Concepts

• **Software and its engineering → Software post-development issues**; **Automatic programming**.

Keywords

Repository-Level Issue Resolution, Large Language Models, Software Engineering Agents, Automated Program Repair

ASE '26, Munich, Germany

ACM ISBN 979-8-4007-2882-2/2026/10
https://doi.org/10.1145/3832783.3837451

ACM Reference Format:
Yibo Wang, Guotian Wang, Zhihao Peng, Ying Wang, Chang Xu, and Zhiliang Zhu. 2026. Prism: A Multi-Solution Reasoning and Synthesis Framework for Repository-Level Issue Resolution. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3832783.3837451

1 Introduction

In recent years, large language models (LLMs) have made significant progress in code generation and issue resolution within the software engineering domain [9, 14, 41]. LLM-powered agents leverage autonomous exploration and deep understanding of large-scale code repositories to perform tasks such as fault localization, patch generation, and code refactoring [5]. With the advent of repository-level benchmarks like *SWE-bench* [18], the field has shifted from pipeline-based approaches to agentic architectures that support planning and tool invocation [26, 54, 57]. State-of-the-art (SOTA) agents such as *SWE-Agent* [50], *OpenHands* [44], and *Trae* [10] achieve strong performance on these benchmarks. However, systematic analyses reveal that existing agents still struggle with the complexity of repository-level issue resolution—particularly in fault localization, context understanding, and patch generation [17, 27].

Effective patches for repository-level issue resolution often require coordinated modifications across multiple files, making it difficult for a single solution to address all aspects of a problem [10, 24]. Leveraging multiple diverse reasoning paths enables examining issues from different perspectives, expanding the search space for correct patches [29]. This aligns with *Test-Time Scaling* (TTS), which allocates additional inference-time resources to generate multiple candidate solutions without updating model parameters [36, 51]. Within this framework, *Best-of-N sampling* has become an important strategy for improving issue resolution rates [6, 29]. However, the effectiveness of TTS strategies heavily depends on the output diversity of the base model [23, 58]. Although LLMs trained with *Reinforcement Learning from Human Feedback* (RLHF) produce higher-quality outputs, they often do so at the cost of reduced diversity, limiting the practical effectiveness of *Best-of-N* [30]. Experiments on *SWE-bench* reveal a robust positive correlation between solution diversity and issue resolution success rates. As the number of candidate solutions increases, the resolution success rate shows a steady increase [29, 63].

However, existing repository-level issue resolution agent frameworks [10, 24] typically adopt sampling-based strategies (e.g., *temperature* adjustment or top-k sampling) to generate multiple candidate solutions, followed by self-reflection for iterative refinement, and ultimately select a single patch via ranking or voting. Although these approaches yield some performance gains, they face key bottlenecks when handling complex repository-level issues:

- *Lack of Solution Diversity Coupled with Misleading Self-reflection.* Existing agent frameworks attempt to enhance diversity via high-*temperature* sampling, but their effectiveness is constrained by two inherent factors of LLMs: *mode collapse* and *misleading self-reflection*. RLHF-fine-tuned models tend to produce low diversity content—keeping sampled solutions homogeneous even with high-*temperature* sampling [2]. Simultaneously, erroneous self-reflection feedback can cause agents to abandon promising exploration paths. Together, these factors can limit the generation of diverse solutions and lead to premature convergence on homogeneous, incorrect solutions (see Section 2.1).
- *Lack of Solution Synthesis Capabilities.* Existing agent frameworks evaluate candidate solutions in isolation, relying on ranking or voting to select a single patch. This paradigm overlooks the complementary strengths across diverse solutions [40], which capture different aspects of a problem. For instance, one solution may identify the root cause while another addresses cross-file coordination logic. By evaluating candidates in isolation, such mechanisms fail to leverage complementarity and cannot synthesize a complete patch covering all critical dimensions (see Section 2.2).

How to encourage substantive exploration of diverse solutions and synthesize complementary partial solutions into comprehensive patches remains underexplored. To address these bottlenecks, we propose Prism, a multi-solution reasoning and synthesis framework that adopts a coarse-to-fine paradigm to systematically generate, refine, and synthesize diverse fixing solutions through three stages: *global exploration*, *local exploration*, and *solution synthesis*. Its key innovations are: (1) **A global exploration mechanism using contrastive semantic constraints** that injects the root cause, expected behavior, modification strategy, and relevant context of historical solutions into the prompt, encouraging new solutions to maintain semantic divergence and thereby broadening the search space. (2) ***An A* algorithm-inspired*** [12, 55] ***local exploration mechanism*** that uses heuristic scoring to identify suitable branching points for generating locally divergent sub-solutions. This mechanism identifies the *suitable branching point* in the agent's execution trajectory, where sufficient context has been accumulated but the specific modification strategy remains undetermined—thereby balancing exploration and exploitation to generate multiple locally divergent sub-solutions. (3) **A multi-agent collaborative synthesis process** where a reviewer agent evaluates the counterpart solution for completeness and regression risk, while a judge agent performs defect verification, complementarity analysis, and semantic synthesis, ultimately producing an integrated final solution.

To evaluate the effectiveness of Prism, we conducted experiments on two benchmarks: the widely recognized *SWE-bench Verified* [18] and the leakage-reduced *SWE-bench-Live Lite* [64]. Additionally, we assessed the contribution of each component of Prism using *DeepSeek-V3.2-Reasoner* on the *SWE-bench Verified* benchmark. The main findings are as follows: On *SWE-bench Verified*, Prism achieves 82.0% *Pass@1* with *Claude-4.6-Sonnet* as the base model and 80.0% *Pass@1* with the open-source *DeepSeek-V3.2-Reasoner*, outperforming the compared baseline agents. Under the same *DeepSeek-V3.2-Reasoner* configuration, Prism achieves a 10.0% improvement over *Mini-SWE-Agent* (70.0%). On *SWE-bench-Live Lite*, Prism achieves 44.67% *Pass@1* with *Claude-4.6-Sonnet* and 42.33% *Pass@1* with *DeepSeek-V3.2-Reasoner*, outperforming the reported *SWE-Agent* (with *Claude-4.5-Sonnet* achieving 36.0%). These results provide complementary evidence of Prism's generalization under reduced data leakage concerns. A stage-wise evaluation of the global and local exploration stages shows that Prism expands the solution space to generate high-quality patches, rather than relying on increasing the number of candidate solutions to boost performance. Experiments further show the unique contribution of the *solution synthesis stage*, revealing 15 issues that only the integrated solutions could resolve.

In summary, the main contributions of this paper are as follows:

- ***An Effective Issue Resolution Agent Framework:*** We introduce Prism, a multi-path reasoning and synthesis framework for repository-level issue resolution that generates and integrates diverse repair solutions across three stages.
- ***Thorough Evaluation:*** We evaluate Prism on *SWE-bench Verified* and *SWE-bench-Live Lite*, demonstrating that it achieves SOTA performance with both *DeepSeek-V3.2-Reasoner* and *Claude-4.6-Sonnet* against existing issue resolution agents.
- ***Experimental Results:*** With *Claude-4.6-Sonnet* and *DeepSeek-V3.2-Reasoner*, Prism achieves *Pass@1* rates of 82.0% and 80.0% on *SWE-bench Verified*, respectively, outperforming the compared baseline agents. On *SWE-bench-Live Lite*, Prism + *DeepSeek-V3.2-Reasoner* achieves 42.33%, surpassing the leading closed-source combination (*SWE-Agent* + *Claude-4.5-Sonnet*, 36.0%).

2 Limitations of Existing Work

To illustrate the limitations of existing issue resolution agents, we analyze *Django#33413*[7] (a representative issue from *SWE-bench Verified*; see Figure 1). The root cause is that `ForeignKey` fails to inherit the `db_collation` attribute from its target field, causing **MySQL Error 3780** during schema migration. Fixing it requires coordinated changes across two files: (1) `related.py`, where the `ForeignKey` must include `collation` in its database parameters to expose the target field's collation; and (2) `schema.py`, where the schema editor must accommodate this collation change and propagate it to the relevant columns.

2.1 *Limitation 1: Lack of Solution Diversity Coupled with Misleading Self-reflection*

Current agents for issue resolution typically rely on sampling-based strategies (e.g., increasing *temperature* or *top-k*) to boost solution diversity. However, their effectiveness is limited by the coupling of intrinsic LLM "*mode collapse*" and "*misleading self-reflection*". RLHF-tuned models tend to generate repetitive, low-diversity content [2], so even with stochasticity, sampled reasoning paths remain highly homogeneous [15, 42, 62]. Compounding this, the self-reflection often produces false-negative feedback, causing agents to abandon correct solutions after exploring them.

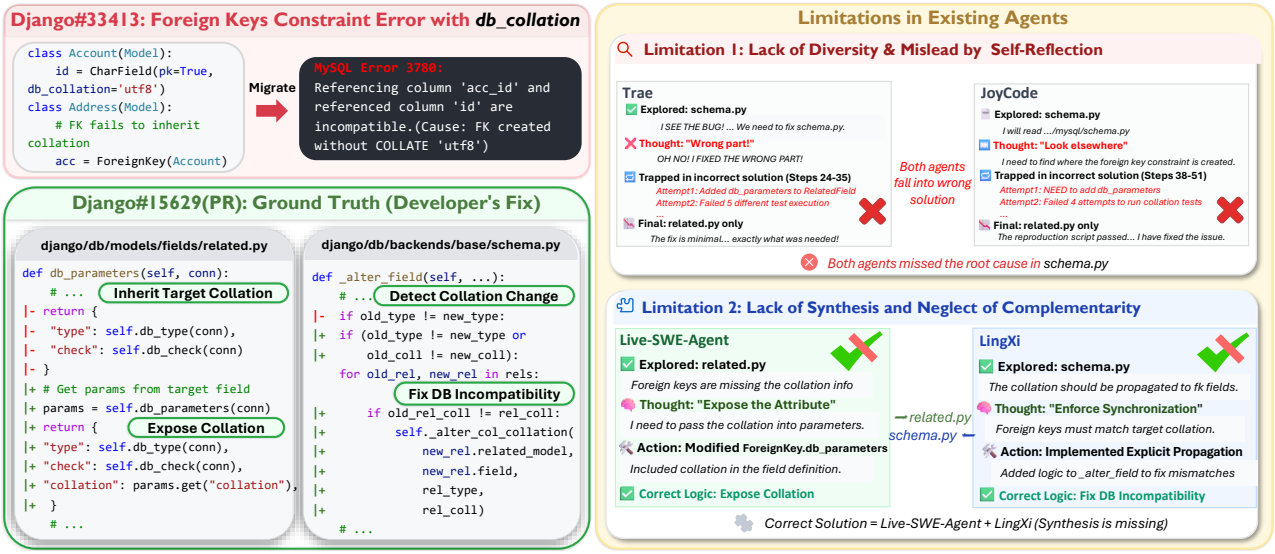


Figure 1: An example for illustrating limitations of the existing agent frameworks based on Django Issue #33413.

The synergy between mode collapse and misleading self-reflection severely restricts the agent's exploration capabilities. Once a correct solution is rejected due to flawed feedback, the agent attempts to regenerate alternatives. However, constrained by mode collapse, it fails to synthesize substantively different reasoning paths. Instead, it stagnates within the erroneous logic, leading to premature convergence on homogeneous, incorrect solutions. Figure 1 demonstrates that although the state-of-the-art agents *Trae*[10] and *JoyCode*[19] utilize high-*temperature* sampling to foster diversity, this approach proves insufficient. Both agents initially targeted `schema.py` (a correct candidate), but were derailed by erroneous self-reflection and prematurely abandoned this direction. More critically, despite high *temperature* sampling, their subsequent attempts to generate solutions for `related.py` remained highly homogeneous in their underlying logic, ultimately resulting in incorrect patches:

- ***Trae* [10]:** Upon analyzing `schema.py`, the self-reflection mistakenly flagged it as the "*wrong part!*", forcing an immediate shift to `related.py`. Even after persistent resampling (*Steps 24–35*), *Trae* showed no strategic divergence, remaining trapped in a local optimum of "adding `db_parameters`" and failing to identify distinct solutions like "exposing `collation`". This inability reveals a critical bottleneck in generating diverse solutions.
- ***JoyCode* [19]:** Upon analyzing `schema.py`, *JoyCode* was similarly derailed by erroneous reflection ("*look elsewhere*"), forcing a pivot to `related.py` (*Steps 38–51*). Critically, despite repeated dynamic test failures, the agent failed to leverage its sampling to generate different alternatives. Instead, it became trapped in a vicious cycle: generating an incorrect patch, failing the test, and subsequently regenerating a highly similar erroneous patch.

2.2 *Limitation 2: Lack of Solution Synthesis Capabilities*

Existing agent frameworks typically evaluate generated solutions in isolation, relying on ranking or voting to identify a single optimal patch. This paradigm overlooks the potential complementarity among different solutions. As illustrated in Figure 1, the correct resolution for *Django#33413* necessitates coordinated modifications across both files `related.py` and `schema.py`. **Notably, two agents independently addressed partial aspects of the issue. However, due to the framework's inability to synthesize these complementary solutions, it failed to generate a complete and correct patch**:

- ***Live-SWE-Agent* [47]:** It correctly focused on `related.py`. Its reasoning paths ("*expose the attribute*") accurately identified that foreign keys were missing `collation` information. Based on this, it modified `ForeignKey.db_parameters` to expose the `collation` attribute (*Correct Logic: Expose Collation*). However, *Live-SWE-Agent* overlooked the issue located in `schema.py`.
- ***LingXi* [52]:** In contrast to *Live-SWE-Agent*, *LingXi* accurately pinpointed the root cause within `schema.py`. Its reasoning paths ("*enforce synchronization*)" recognized that synchronization between the foreign key and the target collation must be enforced during schema migration. It implemented the crucial explicit propagation logic within the `_alter_field` function, thereby fixing the underlying database incompatibility (*Correct Logic: Fix DB Incompatibility*). Unfortunately, *LingXi* failed to locate the issue in `related.py`.

3 Approach

Insight: To address **limitation 1** (lack of solution diversity coupled with misleading self-reflection), **we propose contrastive semantic constraints**, which guide agents to avoid repeating previously explored fixing logic along dimensions such as root cause hypotheses and expected behaviors, thereby systematically broadening the solution space. To tackle **limitation 2** (lack of solution synthesis), **we introduce a multi-agent review and synthesis mechanism**, where multiple agents critically review and integrate candidate solutions, extracting strengths from each to synthesize a globally complete final solution from local optima.

Architecture Overview: We propose Prism, a novel agent framework that overcomes existing bottlenecks in issue resolution through

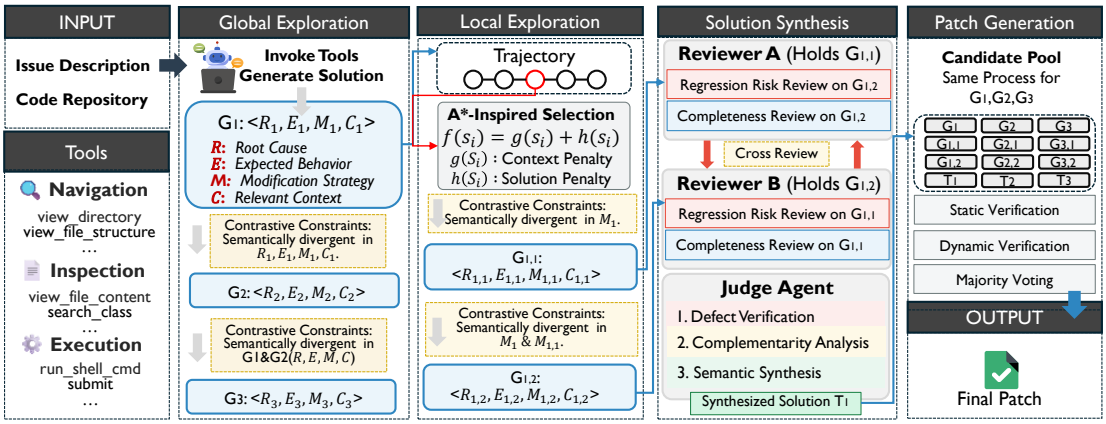


Figure 2: The workflow of Prism

multi-path reasoning and synthesis. It adopts a coarse-to-fine exploration and synthesis paradigm, promoting solution diversity while progressively converging to high-quality patches. Given an issue description and a code repository, Figure 2 illustrates its workflow:

- ***Global Exploration***: This stage serves as the foundation for achieving broad solution coverage. Led by the *plan agent* and guided by ***contrastive semantic constraints***, it iteratively generates N solutions $\{G_1, G_2, \ldots, G_N\}$, with explicit constraints that promote semantic differences across four dimensions: *root cause hypothesis*, *expected behavior*, *modification strategy* and *relevant context*. This mechanism helps mitigate the solution homogeneity problem associated with *mode collapse*, enabling the *global exploration stage* to cover a broader solution space.
- ***Local Exploration***: This stage aims to fully leverage the context accumulated during *global exploration* for in-depth expansion along modification location and strategy. We define the execution trajectory $T = \{s_0, s_1, \ldots, s_L\}$, where each state s_i records a single agent–repository interaction (tool calls, feedback, and context). To balance exploration and exploitation, Prism introduces a *critical state selection mechanism* inspired by the A* algorithm [12, 55]. Using a heuristic scoring strategy, it selects a *critical state* within T as a *suitable branching point*, where sufficient context has been gathered but the exact modification location and strategy remain underdetermined. From such a state, Prism diverges to generate N_L sub-solutions, each with a distinct modification plan.
- ***Solution Synthesis***: This stage achieves complementary strengths across solutions. Prism builds a multi-agents with *Reviewer* and *Judge* roles. Structured cross-review identifies omissions and regression risks in individual sub-solutions. Then, the *Judge* performs semantic-level synthesis, producing N_G integrated solutions that combine advantages of multiple candidate approaches.
- ***Patch Generation***: This stage converts candidate solutions into patches, filters invalid ones through static checks and visible repository tests, and selects one final patch via majority voting following *Trae* [10].

3.1 Global Exploration

The goal of the *global exploration* is to generate a set of diverse solutions. This stage is led by a *plan agent*, which conducts preliminary exploration across the codebase and formulates resolution

Task: You are a planning agent for resolving a repository issue. Generate one concrete repair plan that is semantically different from previously generated plans.
Contrastive constraint:
When generating the new plan G_k, avoid repeating any prior fixing logic in H. The new plan should differ from previous plans under semantic judgment: **1. Root cause hypothesis**: [Explain the underlying mechanism that leads to the issue.] 2.**Expected behavior after the fix:** [Describe the intended program behavior/state after the bug is resolved.] **3. Modification location and strategy:** [Specify the files/classes/functions to modify and the corresponding repair strategy.] **4. Relevant repository context:** [List relevant code snippets/dependencies that support this solution.]
Input: - [Issue] - [Repository] - [Historical Solutions $H=\{G_1 \ldots G_{k-1}\}$]
Output: Modification Solution G_k.

Figure 3: Prompt Templates of Contrastive Constraint (simplified)

strategies. As noted in Section 2.1, even when existing agents adopt high-*temperature sampling*, the resulting solutions remain highly homogeneous in terms of their underlying issue resolution logic.

We define **semantic divergence** as differences in the underlying issue-fixing logic among candidate solutions. To promote such divergence during generation, Prism introduces *contrastive semantic constraints*. These constraints guide each generated solution to avoid repeating the fixing logic of prior solutions across several semantic dimensions, thereby systematically expanding the solution space. The four semantic dimensions are defined as follows:

- ***Root Cause Analysis (R)***: An explanation of the underlying mechanism that leads to the issue, e.g., "the `ForeignKey` does not inherit the `db_collation` attribute from the target field."
- ***Expected Behavior (E)***: The intended program state after the bug is resolved, e.g., "`ForeignKey` should automatically propagate the collation settings of the target field."
- ***Modification Strategy (M)***: A natural language description of the specific locations to be modified (e.g., code files, classes, functions) along with the corresponding modification strategy.
- ***Relevant Context (C)***: Contextual information that supports the proposed solution, including the issue description and relevant code snippets from the repository.

In Prism, semantic divergence is not computed as a distance score. Instead, it is enforced as a generation-time qualitative constraint through a contrastive prompt (see Figure 3), motivated by prompting-based methods for improving LLM output diversity [58, 62]. Specifically, the *plan agent* is instructed to output

each solution as a structured four-tuple $G_i = \langle R_i, E_i, M_i, C_i \rangle$. When generating the k-th solution, we inject the root cause hypotheses $\{R_1, \ldots, R_{k-1}\}$, expected behaviors $\{E_1, \ldots, E_{k-1}\}$, modification strategies $\{M_1, \ldots, M_{k-1}\}$, and contextual information $\{C_1, \ldots, C_{k-1}\}$ from the historical solution set into the prompt. Through contrastive instructions, we require the *plan agent* to avoid semantically equivalent fixing directions in these dimensions, thereby promoting semantic diversity among solutions during generation.

Algorithm 1: Global Exploration with Contrastive Semantic Constraints

Input: Issue description $\mathcal{I}$, Repository $\mathcal{R}$, Number of solutions N_G
Output: A diverse set of candidate solutions H

```
1  H ← ∅ ; // Initialize the set of historical solutions
2  for k = 1 to N_G do
       /* 1: Formulate contrastive semantic constraints */
3      if k = 1 then
4          P_cons ← ∅;
5      else
           /* Construct constraints to avoid repeating
              prior fixing logic in Root Cause (R),
              Expected Behavior (E), Modification Strategy
              (M), and Relevant Context (C)            */
6          P_cons ← Prompt requiring G_k to avoid fixing logic that is
             semantically equivalent to that of any prior G_i with
             respect to R, E, M, and C;
       /* 2: Context exploration                       */
7      C_k ← EXPLORE_REPOSITORY(I, R, H, P_cons);
       /* 3: Solution generation                       */
8      G_k ← GENERATE_SOLUTION(I, C_k, H, P_cons);
9      Let G_k = ⟨R_k, E_k, M_k, C_k⟩;
       /* 4: Update the historical solution set        */
10     H ← H ∪ {G_k};
11 return H
```

The *global exploration* process is presented in Algorithm 1. Given an issue description $\mathcal{I}$ and a code repository $\mathcal{R}$, the *plan agent* iteratively applies contrastive constraints to promote semantic diversity among the N_G generated fixing solutions. Specifically, during the initial solution generation phase ($k = 1$), the contrastive constraint instruction P_{cons} in the prompt is initialized as empty, as the historical solution set H is currently empty and no contrastive constraints need to be applied (Lines 3–4). The *plan agent* first invokes tools to gather context C_1 from the code repository $\mathcal{R}$ (Line 7), generates the first detailed fixing solution G_1 (Line 8), and adds it to the historical solution set H (Line 10). Subsequently, the iterative generation phase begins ($k \geq 2$). When generating the k-th solution, the *plan agent* takes the historical solution set H as input and imposes explicit contrastive constraint instructions within the prompt, requiring the new solution G_k to avoid fixing logic that is semantically equivalent to prior solutions in H across the dimensions of root cause hypothesis R, expected behavior E, modification strategy M, and relevant context C (Lines 5–6). Under this constraint, the *plan agent* again performs context exploration (Line 7) and solution generation (Line 8), then appends the new solution to H (Line 10). Notably, the strictness of this constraint automatically increases with the iteration round: for example, generating G_3 requires avoiding fixing logic that is semantically equivalent to either G_1 or G_2. Through this iterative exploration mechanism with semantic constraints, PRISM discourages the agent from converging to a single fixing direction, thereby promoting broader coverage of the solution space during the *global exploration* stage.

3.2 Local Exploration

The *local exploration* stage aims to further extend and optimize the fixing solutions generated during the *global exploration* stage. In contrast to *global exploration*, *local exploration* does not re-explore different root cause hypotheses R or expected behaviors E; instead, it fully leverages the context accumulated in the *global exploration* stage and focuses on generating diverse sub-solutions along the modification strategy dimension (M).

However, how to reuse the contextual information accumulated during *global exploration* is crucial. We define the sequence of states generated by the agent's interactions with the repository during issue resolution as an execution trajectory $T = \{s_0, s_1, \ldots, s_L\}$, where each state s_i records *a single interaction between the agent and the code repository, including tool calls, environment feedback, and contextual information.* **The core challenge of *local exploration* lies in determining at which state s_i of the trajectory T to perform branching exploration in order to efficiently obtain differentiated sub-solutions.** This involves an *exploration-exploitation trade-off*: if we restart from the trajectory's starting point s_0, the accumulated contextual information will be wasted; conversely, if we branch from the trajectory's endpoint s_L, the execution trajectory at that stage has often converged to a fixed direction, making it difficult to generate divergent sub-solutions. Therefore, a suitable *branching point*(i.e., a *critical state*) should satisfy two conditions: (1) *sufficient repository context has been accumulated*, and (2) *the specific modification strategies remain underdetermined.*

Inspired by the A^* *algorithm* [12, 55], we propose a critical state selection mechanism to address the aforementioned trade-off. The traditional A^* *algorithm* balances exploration and exploitation by combining two independent heuristic functions to guide the search direction. Drawing on this idea, rather than performing full A^* *algorithm* graph search, we define a heuristic branching cost function $f(s_i)$ for each state s_i in the existing trajectory. A smaller value of $f(s_i)$ indicates that the corresponding s_i is more suitable as **a *critical state* for diverging into new solutions:**

$$f(s_i) = g(s_i) + h(s_i) \tag{1}$$

$$s^* = \arg\min_{s_i \in T} f(s_i) \tag{2}$$

- $g(s_i) = 1 - \frac{i}{L}$ denotes the context deficiency cost, which penalizes states s_i that lie early in the trajectory, where L is the total length of the trajectory T. A smaller i indicates that s_i is closer to the start of the trajectory, resulting in a larger $g(s_i)$ value, implying that the state s_i has accumulated insufficient contextual information. By discouraging exploration from states that are too shallow in the execution trajectory, this penalty term favors states with a sufficient contextual foundation.
- $h(s_i)$ denotes the solution convergence cost, which penalizes states s_i where the fixing solution has already converged. We introduce an LLM-driven evaluator to rank all candidate states. The evaluation considers the following factors: (a) whether state

Task: Rank the states in the given execution trajectory to identify the suitable **branching point** for generating diverse modification strategy.
Criteria:
A state s_i should be ranked higher if it satisfies both of the following conditions:
(a) Undetermined Modification Location: No specific files, classes, or methods identified yet.
(b) Undetermined Modification Strategy: No code modification steps or fix direction committed yet.
Input: - [Issue] - [Trajectory_States]
Output: A JSON array of states indices ranked.

(a) Prompt Template of Critical State Selection

Task: Using held solution as a reference baseline, identify defects in counterpart solution across two dimensions.
Criteria :
(a) Completeness Review: Does counterpart solution miss any critical modification locations or fix logic?
(b) Regression Risk Review: Does counterpart solution introduce risks to existing code structure?
Input: - [Issue] - [Held Solution] - [Counterpart Solution]
Output: Critiques (Held solution → counterpart solution)

(b) Prompt Template of Reviewer Agents

Task: Act as a *judge agent* to synthesize two candidate solutions into a globally solution.
Steps:
1.Defect Verification: Validate each critique against the corresponding code context.
2.Complementarity Analysis : identify relative strengths of each surviving solution across dimensions<*R*,*E*,*M*,*C*>.
3.Semantic Synthesis : Rewrite retained strengths from both solutions into a unified repair plan.
Input: $G_{i,1}$, $G_{i,2}$, **Critiques**($G_{i,1} \rightarrow G_{i,2}$), **Critiques**($G_{i,2} \rightarrow G_{i,1}$)
Output: Integrated Solution <*R*, *E*, *M*, *C*>

(c) Prompt Template of Judge Agents

Figure 4: Prompt Templates of Prism (simplified)

s_i has already pinpointed specific modification locations (e.g., identifying concrete files, classes, and methods to be modified); (b) whether state s_i has begun substantially planning concrete code modification steps (e.g., having started to outline specific modification steps within a class). The LLM outputs a list of states ranked in ascending order of solution convergence, such as $\{s_4, s_2, \ldots, s_L\}$. Let $Rank(s_i)$ denote the index position of state s_i in this list, which is then converted into the h value, i.e., $h(s_i) = Rank(s_i)/L$. For example, if state s_4 ranks first after sorting (i.e., $Rank(s_4) = 0$), then its $h(s_i)$ is 0. This mechanism favors states whose modification locations and strategies remain underdetermined by assigning them a lower h penalty.

In summary, $g(s_i)$ penalizes states with insufficient context, while $h(s_i)$ penalizes states where the repair solution has already converged. Their combination provides a heuristic score for selecting a suitable branching point that has accumulated sufficient context while the solution remains unconverged (see Figure 4(a)).

3.3 Solution Synthesis

Guided by the contrastive semantic constraint mechanism, the *local exploration* stage generates for each global solution G_i a set of candidate sub-solutions that exhibit substantial differences in modification locations and strategies. Such diversity often enables different sub-solutions to cover distinct dimensions of the problem. However, if evaluated in isolation and selected individually, the complementary strengths among these sub-solutions would be discarded, potentially resulting in a final patch that still fails to fully resolve the issue. To address this, Prism introduces a multi-agent review and solution synthesis mechanism: for two local sub-solutions $G_{i,1}$ and $G_{i,2}$ generated under a global solution G_i, we construct a review environment consisting of two *reviewer agents* and one *judge agent* (see Figure 4(b-c) for the prompt template):

(1) ***Reviewer Agents:*** Each *reviewer agent* holds one of the sub-solutions and, using its own solution as a reference baseline, performs reviews on the counterpart sub-solution across the following two dimensions:

- ***Completeness Review***: The repair of complex issues often involves coordinated modifications across multiple files and functions, making it easy for a single solution to overlook modification locations or edge cases. During the completeness review phase, the *reviewer agent* takes the modification locations and logic covered by its own solution as a reference baseline, focusing on identifying missing modification locations in the counterpart solution, including uncovered cross-file dependencies, overlooked collaborative modification logic, and unaddressed edge cases.
- ***Regression Risk Review***: Introducing new defects while fixing a bug represents a failure mode equally important as under-repair. During the regression risk review phase, the *reviewer agent* takes the handling of the existing code structure in its own solution as a reference baseline, focusing on identifying modification logic in the counterpart solution that may disrupt existing functionality, including disruptions to data flow, violations of interface usage conventions, and the introduction of unhandled exceptions.

(2) ***Judge Agent:*** Unlike the localized critical perspective of the *reviewer agent*, the *judge agent* takes as input both sub-solutions $G_{i,1}$ and $G_{i,2}$, along with their respective defect reports. This enables the *judge agent* to perform a cross-solution, semantic-level synthesis process comprising the following three steps:

- ***Defect Verification.*** The *judge agent* independently evaluates each defect identified in the two defect reports against the corresponding code context to determine whether the defect indeed exists in the target solution. For defects that are verified, the associated repair logic is flagged as problematic and subject to replacement or correction during subsequent synthesis. For defects lacking sufficient evidence to be verified, they are excluded to avoid interfering with the subsequent synthesis process.
- ***Complementarity Analysis.*** After removing the problematic repair logic, the *judge agent* conducts a complementarity analysis on the valid parts of the two solutions. Using the repair plan quadruple $G = \langle R, E, M, C \rangle$ as the unit of analysis, the *judge agent* identifies the relative strengths of each solution across each dimension. For instance, if $G_{i,1}$ covers more cross-file modifications while $G_{i,2}$ handles boundary conditions more comprehensively, the strengths of the two solutions are distributed across different sub-tasks, exhibiting semantic complementarity. The *judge agent* is the only role with simultaneous access to both solutions and their respective defect reports, enabling it to make cross-solution judgments and synthesis decisions with complete information.
- ***Semantic Synthesis.*** The *judge agent* reorganizes the strengths of each solution at the semantic level of the repair plan quadruple $G = \langle R, E, M, C \rangle$, producing an integrated solution.

3.4 Patch Generation

After undergoing the *global exploration*, *local exploration*, and *solution synthesis* stages, Prism constructs a candidate pool comprising multiple repair solutions. The objective of this phase is to translate these natural language-based solutions into actual code patches and select the final submission version. To this end, we introduce

a *coder agent*, which takes as input the issue description along with the candidate solution, and converts the semantic-level repair strategies into concrete code modifications, thereby generating a corresponding code patch for each candidate solution.

We first adopt the approach from AGENTLESS [46] by introducing a dual verification mechanism consisting of static verification and dynamic verification to progressively sanitize the patches generated from the candidate repair pool. Static verification performs syntactic checks on all patches, filtering out invalid patches that introduce syntax errors or references to undefined variables. Dynamic verification executes only visible repository tests selected from the issue context in an isolated sandbox; it does not access the benchmark's hidden evaluation tests.

Finally, following Trae [10], we apply *majority voting* over verified patches to select the highest-consensus one as the final repair submission. By following this pipeline of *generation*, *verification*, and *voting*, PRISM effectively filters out low-quality patches and maximizes the correctness of the final output patch.

4 Evaluation

This section evaluates the proposed PRISM framework by addressing four core research questions:

- **RQ1 (*Overall Performance*)**: *How effective is PRISM in repository-level issue resolution compared to SOTA baseline agents?*
- **RQ2 (*Effectiveness of Exploration Mechanism*)**: *How do the **Global** and **Local Exploration** mechanisms contribute to PRISM's issue resolution rate?*
- **RQ3 (*Effectiveness of Synthesis Mechanism*)**: *Can **Solution Synthesis** mechanism effectively combine the advantages of different candidate solutions to improve PRISM's issue resolution rate?*
- **RQ4 (*Cost-Efficiency Analysis*)**: *How does PRISM's issue resolution rate vary with its token consumption and wall-clock time?*

4.1 Experiment Setup

Baselines. To comprehensively evaluate our proposed approach, we select baselines primarily based on their ranking on the public *SWE-bench Verified* and *SWE-bench-Live Lite* leaderboards, prioritizing top-ranked agents. Beyond rank, these agents also cover different strategies (e.g., large-scale sampling, dynamic tool synthesis, experience-based repair), allowing us to compare PRISM's exploration-synthesis architecture against other approaches.

- ***Agent Selection***: (1) ***SWE-Agent*** [50] optimizes the interaction interface between LLMs and code repositories (e.g., file editing, repository navigation, and test execution), serving as a widely recognized baseline for issue resolution. (2) ***Mini-SWE-Agent*** [50] is a minimalist variant of SWE-Agent that relies solely on shell commands for LLM interaction, designed to assess model capabilities in highly constrained environments. (3) ***Live-SWE-Agent*** [47] dynamically and autonomously creates tools during issue resolution, achieving SOTA performance on the *SWE-bench Verified* leaderboard. (4) ***Trae Agent*** [10] introduces a set reasoning paradigm that formulates problem-solving as an optimal solution search, leveraging pruning and selection to efficiently navigate massive solution spaces. (5) ***Lingxi*** [52] leverages repair experience extracted from historical bug fixes, enabling the agent to comprehend issue resolution mechanisms from multiple perspectives. (6) ***OpenHands*** [44] is a widely adopted open-source platform designed for general-purpose AI software development. (7) ***JoyCode*** [19] is an AI-powered IDE tailored for enterprise-level scenarios, demonstrating exceptional performance in repository-level issue resolution. (8) ***Sonar Foundation Agent*** [37] builds on *AutoCodeRover* [65] with agentic search and tool-calling, jointly topping the *SWE-bench Verified* leaderboard. (9) ***EPAM AI/Run Developer Agent*** [8] is an enterprise-oriented AI agent for end-to-end repository-level issue resolution.
- ***LLM Selection***: We selected two LLMs representing distinct technical paradigms: an open-source model and a proprietary model. (1) ***DeepSeek-V3.2-Reasoner*** [25] exhibits strong reasoning capabilities, excelling in complex problem-solving and code analysis tasks, and represents the SOTA performance among open-source models in this domain. (2) ***Claude-4.6-Sonnet*** [4], an LLM variant developed by *Anthropic*, is widely recognized as a leading model for coding, agentic workflows, and long-horizon autonomous tasks, demonstrating exceptional proficiency in code planning and resolving complex, multi-system bugs.

Benchmark. To ensure rigorous, comprehensive, and realistic evaluation, we use two complementary benchmarks, enabling fair comparison with prior work while mitigating data leakage risks.

- ***SWE-bench Verified*** [18]: This benchmark consists of 500 high-quality issue instances, rigorously filtered from the original *SWE-bench* and human-verified, covering 12 popular Python repositories. Leveraging this widely recognized subset enables our results to be directly compared against all leading SOTA agents.
- ***SWE-bench-Live Lite*** [64]: Recent empirical studies [1] reveal fundamental limitations in static benchmarks: over 94% of *SWE-bench* instances predate modern LLMs' knowledge cutoff, and roughly 32.67% exhibit "solution leakage" in issue descriptions or comments. To address this concern and assess generalization under reduced data-leakage risk, we use *SWE-bench-Live Lite* as a complementary benchmark, which provides dynamically updated issue instances from actively evolving repositories.

Evaluation Metrics. We adopt ***Pass@1*** (*Resolution Rate*) as our core quantitative evaluation metric. It is important to note that, since PRISM generates multiple candidate patches during its exploration and synthesis stages, we introduce a majority voting mechanism to strictly adhere to the definition of *Pass@1*—which *permits only one final submission attempt.* At the final output stage, PRISM evaluates all candidates in the pool for consistency, performs a voting process, and ultimately selects a single patch for evaluation.

Implementation Details. In PRISM, we set the sampling number in the *global exploration* stage to $N_G = 3$ and that in the *local exploration* stage to $N_L = 2$. This configuration is motivated by two considerations: (1) The *global exploration* stage is structurally equivalent to the layer-wise expansion of a search tree. Following tree-structured reasoning methods such as ToT [53], MCTS [60], and their subsequent works [11, 49], which commonly use a branching factor of 3 to balance coverage and cost, we set $N_G = 3$. (2) Each globally explored solution independently undergoes local exploration and synthesis. We set $N_L = 2$, the minimum required to activate the synthesis mechanism, since larger values would introduce multiplicative overhead with the global sampling factor $N_G \times N_L$. Under this configuration, the complete workflow generates 3 global solutions, 6 local solutions (3×2 local solutions

Table 1: Overall Effectiveness of Issue Resolution

Approach	*Base Model*	*%Resolved*	*Pass@1*
SWE-bench Verified			
Live-SWE-Agent	*Claude 4.5 Opus Medium*	79.20%	-
Sonar Foundation Agent	*Claude 4.5 Opus*	79.20%	-
Trae	*Doubao-Seed-Code*	78.80%	√
Live-SWE-Agent	*Gemini3-Pro*	77.40%	-
EPAM AI	*Claude 4 Sonnet*	76.80%	-
mini-SWE-agent	*Claude 4.5 Opus High*	76.80%	√
JoyCode	*Claude 4 Sonnet & GPT-4.1*	74.60%	-
Lingxi-V1.5	*Claude 4 Sonnet*	74.60%	√
Mini-SWE-Agent	*Claude 4.5 Opus Medium*	74.40%	√
OpenHands	*GPT-5*	71.80%	√
Mini-SWE-Agent	*DeepSeek-V3.2-Reasoner*	70.00%	√
Our Agent			
Prism	*DeepSeek-V3.2-Reasoner*	**80.00%**	√
Prism	*Claude-4.6-Sonnet*	**82.00%**	√
SWE-bench-Live Lite			
SWE-agent	*Claude-4.5-Sonnet*	**36.00%**	√
OpenHands	*Qwen3-Coder-480B-A35B*	**24.67%**	√
SWE-agent	*GPT5-Thinking*	**24.30%**	√
OpenHands	*Claude 3.7 Sonnet*	**17.67%**	√
Our Agent			
Prism	*DeepSeek-V3.2-Reasoner*	**42.33%**	√
Prism	*Claude-4.6-Sonnet*	**44.67%**	√

'-' indicates that the agent's resolution rate is not explicitly reported as *Pass@1*; The **%Resolved** results for baselines are derived from the official leaderboards of *SWE-bench Verified* and *SWE-bench-Live Lite*.

across the 3 global branches), and 3 integrated solutions, forming a candidate pool of 12 patches.

The *Resolution Rate* for all baselines is sourced from the official leaderboards. For *SWE-bench-Live Lite*, our comparative analysis focuses on *SWE-Agent* and *OpenHands*, which are the baselines with officially reported results on the *SWE-bench-Live* leaderboard.

4.2 RQ1: *Overall Performance*

Methodology: To rigorously assess the effectiveness of our proposed Prism on real-world repository-level issue resolution tasks, we perform comparative evaluations on *SWE-bench Verified* and *SWE-bench-Live Lite*. The evaluation protocol follows the standard agent testing paradigm: the agent is given only (1) the original issue description and (2) the associated code repository. Operating end-to-end within an isolated sandbox environment, it autonomously navigates the code repository, performs editing operations, and executes only visible repository tests, not hidden benchmark tests, before producing one final patch through majority voting.

Results: Experimental results on the *SWE-bench Verified* demonstrate that Prism outperforms existing SOTA baselines across all foundation model configurations (see Table 1 for details). Specifically, when integrated with *Claude-4.6-Sonnet, Prism achieves a* Pass@1 *of 82.0%*, yielding substantial improvements ranging from 2.8% to 12.0% over various baseline methods. Compared to *Trae*, which also adopts *Pass@1* as its evaluation metric, Prism achieves an improvement of 3.2%. These results demonstrate the strong performance of Prism among existing issue resolution agents.

It is noteworthy that Prism also demonstrates effectiveness across different LLM. When underpinned by the *DeepSeek-V3.2-Reasoner*, Prism attains a *Pass@1* of 80.0%. This performance not only outperforms all baselines, whose resolution rates range from 70.0% to 79.2%, but also surpasses certain agents leveraging proprietary models, exemplified by *Live-SWE-Agent* with *Claude 4.5 Opus*. The performance enhancement is especially pronounced under the identical *DeepSeek-V3.2-Reasoner* configuration: Prism elevates the *Pass@1* from 70.0% (achieved by *Mini-SWE-Agent*) to 80.0%. This absolute gain of 10.0% suggests that the observed improvement is not solely attributable to the underlying LLM, but is also associated with Prism's multi-solution exploration and synthesis mechanisms. Notably, this 80.0% also exceeds *Mini-SWE-Agent*'s 76.8% achieved with the *Claude 4.5 Opus* (high reasoning).

On the dynamic *SWE-bench-Live Lite* benchmark, which is specifically designed to reduce data leakage risks, Prism also achieves strong performance. Specifically, powered by *DeepSeek-V3.2-Reasoner*, Prism achieves a *Pass@1* of 42.33%, surpassing the current SOTA combination of *SWE-Agent* with *Claude-4.5-Sonnet* (36.0%, which relies on a proprietary model) by a substantial margin of 6.3%. This result provides complementary evidence of Prism's generalization with reduced data leakage concerns.

API Cost: When powered by *DeepSeek-V3.2-Reasoner*, Prism consumes an average of 12.88M input tokens (12.30M cache hits) and 0.19M output tokens per issue, incurring an API cost of approximately $2.45/issue. When powered by *Claude-4.6-Sonnet*, the token consumption is lower (3.03M input, 2.81M cache hits, 0.07M output), with $2.51/issue. To contextualize these costs, we compare them against the human cost of bug repair. Prior works report that the median time for fixing a real-world bug is 0.68 hours [21, 61]. At $40/h [38], the estimated human cost per issue is $27.2, providing a coarse reference point for interpreting Prism's API cost.

> ***Conclusion:*** On *SWE-bench Verified*, Prism achieves new SOTA results: 82.0% *Pass@1* with *Claude-4.6-Sonnet* and 80.0% with *DeepSeek-V3.2-Reasoner*. On the *SWE-bench-Live Lite*, Prism (with *DeepSeek-V3.2-Reasoner*) surpasses the leading closed-source SOTA combination (*SWE-Agent + Claude-4.5-Sonnet*, 36.0%), achieving 42.33%. This result provides complementary evidence of Prism's generalization under reduced leakage concerns, while same-model results support its architectural contribution.

4.3 RQ2: *Effectiveness of Exploration Mechanism*

Methodology: A standard remove-one-stage ablation is not directly applicable to Prism, as its stages are contextually dependent: local exploration branches from trajectories accumulated during global exploration, and solution synthesis requires paired local sub-solutions. Removing earlier stages would therefore break the required inputs for later stages. However, solutions produced at each stage can still be passed to patch generation. Thus, we adopt the experimental methodology established in prior work [22, 28, 34, 46] and use *candidate pool partitioning* to quantify each stage's contribution to the issue resolution rate. Specifically, we execute Prism's complete workflow, record all natural-language solutions generated at each stage, and feed each stage-specific solution pool to the coder agent for patch generation and majority voting. We compute *Pass@1* to estimate each stage's contribution:

- ***Baseline (High-Temperature Sampling)***: This configuration does not employ Prism's *exploration* or *synthesis* mechanisms. It generate 12 solutions through *high-temperature sampling*.
- ***Prism*** ($N_G = 1, N = 4$)***:*** This candidate pool is produced by narrowing *global exploration* to one direction ($N_G = 1$) while

Table 2: Performance Analysis of Prism across Exploration Stages

Approach	*Candidate Plans (N)*	*Pass@1(%)*	*Pass@N(%)*
Prism	12	**80.00%**	**84.60%**
Baseline	12	71.8%(↓**8.2%**)	75.8%(↓**8.8%**)
Prism (N_G**=1)**	4	73.2%(↓**6.8%**)	77.2%(↓**7.4%**)
Global-Only	3	74.8%(↓**5.2%**)	80.0%(↓**4.6%**)
Local-Only	6	75.0%(↓**5.0%**)	80.4%(↓**4.2%**)

retaining *local exploration* and *synthesis*, consisting of one global, two local sub-solutions, and one synthesized solution.

- ***Global-Only Candidates (N=3)***: This candidate pool consists of the 3 initial global solutions generated by *global exploration*.
- ***Local-Only Candidates (N=6)***: This candidate pool contains only the 6 sub-solutions refined through *local exploration* from the 3 global branches, with 2 local sub-solutions per branch.

All configurations use *DeepSeek-V3.2-Reasoner* and apply majority voting for *Pass@1*. Because candidates vary in size (N = 3, 4, 6, 12), *Pass@1* may be affected by the number of candidates in the voting. We therefore treat these comparisons as stage-level contribution analyses rather than size-controlled ablations, and report *Pass@N* to reflect each pool's coverage of the solution space.

Results: Table 2 presents a performance comparison between the complete framework and its individual stages. With the identical total number of candidate solutions (N=12), the complete Prism framework achieves a *Pass@1* of 80.0% and a *Pass@12* of 84.6%. In contrast, the baseline employing only high-temperature sampling attains 71.8% (*Pass@1*) and 75.8% (*Pass@12*). This 8.2% improvement in the *Pass@1* issue resolution rate substantiates that the performance gains are not attributable to generating a larger volume of solutions. Instead, they stem from the framework's global and local exploration mechanisms, which not only broaden the search space for correct solutions (as evidenced by the enhanced *Pass@N*) but also improve the consistency of high-quality candidates, thereby enabling the majority voting to pinpoint the correct patch.

Moving from *global exploration* to *local exploration*, Prism resolves more issues. Using only the initial *global exploration* solutions, Prism achieves 74.8% *Pass@1* and 80.0% *Pass@3*. Remarkably, compared to the baseline, which generates 12 solutions via high-temperature sampling, *global exploration* attains a 3% higher resolution rate using only 3 solutions. With the incorporation of *local exploration*, the *Pass@N* further improves to 80.4%. These results indicate that Prism's exploration mechanism effectively captures correct solutions that are easily overlooked from a single perspective, thereby enhancing the overall quality of the solution pool. We further examine the breadth of *global exploration* by reducing N_G from 3 to 1 while retaining *local exploration* and *synthesis*. This configuration achieves 73.2% *Pass@1* and 77.2% *Pass@N*, indicating that broader *global exploration* contributes to the resolution rate.

Case Study. Figure 5(a) illustrates a cross-file issue that requires three coordinated fixes: adding fallback logic in *RelatedFieldListFilter* (`TODO#1`), passing ordering parameters in *RelatedOnlyFieldListFilter* (`TODO#2`), and fixing lower-level ordering logic in `fields/init.py` and `reverse_related.py` (`TODO#3`). Existing agents tend to follow a single repair path and thus produce partial fixes: *Trae* stops at the *filters.py*-level changes, while *Live-SWE-Agent*, despite extracting helper functions, still misses the field-ordering logic. This illustrates how single-path exploration can converge to incomplete local optima on multi-file issues.

Prism avoids this premature convergence through staged exploration. In global exploration, it generates semantically different directions: G_2 focuses on filter-level fallback logic, whereas G_3 inspects the underlying field-ordering behavior. Local exploration then expands these directions: $G_{2,1}$ recovers the missing *RelatedOnlyFieldListFilter* change, while $G_{3,1}$ further incorporates filter-level fallback logic. This staged process constructs a candidate pool that covers complementary repair dimensions, enabling later stages to approach the developer's complete fix.

Conclusion: The progressive exploration mechanism of Prism shows a significant advantage over random sampling. With the same candidate pool size (N = 12), Prism achieves a *Pass@1* of 80.0%, compared to 71.8% for the *high-temperature sampling* baseline. This indicates that Prism effectively navigates the solution space to generate high-quality patches, rather than merely relying on a larger number of candidate solutions.

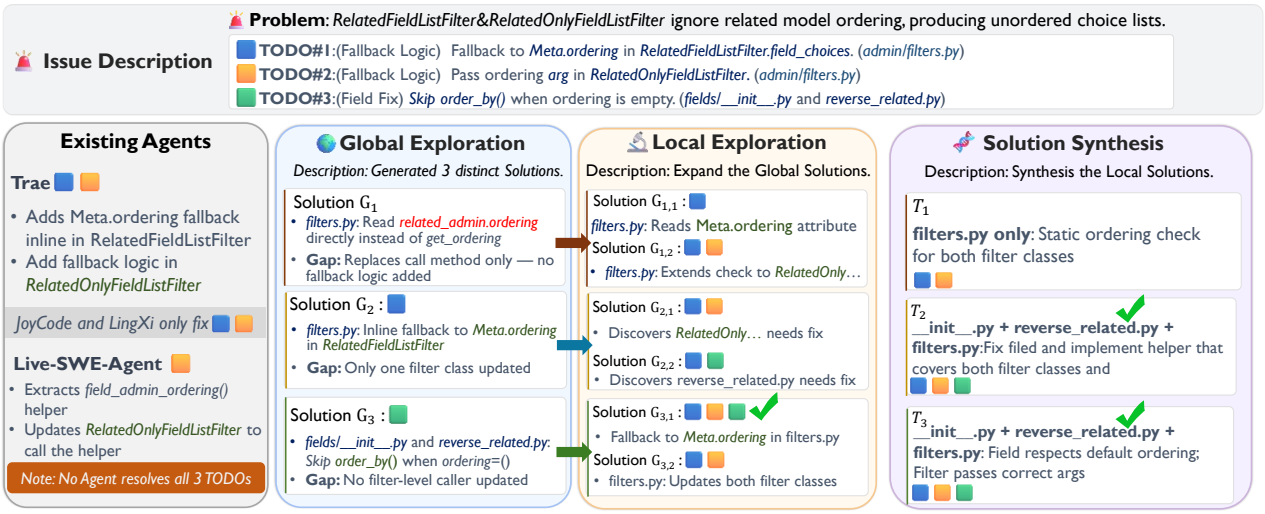
4.4 RQ3: *Effectiveness of Synthesis Mechanism*

Methodology: RQ3 evaluates the contribution of the *solution synthesis* to Prism's overall resolution rate. Building on RQ2 results, we compare the performance of synthesis-stage candidates against those from the preceding exploration stages. To determine whether synthesis can generate entirely new patches not identified during exploration, we extract the sets of successfully resolved issues from each stage and analyze them using *UpSet plots* [20].

Results: The results further reveal the critical role of the *solution synthesis*. The *global exploration* generated three candidates, achieving a *Pass@1* of 74.8%. Following this, the six solutions produced during the *local exploration* phase yielded a *Pass@1* of 75.0%. The *solution synthesis* stage combined these six solutions in pairs to form three solutions, resulting increase in *Pass@1* to 79.2%, with *Pass@N* reaching 83.4%. This shows that the *synthesis* mechanism combines the complementary strengths of multiple solutions, addressing the limitations of individual solutions from the *local exploration* stage and thereby improving the overall issue resolution rate.

Figure 6 underscores the necessity of retaining all candidate solutions (N=12) across Prism's three stages: (1) ***High Overlap:*** Most successful cases (350) are resolved by all stages, confirming Prism's capability on routine issues. (2) ***Stage Complementarity:*** Each stage also contributes exclusively to a unique set of successes. The *solution synthesis* stage exclusively resolves 15 cases. Similarly, the *global exploration* and *local exploration* stages contribute 8 and 2 issues, respectively. By harnessing this complementarity, the Prism combines solutions from all stages into its final candidate pool. This holistic strategy elevates the theoretical upper bound (*Pass@N*) to 84.6% and, achieves an SOTA *Pass@1* performance of 80.0%.

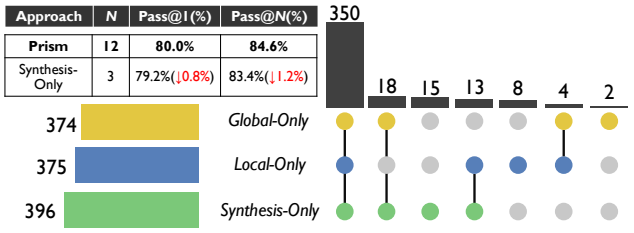
Case Study: As shown in Figure 5(b), although the *local exploration* stage generated solutions covering different combinations of `TODO`s, most of these solutions still had limitations. Among the three integrated solutions, *T1* attempted to unify the logic solely within *filters.py* (addressing `TODO#1` and `TODO#2`), but remained incomplete. In contrast, *T2* identified the need for cross-file repair: it extracted the underlying field fix logic (`TODO#3`) and integrated it with the filter fallback logic (`TODO#1` and `TODO#2`), producing an integrated patch that covers all three modules. It ensures that the underlying fields respect default ordering while enabling filters to pass correct parameters, resolving all three `TODO`s.



(a) Global and Local Exploration: Prism generates three globally diverse solutions (G_1–G_3) and refines them into six local sub-solutions ($G_{1,1}$ – $G_{3,2}$).

(b) Synthesis : Prism synthesizes local solutions into three synthesized solutions(T_1 – T_3).

Figure 5: An example (*issue Django#11400*) for illustrating the effectiveness of exploration mechanism



(1) The horizontal bar chart (left) displays the number of issues successfully resolved by each individual stage. **(2)** The vertical bar chart (top) illustrates the intersection sizes of issues resolved by different stage combinations. **(3)** Solid circles connected by vertical lines indicate the stages participating in each combination, while gray circles represent stages that did not contribute to resolving issues in that particular combination.

Figure 6: Intersection analysis of successfully resolved issues across global exploration, local exploration, and solution synthesis stages

This case highlights the value of *solution synthesis* mechanism: rather than simply selecting a solution, it reconstructs multiple exploratory solutions, each with partial defects yet complementary strengths, into a comprehensive final solution. To further examine whether these synthesis-exclusive successes result from integration rather than simple candidate selection, we manually analyzed the 15 cases: 10 perform cross-file integration, 3 perform within-file integration, and 2 perform boundary-condition fusion; no case simply selects or lightly rewrites a single candidate solution. This explains the *Pass@1* gain from 75.0% to 79.2%, suggesting that the improvement comes from synthesis-driven integration rather than candidate selection alone.

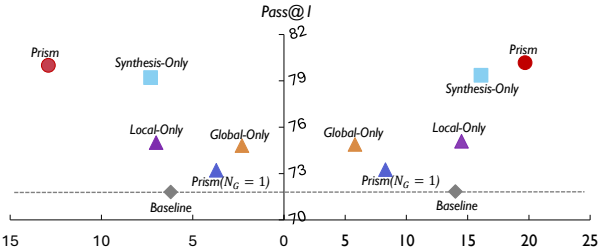
Conclusion: The synthesis stage is critical: with only three integrated candidates, Prism achieves 79.2% *Pass@1*, outperforming the *global exploration* (74.8% with 3) and *local exploration* (75.0% with 6). An *UpSet plot* confirms its contribution, with 15 exclusively resolved issues.

4.5 RQ4: *Cost-Efficiency Analysis*

Methodology: To further evaluate the practical efficiency of Prism, we compare its issue resolution rate, *token consumption*, and *wall-clock time* across the configurations studied in RQ2 and RQ3. All

Table 3: Efficiency Analysis of Prism across Evaluated Settings

Approach	*Tokens (M)*	*Time (min)*	*Pass@1(%)*
Prism	**12.88**	**19.7**	**80.0%**
Baseline	6.24	14.0	71.8%
Prism (N_G=1)	3.73	8.3	73.2%
Global-Only	2.32	5.8	74.8%
Local-Only	6.96	14.5	75.0%
Synthesis-Only	7.34	16.1	79.2%



(a) Token Consumption (*M*) vs *Pass@1* **(b) Wall-clock Time (*min*) vs *Pass@1***

Figure 7: Token and latency analysis on *SWE-bench Verified*.

measurements are conducted on *SWE-bench Verified* using *DeepSeek-V3.2-Reasoner* as the base model, consistent with the setting used in RQ2 and RQ3. *Token consumption* is computed by averaging the total number of input and output API tokens over all evaluated issues. *Wall-clock time* is computed by averaging the end-to-end latency per issue over all evaluated issues for each setting.

Results: Table 3 and Figure 7 show that Prism's gains are not merely caused by larger test-time compute. *Global-Only* consumes 2.32M tokens per issue and achieves 74.8% *Pass@1*, outperforming the high-temperature sampling baseline, which consumes 6.24M tokens but achieves only 71.8%. Similarly, Prism with $N_G = 1$ consumes 3.73M tokens and achieves 73.2% *Pass@1*, still surpassing the baseline with fewer tokens. The same trend holds for wall-clock time. *Global-Only* reaches 74.8% *Pass@1* in 5.8 minutes, outperforming the baseline

at lower latency. *Local-Only* achieves 75.0% in 14.5 minutes, while *Synthesis-Only* improves *Pass@1* to 79.2% in 16.1 minutes.

The full PRISM achieves the best *Pass@1* of 80.0% with 12.88M tokens and an average wall-clock time of 19.7 minutes per issue. Under the same candidate budget as unstructured high-temperature sampling ($N = 12$), this result highlights the benefit of PRISM's structured exploration-synthesis design. Since prior SWE-bench studies often emphasize resolution rate and provide limited wall-clock latency reporting [6, 48, 50, 64], we additionally report end-to-end latency to quantify the cost of PRISM. The measured latency captures the runtime of multiple dependent stages, including global exploration, local branching, synthesis, and final patch selection.

Conclusion: Reporting *Pass@1* against both token consumption and wall-clock time shows that PRISM's gains are not merely due to larger test-time compute. *Global-Only* and PRISM($N_G = 1$) outperform high-temperature sampling with lower observed token cost, while full PRISM achieves the highest *Pass@1* with reported token and latency costs.

5 Threats to Validity

Data Leakage: "Data leakage" refers to evaluation data overlapping with LLM pretraining corpora, which can lead to overfitting. To mitigate this, we additionally evaluate PRISM on *SWE-bench-Live Lite* [64], which reduces leakage risk by using issues postdating existing LLMs' knowledge cutoff. The results provide complementary evidence of PRISM's generalization under reduced leakage concerns.
Computational Cost: Direct cost comparison is limited because many agents do not report per-issue cost or token consumption. *Mini-SWE-Agent* [50] and *Live-SWE-Agent* [47] report costs of $0.56 and $0.68 per issue, achieving 70.6% and 75.4% with Claude 4.5 Sonnet, respectively. In contrast, recent test-time scaling methods such as *Trae Agent* do not disclose monetary cost or token usage [10]. To improve transparency, RQ4 reports token consumption and measured end-to-end latency for each evaluated PRISM setting.
Semantic Diversity: PRISM uses semantic divergence as a generation-time constraint to encourage different repair directions. While our stage-wise results and case studies provide evidence of its benefit, a fine-grained quantitative characterization of solution diversity remains an interesting direction for future work.
Reproducibility: LLM-based agents are subject to stochastic sampling during inference, which may affect the reproducibility. To address this concern, we strictly adhere to the official evaluation protocols of both benchmarks and fix all PRISM configurations, including the global exploration (N_G=3) and local exploration (N_L=2).

6 Related Work

Diversity in LLM Reasoning. *Test-Time Scaling* (TTS) served as an important paradigm for enhancing the reasoning performance of LLMs. By allocating additional inference-time resources, it generates multiple candidate solutions via *Best-of-N sampling strategy* or tree search, and subsequently selects the optimal solution [6, 29, 36]. Existing methods primarily relied on temperature scaling or *top-k* sampling to increase candidate diversity [43, 58]. For repository-level issue resolution tasks, *SWE-Search* [3] leveraged the tree-structured architecture of MCTS to explore multiple reasoning trajectories across different branches; *Satori-SWE* [59] achieved TTS by enabling LLMs to self-improve based on the scores of reward models regarding their historically generated outputs; *Trae* [10] achieved ensemble reasoning through an agent architecture to further enhance the capability to resolve complex issues. However, the diversity of these approaches relied on stochastic sampling approaches. Constrained by the mode collapse effect of RLHF fine-tuned models [2, 15, 62], their exploration effectiveness was limited.

Unlike prior approaches, PRISM enforces semantic divergence among candidate solutions across four dimensions through generation time contrastive semantic constraints.
Multi-Agent Collaboration and Solution Synthesis. Multi-agent collaboration and agentic workflows represented a fundamental paradigm that improves LLM performance by leveraging diverse roles, debate, and iterative reflection [26, 35, 39, 45]. Such paradigms have demonstrated broad effectiveness across various tasks, including requirements engineering, code generation, and software maintenance [13]. Relatedly, component-based synthesis constructs programs by composing reusable components under formal specifications [31]. While both lines of work involve combining partial program artifacts, PRISM addresses repository-level issue resolution, where explicit formal specifications are rarely available; its synthesis process instead leverages semantic analysis of candidate solutions. For instance, *EntroPO* [56] proposed a multi-agent framework integrating debate, verification, and debugging to improve automated programming robustness. *Sami et al.*[33] employed agents with distinct roles (e.g., Product Owner, Developer, and QA) to independently analyze requirements, and then synthesized their outputs via a Manager agent. In repository-level issue resolution [17, 26], existing works (e.g., *SWE-Agent* [50], *Lingxi* [52], *Live-SWE-Agent* [47]) employed multi-agent architectures to generate candidate solutions. However, they typically evaluated each candidate solution in isolation, relying on ranking or voting [16, 32, 40]. This paradigm overlooked complementarity among the solutions.

In contrast, PRISM introduces a structured *solution synthesis* mechanism, where the *reviewer agents* identify defects and complementary strengths across solutions via cross-review, and the *judge agent* integrates them into a complete final solution.

7 Conclusion

In this paper, we propose PRISM, a multi-solution reasoning and synthesis framework that addresses diversity and synthesis limitations in repository-level issue resolution. Following a coarse-to-fine paradigm, PRISM broadens the solution space with contrastive semantic constraints, generates divergent sub-solutions via an A^* *algorithm*-inspired mechanism, and integrates complementary patches through multi-agent collaboration. Evaluations on *SWE-bench Verified* and *SWE-bench-Live Lite* show that PRISM achieves state-of-the-art performance.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 92582201), the National Key R&D Program of China (No. 2024YFF0908000), the Liaoning Provincial Central-Guided Local S&T Development Special Project–Regional Innovation System Construction (No. 2026JH6/100400004), the Open Fund of the National Key Laboratory of Parallel and Distributed Computing (PDL) (No. 2026RJ0601), and the China Postdoctoral Science Foundation (No. 2024M750375).

Data Availability Statement

A reproduction package containing ground truth datasets, tools, and raw experimental data is available at https://doi.org/10.5281/zenodo.19249832 to support further research. Our project page is available at https://prism-agent-code.github.io/.

References

[1] Reem Aleithan, Haoran Xue, Mohammad Mahdi Mohajer, Elijah Nnorom, Gias Uddin, and Song Wang. 2024. Swe-bench+: Enhanced coding benchmark for llms. *arXiv preprint arXiv:2410.06992* (2024). doi:10.48550/arXiv.2410.06992

[2] Oron Anschel, Alon Shoshan, Adam Botach, Shunit Haviv Hakimi, Asaf Gendler, Emanuel Ben Baruch, Nadav Bhonker, Igor Kviatkovsky, Manoj Aggarwal, and Gerard Medioni. 2025. Group-aware reinforcement learning for output diversity in large language models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 32382–32403. doi:10.18653/v1/2025.emnlp-main.1649

[3] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. Swe-search: Enhancing software agents with monte carlo tree search and iterative refinement. *arXiv preprint arXiv:2410.20285* (2024). doi:10.48550/arXiv.2410.20285

[4] Claude. 2026. Claude 4.6 Sonnet. https://code.claude.com

[5] Jonathan Cordeiro, Shayan Noei, and Ying Zou. 2024. An empirical study on the code refactoring capability of large language models. *ACM Transactions on Software Engineering and Methodology* (2024). doi:10.1145/3801158

[6] Yifeng Ding and Lingming Zhang. 2026. SWE-Replay: Efficient Test-Time Scaling for Software Engineering Agents. *arXiv preprint arXiv:2601.22129* (2026). doi:10.48550/arXiv.2601.22129

[7] Django. 2022. Django. https://code.djangoproject.com/ticket/33413

[8] EPAM. 2026. EPAM AI. https://www.epam.com

[9] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. 2023. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*. IEEE, 31–53. doi:10.1109/ICSE-FoSE59343.2023.00008

[10] Pengfei Gao, Zhao Tian, Xiangxin Meng, Xinchen Wang, Ruida Hu, Yuanan Xiao, Yizhou Liu, Zhao Zhang, Junjie Chen, Cuiyun Gao, et al. 2025. Trae agent: An llm-based agent for software engineering with test-time scaling. *arXiv preprint arXiv:2507.23370* (2025). doi:10.48550/arXiv.2507.23370

[11] Rui Ha, Chaozhuo Li, Rui Pu, Litian Zhang, Xi Zhang, and Sen Su. 2025. DSG-MCTS: A Dynamic Strategy-Guided Monte Carlo Tree Search for Diversified Reasoning in Large Language Models. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 10541–10555. doi:10.18653/v1/2025.emnlp-main.532

[12] Peter E Hart, Nils J Nilsson, and Bertram Raphael. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics* 4, 2 (1968), 100–107. doi:10.1109/TSSC.1968.300136

[13] Junda He, Christoph Treude, and David Lo. 2025. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Transactions on Software Engineering and Methodology* 34, 5 (2025), 1–30. doi:10.1145/3712003

[14] Xinyi Hou, Yanjie Zhao, Yue Liu, Zhou Yang, Kailong Wang, Li Li, Xiapu Luo, David Lo, John Grundy, and Haoyu Wang. 2024. Large language models for software engineering: A systematic literature review. *ACM Transactions on Software Engineering and Methodology* 33, 8 (2024), 1–79. doi:10.1145/3695988

[15] Xiuyuan Hu, Guoqing Liu, Yang Zhao, Jieran Li, Dongbiao Sun, José Miguel Hernández-Lobato, Hao Zhang, and Xue Liu. 2025. Exploring the Trade-off between Quality and Diversity of Language Models during Reinforcement Learning. *OpenReview* (2025).

[16] Audrey Huang, Adam Block, Qinghua Liu, Nan Jiang, Akshay Krishnamurthy, and Dylan J Foster. 2025. Is best-of-n the best of them? coverage, scaling, and optimality in inference-time alignment. *arXiv preprint arXiv:2503.21878* (2025). doi:10.48550/arXiv.2503.21878

[17] Zhonghao Jiang, David Lo, and Zhongxin Liu. 2025. Agentic Software Issue Resolution with Large Language Models: A Survey. *arXiv preprint arXiv:2512.22256* (2025). doi:10.48550/arXiv.2512.22256

[18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770* (2023). doi:10.48550/arXiv.2310.06770

[19] JoyCode. 2026. JoyCode-Agent. https://github.com/jd-opensource/joycode-agent

[20] Alexander Lex, Nils Gehlenborg, Hendrik Strobelt, Romain Vuillemot, and Hanspeter Pfister. 2014. UpSet: visualization of intersecting sets. *IEEE transactions on visualization and computer graphics* 20, 12 (2014), 1983–1992. doi:10.1109/TVCG.2014.2346248

[21] Chenglin Li, Yangyang Zhao, Yibiao Yang, Yuming Zhou, Liming Nie, and Zuohua Ding. 2023. Investigating the Impact of Bug Dependencies on Bug-Fixing Time Prediction. In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, 1–12. doi:10.1109/ESEM56168.2023.10304804

[22] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science* 378, 6624 (2022), 1092–1097. doi:10.1126/science.abq1158

[23] Ziniu Li, Congliang Chen, Tian Xu, Zeyu Qin, Jiancong Xiao, Zhi-Quan Luo, and Ruoyu Sun. 2024. Preserving diversity in supervised fine-tuning of large language models. *arXiv preprint arXiv:2408.16673* (2024). doi:10.48550/arXiv.2408.16673

[24] Jiaye Lin, Yifu Guo, Yuzhen Han, Sen Hu, Ziyi Ni, Licheng Wang, Mingguang Chen, Hongzhang Liu, Ronghao Chen, Yangfan He, et al. 2025. Se-agent: Self-evolution trajectory optimization in multi-step reasoning with llm-based agents. *arXiv preprint arXiv:2508.02085* (2025). doi:10.48550/arXiv.2508.02085

[25] Aixin Liu, Aoxue Mei, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, et al. 2025. Deepseek-v3.2: Pushing the frontier of open large language models. *arXiv preprint arXiv:2512.02556* (2025). doi:10.48550/arXiv.2512.02556

[26] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. 2024. Large language model-based agents for software engineering: A survey. *ACM Transactions on Software Engineering and Methodology* (2024). doi:10.1145/3796507

[27] Simiao Liu, Fang Liu, Liehao Li, Xin Tan, Yinghao Zhu, Xiaoli Lian, and Li Zhang. 2025. An empirical study on failures in automated issue solving. *arXiv preprint arXiv:2509.13941* (2025). doi:10.48550/arXiv.2509.13941

[28] Mingyu Derek Ma, Yanna Ding, Zijie Huang, Jianxi Gao, Yizhou Sun, and Wei Wang. 2025. Inferring from Logits: Exploring Best Practices for Decoding-Free Generative Candidate Selection. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 33125–33144. doi:10.18653/v1/2025.acl-long.1589

[29] Yingwei Ma, Yongbin Li, Yihong Dong, Xue Jiang, Yanhao Li, Yue Liu, Rongyu Cao, Jue Chen, Fei Huang, and Binhua Li. 2025. Thinking longer, not larger: Enhancing software engineering agents via scaling test-time compute. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 3730–3741. doi:10.1109/ase63991.2025.00309

[30] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, et al. 2021. Webgpt: Browser-assisted question-answering with human feedback. *arXiv preprint arXiv:2112.09332* (2021). doi:10.48550/arXiv.2112.09332

[31] Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (2021), 1–29. doi:10.1145/3485535

[32] Amin Rakhsha, Kanika Madan, Tianyu Zhang, Amir-massoud Farahmand, and Amir Khasahmadi. 2025. Majority of the Bests: Improving Best-of-N via Bootstrapping. *arXiv preprint arXiv:2511.18630* (2025). doi:10.48550/arXiv.2511.18630

[33] Malik Abdul Sami, Muhammad Waseem, Zheying Zhang, Zeeshan Rasheed, Kari Systä, and Pekka Abrahamsson. 2024. AI based multiagent approach for requirements elicitation and analysis. *arXiv preprint arXiv:2409.00038* (2024). doi:10.48550/arXiv.2409.00038

[34] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems* 36 (2023), 8634–8652. doi:10.5555/3666122.3666499

[35] Andries Smit, Paul Duckworth, Nathan Grinsztajn, Thomas D Barrett, and Arnu Pretorius. 2023. Should we be going mad? a look at multi-agent debate strategies for llms. *arXiv preprint arXiv:2311.17371* (2023). doi:10.48550/arXiv.2311.17371

[36] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. 2024. Scaling llm test-time compute optimally can be more effective than scaling model parameters. *arXiv preprint arXiv:2408.03314* (2024). doi:10.48550/arXiv.2408.03314

[37] Sonar. 2026. Sonar Foundation Agent. https://www.sonarsource.com

[38] stackoverflow. 2026. stackoverflow Survey. https://survey.stackoverflow.co/2025/work

[39] Khanh-Tung Tran, Dung Dao, Minh-Duong Nguyen, Quoc-Viet Pham, Barry O'Sullivan, and Hoang D Nguyen. 2025. Multi-agent collaboration mechanisms: A survey of llms. *arXiv preprint arXiv:2501.06322* (2025). doi:10.48550/arXiv.2501.06322

[40] Fernando Vallecillos-Ruiz, Max Hort, and Leon Moonen. 2025. Wisdom and Delusion of LLM Ensembles for Code Generation and Repair. *arXiv preprint arXiv:2510.21513* (2025). doi:10.48550/arXiv.2510.21513

[41] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. 2024. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering* 50, 4 (2024), 911–936. doi:10.1109/TSE.2024.3368208

[42] Qihan Wang, Shidong Pan, Tal Linzen, and Emily Black. 2025. Multilingual prompting for improving llm generation diversity. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*. 6378–6400.

doi:10.18653/v1/2025.emnlp-main.324

[43] Tianchun Wang, Zichuan Liu, Yuanzhou Chen, Jonathan Light, Weiyang Liu, Haifeng Chen, Xiang Zhang, and Wei Cheng. 2025. On the Effect of Sampling Diversity in Scaling LLM Inference. *arXiv preprint arXiv:2502.11027* (2025). doi:10.48550/arXiv.2502.11027

[44] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024). doi:10.48550/arXiv.2407.16741

[45] Haolun Wu, Zhenkun Li, and Lingyao Li. 2025. Can LLM Agents Really Debate? A Controlled Study of Multi-Agent Debate in Logical Reasoning. *arXiv preprint arXiv:2511.07784* (2025). doi:10.48550/arXiv.2511.07784

[46] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. 2024. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489* (2024). doi:10.48550/arXiv.2407.01489

[47] Chunqiu Steven Xia, Zhe Wang, Yan Yang, Yuxiang Wei, and Lingming Zhang. 2025. Live-SWE-agent: Can Software Engineering Agents Self-Evolve on the Fly? *arXiv preprint arXiv:2511.13646* (2025). doi:10.48550/arXiv.2511.13646

[48] Chengxing Xie, Bowen Li, Chang Gao, He Du, Wai Lam, Difan Zou, and Kai Chen. 2025. Swe-fixer: Training open-source llms for effective and efficient github issue resolution. In *Findings of the Association for Computational Linguistics: ACL 2025*. 1123–1139. doi:10.18653/v1/2025.findings-acl.62

[49] Yuxi Xie, Anirudh Goyal, Wenyue Zheng, Min-Yen Kan, Timothy P Lillicrap, Kenji Kawaguchi, and Michael Shieh. 2024. Monte carlo tree search boosts reasoning via iterative preference learning. *arXiv preprint arXiv:2405.00451* (2024). doi:10.48550/arXiv.2405.00451

[50] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652. doi:10.5555/3737916.3739517

[51] Joy Qiping Yang, Salman Salamatian, Ziteng Sun, Ananda Theertha Suresh, and Ahmad Beirami. 2024. Asymptotics of language model alignment. In *2024 IEEE International Symposium on Information Theory (ISIT)*. IEEE, 2027–2032. doi:10.1109/isit57864.2024.10619456

[52] Xu Yang, Jiayuan Zhou, Michael Pacheco, Wenhan Zhu, Pengfei He, Shaowei Wang, Kui Liu, and Ruiqi Pan. 2025. Lingxi: Repository-Level Issue Resolution Framework Enhanced by Procedural Knowledge Guided Scaling. *arXiv preprint arXiv:2510.11838* (2025). doi:10.48550/arXiv.2510.11838

[53] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. *Advances in neural information processing systems* 36 (2023), 11809–11822. doi:10.5555/3666122.3666639

[54] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*. doi:10.48550/arXiv.2210.03629

[55] Ryo Yonetani, Tatsunori Taniai, Mohammadamin Barekatain, Mai Nishimura, and Asako Kanezaki. 2021. Path Planning using Neural A* Search. arXiv:2009.07476 [cs.LG] doi:10.48550/arXiv.2009.07476

[56] Jiahao Yu, Zelei Cheng, Xian Wu, and Xinyu Xing. 2025. Building Coding Agents via Entropy-Enhanced Multi-Turn Preference Optimization. *arXiv preprint arXiv:2509.12434* (2025). doi:10.48550/arXiv.2509.12434

[57] Yuanqing Yu, Zhefan Wang, Weizhi Ma, Zhicheng Guo, Jingtao Zhan, Shuai Wang, Chuhan Wu, Zhiqiang Guo, and Min Zhang. 2025. StepTool: Enhancing Multi-Step Tool Usage in LLMs via Step-Grained Reinforcement Learning. In *Proceedings of the 34th ACM International Conference on Information and Knowledge Management*. doi:10.1145/3746252.3761391

[58] Longfei Yun, Chenyang An, Zilong Wang, Letian Peng, and Jingbo Shang. 2025. The price of format: Diversity collapse in llms. *arXiv preprint arXiv:2505.18949* (2025). doi:10.48550/arXiv.2505.18949

[59] Guangtao Zeng, Maohao Shen, Delin Chen, Zhenting Qi, Subhro Das, Dan Gutfreund, David Cox, Gregory Wornell, Wei Lu, Zhang-Wei Hong, et al. 2025. Satori-SWE: Evolutionary Test-Time Scaling for Sample-Efficient Software Engineering. *arXiv preprint arXiv:2505.23604* (2025). doi:10.48550/arXiv.2505.23604

[60] Dan Zhang, Sining Zhoubian, Ziniu Hu, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024. Rest-mcts*: Llm self-training via process reward guided tree search. *Advances in Neural Information Processing Systems* 37 (2024), 64735–64772. doi:10.52202/079017-2066

[61] Feng Zhang, Foutse Khomh, Ying Zou, and Ahmed E Hassan. 2012. An empirical study on factors impacting bug fixing time. In *2012 19th Working conference on reverse engineering*. IEEE, 225–234. doi:10.1109/WCRE.2012.32

[62] Jiayi Zhang, Simon Yu, Derek Chong, Anthony Sicilia, Michael R Tomz, Christopher D Manning, and Weiyan Shi. 2025. Verbalized sampling: How to mitigate mode collapse and unlock llm diversity. *arXiv preprint arXiv:2510.01171* (2025). doi:10.48550/arXiv.2510.01171

[63] Kexun Zhang, Shang Zhou, Danqing Wang, William Yang Wang, and Lei Li. 2025. Scaling llm inference efficiently with optimized sample compute allocation. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. 7959–7973. doi:10.18653/v1/2025.naacl-long.404

[64] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, et al. 2025. Swe-bench goes live! *arXiv preprint arXiv:2505.23419* (2025). doi:10.48550/arXiv.2505.23419

[65] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604. doi:10.1145/3650212.3680384

Received 2026-03-27; accepted 2026-06-18